Adaptively Secure Streaming Functional Encryption

Pratish Datta* Jiaxin Guan† Alexis Korb‡ Amit Sahai§

February 2024

Abstract

This paper introduces the *first adaptively secure* streaming functional encryption (sFE) scheme for $\mathsf{P/Poly}$. sFE stands as an evolved variant of traditional functional encryption (FE), catering specifically to contexts with vast and/or dynamically evolving data sets. sFE is designed for applications where data arrives in a streaming fashion and is computed on in an iterative manner as the stream arrives. Unlike standard FE, in sFE: (1) encryption is possible without knowledge of the full data set, (2) partial decryption is possible given only a prefix of the input.

Guan, Korb, and Sahai introduced this concept in their recent publication [CRYPTO 2023], where they constructed an sFE scheme for $\mathsf{P/Poly}$ using a compact standard FE scheme for the same. However, their sFE scheme only achieved semi-adaptive-function-selective security, which constrains the adversary to obtain all functional keys prior to seeing any ciphertext for the challenge stream. This limitation severely limits the scenarios where sFE can be applied, and therefore fails to provide a suitable theoretical basis for sFE.

In contrast, the adaptive security model empowers the adversary to arbitrarily interleave requests for functional keys with ciphertexts related to the challenge stream. Guan, Korb, and Sahai identified achieving adaptive security for sFE as the key question left open by their work.

We resolve this open question positively by constructing an adaptively secure sFE construction from indistinguishability obfuscation for $\mathsf{P/Poly}$ and injective PRGs. By combining our work with that of Jain, Lin, and Sahai [STOC 2021, EUROCRYPT 2022], we obtain the first adaptively secure sFE scheme for $\mathsf{P/Poly}$ based on sub-exponential hardness of well-studied computational problems.

Don't panic! One does not need to read all 330 pages of this paper to understand it. For researchers that want all the details of every hybrid, we include them here.

*NTT Research. Email: pratish.datta@ntt-research.com.

†New York University. Email: jiaxin@guan.io. Part of this research was conducted while this author was a Ph.D. student at Princeton University.

‡UCLA. Email: alexiskorb@cs.ucla.edu.

§UCLA. Email: sahai@cs.ucla.edu.

Contents

1 Introduction

In this work, we resolve the main open question posed by the work of Guan, Korb, and Sahai [GKS23], and show how to construct *adaptively secure* streaming functional encryption (sFE). As we argue below, adaptive security is crucial to providing a meaningful theoretical foundation for almost all applications of sFE. We now elaborate.

A motivating medical research scenario. Imagine that a medical research institute wants to determine the appropriate vaccine dosage for patients with compromised health. To do this, the institute needs access to the records of patients held by a major health organization. However, these records contain highly sensitive private information. Naturally, the health organization would prefer to share only the necessary details for the study, withholding additional sensitive information.

To meet this objective, the health organization might turn to functional encryption (FE). Functional encryption [SW05, BSW11, O'N10] is an advanced form of encryption that departs from the traditional "all-or-nothing" encryption model. With FE, an authority can generate function-specific keys using a master secret key. Given a function key for f and an encrypted piece of data x, decryption should yield only $f(x)$ and nothing more. Hence, to accomplish this goal, the health organization can encrypt its medical records with FE and grant the research institute a function key for some function that is appropriate to their research. This key might allow, for example, the institute to extract only the results of certain statistical analyses on private patient data.

While FE may already sound too good to be true, after a long series of works [SW05, GGH$^+$13, SW14, GGHZ16, GKP$^+$13, BGG$^+$14, GVW15, ABSV15, AJ15, BV15, Lin16, Lin17, GPSZ17, GPS16, LV16, AS17, LT17, AJS18, AJL$^+$19, Agr19, JLMS19], the recent pivotal works of [JLS21, JLS22] successfully constructed FE schemes for general polynomial-sized circuits, using well-studied computational assumptions.

However, while FE offers a promising theoretical framework to enable the aforementioned medical study and other similar privacy-preserving computational challenges, it is not without its drawbacks. In terms of both functionality and privacy, FE presents certain limitations when applied to scenarios like the one described above. For example:

1. FE permits the medical research institute to access information only from the records available when the function key is initially provided. If new medical records emerge during the study, the institute would need the health organization to re-encrypt its entire database, inclusive of the new records. They would then apply the function key to this newly encrypted database. Thus, for assimilating newly obtained data, the study must essentially begin anew.

2. The institute cannot obtain interim results. They must await the function key's completion of the decryption process, a duration proportional to the database's size. If, due to power or connectivity issues, the server housing the encrypted records goes offline during decryption, the process has to start over from scratch.

From a privacy perspective, using FE in the aforementioned scenarios raises even more significant concerns. Given that the database of medical records is continuously evolving, if the health organization chooses to encrypt records in batches and then share those ciphertexts with the research institute, the institute could discern the output of the learning function for each batch individually. Consequently, the research institute might either gain excessive information, or insufficient insight if the health organization adds substantial noise to each output to preserve privacy. Therefore, the only viable strategy for the health organization involves periodically encrypting its ever-expanding database, which incurs asymptotically large computational overhead.

FE's primary challenges, especially in applications like medical studies which involve extensive and evolving data sets, stem from its inherent design. Specifically, FE requires the entire data set to be present at the time of encryption, and decryption can only be performed on the ciphertext encrypting the entire data set at once.

Streaming functional encryption. To address these issues, recently Guan, Korb, and Sahai [GKS23] introduced streaming FE (sFE). Essentially, sFE caters to situations where data is received in a streaming manner and is sequentially processed as it's received.

In simple terms, a streaming function is a stateful function that takes as input a state st_i and a value x_i and outputs the next state st_{i+1} and a value y_i. In an sFE scheme, the input $x = x_1 \dots x_n$ is encrypted piece by piece as it arrives in a streaming fashion and the streaming function of the encrypted input is derived by decrypting the piece-wise ciphertext of the input stream as it arrives. More precisely, in an sFE scheme, encryption requires the ability to individually generate ciphertexts ct_i for the i^{th} input x_i given only the master public key, x_i, the index i, and an encryption state (which is generated once for x using only the master public key). The decryption algorithm will itself be a streaming function that takes as input the i^{th} ciphertext ct_i, the index i, the function key sk_f, and the current decryption state $\mathsf{Dec.st}_i$ (which roughly speaking encrypts st_i), and outputs the next output value y_i where $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$ and the next decryption state $\mathsf{Dec.st}_{i+1}$. For non-triviality, it is required that an sFE scheme be streaming efficient, meaning that the runtime of the algorithms should not depend on the total length n of the data stream $x = x_1 \dots x_n$ that is encrypted. More formally, we require that the size and runtime of all algorithms of sFE with security parameter λ, function size $\ell_{\mathcal{F}}$, state size $\ell_{\mathcal{S}}$, input size $\ell_{\mathcal{X}}$, and output size $\ell_{\mathcal{Y}}$ are $\mathsf{poly}(\lambda, \ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}})$.

Given its design, sFE naturally addresses the issues raised in our medical research example, avoiding the associated operational/security constraints sketched above. sFE also has potential uses in other privacy-focused computation scenarios as highlighted in [GKS23]. Applications include executing privacy-preserving machine learning algorithms on voluminous, evolving private data, analyzing sensitive live-streamed video content, and outsourcing the assessment of confidential user data as it becomes available.

In their work, Guan, Korb, and Sahai [GKS23] constructed the first sFE system using standard FE. They demonstrated that assuming (1) a selectively secure, public-key FE scheme for $\mathsf{P/Poly}$, and (2) a strongly-compact selectively secure, secret-key FE scheme for $\mathsf{P/Poly}$, then a *semi-adaptive function-selective secure*, public-key sFE scheme for $\mathsf{P/Poly}$ exists. Here, semi-adaptive function-selective security for sFE requires that in the security game, adversaries must present all function key queries immediately after obtaining the master public key. Only after this step can they request ciphertexts for the challenge stream. Unfortunately, as we now argue, this model of security leaves a lot to be desired, and indeed, the work of [GKS23] explicitly stated that achieving full (adaptive) security for sFE is a major open problem.

The insufficiency of function-selective security. The function-selective security guarantee achieved by [GKS23] not only dampens the sFE's privacy robustness, rather it is so restrictive that it does not suffice for various natural applications of sFE. Take, for instance, the medical research scenario introduced earlier. Here, encrypted medical records of ill patients might have been accumulated long before the research institute decided to initiate the study. Using a semi-adaptive-function-selective sFE scheme would necessitate the health organization to re-encrypt the entire database after the function key is given to the research institute. Furthermore, if two distinct research institutes request computations on the same database at different times, the

health organization must ensure that the institute receiving a function key later cannot access the encrypted database created for the previous institute.

These limitations are so onerous as to render function-selective sFE unsatisfactory as a theoretical foundation for the security scenarios that we aim to address.

What we want: adaptive security. The security guarantee which addresses these issues in sFE applications is *adaptive security*. Specifically, adaptive security guarantees security against adversaries that demand ciphertexts for the challenge stream and function keys for multiple streaming functions in an arbitrarily intertwined manner. As we just pointed out, such intertwining of function key requests and the encryption of different parts of the data stream would naturally arise in almost any scenario where streaming functional encryption would be used.

Traditionally, it has been possible to upgrade from selective to adaptive security for standard FE via "complexity leveraging" [BB11,SW05] and its more advanced variant, namely "the piecewise guessing framework" [JW16, KW20, JKK+17]. These approaches involve predicting the challenge ciphertext in the adaptive security game and halting if the prediction is wrong. As such, these approaches incur a security loss "super" polynomial in the length of the challenge ciphertext. However, in streaming FE, the challenge stream's length is theoretically limitless, preventing the use of complexity leveraging or the piecewise guessing framework for attaining adaptive security for sFE. Indeed, this is why Guan, Korb, and Sahai identified achieving adaptive security for sFE as a particularly intriguing open problem [GKS23].

Our Results. In this paper, we resolve this problem by constructing the first adaptively secure sFE scheme for P/Poly. Our main result is summarized below.

Theorem 1.1 (Main Result, Informal)**.** *Assuming a secure indistinguishability obfuscator (iO) for* P/Poly *and injective pseudorandom generators (*PRG*s), there exists an adaptively secure* sFE *scheme for* P/Poly*.*

An obfuscator, as defined in [BGI+01], is a tool that converts a circuit into an equivalent one, *i.e.* preserving its input-output behavior, while concealing the original circuit's confidential data. An indistinguishability obfuscator iO is a specific type of obfuscator which ensures that any two equivalent circuits' obfuscations are indistinguishable. The utility of iO is extensive, enabling a broad range of applications in both cryptography and complexity theory [GGH+13, SW14, BFM14, GGG+14, HSW13, KLW15, BPR15, CHN+16, GPS16, HJK+16].

Given the extensive applications of iO in cryptography, it has been extensively researched [GGH12, GGH+13, BGK+14, BR14, PST14, AGIS14, BMSZ16, CLT13, CLT15, GGH15, CHL+15, BWZ14, CGH+15, HJ15, BGH+15, Hal15, CFL+16, MSZ16, DGG+18, Lin16, LV16, AS17, Lin17, LT17, GJ18, AJS18, Agr19, LM18, JLMS19, BIJ+20, AP20, BDGM20], culminating in recent advancements [BSW11, O'N10] constructing iO from the following well-established computational assumptions.

Theorem 1.2 ([JLS22], Informal)**.** *Assume sub-exponential security of the following assumptions:*

- *the Learning Parity with Noise* (LPN) *assumption over general prime fields* $\mathbb{F}_p$ *with polynomially many LPN samples and error rate* $1/k^\delta$*, where* k *is the dimension of the* LPN *secret, and* $\delta > 0$ *is any constant (Definition A.2);*
- *the existence of a Boolean Pseudo-Random Generator* (PRG) *in* NC^0 *with stretch* $n^{1+\tau}$*, where* n *is the length of the* PRG *seed, and* $\tau > 0$ *is any constant (Definition B.1);*

- *the Decision Linear* (DLIN) *assumption on symmetric bilinear groups of prime order (Definition A.4).*

Then, there exists (subexponentially secure) $i\mathcal{O}$ for P/Poly.

Theorems 1.1 and 1.2 together imply the following result.

Corollary 1.3 (Informal)**.** *Assume injective* PRG*s and the sub-exponential security of the following assumptions:*

- *the Learning Parity with Noise* (LPN) *assumption over general prime fields $\mathbb{F}_p$ with polynomially many LPN samples and error rate $1/k^\delta$, where k is the dimension of the* LPN *secret, and $\delta > 0$ is any constant (Definition A.2);*
- *the existence of a Boolean Pseudo-Random Generator* (PRG) *in* NC^0 *with stretch $n^{1+\tau}$, where n is the length of the* PRG *seed, and $\tau > 0$ is any constant (Definition B.1);*
- *the Decision Linear* (DLIN) *assumption on symmetric bilinear groups of prime order (Definition A.4).*

Then, there exists an adaptively secure sFE *scheme for* P/Poly.

In the next section, we elaborate extensively on our technical approach. Briefly, our adaptive sFE scheme is developed in two steps.

Step 1: Build Message-Selective sFE**.** We start by constructing a semi-adaptive *message-selective* sFE scheme essentially for P/Poly. More precisely, we prove the following theorem.[1]

Theorem 1.4. *(Message-Selective* sFE*) Assuming a secure $i\mathcal{O}$ for* P/Poly *and injective* PRG*s, there exists a semi-adaptive message-selective* sFE *scheme for the function class $\mathcal{F}_\perp = \{$*two-input *$f \in$* P/Poly *$: \forall s, f(\perp, s) = \perp\}$ where $\perp$ is a special symbol.*

Message-selective security is the dual notion to function-selective security. More precisely, in the message-selective security model, the adversary must output the entire challenge stream before querying any function key. Constructing a message-selective sFE scheme is highly non-trivial and we develop innovative technical ideas to tackle it.

Our approach modifies and adapts $i\mathcal{O}$-friendly "authentication[2]" techniques pioneered by Koppula et al. [KLW15] in the context of developing $i\mathcal{O}$ for Turing Machines. These techniques were originally devised for managing computations which take the entire input at once, and produce output only after the entire iterative computation concludes. In contrast, in the context of sFE, we encounter new inputs at each iterative step of our computation, and we must produce outputs visible to the adversary after each step of computation. At a high level, instead of authenticating a

[1] The message-selective sFE scheme we directly build in this paper has a weaker security guarantee than the scheme promised by Theorem 1.4 in that it is a secret-key scheme which is only secure against adversaries who are given just one function key and one encrypted challenge stream. As this weaker scheme is sufficient for building our final adaptive scheme, we do not need to enhance its security. Nevertheless, we can build the standalone message-selective scheme promised in Theorem 1.4 (which is a semi-adaptive, message-selective, public key sFE scheme) either (1) directly by bootstrapping our weaker message-selective scheme using the same bootstrapping technique we use for our adaptive scheme (see Section 7), or (2) as a byproduct of our final adaptive scheme which by definition, is also semi-adaptive, message-selective secure.

[2] Very roughly speaking, what we mean by an $i\mathcal{O}$-friendly authentication mechanism is one that *only* allows a special circuit to be evaluated on one particular input, and not on any others. We refer the reader to the technical overview for a much more accurate description.

single iterative computation path for a single fixed input, as was the case in prior work, we develop a more flexible authentication system that is able to authenticate a "sliding window" consisting of the inputs and intermediate states of the computation in two adjacent steps.

Step 2: Combine Message-Selective and Function-Selective Schemes to Achieve Full Adaptive Security. Having crafted schemes with adaptive security's two facets (the message-selective one above, and the function-selective one from [GKS23]), we develop a novel "gluing" technique to combine the two, resulting in full-fledged adaptive security. Our gluing mechanism is highly non-black-box and relies on specific properties of the two underlying schemes to merge the two "halves" of adaptive security.

We also remark that our technique for achieving adaptive security departs from the "dual system encryption" paradigm invented by Waters [Wat09,LW10,LOS+10] and could potentially pave a new pathway towards adaptive security for FE in scenarios similar to ours.

Related Work. Perhaps the two variants of FE that are closest to sFE are: FE for Turing machines [GKP+13, AS16] and multi-input FE [GGG+14, ACF+19, BKS16, GJO16]. However, crucial distinctions exist between sFE and these FE variants. While Turing machines inherently use iterative operations reminiscent of a streaming function, FE for Turing machines necessitates knowing the entire input at the time of (the first and only) encryption and doesn't yield any output until the Turing machine's computation concludes. On the other hand, though multi-input FE envisions segmenting a message into multiple parts and encrypting these segments incrementally, in order to successfully decrypt, the function key needs to be applied collectively to all these ciphertexts. Consequently, it lacks the capability to support incremental outputs like sFE.

2 Technical Overview

Our goal is to build an adaptively secure sFE scheme. Due to a bootstrapping theorem already present in [GKS23], it suffices to build a weaker primitive: an adaptively secure single-key, single-ciphertext secret-key sFE scheme, that is only required to be adaptively secure against adversaries who are given just one function key and one encrypted challenge stream. Thus, we prove our main theorem using the following two steps:

1. We construct a single-key, single-ciphertext, *adaptively* secure, secret-key sFE scheme One-sFE. We prove the following:

   Theorem 2.1. *Assuming $i\mathcal{O}$ for* P/Poly *and injective* PRG*s, there exists a single-key, single-ciphertext, adaptively secure, secret-key sFE scheme for* P/Poly*.*

2. Then, we use the following theorem, which is implied by the work of [GKS23], to bootstrap One-sFE into a public-key, sFE scheme.[3]

   Theorem 2.2. *Assuming (1) a selectively secure, public-key FE scheme for* P/Poly*, and (2) a single-key, single-ciphertext, adaptively secure, secret-key sFE scheme for* P/Poly*, there exists an adaptively secure, public-key sFE scheme for* P/Poly*.*

   For a technical overview of this bootstrapping scheme, we refer the reader to [GKS23].

Since we can build a selectively secure, public-key FE scheme from $i\mathcal{O}$ and OWFs [Wat15], and OWFs are implied by PRGs, then together, these two theorems imply our main theorem.

2.1 Towards Building Single-Key, Single-Ciphertext sFE

Recall that we need to build an adaptively secure secret-key sFE scheme One-sFE which is only required to be secure against adversaries who are given just one function key and one encrypted challenge stream. To construct this object, we first observe the challenges present when trying to apply prior work.

2.1.1 Prior Work

Problems with adapting the function-selective scheme from [GKS23]. The work of [GKS23] builds a *function-selective* variant of One-sFE, where security holds only if the function query is asked first. Let us call this scheme Post-One-sFE since the ciphertext queries must come *after* the functional key queries.

We first see whether we can modify Post-One-sFE to make it adaptively secure. Unfortunately, the proof of security for Post-One-sFE is heavily dependent on the adversary receiving the function query before the message queries. This is because the security proof of Post-One-sFE works by embedding each output value y_i into the corresponding i^{th} ciphertext. This allows the proof to later remove all information about the input stream from the ciphertexts. However, this only works since the challenger can compute y_i (which requires knowing the function f) before it needs to give out the i^{th} ciphertext. Thus, this programming cannot occur if the message queries are made before the function query.

[3] Technically, [GKS23] only proves this theorem for function-selective sFE schemes. However, as they remark in their paper, the same construction and essentially the same proof can be used to show that this bootstrapping step also applies for adaptively secure sFE schemes. For completeness, we provide a full proof of Theorem 2.2 in Section 7.

In fact, Post-One-sFE is simulation secure. However, selective simulation secure sFE, where the ciphertext queries are asked before the functional key queries, is impossible even in the single-key, single-ciphertext setting. This is because we must first simulate arbitrarily many ciphertexts (one for each element of the stream) without knowing any of the stream or output values, and then must simulate an a priori bounded-size function key that somehow provides the correct output values for all of the ciphertexts (c.f. [BSW11]). Thus, to build an adaptive scheme or even just a message-selective scheme, it seems likely we will need other techniques apart from the simulation techniques used in building Post-One-sFE.

A source of inspiration: FE for Turing machines from [AS16]. Our main source of inspiration is [AS16] which uses techniques from [KLW15] to build FE for Turing machines from $i\mathcal{O}$ and injective PRGs. Since Turing machines also involve iterative computation, it seems plausible that some of their techniques may also be applicable to the streaming setting.

Unfortunately, the construction from [AS16] seems incompatible with the streaming setting. In particular, their function keys depend on the size of the input to the Turing machine. In the streaming setting, this would mean that the size of our function keys would depend on the total length of our input stream, breaking our efficiency requirements. Unfortunately, this dependence seems to be an inherent property of the technique they use to achieve adaptive security, even in just the single-key, single-ciphertext setting.

Indeed, even if we only want selective security, there are still several problems with the construction. The main issues are

- The input to the Turing machine must be known all at once at encryption time. There is no way to adaptively add new input as in the streaming setting.
- There is no mechanism for outputting intermediate output values y_i. In particular, their security proof works by hardwiring in the final output value and erasing each of the steps of computation one by one. This works in the Turing machine setting since intermediate steps do not provide output. However, it is problematic in the streaming setting as we cannot hardwire in every output value, so we cannot erase all intermediate states without jeopardizing our ability to continue the computation.

These issues turn out to be very non-trivial to overcome. However, while we cannot use their construction or security proof, our final scheme owes a great intellectual debt to [AS16] and [KLW15] and adapts several of their techniques. We provide a more detailed explanation of the technical difficulties with adapting their work along with a comparison of our techniques in Section 2.2 of the Technical Overview.

2.1.2 Building a Selectively Secure One-sFE Scheme

To get started, we begin with a simpler goal: building a (semi-adaptive) *selectively* secure variant of One-sFE, where security holds only if the message queries are asked first. We call such a scheme Pre-One-sFE since the message queries must come *before* the function queries. Note that since any adaptively secure scheme must also be selectively secure, this is a natural starting point. Additionally, achieving even this weaker notion of security was left as an open problem in the prior work.

Constructing such a scheme turns out to be rather complex. Indeed one of the main technical contributions of our work is building Pre-One-sFE. We give an overview of the construction and security proof in Section 2.2 of the technical overview. But for the moment, we will continue this technical overview assuming that we succeed in building Pre-One-sFE.

2.1.3 Combining Pre-One-sFE and Post-One-sFE to Build One-sFE

We would now like to get adaptive security. Observe that we have two schemes, each with half of the security that we want:

- Pre-One-sFE: A (semi-adaptive) selectively secure scheme which we build in this paper.
- Post-One-sFE: A (semi-adaptive) function-selectively secure scheme from [GKS23].

We will attempt to combine these two schemes to somehow get adaptive security.

Composing the two schemes. For our approach, we will treat each scheme as an individual building block, and then try to combine these building blocks together to create an adaptive scheme. The main idea is to use the security of Pre-One-sFE to deal with all message queries given before the function query, and to use the security of Post-One-sFE to deal with all message queries given after the function query.

Combining the two schemes is highly non-trivial and is in fact one of the main technical contributions of this paper. In particular, there are two major difficulties: (1) determining how to syntactically compose the two schemes, and (2) bridging the security proof from Pre-One-sFE to Post-One-sFE after receiving the function query. We will focus on the first issue for now.

Note that we cannot simply give out ciphertexts and function keys for both schemes simultaneously. This is because each scheme is only secure for some of the messages (either the ones given before or after the function query), so using both schemes all the time compromises security.

Our key observation is that the decryption algorithm of an sFE scheme is itself a streaming function that acts on the stream of sFE ciphertexts. This means we can encrypt the output of an sFE scheme with another sFE scheme. Thus, in our construction, we will compose the two schemes by encrypting first with an inner Post-One-sFE scheme and then encrypting again with an outer Pre-One-sFE scheme.

In more detail, the inner scheme Post-One-sFE will encrypt the actual stream $x = x_1, x_2, x_3, \ldots$ and create a function key for the actual streaming function f. The outer scheme Pre-One-sFE will encrypt the stream of Post-One-sFE ciphertexts: $\mathsf{Post.ct}_1, \mathsf{Post.ct}_2, \mathsf{Post.ct}_3, \ldots$ and create a function key for Post-One-sFE's decryption algorithm. The ciphertexts and function keys for One-sFE are defined to be those output by the outer Pre-One-sFE scheme. To decrypt, we simply decrypt using Pre-One-sFE which gives us the output of running Post-One-sFE's decryption algorithm on Post-One-sFE's ciphertexts, which by the correctness of Post-One-sFE gives us the output of running f on the actual stream x. Thus, One-sFE outputs the correct values. The construction is depicted in Figure 1.

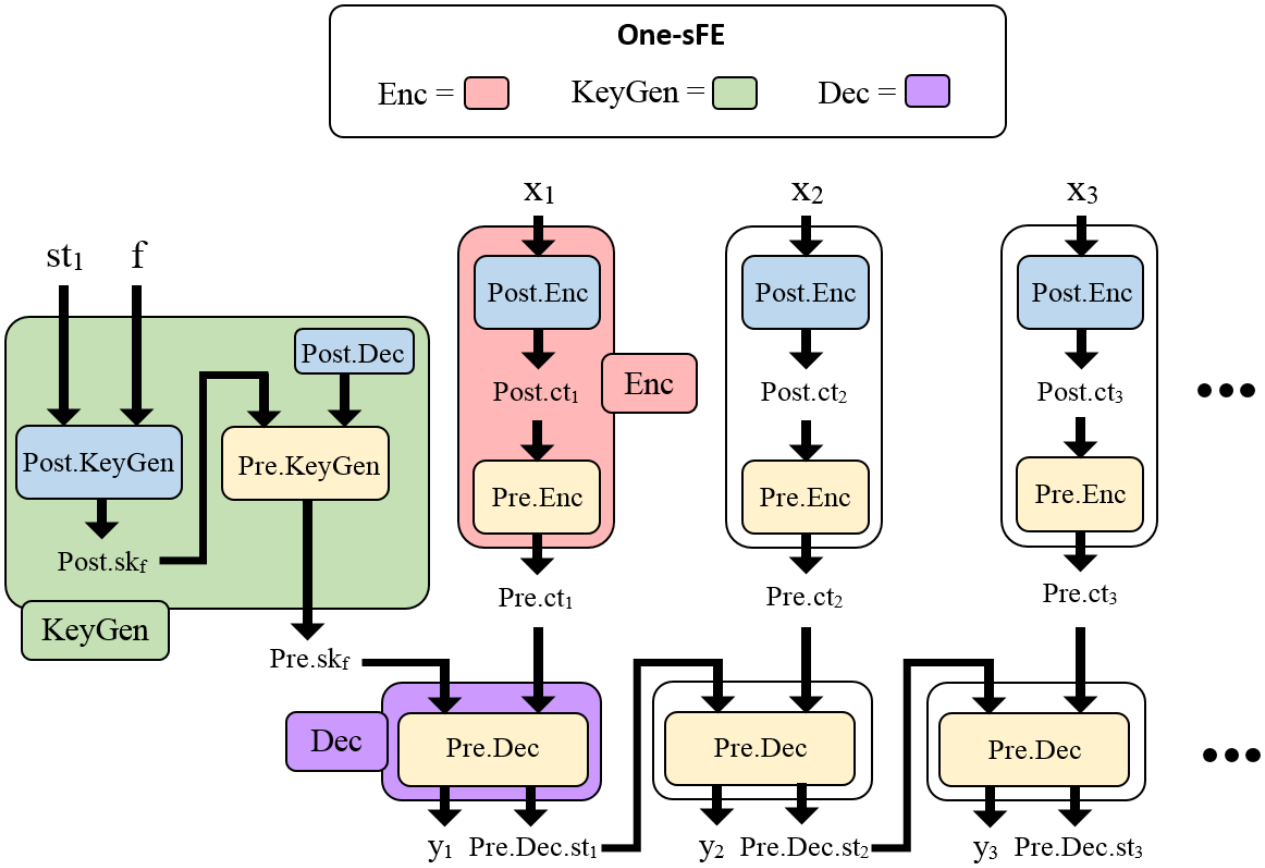


Figure 1: Construction of One-sFE.

For notational convenience, in Figure 1:

- We denote the algorithms of Pre-One-sFE and Post-One-sFE with the prefixes Pre and Post respectively.
- We have omitted EncSetup and Enc.st from all algorithms as they are not needed in the single-key, single-ciphertext sFE scheme.
- We have KeyGen additionally take the starting state as input.
- We have Dec takes in only two inputs: the current decryption state $\mathsf{Dec.st}_i$ and the current ciphertext ct_i. The function key sk_f is now defined to be $\mathsf{Dec.st}_1$. This change is easy to implement by Remark 3.36.

Bridging the security proofs. To prove security, we need to move from an encryption of stream $x^{(0)} = x_1^{(0)} x_2^{(0)} x_3^{(0)} \dots x_n^{(0)}$ to an encryption of stream $x^{(1)} = x_1^{(1)} x_2^{(1)} x_3^{(1)} \dots x_n^{(1)}$. Our high-level idea is the following:

1. First, use the security of Pre-One-sFE to prove indistinguishability for all stream queries given before the function query.

2. Then, use the security of Post-One-sFE to prove indistinguishability for all stream queries given after the function query.

The first step proceeds smoothly. Let t^* be the number of stream queries given before the function query, and let $\mathsf{Post.ct}_i^{(b)}$ be a Post-One-sFE encryption of $x_i^{(b)}$ for $b \in \{0,1\}$. The security of Pre-One-sFE lets us exchange an encryption of the stream $\mathsf{Post.ct}_1^{(0)}, \mathsf{Post.ct}_2^{(0)}, \mathsf{Post.ct}_3^{(0)}, \ldots$ with an encryption of the stream $\mathsf{Post.ct}_1^{(1)}, \mathsf{Post.ct}_2^{(1)}, \mathsf{Post.ct}_3^{(1)}, \ldots$ for all indices $i \leq t^*$. Since these Post-One-sFE ciphertext streams represent encryptions of $x^{(0)}$ and $x^{(1)}$ respectively, then we have successfully swapped out ciphertexts of $x^{(0)}$ under our scheme for ciphertexts of $x^{(1)}$ for all indices $i \leq t^*$.

The second step is much more difficult. The issue is that we want to start running the security proof of Post-One-sFE from midway through the stream. To deal with this issue, we will need to use specific properties of both schemes. This allows us to glue together the two security proofs with "very non-black-box glue".

We first observe that in the construction of Post-One-sFE from [GKS23], the security of each ciphertext only depends on a local portion of the master secret key. Thus, a ciphertext can remain secure even if we reveal the secret values used to encrypt another ciphertext as long as the two ciphertexts are encrypted using non-overlapping portions of the master secret key. Our goal is to use this property to decouple the security of the ciphertexts given after the function query from the ciphertexts given before the function query. We will also need to use an additional property of Post-One-sFE which allows us to directly generate ciphertexts for intermediate states using the same local portions of the master secret key. Together, these properties allow us to start the security proof of Post-One-sFE from midway through the computation, while ignoring any secret values contained in ciphertexts we have already given out.

In fact, we are able to show that even if we revealed every value used to encrypt ciphertexts $\{\mathsf{Post.ct}_i\}_{i<t^*}$, we can still prove security for ciphertexts $\{\mathsf{Post.ct}_i\}_{i>t^*}$ as if we had started midway through the computation at step t^*+1. However, this is not true if we reveal the secret values contained in ciphertext $\mathsf{Post.ct}_{t^*}$.

This is where our Pre-One-sFE scheme comes in. The security proof for Pre-One-sFE allows us to hardwire and erase a constant number of stream values. Thus, we will use Pre-One-sFE to erase the problematic Post-One-sFE ciphertext at the midway step t^*. This allows us to finish the security proof by using the security of Post-One-sFE to swap ciphertexts for stream $x^{(0)}$ with ciphertexts for $x^{(1)}$ for all indices $i \geq t^*+1$.

Note that because we needed to use specific properties of our ingredient schemes, although we are able to describe the construction of our final scheme modularly in terms of our ingredient schemes, we give the proof of security for our final scheme in a monolithic manner.

2.2 Building Pre-One-sFE

We now provide an overview of how we build our message-selective, secret-key sFE scheme Pre-One-sFE which is only required to be secure against adversaries who are given just one function key and one encrypted challenge stream. We prove the following:

Theorem 2.3. *Assuming $i\mathcal{O}$ for* P/Poly *and injective* PRG*s, there exists a single-key, single-ciphertext, selectively secure, secret-key* sFE *scheme for the function class* $\mathcal{F}_\perp = \{\text{two-input } f \in \mathsf{P/Poly} : \forall s, f(\perp, s) = \perp\}$.

Note that restricting the function class to $\mathcal{F}_\perp$ does not hinder the usability of our scheme since for every (two-input) streaming function $f \in \mathsf{P/Poly}$, we can construct a function $f' \in \mathcal{F}_\perp$ with

essentially the same functionality by defining $f'(z,s) = \begin{cases} f(x,s) & \text{if } z = 1||x \text{ for some } x \\ \perp & \text{else} \end{cases}$.

Additionally, our adaptive scheme One-sFE is not restricted to $\mathcal{F}_\perp$ and works for the function class P/Poly since the issue which led to this restriction is dealt with by Post-One-sFE.

2.2.1 On the Insufficiency of Directly Using [KLW15]

As mentioned earlier, in constructing our Pre-One-sFE scheme, we owe a great intellectual debt to techniques developed in [KLW15]. However there are many technical difficulties involved with trying to adapt their techniques for our uses. Here we elaborate on these difficulties and provide a comparison of our techniques. As this will require delving into the intricacies of the proof of [KLW15], we suggest that readers unfamiliar with this work skip this section upon initial reading.

The techniques of [KLW15] were developed to construct a machine-hiding encoding scheme for Turing machines which was then used to build $i\mathcal{O}$ for Turing machines. A machine-hiding encoding scheme can be interpreted (as in [AS16]) as a selectively-secure secret-key FE scheme for Turing machines which is only secure against adversaries who are given just one ciphertext and one function key. Additionally, the scheme is efficient with respect to the total runtime of the Turing machine in that the encryption and key generation algorithms do not depend on this runtime. Such a scheme is similar to what we desire for Pre-One-sFE in that it is a single-key, single-ciphertext, selectively-secure, secret key FE scheme whose encryption and key generation algorithms do not depend on the total number of computation steps (as defined by either the runtime of the Turing machine or the length of the stream). However, we want Pre-One-sFE to be an FE scheme for streaming functions, rather than for Turing machines.

Now, since both streaming functions and Turing machines involve iterative computation, it is tempting to think that one could use the techniques of [KLW15] to directly build Pre-One-sFE. However, there are some key differences between the two schemes which cause difficulties. In particular, an FE scheme for Turing machines takes in the entire input at encryption time and reveals only the final output value of the Turing machine at decryption. This means that the intermediate steps of computation should not be visible to the adversary during the security game. In contrast, in a streaming FE scheme, at each step of computation, we must take in a new stream value x_i and output a value y_i. Therefore, during the security game, each step of computation will convey information to the adversary. This is at the heart of the technical difficulties in using [KLW15] to build streaming FE. In particular, a crucial element of the security proof of [KLW15] is the fact that when using Turing machines, the intermediate computation is not revealed to the adversary.

For further insight, we now elaborate on how [KLW15] works. At a high level, the function key for a Turing machine M consists of the obfuscated "next step" function of M. To hide the input and the intermediate steps of the computation, the tape values are encrypted whenever they appear outside of the obfuscated program. This means that the obfuscated program must decrypt and re-encrypt the tape values before and after performing each computational step. Now, let T be the runtime of the Turing machine on the input. The general structure of their security proof is as follows:

1. First, they hardwire the output of the Turing machine into the final step T of the program. Since the Turing machine does not continue computation after halting, then there is no need to output any encrypted tape value at step T. This provides a useful endpoint for which to start their proof.

2. Next, they iteratively hardwire and erase each step of computation (by replacing the encrypted tape value with an encryption of $\perp$) starting from step $T-1$ and going backwards to step 1.

It is crucial in their proof that they erase the steps in the backwards direction. Otherwise, the encrypted tape value output at some step i may need to be correctly decrypted by the program at a later step. But this would prevent us from erasing the tape value at the current step since the keys used to encrypt that tape value would still be needed by the program at that later step. By erasing the computation from back to front, we remove the need to correctly decrypt future tape values, enabling the erasure of each tape value at the step it is first computed.

3. Once the entire computation is erased, there is now no dependence on the input apart from the final output of the Turing machine. Roughly speaking, this completes their security proof.

Now, several issues immediately arise when trying to use this security proof for streaming FE. First, and most importantly, in streaming FE, we cannot erase each step of computation during the security proof. This is because each iteration of the streaming function computation will produce an output y_i in the clear. If we erase the intermediate states of computation, then we can no longer correctly compute these output values from the input. Thus, we must either neglect to output these values or hardwire each of these values into the program. The former breaks the security proof while the latter breaks the efficiency requirements by requiring us to hardwire a number of steps equal to the stream length.

To solve this issue, we develop a new "sliding window" version of [KLW15]. As an initial change from [KLW15], we will switch to an indistinguishability-based notion of security rather than the simulation-based security considered (as an intermediate goal) by [KLW15]. This means that our goal is to move from an encryption of some stream $x^{(0)}$ to an encryption of a different stream $x^{(1)}$. This will be a crucial change. Unlike in [KLW15], our proof will proceed in the *forward* direction, starting from index 1 and continuing to the final step. At each step i, we will swap the i^{th} ciphertexts of stream $x^{(0)}$ with those of stream $x^{(1)}$. We can perform this swap by hardwiring a "window" consisting of Step $i-1$ (in which we encrypt the outgoing i^{th} state) and Step i (in which we decrypt the incoming i^{th} stream value and state). Now, suppose we want to move the window up one step so that we are instead hardwiring steps i and $i+1$. Hardwiring the new step $i+1$ is done by adapting tools from [KLW15], as we elaborate on below. However, we must also unhardwire step $i-1$. Since the program must continue to output the correct intermediate output y_i at step $i-1$, then we can only unhardwire this step if the program is able to correctly compute y_i from its inputs. Note that if we had erased the computation at each step as in [KLW15], then this would be impossible since we cannot compute an arbitrary value y_i from an erased input. However, because we are instead moving from the encryption of one stream to that of another, we are essentially able to restore the computation behind us by computing using the newly added $x^{(1)}$ stream. In particular, when we need to unhardwire step $i-1$, our input at this step will corresponds to stream $x^{(1)}$ which can be used to compute the correct output value y_i. This is possible since the security definition of sFE requires that computing the streaming function on the two challenge streams results in the same output y_i values. Therefore, we are able to move the sliding window up a step, which means we never need to hardwire more than a constant number of steps at a time! By sliding the window through the entire computation, we are able to entirely swap the encryption of stream $x^{(0)}$ for that of stream $x^{(1)}$, proving security.

Our main technical contributions here are our sliding window technique, the idea of restoring the computation behind us by switching to a different stream, and the methods of adapting mechanisms from [KLW15] to our new purposes. Thus, we use tools from [KLW15], especially splittable signatures and iterators, within our new proof structure. We also must make several other changes to the lower-level proof strategies employed by [KLW15] as we detail below.

In addition to this main issue, there are also several other technical difficulties. For one, our streaming FE scheme must allow for new stream values at each step of computation. To solve this, we deal with the stream and state values of the input separately. We handle the state values similarly to how tape values are handled in [KLW15], and we separately handle the input values by creating additional instances of some of the mechanisms from [KLW15]. In contrast, in [KLW15], both the input and intermediate states were encapsulated within the tape which could not be later added to by an outside entity, unlike our input stream values.

As a final issue, although we require our scheme to only be message-selectively *secure*, we still need it to be adaptively *correct* in that it should work no matter which order the stream and function queries arrive. In particular, even if the adversary only queries a stream of length n before the function query, it does not make syntactic sense to have the output of the obfuscated program at step n indicate that it is the final step and prevent future computation. This means that like all previous steps, step n should output an encrypted next state. In contrast, when using Turing machines, it does make sense to treat the final step of the Turing machine computation as an endpoint since a Turing machine does not continue computation after halting. This meant that [KLW15] could neglect to output an encrypted tape value at their final step which provided a useful endpoint to their chain of computation. Unfortunately, we are unable do the same. Instead, we must handle the additional encrypted state output at step n. If we are using Pre-One-sFE as a standalone scheme, we deal with this by adding a "dummy" stream value to the end of the stream which produces empty output states. If we are using Pre-One-sFE as a building block of our adaptive scheme, we deal with the additional encrypted state using our function-selective scheme Post-One-sFE, as was touched upon in a previous section.

We further remark that [KLW15] is a notoriously complex and intricate proof which makes modifying any aspect of it quite challenging. Indeed, to our knowledge, all prior works [AS16, AJS17a, AJS17b, BFK+19, CCC+16, DKW16, AJS15, CCHR16, CCC+16] that employed the techniques of [KLW15] essentially utilized it as a black box in that they made only minor modfications which did not alter the structure of the security proof. Our work is the first to really get inside the inner mechanisms of [KLW15] and modify it to work for other advanced functionalities.

Note that the complexity of [KLW15] and our scheme partly arises due to the difficulty of using $i\mathcal{O}$-based "punctured programming" techniques [SW14] with any iterative computation that creates a chain of dependencies between steps of computation. In particular, the punctured programming paradigm involves modifying the obfuscated program one input at a time by hardwiring the program at all places which depend upon that particular input. However, in a streaming function or Turing machine computation, each step depends on the next, which means that any particular input may be related to every step of computation. As we cannot hardwire all of these steps at once, we must use more complex methods to prove security, which involve "authenticating" one computation path in a way that deactivates other computation paths.

2.2.2 First Attempt: A Simple $i\mathcal{O}$-based Construction

Consider the following natural $i\mathcal{O}$-based candidate construction of Pre-One-sFE:

First Attempt at Building Pre-One-sFE

Let $i\mathcal{O}$ be an indistinguishability obfuscation scheme, SKE be a secret key encryption scheme, and PRF be a (puncturable) pseudorandom function.

- We use a PRF key K to generate SKE keys k_i for $i \in [2^\lambda]$.

- To encrypt stream values x_i, we output an SKE encryption $\mathsf{ct}_{\mathsf{inp},i}$ of x_i under key k_i.
- To create a function key for f, we first create an SKE encryption $\mathsf{ct}_{\mathsf{st},1}$ of the starting state st_1 under key k_1. We then output $\mathsf{ct}_{\mathsf{st},1}$ along with an obfuscation of the following program:

  $\mathsf{Program}_1(i, \mathsf{ct}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i})$
  Hardwired values: function f, PRF key K

  1. Compute SKE keys k_i, k_{i+1} from PRF key K.
  2. Decrypt $(\mathsf{ct}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i})$ with k_i to get (x_i, st_i).
  3. Compute $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$.
  4. Encrypt st_{i+1} with k_{i+1} to get $\mathsf{ct}_{\mathsf{st},i+1}$.
  5. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$.

- To decrypt, we iteratively run the obfuscated program on $(i, \mathsf{ct}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i})$ to get $(y_i, \mathsf{ct}_{\mathsf{st}_{i+1}})$. It is easy to see that the output values y_i will be correct.

Now, we want to prove that our scheme is selectively secure. Recall that to prove security, we will need to move[4] from an encryption of stream $x^{(0)} = x_1^{(0)} x_2^{(0)} x_3^{(0)} \dots x_n^{(0)}$ to an encryption of stream $x^{(1)} = x_1^{(1)} x_2^{(1)} x_3^{(1)} \dots x_n^{(1)}$.

Using $i\mathcal{O}$ to hide the SKE keys. Intuitively, we want to use the security of SKE to exchange ciphertexts of $x^{(0)}$ for ciphertexts of $x^{(1)}$. Unfortunately, we cannot do this since the PRF key K is embedded into the program, and thus the SKE keys are not hidden. However, since the program is obfuscated, the hope is that $i\mathcal{O}$ techniques can ensure that these keys are "hidden enough" for us to perform these swaps.

Punctured programming based $i\mathcal{O}$ techniques [SW14] work by hardwiring the input and output to the obfuscated program at every step which depends on the secret values we wish to hide. Then, since the program does not need to compute these steps by hand, it will not need to know whatever secret values were needed to compute them and can remove these values from its code.

Therefore, to exchange the i^{th} ciphertext, we need to hardwire the input and output of $\mathsf{Program}_1$ at all steps that depend on the SKE key k_i. In particular, this requires us to hardwire step $i-1$ (which uses k_i to encrypt the outgoing state) and step i (which uses k_i to decrypt its input). Note that we will not need to hardwire the other steps even though they depend on the PRF key K, since standard techniques allow us to use a *puncturable* PRF to remove the dependency between K (when evaluated at points $j \neq i$) and k_i.

Problem: Too many values to hardwire. Unfortunately, hardwiring any step i would require us to hardwire exponentially many values, since there are exponentially many possible inputs $(i, \mathsf{ct}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i})$ to the program at step i. This would break the efficiency requirements of our algorithms.

[4] In the paper, we define security as requiring computational indistinguishability between an encryption of $x^{(0)}$ and an encryption of $x^{(1)}$. However, in our security proof, we define security using an equivalent formulation where the adversary receives an encryption of $x^{(b)}$ for a random bit b and needs to guess bit b with probability greater than $\frac{1}{2} + \mathsf{negl}(\lambda)$. Thus, in our actual security proof, we will move from an encryption of $x^{(b)}$ for a random bit b to a (non-standard) encryption of $x^{(0)}$ which is independent of b. To do so, we use the same techniques described here.

To restrict the number of inputs we need to hardwire, we could try using a different SKE key for *every possible message*. Then, to exchange the i^{th} ciphertext for some state st_i (which must remain hidden), rather than needing to hardwire all of steps $i-1$ and i, we would only need to hardwire the specific inputs to the program that require us to encrypt or decrypt st_i. However, even with this change, there could still be exponentially many of these inputs. This is because there could be exponentially many values $(x_{i-1}, \mathsf{st}_{i-1})$ such that $f(x_{i-1}, \mathsf{st}_{i-1}) = (y_{i-1}, \mathsf{st}_i)$ for some y_{i-1}, and thus there could be exponentially many inputs $(i-1, \mathsf{ct}_{\mathsf{inp},i-1}, \mathsf{ct}_{\mathsf{st},i-1})$ to the program that would require us to encrypt st_i. Thus, this unfortunately does not work, so we will need to use other techniques.

Next Steps. Section 2.2.3 will be focused on this key challenge: Determining how to hardwire a step i without needing to hardwire more than a constant number of inputs and outputs at once. To prove this, we use techniques inspired by [KLW15]. Finally, Section 2.2.4 will show how to use this hardwiring technique to complete the proof of security.

2.2.3 Hardwiring a Step of the Program

Using signatures to enforce certain inputs. To hardwire step i in the previous construction, we needed to hardwire exponentially many values. Let us now look at the intuition for why this was needed. The main issue is that there is no restriction on which values the adversary can input to the obfuscated program. Indeed the adversary could run the program on any inputs it generates itself. And since we don't know what these inputs might be, the program needs to be prepared to handle any possible input. Thus, the program must hardwire all of the exponentially many possible inputs.

But what if we could somehow restrict the program so that it can only run on inputs which are either (1) directly generated by the challenger, or (2) previously output by the program? Then, since the challenger only ever gives out ciphertexts for one stream (since we're in the single-key, single-ciphertext setting), then the program should only be able to run on inputs which correspond to the execution of the queried function on the challenge stream. This means that there'd only be one valid input per step of computation, so we'd only need to hardwire one value at each step.

To implement this restriction, we introduce a signature scheme. The challenger will sign the stream ciphertexts and the starting state ciphertext during encryption and key generation respectively, and the program will sign its outgoing state ciphertexts. The program will reject (by immediately outputting $\perp$) any inputs that do not come with a valid signature. Intuitively, the unforgeability of the signature scheme should prevent the adversary from running the program on any inputs not generated by the challenger or the program itself. This gives us the following candidate construction:

Second Attempt at Building $\mathsf{Pre\text{-}One\text{-}sFE}$

Let $i\mathcal{O}$ be an indistinguishability obfuscation scheme, SKE be a secret key encryption scheme, PRF be a (puncturable) pseudorandom function, and Sig be a signature scheme. We will use different SKE and Sig keys for each step of computation.

- We use PRF keys $K, K_{\mathsf{inp}}, K_{\mathsf{st}}$ to generate SKE keys k_i, stream signature keys $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i})$, and state signature keys $(\mathsf{sgk}_{\mathsf{st},i}, \mathsf{vk}_{\mathsf{st},i})$ respectively for $i \in [2^\lambda]$.
- To encrypt stream value x_i, we encrypt x_i using key k_i and sign the resulting ciphertext

using $\mathsf{sgk}_{\mathsf{inp},i}$ to get $(\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$, which we output.

- To create a function key for f, we encrypt state st_1 using key k_1 and sign the resulting ciphertext using $\mathsf{sgk}_{\mathsf{st},1}$ to get $(\mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1})$. We then output these values along with an obfuscation of the following program:

  $\mathsf{Program}_2(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i})$
  Hardwired values: function f, PRF keys $K, K_{\mathsf{inp}}, K_{\mathsf{st}}$

  1. Compute Sig keys $\mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{st},i}, \mathsf{sgk}_{\mathsf{st},i+1}$ from PRF keys $K_{\mathsf{inp}}, K_{\mathsf{st}}$.
  2. If $\mathsf{Sig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$ or $\mathsf{Sig.Verify}(\mathsf{vk}_{\mathsf{st},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.
  3. Compute SKE keys k_i, k_{i+1} from PRF key K.
  4. Decrypt $(\mathsf{ct}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i})$ with k_i to get (x_i, st_i).
  5. Compute $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$.
  6. Encrypt st_{i+1} with k_{i+1} to get $\mathsf{ct}_{\mathsf{st},i+1}$.
  7. Sign $\mathsf{ct}_{\mathsf{st},i+1}$ with $\mathsf{sgk}_{\mathsf{st},i+1}$ to get $\sigma_{\mathsf{st},i+1}$.
  8. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1})$.

- To decrypt, we iteratively run the obfuscated program on $(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i})$ to get $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1})$. It is easy to see that the output values y_i will be correct.

Does this work? Now, suppose we want to hardwire some step i. Consider our intuition from before. We claimed that by the unforgeability of the signature scheme, the adversary should be unable to obtain any inputs that contain valid signatures apart from those corresponding to the execution of the challenge function on the challenge stream. We then implied that this means that to hardwire step i, we only need to hardwire the single input corresponding to the challenge execution path and can output $\perp$ on all other inputs (since the program outputs $\perp$ when given invalid signatures and the adversary is unable to construct valid signatures for other inputs). Thus, we can hardwire step i without breaking our efficiency requirements, which means the rest of the security proof will follow.

There are two main issues with this argument:

1. First, unforgeability only holds when the signing keys are hidden. However, the state signing keys $\mathsf{sgk}_{\mathsf{st},i}$ are embedded into the program, and thus are not hidden.[5]

2. Second, in order to hardwire values into the program, we implicitly relied on the security of $i\mathcal{O}$. However, when using the security properties of $i\mathcal{O}$, we can only argue indistinguishability of obfuscated programs that have identical input/output behavior. This means that we must maintain the same behavior on *all* inputs, even ones that the adversary cannot efficiently generate. Indeed, there are exponentially many possible inputs which *could* contain valid signatures and thus that could cause the program to output a non-$\perp$ value. Therefore, even

[5]Technically, the PRF keys K_{inp} and K_{st} are embedded into the program, not the signing keys. However, we can use a puncturable PRF to remove the dependency between K_{inp} and K_{st} (when evaluated at points $j \neq i$) and the i^{th} signing keys $\mathsf{sgk}_{\mathsf{inp},i}$ and $\mathsf{sgk}_{\mathsf{st},i}$. Then, the only parts of the program that will depend on the signing keys are the steps that actually make use of these keys to sign messages. Since the stream signing keys $\mathsf{sgk}_{\mathsf{inp},i}$ are never used, they can be removed entirely.

though the adversary may be unable to find these inputs and signatures, we cannot use $i\mathcal{O}$ to hardwire the program at step i so that it outputs $\perp$ on all but one input as this would change the behavior of the program.

Fortunately, we can solve the second issue, at least with regards to the stream inputs $\mathsf{ct}_{\mathsf{inp},i}$. (The first issue provides further complications when dealing with the state inputs $\mathsf{ct}_{\mathsf{st},i}$.) For this, we use a *splittable* signature scheme, introduced in [KLW15]. Splittable signatures have the property that if you only ever sign one message, then you can indistinguishably exchange the verification key for a one-message verification key that can only verify this one message.[6] Thus, for the stream of inputs, since the challenger only ever signs one stream ciphertext per step, if we want to hardwire step i, we can first exchange the stream verification key $\mathsf{vk}_{\mathsf{inp},i}$ embedded in the program with a restricted key $\mathsf{vk}^*_{\mathsf{inp},i}$ that can only verify this one value. This means that, as desired, our program will indeed output $\perp$ on any stream values that do not correspond to the challenge execution path. Thus, with respect to the stream values, we can use $i\mathcal{O}$ to hardwire the program as originally intended!

Using induction to deal with the state signatures. Unfortunately, due to the first issue, we cannot do the same for the state signatures. The problem is that the state signing keys $\mathsf{sgk}_{\mathsf{st},i}$ are embedded into the program and thus could be used to sign multiple messages. This means that at step i, we are unable to change the verification key to one that can only verify a single message. Thus, we cannot enforce that the program will only accept one state ciphertext at step i, and therefore cannot hardwire the program in the manner we desire.

Nevertheless, since this is a single-key, single-ciphertext scheme, then intuitively it should be true that the adversary will only ever be able to *obtain* one valid state signature per step. This intuition is motivated by the following inductive reasoning: Suppose that at some step i, the adversary has only one signed stream ciphertext and one signed state ciphertext. Then, by the unforgeability of the signature scheme and the hiding properties of $i\mathcal{O}$, the adversary should only be able to get the program to output a value other than $\perp$ on these specific inputs. Thus, the adversary should only be able to obtain one signed state ciphertext at step $i+1$, namely the one output by the program on the signed values of step i.

We begin by trying to directly implement our inductive reasoning with a splittable signature scheme SSig.

Notation: For simplicity, in the technical overview, we may refer to directly signing or verifying an input x_i or a state st_i. However, in our actual proof, we will instead sign and verify inputs containing the corresponding ciphertexts $\mathsf{ct}_{\mathsf{inp},i}$ and $\mathsf{ct}_{\mathsf{st},i}$ which encrypt these values.

1. At each step i, the program verifies incoming messages with verification keys $(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{st},i})$, and signs the outgoing state using signing key $\mathsf{sgk}_{\mathsf{st},i+1}$ as shown in the diagram below. We omit the signing keys $\mathsf{sgk}_{\mathsf{inp},i}$ for the inputs x_i from the diagram as these are handled outside the program.

	Step 1	Step 2	Step 3	Step 4
Verify With	$(\mathsf{vk}_{\mathsf{inp},1}, \mathsf{vk}_{\mathsf{st},1})$	$(\mathsf{vk}_{\mathsf{inp},2}, \mathsf{vk}_{\mathsf{st},2})$	$(\mathsf{vk}_{\mathsf{inp},3}, \mathsf{vk}_{\mathsf{st},3})$	$(\mathsf{vk}_{\mathsf{inp},4}, \mathsf{vk}_{\mathsf{st},4})$
Sign Using	$\mathsf{sgk}_{\mathsf{st},2}$	$\mathsf{sgk}_{\mathsf{st},3}$	$\mathsf{sgk}_{\mathsf{st},4}$	$\mathsf{sgk}_{\mathsf{st},5}$

[6] Splittable signatures also have additional properties which we will utilize later. Refer to the paragraph *"Splitting the signature scheme"* on page 23 for more details.

2. Since the program only signs outgoing steps, it will never need to sign st_1. Thus, we can change the program's signing step so that at step $i = 0$ (which is unused), if the outgoing state is equal to st_1, we will output a hardwired signature $\sigma_{\mathsf{st},1}$ for st_1 (and will output $\perp$ otherwise).

	Step 0	Step 1	Step 2	Step 3	Step 4
Verify With		$(\mathsf{vk}_{\mathsf{inp},1}, \mathsf{vk}_{\mathsf{st},1})$	$(\mathsf{vk}_{\mathsf{inp},2}, \mathsf{vk}_{\mathsf{st},2})$	$(\mathsf{vk}_{\mathsf{inp},3}, \mathsf{vk}_{\mathsf{st},3})$	$(\mathsf{vk}_{\mathsf{inp},4}, \mathsf{vk}_{\mathsf{st},4})$
Sign Using	$\sigma_{\mathsf{st},1}$ if st_1 $\perp$ else	$\mathsf{sgk}_{\mathsf{st},2}$	$\mathsf{sgk}_{\mathsf{st},3}$	$\mathsf{sgk}_{\mathsf{st},4}$	$\mathsf{sgk}_{\mathsf{st},5}$

3. Since we will now only output a single input and state signature for stream index 1, we can use the properties of SSig to swap the verification keys $(\mathsf{vk}_{\mathsf{inp},1}, \mathsf{vk}_{\mathsf{st},1})$[7] for step 1 with hardwired one-message verification keys $(\mathsf{vk}^*_{\mathsf{inp},1}, \mathsf{vk}^*_{\mathsf{st},1})$ that will only verify x_1 and st_1 respectively.[8]

	Step 0	Step 1	Step 2	Step 3	Step 4
Verify With		$(\mathsf{vk}^*_{\mathsf{inp},1}, \mathsf{vk}^*_{\mathsf{st},1})$	$(\mathsf{vk}_{\mathsf{inp},2}, \mathsf{vk}_{\mathsf{st},2})$	$(\mathsf{vk}_{\mathsf{inp},3}, \mathsf{vk}_{\mathsf{st},3})$	$(\mathsf{vk}_{\mathsf{inp},4}, \mathsf{vk}_{\mathsf{st},4})$
Sign Using	$\sigma_{\mathsf{st},1}$ if st_1 $\perp$ else	$\mathsf{sgk}_{\mathsf{st},2}$	$\mathsf{sgk}_{\mathsf{st},3}$	$\mathsf{sgk}_{\mathsf{st},4}$	$\mathsf{sgk}_{\mathsf{st},5}$

4. Since x_1 and st_1 are now the only inputs that can be verified at step 1, we know that if the obfuscated program does not output $\perp$ at step 1, then the output state must be st_2 where $(y_1, \mathsf{st}_2) = f(x_1, \mathsf{st}_1)$. Thus, we can change the program's signing step at step 1, so that if the outgoing state is equal to st_2, then we will output a hardwired signature $\sigma_{\mathsf{st},2}$ for st_2.

	Step 0	Step 1	Step 2	Step 3	Step 4
Verify With		$(\mathsf{vk}^*_{\mathsf{inp},1}, \mathsf{vk}^*_{\mathsf{st},1})$	$(\mathsf{vk}_{\mathsf{inp},2}, \mathsf{vk}_{\mathsf{st},2})$	$(\mathsf{vk}_{\mathsf{inp},3}, \mathsf{vk}_{\mathsf{st},3})$	$(\mathsf{vk}_{\mathsf{inp},4}, \mathsf{vk}_{\mathsf{st},4})$
Sign Using	$\sigma_{\mathsf{st},1}$ if st_1 $\perp$ else	$\sigma_{\mathsf{st},2}$ if st_2 $\perp$ else	$\mathsf{sgk}_{\mathsf{st},3}$	$\mathsf{sgk}_{\mathsf{st},4}$	$\mathsf{sgk}_{\mathsf{st},5}$

5. Since we will now only output a single input and state signature for stream index 2, we can use SSig again to swap the verification keys $(\mathsf{vk}_{\mathsf{inp},2}, \mathsf{vk}_{\mathsf{st},2})$ for step 2 with hardwired one-message verification keys $(\mathsf{vk}^*_{\mathsf{inp},2}, \mathsf{vk}^*_{\mathsf{st},2})$ that can only verify x_2 and st_2 respectively.

	Step 0	Step 1	Step 2	Step 3	Step 4
Verify With		$(\mathsf{vk}^*_{\mathsf{inp},1}, \mathsf{vk}^*_{\mathsf{st},1})$	$(\mathsf{vk}^*_{\mathsf{inp},2}, \mathsf{vk}^*_{\mathsf{st},2})$	$(\mathsf{vk}_{\mathsf{inp},3}, \mathsf{vk}_{\mathsf{st},3})$	$(\mathsf{vk}_{\mathsf{inp},4}, \mathsf{vk}_{\mathsf{st},4})$
Sign Using	$\sigma_{\mathsf{st},1}$ if st_1 $\perp$ else	$\sigma_{\mathsf{st},2}$ if st_2 $\perp$ else	$\mathsf{sgk}_{\mathsf{st},3}$	$\mathsf{sgk}_{\mathsf{st},4}$	$\mathsf{sgk}_{\mathsf{st},5}$

6. We can repeat steps 4 and 5 until we reach the step i we wish to hardwire.

	Step 0	Step 1	Step 2	Step 3	...	Step i
Verify With		$(\mathsf{vk}^*_{\mathsf{inp},1}, \mathsf{vk}^*_{\mathsf{st},1})$	$(\mathsf{vk}^*_{\mathsf{inp},2}, \mathsf{vk}^*_{\mathsf{st},2})$	$(\mathsf{vk}^*_{\mathsf{inp},3}, \mathsf{vk}^*_{\mathsf{st},3})$	...	$(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{vk}^*_{\mathsf{st},i})$
Sign Using	$\sigma_{\mathsf{st},1}$ if st_1 $\perp$ else	$\sigma_{\mathsf{st},2}$ if st_2 $\perp$ else	$\sigma_{\mathsf{st},3}$ if st_3 $\perp$ else	$\sigma_{\mathsf{st},4}$ if st_4 $\perp$ else	...	$\mathsf{sgk}_{\mathsf{st},i}$

[7] Technically, the verification keys of SSig are not directly hardwired into the programs, as these are computed based on puncturable PRFs. However, by appropriately puncturing the PRFs, we can "pretend" that the verification key of SSig is directly hardwired for the purpose of this technical overview.

[8] Observe that this is only possible because we are trying to build $\mathsf{Pre\text{-}One\text{-}sFE}$ – where the function key (with the obfuscated program inside) only gets queried *after* all the challenge ciphertexts have been issued. Thus, we will indeed already know the value of x_1 before needing to build our obfuscated program.

This process ensure that, as desired, there is only one set of possible inputs that can be verified at step i. Thus, to hardwire our final step i, we will only need to hardwire into the program the values corresponding to this single set of inputs. This fixes our original problem of needing to hardwire step i for each of the exponentially many inputs that used to be valid.

However, this process also requires hardwiring all of the state signatures and one-message verification keys up to the step we wish to hardwire. This means we need to hardwire a number of values that grows with the stream length, which is still too many.

Issues with restoring the computation. To reduce the number of values we need to simultaneously hardwire, we will try a sliding window approach in which we un-hardwire some values as we go. Let us consider how this might work with a first attempt (that will fail and need repair).

1. Suppose that we have both the signature schemes for i and $i+1$ hardwired and wish to remove the hardwiring of the i^{th} signature scheme.

	Step $i-2$	Step $i-1$	Step i	Step $i+1$
Verify With	$\ldots$	$(\mathsf{vk}_{\mathsf{inp},i-1}, \mathsf{vk}_{\mathsf{st},i-1})$	$(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{vk}^*_{\mathsf{st},i})$	$(\mathsf{vk}^*_{\mathsf{inp},i+1}, \mathsf{vk}^*_{\mathsf{st},i+1})$
Sign Using	$\mathsf{sgk}_{\mathsf{st},i-1}$	$\sigma_{\mathsf{st},i}$ if st_i $\perp$ else	$\sigma_{\mathsf{st},i+1}$ if st_{i+1} $\perp$ else	$\mathsf{sgk}_{\mathsf{st},i+2}$

2. We can use the properties of SSig to first change the one-message verification keys $(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{vk}^*_{\mathsf{st},i})$ at step i back to regular verification keys $(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{st},i})$.

	Step $i-2$	Step $i-1$	Step i	Step $i+1$
Verify With	$\ldots$	$(\mathsf{vk}_{\mathsf{inp},i-1}, \mathsf{vk}_{\mathsf{st},i-1})$	$(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{st},i})$	$(\mathsf{vk}^*_{\mathsf{inp},i+1}, \mathsf{vk}^*_{\mathsf{st},i+1})$
Sign Using	$\mathsf{sgk}_{\mathsf{st},i-1}$	$\sigma_{\mathsf{st},i}$ if st_i $\perp$ else	$\sigma_{\mathsf{st},i+1}$ if st_{i+1} $\perp$ else	$\mathsf{sgk}_{\mathsf{st},i+2}$

3. Now we wish to replace $\sigma_{\mathsf{st},i}$ with the regular signing key $\mathsf{sgk}_{\mathsf{st},i}$. Unfortunately, we are unable to do this using $i\mathcal{O}$ since this would change the behavior of our program. Observe that since we are using the regular verification keys $(\mathsf{vk}_{\mathsf{inp},i-1}, \mathsf{vk}_{\mathsf{st},i-1})$ at step $i-1$, then there is no guarantee that valid incoming messages (and thus valid outgoing states) need to correspond to those on our chosen computation path. Thus, switching to using $\mathsf{sgk}_{\mathsf{st},i}$ would change the behavior of the program since we would now be able to output valid signatures for multiple possible output states, rather than outputting $\perp$ on all but our chosen output state.

Perhaps if our sliding window started with a different configuration, that could solve the problem above?

1b. Suppose that in addition to having the signature schemes for i and $i+1$ hardwired, we also have the verification key for the $i-1^{th}$ scheme hardwired.

	Step $i-2$	Step $i-1$	Step i	Step $i+1$
Verify With	$\ldots$	$(\mathsf{vk}^*_{\mathsf{inp},i-1}, \mathsf{vk}^*_{\mathsf{st},i-1})$	$(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{vk}^*_{\mathsf{st},i})$	$(\mathsf{vk}^*_{\mathsf{inp},i+1}, \mathsf{vk}^*_{\mathsf{st},i+1})$
Sign Using	$\mathsf{sgk}_{\mathsf{st},i-1}$	$\sigma_{\mathsf{st},i}$ if st_i $\perp$ else	$\sigma_{\mathsf{st},i+1}$ if st_{i+1} $\perp$ else	$\mathsf{sgk}_{\mathsf{st},i+2}$

Unfortunately, in this configuration, we have no way of changing the one-message verification keys $(\mathsf{vk}^*_{\mathsf{inp},i-1}, \mathsf{vk}^*_{\mathsf{st},i-1})$ back to $(\mathsf{vk}_{\mathsf{inp},i-1}, \mathsf{vk}_{\mathsf{st},i-1})$ since we can only perform this change whenever we are *not* using $\mathsf{sgk}_{\mathsf{st},i-1}$.

1c. If in addition to the previous change, we supposed that rather than using $\mathsf{sgk}_{\mathsf{st},i-1}$ in step $i-2$, we still had $\sigma_{\mathsf{st},i-1}$ hardwired into the program as below,

	Step $i-2$	Step $i-1$	Step i	Step $i+1$
Verify With	$\ldots$	$(\mathsf{vk}^*_{\mathsf{inp},i-1}, \mathsf{vk}^*_{\mathsf{st},i-1})$	$(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{vk}^*_{\mathsf{st},i})$	$(\mathsf{vk}^*_{\mathsf{inp},i+1}, \mathsf{vk}^*_{\mathsf{st},i+1})$
Sign Using	$\sigma_{\mathsf{st},i-1}$ if $\mathsf{st}_i - 1$ $\perp$ else	$\sigma_{\mathsf{st},i}$ if st_i $\perp$ else	$\sigma_{\mathsf{st},i+1}$ if st_{i+1} $\perp$ else	$\mathsf{sgk}_{\mathsf{st},i+2}$

then in fact we have just changed the inductive statement so that we have the $i-1^{th}$, i^{th}, and $i+1^{th}$ signature schemes hardwired. Thus, we have not made any progress in removing the hardwiring.

Our main issue is that by moving from a program where we can sign only one outgoing message to a program where we can sign arbitrary outgoing messages, we change the behavior of the program and thus cannot use $i\mathcal{O}$.

Here, we see the heart of our quandary: In order to hardwire our final step, we need to ensure our hardwired mode can only sign and verify a single computational path. However, in order to move back to our regular mode using $i\mathcal{O}$, we need our hardwired mode to be able to sign arbitrary messages.

Splitting the signature scheme. To solve this issue, we will now use additional properties of splittable signature schemes. Splittable signature schemes allow us to isolate a particular message m by splitting the signing and verification keys into two parts: one part that deals with the chosen message m, and another that deals with all other messages. In particular splitting the signature scheme on a message m will

- split the signing key sgk into a signature σ_{one} for m and a signing key $\mathsf{sgk}_{\mathsf{abo}}$ that can only sign messages not equal to m,
- and split the verification key vk into a one-message verification key $\mathsf{vk}_{\mathsf{one}}$ that can only verify m and an all-but-one verification key $\mathsf{vk}_{\mathsf{abo}}$ that can only verify messages not equal to m.

The splittable signature scheme has several useful properties:

- If instead of using sgk, you only use σ_{one} (or respectively $\mathsf{sgk}_{\mathsf{abo}}$) then you can indistinguishably change vk to $\mathsf{vk}_{\mathsf{one}}$ (or respectively to $\mathsf{vk}_{\mathsf{abo}}$). This is the property we were previously relying on.
- If you sign no messages and do not use sgk at all, you can indistinguishably change vk to an always-reject verification key $\mathsf{vk}_{\mathsf{rej}}$ which rejects all signatures.
- You can merge $(\sigma_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{vk}_{\mathsf{abo}})$ back into the original signing and verification keys $(\mathsf{sgk}, \mathsf{vk})$. This works even if the $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}})$ parts come from one signature scheme and the $(\mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{abo}})$ parts come from a completely independent signature scheme!

We will use the merging property to enable the unhardwiring of each step. In particular, our original state signature scheme (which we will now call an A-type signature scheme) will be used as before. However, in the hardwiring mode, we will divert all other branches of computation to a new independent B-type signature scheme. The single computation path enforced by the A-type scheme will enable us to hardwire our step as we desire. However, the alternate B-type branches will allow us to transition back to the regular computational path by merging the A and B type schemes together into one scheme. We elaborate on this below.

Second attempt at our inductive argument. We will now reattempt our inductive argument using the additional properties of splittable signatures. Our second attempt will nearly work except for one issue we detail below. Recall that out overall goal is to hardwire some chosen step of the program.

1. We begin with our original scheme. At each step i, the program verifies incoming messages with verification keys $(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{A,i})$, and signs the outgoing state using signing key $\mathsf{sgk}_{A,i+1}$. Note that we have renamed the state signature keys from $(\mathsf{vk}_{\mathsf{st},i}, \mathsf{sgk}_{\mathsf{st},i})$ to $(\mathsf{vk}_{A,i}, \mathsf{sgk}_{A,i})$ as we will sometimes refer to them as A-type signature keys. We omit the signing keys $\mathsf{sgk}_{\mathsf{inp},i}$ for the inputs x_i from the diagram as these are handled outside the program.

	Step 1	Step 2	...	Step i
Verify With	$(\mathsf{vk}_{\mathsf{inp},1}, \mathsf{vk}_{A,1})$	$(\mathsf{vk}_{\mathsf{inp},2}, \mathsf{vk}_{A,2})$	...	$(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{A,i})$
Sign Using	$\mathsf{sgk}_{A,2}$	$\mathsf{sgk}_{A,3}$	...	$\mathsf{sgk}_{A,4}$

2. We will now introduce a new B type signature scheme up to and including step $i-1$. In particular, we will change our program so that it will first try verifying the state with the usual A-type verification key, but if that fails, it will then try to verify with a B-type key. If it verifies using the A-type key, it will sign the outgoing state with an A-type signature, and if it verifies with the B-type key, it will sign the outgoing state with a B-type signature.

	Step 1	Step 2	...
Verify With	$\mathsf{vk}_{\mathsf{inp},1}$ & $\begin{cases}\mathsf{vk}_{A,1} \text{ first}\\ \mathsf{vk}_{B,1} \text{ second}\end{cases}$	$\mathsf{vk}_{\mathsf{inp},2}$ & $\begin{cases}\mathsf{vk}_{A,2} \text{ first}\\ \mathsf{vk}_{B,2} \text{ second}\end{cases}$	...
Sign Using	$\mathsf{sgk}_{A,2}$ if verified w/ A-type $\mathsf{sgk}_{B,2}$ if verified w/ B-type	$\mathsf{sgk}_{A,3}$ if verified w/ A-type $\mathsf{sgk}_{B,3}$ if verified w/ B-type	...

We do this by first introducing the B-type branch at all relevant steps into just the signing phase of the program. We are able to make this initial change using the security of $i\mathcal{O}$ since we only sign using B-type keys if we verified with B-type keys, and we have not yet added in any B-type verification keys. We are then able to iteratively add in the B-type verification keys starting from index $i-1$ and traveling backwards to index 1. We do this by first introducing the B-type verification branch at the chosen index using an always-reject verification key $\mathsf{vk}_{B,\mathsf{rej}}$ (which cannot verify anything and thus will not change the program's behavior). We can then swap this with a real verification key since at that point in time, the corresponding signing key is never used. This is because the corresponding signing key would only be used if the previous step verified with B-type, but we are adding in the B-type verification keys from larger index to smaller index, so we would not yet have added in the B-type verification key to the previous step. We leave further details to the main proof.

3. Since the program only signs outgoing steps, it will never need to sign st_1. Thus, we can change the program's signing step so that at step $i = 0$ (which is unused), if the outgoing state is equal to st_1, we will output a hardwired A-type signature $\sigma_{A,1}$ for st_1. Otherwise, we will sign with a B-type all-but-one signing key $\mathsf{sgk}'_{B,1}$ that can sign all messages except st_1.

	Step 0	Step 1	Step 2
Verify With		$\mathsf{vk}_{\mathsf{inp},1}$ & $\begin{cases}\mathsf{vk}_{A,1} \text{ first}\\ \mathsf{vk}_{B,1} \text{ second}\end{cases}$	$\mathsf{vk}_{\mathsf{inp},2}$ & $\begin{cases}\mathsf{vk}_{A,2} \text{ first}\\ \mathsf{vk}_{B,2} \text{ second}\end{cases}$
Sign Using	$\sigma_{A,1}$ if st_1 $\mathsf{sgk}'_{B,1}$ else	$\mathsf{sgk}_{A,2}$ if verified w/ A-type $\mathsf{sgk}_{B,2}$ if verified w/ B-type	$\mathsf{sgk}_{A,3}$ if verified w/ A-type $\mathsf{sgk}_{B,3}$ if verified w/ B-type

4. Since we will now only only output one input signature and one A-type state signature for stream index 1, we can use the properties of SSig to swap $(\mathsf{vk}_{\mathsf{inp},1}, \mathsf{vk}_{A,1})$ with one-message verification keys $(\mathsf{vk}^*_{\mathsf{inp},1}, \mathsf{vk}^*_{A,1})$ that can only verify x_1 and st_1 respectively. Additionally, since we are using the all-but-one signing key $\mathsf{sgk}'_{B,1}$, we can also swap $\mathsf{vk}_{B,1}$ with an all-but-one verification key $\mathsf{vk}'_{B,1}$ which can only verify messages not equal to st_1.

	Step 0	Step 1	Step 2
Verify With		$\mathsf{vk}^*_{\mathsf{inp},1}$ & $\begin{cases}\mathsf{vk}^*_{A,1} \text{ first} \\ \mathsf{vk}'_{B,1} \text{ second}\end{cases}$	$\mathsf{vk}_{\mathsf{inp},2}$ & $\begin{cases}\mathsf{vk}_{A,2} \text{ first} \\ \mathsf{vk}_{B,2} \text{ second}\end{cases}$
Sign Using	$\sigma_{A,1}$ if st_1 $\mathsf{sgk}'_{B,1}$ else	$\mathsf{sgk}_{A,2}$ if verified w/ A-type $\mathsf{sgk}_{B,2}$ if verified w/ B-type	$\mathsf{sgk}_{A,3}$ if verified w/ A-type $\mathsf{sgk}_{B,3}$ if verified w/ B-type

5. Now, we will only sign using $\mathsf{sgk}_{A,2}$ if we verified using the A-type verification key. However, since $(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{vk}^*_{A,i})$ can only verify (x_1, st_1), then we will only need to use $\mathsf{sgk}_{A,2}$ if the outgoing state is st_2 where $(y_1, \mathsf{st}_2) = f(x_1, \mathsf{st}_1)$. Thus, we can replace $\mathsf{sgk}_{A,2}$ with a hardwired signature $\sigma_{A,2}$ for st_2.

	Step 0	Step 1	Step 2
Verify With		$\mathsf{vk}^*_{\mathsf{inp},1}$ & $\begin{cases}\mathsf{vk}^*_{A,1} \text{ if } \mathsf{st}_1 \\ \mathsf{vk}'_{B,1} \text{ else}\end{cases}$	$\mathsf{vk}_{\mathsf{inp},2}$ & $\begin{cases}\mathsf{vk}_{A,2} \text{ first} \\ \mathsf{vk}_{B,2} \text{ second}\end{cases}$
Sign Using	$\sigma_{A,1}$ if st_1 $\mathsf{sgk}'_{B,1}$ else	$\sigma_{A,2}$ if st_2 $\mathsf{sgk}_{B,2}$ if verified w/ B-type	$\mathsf{sgk}_{A,3}$ if verified w/ A-type $\mathsf{sgk}_{B,3}$ if verified w/ B-type

6. We would next like to use the properties of SSig to replace $\mathsf{sgk}_{B,2}$ with an all-but-one signing key $\mathsf{sgk}'_{B,2}$ that can sign every value except st_2.

	Step 0	Step 1	Step 2
Verify With		$\mathsf{vk}^*_{\mathsf{inp},1}$ & $\begin{cases}\mathsf{vk}^*_{A,1} \text{ if } \mathsf{st}_1 \\ \mathsf{vk}'_{B,1} \text{ else}\end{cases}$	$\mathsf{vk}_{\mathsf{inp},2}$ & $\begin{cases}\mathsf{vk}_{A,2} \text{ first} \\ \mathsf{vk}_{B,2} \text{ second}\end{cases}$
Sign Using	$\sigma_{A,1}$ if st_1 $\mathsf{sgk}'_{B,1}$ else	$\sigma_{A,2}$ if st_2 $\mathsf{sgk}'_{B,2}$ else	$\mathsf{sgk}_{A,3}$ if verified w/ A-type $\mathsf{sgk}_{B,3}$ if verified w/ B-type

Here lies the issue. In order to perform this swap, we need to ensure that $\mathsf{sgk}_{B,2}$ would never be used to sign st_2. Otherwise, the program's behavior will change. Now, we will only sign using $\mathsf{sgk}_{B,2}$ if we verified with the B-type verification key $\mathsf{vk}'_{B,1}$. Since $\mathsf{vk}'_{B,1}$ cannot verify st_1, we know that the incoming state cannot be st_1. Unfortunately, this is not enough to guarantee that the outgoing state cannot be st_2 (even if we are additionally ensured that the input value is x_1). This is because there could be another state $\mathsf{st}' \neq \mathsf{st}_1$ such that $f(x_1, \mathsf{st}_1) = f(x_1, \mathsf{st}') = (y_1, \mathsf{st}_2)$ for some y_1. Thus, we cannot perform the swap. While this an important issue we must resolve later, let us suppose for now that this was possible to see how the proof would progress.

7. We can now clean up the hardwiring of the first signature scheme by using the merging properties of SSig to merge $(\sigma_{A,1}, \mathsf{sgk}'_{B,1}, \mathsf{vk}^*_{A,1}, \mathsf{vk}'_{B,1})$ into $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1})$. We can additionally restore $\mathsf{vk}^*_{\mathsf{inp},1}$ to the regular $\mathsf{vk}_{\mathsf{inp},1}$.

	Step 0	Step 1	Step 2
Verify With		$(\mathsf{vk}_{\mathsf{inp},1}, \mathsf{vk}_{A,1})$	$\mathsf{vk}_{\mathsf{inp},2}$ & $\begin{cases}\mathsf{vk}_{A,2} \text{ first} \\ \mathsf{vk}_{B,2} \text{ second}\end{cases}$
Sign Using	$\mathsf{sgk}_{A,1}$	$\sigma_{A,2}$ if st_2 $\mathsf{sgk}'_{B,2}$ else	$\mathsf{sgk}_{A,3}$ if verified w/ A-type $\mathsf{sgk}_{B,3}$ if verified w/ B-type

We have now finished sliding our window up by one step from the signing step of step 0 to the signing step of step 1.

8. Continuing in this fashion by repeating steps 3-7 with appropriate indexing, we can reach the signing phase of step $i-1$. Note that since we originally only added in B type signatures up to and including step $i-1$, they are not present at step i and beyond.

	...	Step $i-1$	Step i	Step $i+1$	...
Verify With	...	$(\mathsf{vk}_{\mathsf{inp},i-1}, \mathsf{vk}_{A,i-1})$	$(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{A,i})$	$(\mathsf{vk}_{\mathsf{inp},i+1}, \mathsf{vk}_{A,i+1})$	...
Sign Using	...	$\sigma_{A,i}$ if st_i $\mathsf{sgk}'_{B,i}$ else	$\mathsf{sgk}_{A,i+1}$	$\mathsf{sgk}_{A,i+2}$	...

9. We can then use the properties of the splittable signature to swap $(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{A,i})$ with one-message verification keys $(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{vk}^*_{A,i})$ that only verify x_i and st_i respectively.

	...	Step $i-1$	Step i	Step $i+1$	...
Verify With	...	$(\mathsf{vk}_{\mathsf{inp},i-1}, \mathsf{vk}_{A,i-1})$	$(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{vk}^*_{A,i})$	$(\mathsf{vk}_{\mathsf{inp},i+1}, \mathsf{vk}_{A,i+1})$	...
Sign Using	...	$\sigma_{A,i}$ if st_i $\mathsf{sgk}'_{B,i}$ else	$\mathsf{sgk}_{A,i+1}$	$\mathsf{sgk}_{A,i+2}$	...

Now, we have reached the stage we desired in that we have enforced that the only valid inputs at step i are (x_i, st_i). Thus, we only need to hardwire one value at step i. Additionally, this process never required us to hardwire more than a constant number of values at a time.

After completing the hardwiring of step i, we are unfortunately left with a lingering B-type all-but-one signing key at step $i-1$ along with one-message verification keys $(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{vk}^*_{A,i})$. However, to rid ourselves of these extraneous elements, we can simply do all of the steps up to this point in reverse. This would then complete the hardwiring process if not for the issue present in step 6, which is the final issue we must resolve.

Using an iterator to enforce a step of computation. Let us consider the problem present in step 6. We are trying to replace the B-type signing key $\mathsf{sgk}_{B,2}$ with an all-but-one signing key $\mathsf{sgk}'_{B,2}$ which can sign every message except st_2.

	Step 0	Step 1	Step 2
Verify With		$\mathsf{vk}^*_{\mathsf{inp},1}$ & $\begin{cases}\mathsf{vk}^*_{A,1} \text{ if } \mathsf{st}_1 \\ \mathsf{vk}'_{B,1} \text{ else}\end{cases}$	$\mathsf{vk}_{\mathsf{inp},2}$ & $\begin{cases}\mathsf{vk}_{A,2} \text{ first} \\ \mathsf{vk}_{B,2} \text{ second}\end{cases}$
Sign Using	$\sigma_{A,1}$ if st_1 $\mathsf{sgk}'_{B,1}$ else	$\sigma_{A,2}$ if st_2 ~~$\mathsf{sgk}_{B,2}$ if verified w/ B-type~~ $\mathsf{sgk}'_{B,2}$ else	$\mathsf{sgk}_{A,3}$ if verified w/ A-type $\mathsf{sgk}_{B,3}$ if verified w/ B-type

Now, since we are using $\mathsf{vk}^*_{\mathsf{inp},1}$ and $\mathsf{vk}'_{B,1}$, we know that we will only verify with a B-type verification key at step 1 (and thus will only sign using a B-type signing key at step 1) if the input is x_1 and

the incoming state is *not* equal to st_1. Unfortunately, this does not tell us much about the outgoing state to be signed. In particular, since the streaming function need not be injective, there could be some state $\mathsf{st}' \neq \mathsf{st}_1$ such that $f(x_1, \mathsf{st}') = f(x_1, \mathsf{st}_1) = (y_1, \mathsf{st}_2)$ for some y_1. But this means that we could have a B-type branch where the outgoing state at step 1 could be equal to st_2! This is a problem since we are only able to swap to the all-but-one signing key $\mathsf{sgk}'_{B,2}$ if we are able to ensure that it will *not* be used to sign st_2. However, as we have just shown, this is not necessarily true.

What we would like to do is to enforce some form of injectivity between the incoming and outgoing messages. In particular, if we could have somehow ensured that the outgoing state is only equal to st_2 if the incoming message was (x_1, st_1), then our proof would have worked. Since this is not true in general, we need to add another method of enforcement.

To solve this issue, we will use a cryptographic iterator. An iterator consists of a small state that is updated in an iterative fashion as messages are received. The iterator has two important properties.

1. The iterator remains small regardless of how many messages have been iterated into it.
2. The normal public parameters are computationally indistinguishable from special enforcing parameters which ensure that one chosen iterator state can only be obtained as the outcome of an update to precisely one other state-message pair.

In particular, for each i, let itr_i be an iterator state which has states $\mathsf{st}_1, \ldots, \mathsf{st}_i$ iterated into it. Then, the second property means that we can indistinguishably change the iterator parameters to enforce that for some chosen i, we can only obtain itr_i as the output of the iterator update operation if it was performed on the inputs st_i and itr_{i-1}. In other words, we can enforce that a specified output itr_i can only be obtained as a result of a specific input $(\mathsf{st}_i, \mathsf{itr}_{i-1})$. This is precisely what we need!

To incorporate the iterator into our construction, we will modify our program so that at each step i, the program additionally takes in iterator state itr_{i-1}, incorporates state st_i into the iterator, and then outputs the new iterator state itr_i. In addition, we will have both our A and B type signature schemes sign and verify the iterator state in addition to the streaming state.

This gives us the following construction, which is in fact our final construction of Pre-One-sFE. Note that this construction is similar to our simple $i\mathcal{O}$-based construction, but has been augmented with splittable signatures and iterators in order to allow us to hardwire (and un-hardwire) steps of computation into the obfuscated program.

Final Construction of Pre-One-sFE

Let $i\mathcal{O}$ be an indistinguishability obfuscation scheme, SKE be a secret key encryption scheme, PRF be a (puncturable) pseudorandom function, SSig be a splittable signature scheme, and Itr be a cryptographic iterator.

- We use PRF keys $K, K_{\mathsf{inp}}, K_{\mathsf{st}}$ to generate SKE keys k_i, stream signature keys $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i})$, and state signature keys $(\mathsf{sgk}_{\mathsf{st},i}, \mathsf{vk}_{\mathsf{st},i})$ respectively for $i \in [2^\lambda]$.
- To encrypt stream value x_i, we encrypt x_i using key k_i and sign the resulting ciphertext using $\mathsf{sgk}_{\mathsf{inp},i}$ to get $(\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$, which we output.
- To create a function key for f, we encrypt state st_1 using key k_1 to get $\mathsf{ct}_{\mathsf{st},1}$. We run the iterator setup algorithm to get public parameters pp and initial iterator state $\mathsf{itr}_{\mathsf{st},0}$. We then sign $(1, \mathsf{ct}_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ with $\mathsf{sgk}_{\mathsf{st},1}$ to get $\sigma_{\mathsf{st},1}$. We output $(\mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ along

with an obfuscation of the following program:

$\mathsf{Program}_3(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

Hardwired values: function f, PRF keys $K, K_{\mathsf{inp}}, K_{\mathsf{st}}$, iterator parameters pp

1. Compute Sig keys $\mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{st},i}, \mathsf{sgk}_{\mathsf{st},i+1}$ from PRF keys $K_{\mathsf{inp}}, K_{\mathsf{st}}$.
2. If $\mathsf{Sig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$ or $\mathsf{Sig.Verify}(\mathsf{vk}_{\mathsf{st},i}, (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1}), \sigma_{\mathsf{st},i}) = 0$, output $\perp$.
3. Compute SKE keys k_i, k_{i+1} from PRF key K.
4. Decrypt $(\mathsf{ct}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i})$ with k_i to get (x_i, st_i).
5. Compute $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$.
6. Encrypt st_{i+1} with k_{i+1} to get $\mathsf{ct}_{\mathsf{st},i+1}$.
7. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
8. Sign $(i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_i)$ with $\mathsf{sgk}_{\mathsf{st},i+1}$ to get $\sigma_{\mathsf{st},i+1}$.
9. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

- To decrypt, we iteratively run the obfuscated program on $(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$ to get $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$. It is easy to see that the output values y_i will be correct.

Now, let us see how this construction enables us to fix step 6. Since the A and B type signature schemes additionally sign the iterator, then when we split these schemes, we will split them on a message containing both the iterator and the current state.

	Step 0	Step 1	Step 2
Verify With		$\mathsf{vk}^*_{\mathsf{inp},1}$ & $\begin{cases}\mathsf{vk}^*_{A,1} \text{ if } (\mathsf{st}_1, \mathsf{itr}_0)\\ \mathsf{vk}'_{B,1} \text{ else}\end{cases}$	$\mathsf{vk}_{\mathsf{inp},2}$ & $\begin{cases}\mathsf{vk}_{A,2} \text{ first}\\ \mathsf{vk}_{B,2} \text{ second}\end{cases}$
Sign Using	$\sigma_{A,1}$ if $(\mathsf{st}_1, \mathsf{itr}_0)$ $\mathsf{sgk}'_{B,1}$ else	$\sigma_{A,2}$ if $(\mathsf{st}_2, \mathsf{itr}_1)$ ~~$\mathsf{sgk}_{B,2}$ if verified w/ B-type~~ $\mathsf{sgk}'_{B,2}$ else	$\mathsf{sgk}_{A,3}$ if verified w/ A-type $\mathsf{sgk}_{B,3}$ if verified w/ B-type

Now, we wish to replace $\mathsf{sgk}_{B,2}$ with an all-but-one verification key $\mathsf{sgk}'_{B,2}$ that can sign all messages except $(\mathsf{st}_2, \mathsf{itr}_1)$. First, we can use the enforcing properties of the iterator to ensure that itr_1 will only appear in the output if the input contains $(\mathsf{st}_1, \mathsf{itr}_0)$. Then, since we are using an all-but-one verification key $\mathsf{vk}'_{B,1}$, we know that we will only verify with a B-type key at step 1 if the input does *not* contain $(\mathsf{st}_1, \mathsf{itr}_0)$. But due to the iterator enforcement, this means that the output of the B-type branch cannot contain itr_1! Therefore, $\mathsf{sgk}_{B,2}$ will never need to sign $(\mathsf{st}_2, \mathsf{itr}_1)$ which means we can perform the swap to $\mathsf{sgk}'_{B,2}$ as desired. This completes our fix of step 6, completing the proof.

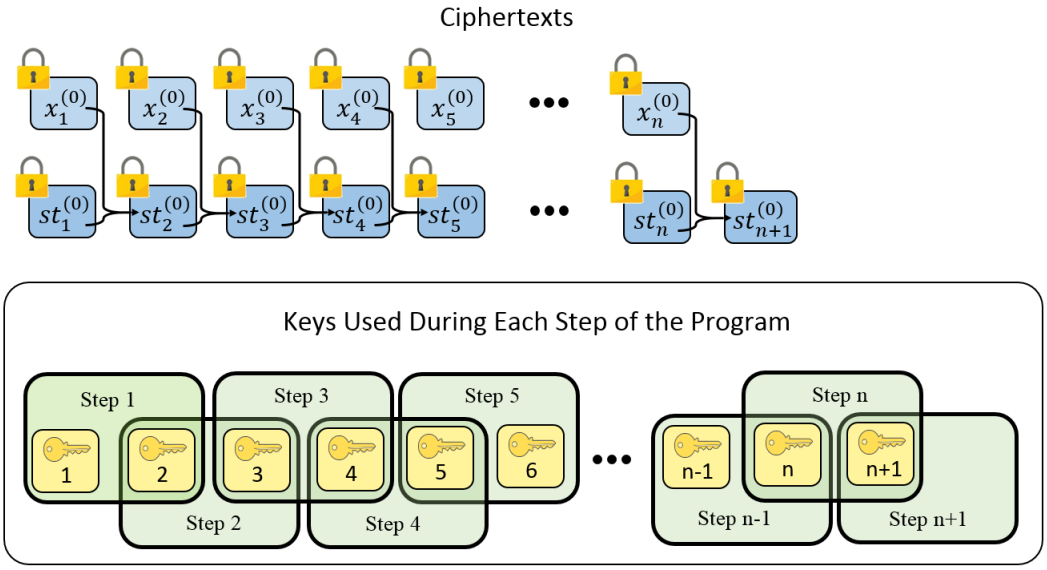
2.2.4 Proving Security

Now that we can hardwire steps into the program, we can finally prove security. At a high level, the benefits of hardwiring a step into the program is that it allows the program to compute the hardwired step without needing to know any SKE keys. Since the program only uses SKE key k_i in step $i-1$ (to encrypt the outgoing state) and step i (to decrypt the input values), then if we

hardwire both steps $i-1$ and i, we can employ punctured programming techniques[9] to remove key k_i from the program. This allows us to swap the i^{th} ciphertext from an encryption of stream $x^{(0)}$ to an encryption of stream $x^{(1)}$.

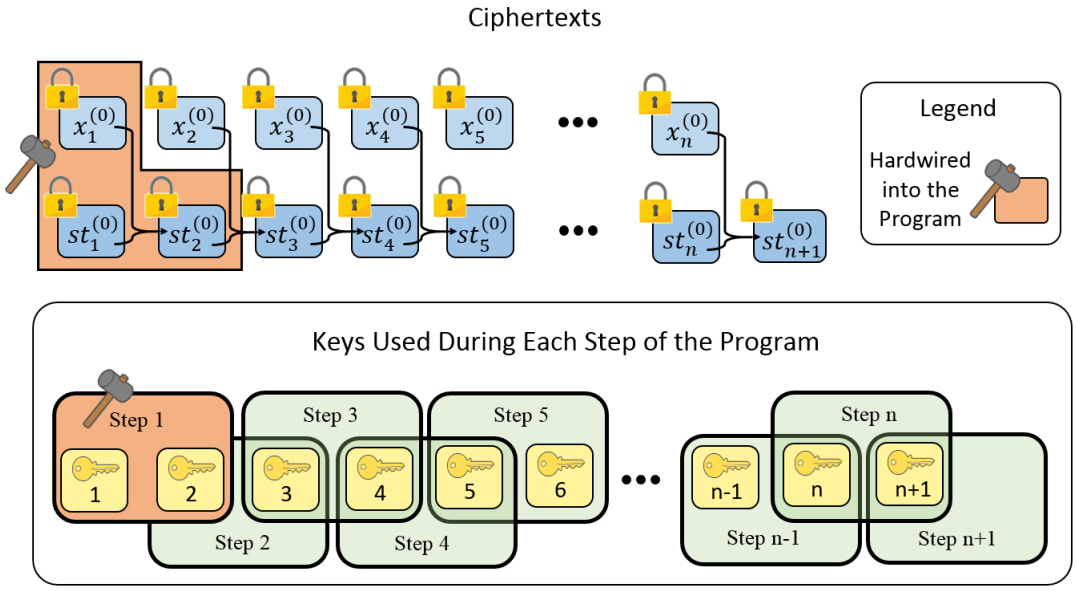
In more detail, we prove security using the following hybrid argument.

1. Our first hybrid is the security game for the adversary in which we encrypt stream $x^{(0)} = x_1^{(0)} x_2^{(0)} \dots x_n^{(0)}$. In the diagram below, we have depicted the SKE ciphertexts for stream $x^{(0)}$, which are produced during encryption. We have also depicted the SKE state ciphertexts which would be output by the obfuscated program during decryption. These correspond to the intermediate states produced by computing the streaming function f on $x^{(0)}$. Observe that for $i \in [n+1]$, the program only needs to use SKE key k_i at steps $i-1$ and i.

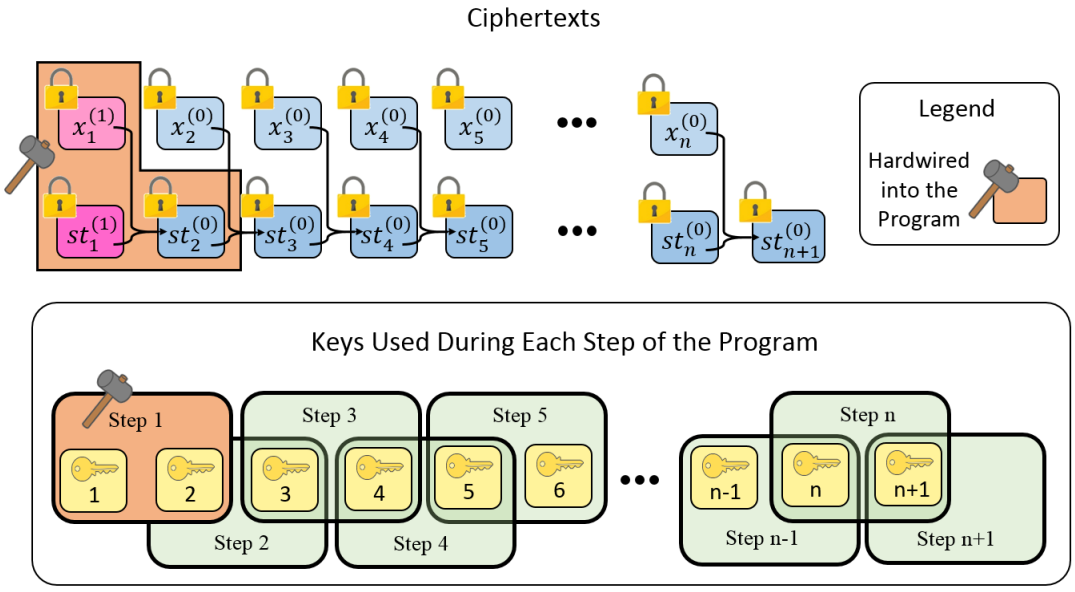


[9]In particular, we will need to first hardwire k_i into steps $i-1$ and i. We can then puncture the PRF key K at index i to remove the dependency between K (when evaluated at points $j \neq i$) and k_i. This ensures that the only steps of the program that depend on k_i are steps $i-1$ and i. Thus, once the inputs and outputs of these steps are hardwired, we can remove k_i.

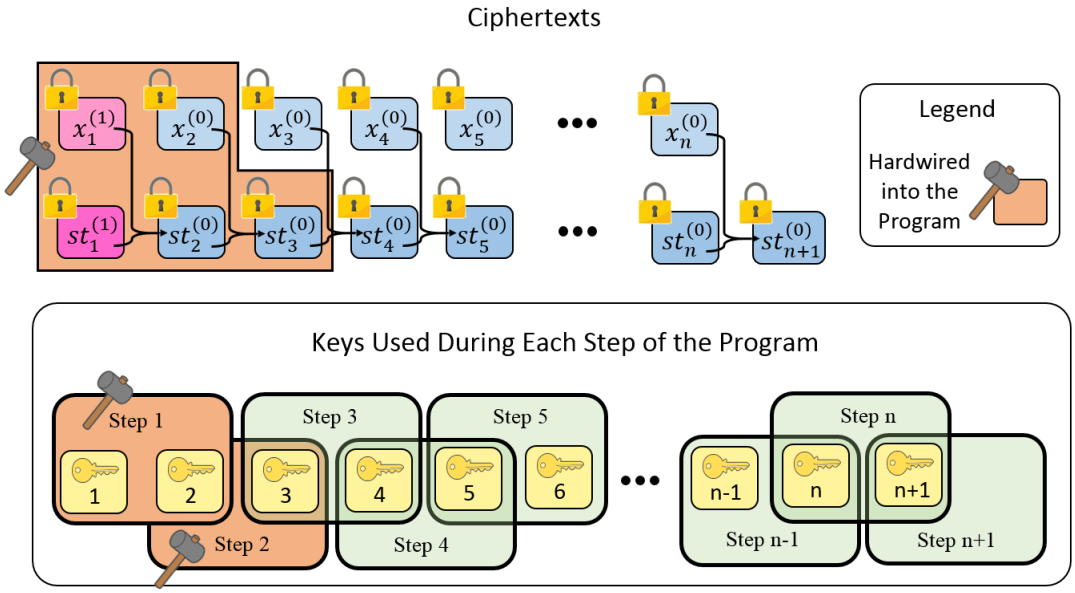
2. We now hardwire step 1 into the program using the techniques from the previous section. To hardwire step 1, we will need to hardwire both the input at step 1, which includes ciphertexts of $x_1^{(0)}$ and $\mathsf{st}_1^{(0)}$, and the output at step 1, which includes a ciphertext of $\mathsf{st}_2^{(0)}$.



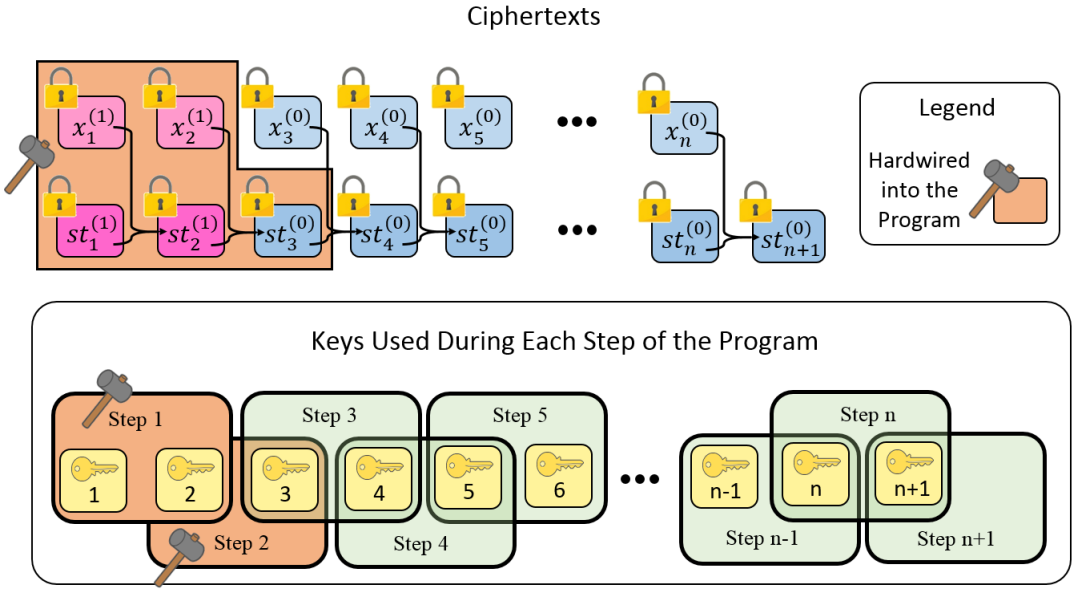
3. Since step 1 is hardwired, it no longer needs to know k_1. But since step 1 was the only step which used k_1, we can use standard techniques to remove k_1 from the obfuscated program. This allows us to swap the SKE ciphertexts of $x_1^{(0)}$ and $\mathsf{st}_1^{(0)}$ for ciphertexts of $x_1^{(1)}$ and $\mathsf{st}_1^{(1)}$ respectively.



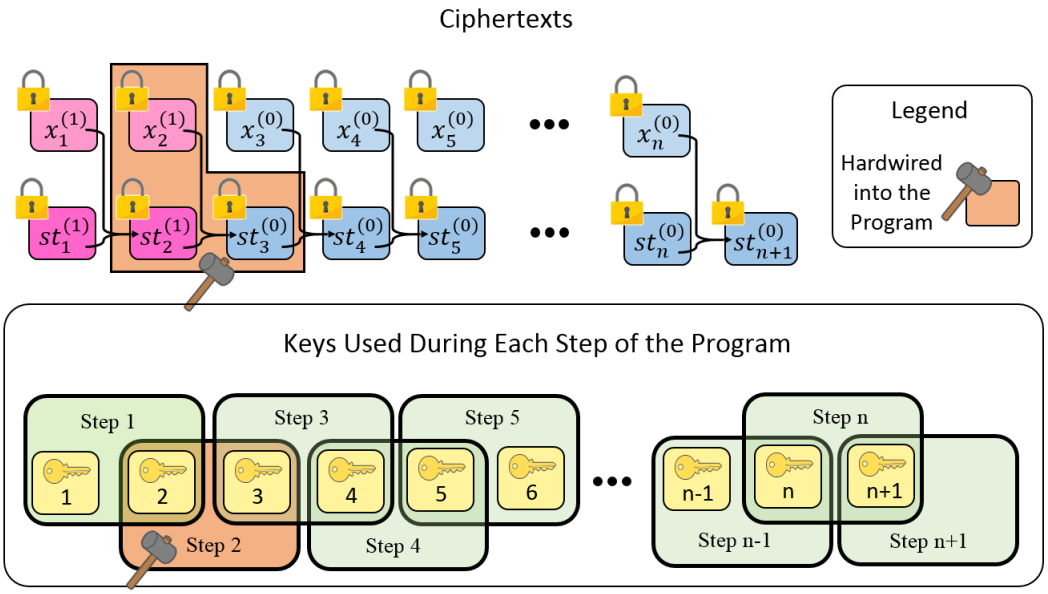
4. We now hardwire step 2 into the program. To hardwire step 2, we will need to hardwire both the input at step 2, which includes ciphertexts of $x_2^{(0)}$ and $\mathsf{st}_2^{(0)}$, and the output at step 2, which includes a ciphertext of $\mathsf{st}_3^{(0)}$.



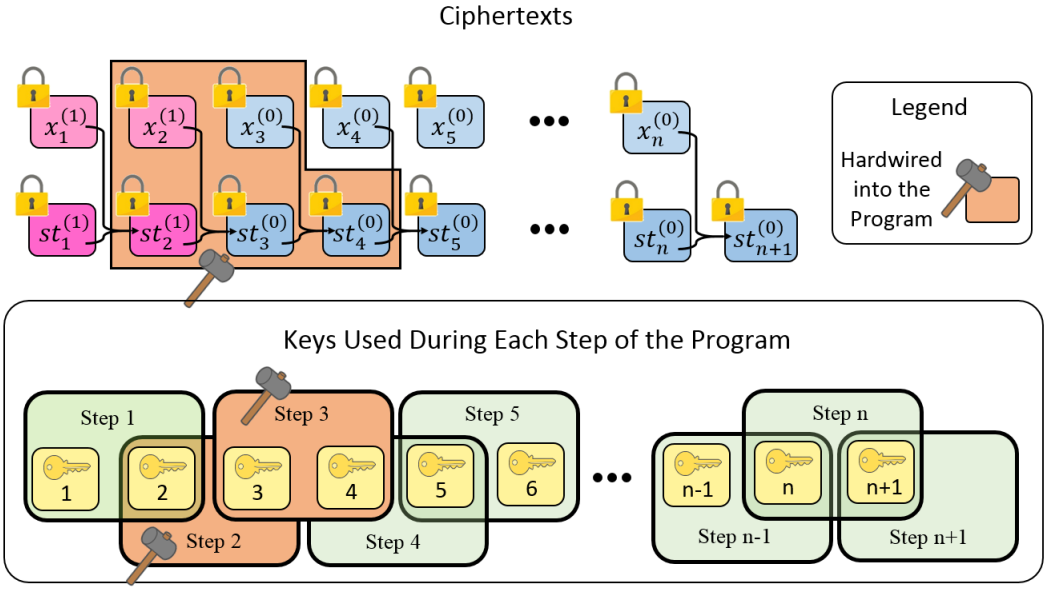
5. Since steps 1 and 2 are hardwired, these steps no longer need to know k_2. But since these steps were the only steps which used k_2, we can use standard techniques to remove k_2 from the obfuscated program. This allows us to swap the SKE ciphertexts of $x_2^{(0)}$ and $\mathsf{st}_2^{(0)}$ for ciphertexts of $x_2^{(1)}$ and $\mathsf{st}_2^{(1)}$ respectively.



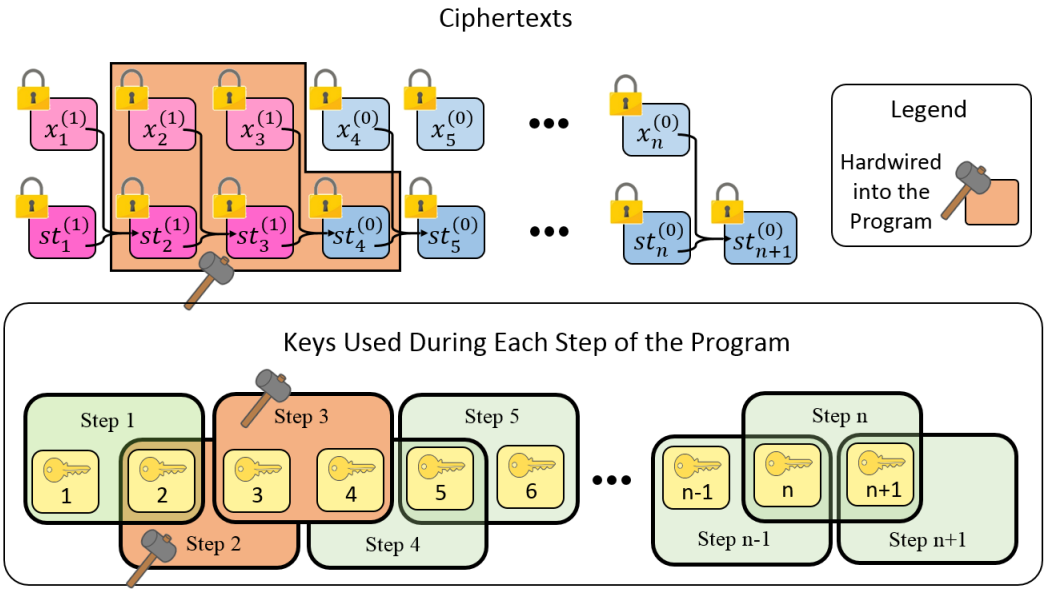
6. We now un-hardwire step 1 using the techniques from the previous section. This works since the hardwired input at step 1 corresponds to the hardwired output at step 1.



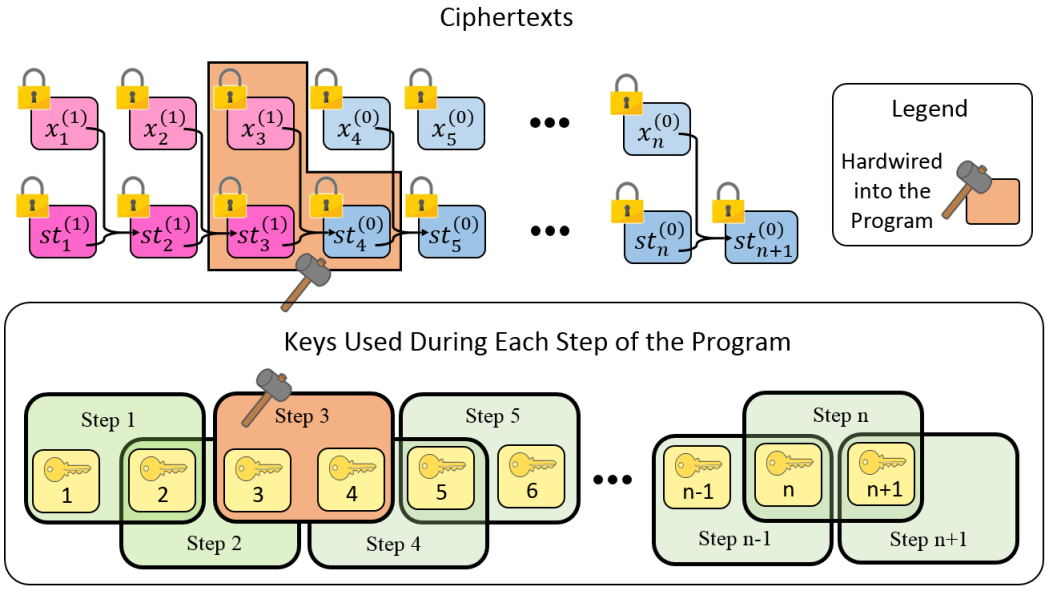
7. Next, we hardwire step 3 into the program. To hardwire step 3, we will need to hardwire both the input at step 3, which includes ciphertexts of $x_3^{(0)}$ and $\mathsf{st}_3^{(0)}$, and the output at step 3, which includes a ciphertext of $\mathsf{st}_4^{(0)}$.



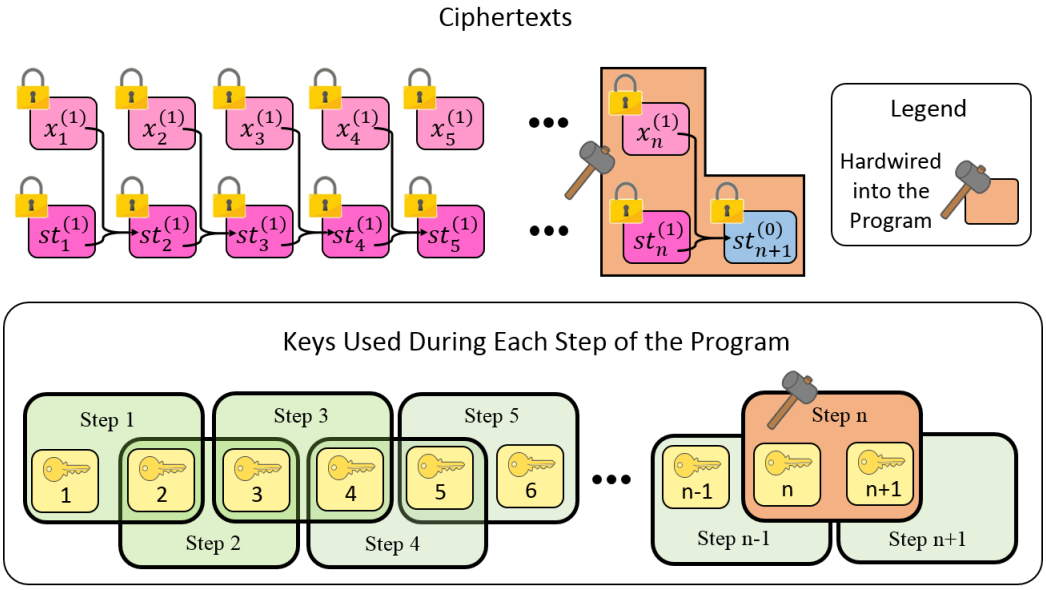
8. Since steps 2 and 3 are hardwired, these steps no longer need to know k_3. But since these steps were the only steps which used k_3, we can use standard techniques to remove k_3 from the obfuscated program. This allows us to swap the SKE ciphertexts of $x_3^{(0)}$ and $\mathsf{st}_3^{(0)}$ for ciphertexts of $x_3^{(1)}$ and $\mathsf{st}_3^{(1)}$ respectively.



9. We now un-hardwire step 2. This works since the hardwired input at step 2 corresponds to the hardwired output at step 2.



10. We continue in a similar manner for the remaining steps $i \in [n]$. On each iteration i, we hardwire step i into the program, exchange the i^{th} ciphertexts, and then un-hardwire step $i-1$ from the program. After iteration $i = n$, we have swapped every ciphertext of $x^{(0)}$ to a ciphertext of $x^{(1)}$. We are now in a hybrid where we encrypt stream $x^{(1)}$ for $i \leq n$ and have step n hardwired into the program with input corresponding to $x^{(1)}$ and output state corresponding to $x^{(0)}$.



All that remains is to un-hardwire step n. Unfortunately, we are unable to do this as we can only un-hardwire steps where the inputs and outputs match. However, the input at step n corresponds to stream $x^{(1)}$ and the output at step n corresponds to the final output state $\mathsf{st}_{n+1}^{(0)}$ of the original stream $x^{(0)}$. Furthermore, we cannot swap the state ciphertext for step $n+1$ since this would require hardwiring step $n+1$ (which depends on key k_{n+1}), which we cannot do since we only have stream inputs up to step n.

We deal with this problem in two separate ways depending on whether we are using Pre-One-sFE as a standalone construction or as a component in the combined construction of One-sFE.

- **Using Pre-One-sFE as a standalone scheme.**
  When using Pre-One-sFE as a standalone scheme, we are only able to prove security for function class $\mathcal{F}_\perp$. If $f \in \mathcal{F}_\perp$, we can add an additional dummy value of $\perp$ to the end of both challenge streams which ensures that the final output state of both streams will be $\perp$. Then, we can un-hardwire the last step since the inputs and outputs will match. We will then be left with the original security game, but where we now encrypt stream $x^{(1)}$.
- **Using Pre-One-sFE as a component of One-sFE.**
  If t^* is the length of the challenge stream before the first function query, then at index t^*, rather than swapping the stream values to those of the other stream, we instead replace them with $(\perp, \perp)$. This allows us to use Post-One-sFE to deal with the problematic final state at index $t^* + 1$ since setting the midway values to $(\perp, \perp)$ breaks the chain of dependencies between ciphertexts given before the function query (those at indices $i \leq t^*$) and ciphertexts given after the function query (those at indices $i \geq t^* + 1$).

However, we can no longer remove the hardwiring at step t^* since computing on $(\bot, \bot)$ would result in an incorrect computation. Therefore, in our actual security proof, rather than moving from an encryption of stream $x^{(0)}$ to an encryption of stream $x^{(1)}$, we instead move from an encryption of stream $x^{(b)}$ for a random bit b to a hybrid which is independent of b. For indices $i \leq t^*$, this hybrid corresponds to a non-standard encryption of stream $x^{(0)}$ which maintains the hardwiring of step t^* and uses $(\bot, \bot)$ for its stream values at index t^*.

3 Preliminaries

Throughout, we will use λ to denote the security parameter.

Notation.

- We say that a function $f(\lambda)$ is negligible in λ if $f(\lambda) = \lambda^{-\omega(1)}$, and we denote it by $f(\lambda) = \mathsf{negl}(\lambda)$.
- We say that a function $g(\lambda)$ is polynomial in λ if $g(\lambda) = p(\lambda)$ for some fixed polynomial p, and we denote it by $g(\lambda) = \mathsf{poly}(\lambda)$.
- For $n \in \mathbb{N}$, we use $[n]$ to denote $\{1, \ldots, n\}$.
- If R is a random variable, then $r \leftarrow R$ denotes sampling r from R. If T is a set, then $i \leftarrow T$ denotes sampling i uniformly at random from T.

Definition 3.1 (Statistical Distance)**.** *Let D_1 and D_2 be two distributions with support in X. The statistical distance between D_1 and D_2 is*

$$\Delta(D_1, D_2) = \frac{1}{2} \sum_{x \in X} \Big| \Pr[D_1 = x] - \Pr[D_2 = x] \Big|$$

Notation. Let A and B be two random variables with support in X. We use $\Delta(A, B)$ to denote the statistical distance $\Delta(P_A, P_B)$ between the underlying distributions of the random variables.

We use the standard definitions of PRGs, PRFs, and symmetric key encryption (SKE) with pseudorandom ciphertexts. We formally define these notions in Appendix B.1.

3.1 Indistinguishability Obfuscation

The recent work of [JLS22] shows how to construct $i\mathcal{O}$ for $\mathsf{P/Poly}$ from well-established computational assumptions (see Theorem 1.2).

Definition 3.2 (Indistinguishability Obfuscation ($i\mathcal{O}$) for Circuits [JLS21])**.** *A uniform PPT algorithm $i\mathcal{O}$ is an indistinguishability obfuscator for polynomial-sized circuits if the following holds:*

- ***Completeness**: For every $\lambda \in \mathbb{N}$, every circuit C with input length n, and every input $x \in \{0,1\}^n$,*
$$\Pr[C'(x) = C(x) : C' \leftarrow i\mathcal{O}(1^\lambda, C)] = 1$$
- ***Indistinguishability**: For every two ensembles $\{C_{0,\lambda}\}, \{C_{1,\lambda}\}$ of polynomial-sized circuits that have the same size, input length, and output length, and are functionally equivalent, that is, $\forall \lambda \in \mathbb{N}$, $C_{0,\lambda}(x) = C_{1,\lambda}(x)$ for every input x, then for all polynomial-time, non-uniform adversaries $\mathcal{A}$, there exists a negligible function μ, such that for all λ,*
$$\Big| \Pr[\mathcal{A}(1^\lambda, i\mathcal{O}(1^\lambda, C_{0,\lambda}))] = 1 - \Pr[\mathcal{A}(1^\lambda, i\mathcal{O}(1^\lambda, C_{1,\lambda}))] = 1 \Big| \leq \mu(\lambda)$$

3.2 Puncturable Pseudorandom Function

A puncturable pseudorandom function (PPRF), first termed by Sahai and Waters [SW14], is a PRF augmented with additional algorithms that allow the user to puncture the key at a point of their choice. The punctured key can be used to correctly evaluate the PRF on all points not equal to the punctured point. Additionally, pseudorandomness holds at the punctured point even if the punctured key is given out.

As shown in [SW14, BW13, BGI14, KPTZ13], the GGM tree-based construction of PRFs from OWFs [GGM86] can be readily modified to build a PPRF. Thus, we can build PPRFs from any OWF.

Definition 3.3 (Puncturable Pseudorandom Function (PPRF))**.** *A puncturable pseudorandom function family with key space $\mathcal{K} = \{\mathcal{K}_{\lambda,n,m}\}_{\lambda,n,m\in\mathbb{N}}$ is a tuple of PPT algorithms* $\mathsf{PPRF} = (\mathsf{PPRF.Setup}, \mathsf{PPRF.Eval}, \mathsf{PPRF.Punc}, \mathsf{PPRF.EvalPunc})$ *where*

- $\mathsf{PPRF.Setup}(1^\lambda, 1^n, 1^m)$ *is a randomized algorithm that takes as input the security parameter λ, an input length n, and an output length m, and outputs a key $K \in \mathcal{K}_{\lambda,n,m}$.*
- $\mathsf{PPRF.Eval}(K, x)$ *is a deterministic algorithm that takes as input a key $K \in \mathcal{K}_{\lambda,n,m}$ and an input $x \in \{0,1\}^n$, and outputs a value $y \in \{0,1\}^m$.*
- $\mathsf{PPRF.Punc}(K, x^*)$ *is a randomized algorithm that takes as input a key $K \in \mathcal{K}_{\lambda,n,m}$ and an input $x^* \in \{0,1\}^n$, and outputs a punctured key $K[x^*]$.*
- $\mathsf{PPRF.EvalPunc}(K[x^*], x)$ *is a deterministic algorithm that takes as input a punctured key $K[x^*]$ and an input $x \in \{0,1\}^n$, and outputs either a value $y \in \{0,1\}^m$ or $\perp$.*

We require the scheme to satisfy correctness under puncturing, and selective pseudorandomness at punctured points as defined below.

Remark 3.4. For convenience, we will sometimes combine PPRF.Eval and PPRF.EvalPunc into one algorithm. This can be done by having the combined algorithm run PPRF.Eval if it receives a regular key K and run PPRF.EvalPunc if it receives a punctured key $K[x^*]$ since the two types of keys are easily distinguishable in the construction from [SW14]. When using the combined algorithm, we will overload notation and refer to it simply by PPRF.Eval.

Definition 3.5 (Correctness under Puncturing)**.** *For all $\lambda, n, m \in \mathbb{N}$ and all $x^* \in \{0,1\}^n$, if $K \leftarrow \mathsf{PPRF.Setup}(1^\lambda, 1^n, 1^m)$ and $K[x^*] \leftarrow \mathsf{PPRF.Punc}(K, x^*)$, then*

$$\mathsf{PPRF.EvalPunc}(K[x^*], x) = \begin{cases} \mathsf{PPRF.Eval}(K, x) & \text{if } x \neq x^* \\ \perp & \text{else} \end{cases}$$

Definition 3.6 (Selective Pseudorandomness at Punctured Points)**.** *There exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PPRF}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PPRF}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

$\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PPRF}}(1^\lambda, b)$

1. ***Parameters**: $\mathcal{A}$ takes as input 1^λ, and outputs an input size 1^n, an output size 1^m, and a message $x^* \in \{0,1\}^n$.*
2. ***Compute Values**:*
   (a) $K \leftarrow \mathsf{PPRF.Setup}(1^\lambda, 1^n, 1^m)$.
   (b) $K[x^*] \leftarrow \mathsf{PPRF.Punc}(K, x^*)$.
   (c) *If $b = 0$, send $(y, K[x^*])$ to $\mathcal{A}$ where $y = \mathsf{PPRF.Eval}(K, x^*)$.*
   (d) *If $b = 1$, send $(r, K[x^*])$ to $\mathcal{A}$ where $r \leftarrow \{0,1\}^m$.*
3. ***Experiment Outcome**: $\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

3.3 Iterators

The following background is taken from Koppula et al. [KLW15] who construct iterators from $i\mathcal{O}$ for P/Poly and OWFs.[10] Informally speaking, a cryptographic iterator consists of a small state that is updated in an iterative fashion as messages are received. An update to incorporate a new message given the current state is performed with the help of some public parameters. Since states will remain relatively small regardless of the number of messages that have been iteratively incorporated, there will in general be many sequences of messages that lead to the same state. However, its security properties require that the normal public parameters should be computationally indistinguishable from specially constructed "enforcing" parameters which ensure that a particular *single* state can only be obtained as the outcome of an update to precisely one other state-message pair. Note that this enforcement is a very localized property to a specific state, and hence can be achieved information-theoretically when we fix ahead of time where exactly this enforcement is desired.

Definition 3.7 (Iterator [KLW15])**.** *A cryptographic iterator with state size $s(\cdot)$ is a tuple of PPT algorithms $\mathsf{Itr} = (\mathsf{Itr.Setup}, \mathsf{Itr.SetupEnforce}, \mathsf{Itr.Iterate})$ where*

- $\mathsf{Itr.Setup}(1^\lambda, 1^n, B)$ *is a randomized algorithm that takes as input the security parameter λ, a message size n, and a bound B (in binary) of the number of iterations, and outputs public parameters pp and an initial iterator state $\mathsf{itr}_0 \in \{0,1\}^{s(\lambda,n,\log(B))}$.*
- $\mathsf{Itr.SetupEnforce}(1^\lambda, 1^n, B, \{m_i\}_{i\in[k]})$ *is a randomized algorithm that takes as input the security parameter λ, a message size n, a bound B (in binary) of the number of iterations, and messages $\{m_i\}_{i\in[k]}$ where $k \leq B$ and each $m_i \in \{0,1\}^n$. It outputs public parameters pp and an initial iterator state $\mathsf{itr}_0 \in \{0,1\}^{s(\lambda,n,\log(B))}$.*
- $\mathsf{Itr.Iterate}(\mathsf{pp}, \mathsf{itr}_{\mathsf{in}}, m)$ *is a deterministic algorithm that takes as input public parameters pp, an iterator state $\mathsf{itr}_{\mathsf{in}} \in \{0,1\}^{s(\lambda,n,\log(B))}$, and a message $m \in \{0,1\}^n$, and outputs an iterator state $\mathsf{itr}_{\mathsf{out}} \in \{0,1\}^{s(\lambda,n,\log(B))}$.*

Security requires that the iterator satisfies indistinguishability of setup and the enforcing property defined below.

[10] The construction of [KLW15] additionally uses puncturable PRFs and a CPA secure PKE, which can be constructed from $i\mathcal{O}$ for P/Poly and OWFs.

Definition 3.8 (Indistinguishability of Setup)**.** *There exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{ltr\text{-}Setup}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{ltr\text{-}Setup}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

$\mathsf{Expt}_{\mathcal{A}}^{\mathsf{ltr\text{-}Setup}}(1^\lambda, b)$

1. ***Parameters**: $\mathcal{A}$ takes as input 1^λ and outputs a message size 1^n, a bound $B \in \Theta(2^\lambda)$, and messages $\{m_i\}_{i\in[k]}$ for some $k \leq B$ and where each $m_i \in \{0,1\}^n$.*
2. ***Compute Values**:*
   (a) $(\mathsf{pp}, \mathsf{itr}_0) \leftarrow \mathsf{ltr.Setup}(1^\lambda, 1^n, B)$.
   (b) $(\mathsf{pp}', \mathsf{itr}'_0) \leftarrow \mathsf{ltr.SetupEnforce}(1^\lambda, 1^n, B, \{m_i\}_{i\in[k]})$.
   (c) *If $b = 0$, send $(\mathsf{pp}, \mathsf{itr}_0)$ to $\mathcal{A}$.*
   (d) *If $b = 1$, send $(\mathsf{pp}', \mathsf{itr}'_0)$ to $\mathcal{A}$.*
3. ***Experiment Outcome**: $\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

Definition 3.9 (Enforcing)**.** *Let $\lambda, n \in \mathbb{N}$, $B \in \Theta(2^\lambda)$, $k < B$, and $\{m_i\}_{i\in[k]}$ where each $m_i \in \{0,1\}^n$. Define*

- $(\mathsf{pp}, \mathsf{itr}_0) \leftarrow \mathsf{ltr.SetupEnforce}(1^\lambda, 1^n, B, \{m_i\}_{i\in[k]})$.
- *For $i \in [k]$,* $\mathsf{itr}_i = \mathsf{ltr.Iterate}(\mathsf{pp}, \mathsf{itr}_{i-1}, m_i)$.

Then, ltr *is* enforcing *if for all $(\mathsf{itr}', m') \in \{0,1\}^{s(\lambda,n,\log(B))} \times \{0,1\}^n$,*

$$\mathsf{itr}_k = \mathsf{ltr.Iterate}(\mathsf{pp}, \mathsf{itr}', m') \implies (\mathsf{itr}', m') = (\mathsf{itr}_{k-1}, m_k)$$

Note that this is an information-theoretic property.

3.4 Splittable Signatures

The following background is taken from Koppula et al. [KLW15] who construct splittable signatures from $i\mathcal{O}$ for P/Poly and injective PRGs.[11] A splittable signature scheme is essentially a normal signature scheme augmented by some additional algorithms that produce alternative signing and verification keys with differing capabilities. More precisely, there are "all-but-one" signing and verification keys which work correctly for all messages except for a specific one, as well as "one" signing and verification keys which work only for a particular message. Additionally, there are "reject" verification keys which always reject signatures.

Definition 3.10 (Splittable Signature (SSig) [KLW15])**.** *A splittable signature scheme with signature size $s(\cdot)$ is a tuple of* PPT *algorithms* $\mathsf{SSig} = (\mathsf{SSig.Setup}, \mathsf{SSig.Sign}, \mathsf{SSig.Verify}, \mathsf{SSig.Split}, \mathsf{SSig.SignAbo})$ *where*

[11] The construction of [KLW15] additionally uses a puncturable PRF which can be constructed from OWFs (which are implied by PRGs).

- $\mathsf{SSig.Setup}(1^\lambda, 1^n)$ *is a randomized algorithm that takes as input the security parameter* λ *and a message size* n*, and outputs a signing key* sgk*, a verification key* vk*, and a rejecting verification key* $\mathsf{vk}_{\mathsf{rej}}$*.*
- $\mathsf{SSig.Sign}(\mathsf{sgk}, m)$ *is a deterministic algorithm that takes as input a signing key* sgk *and a message* $m \in \{0,1\}^n$ *and outputs a signature* $\sigma \in \{0,1\}^{s(\lambda,n)}$*.*
- $\mathsf{SSig.Verify}(\mathsf{vk}, m, \sigma)$ *is a deterministic algorithm that takes as input a verification key* vk*, a message* $m \in \{0,1\}^n$*, and a signature* $\sigma \in \{0,1\}^{s(\lambda,n)}$*, and outputs a bit* $b \in \{0,1\}$*.*
- $\mathsf{SSig.Split}(\mathsf{sgk}, m^*)$ *is a randomized algoirithm that takes as input a signing key* sgk *and a message* $m^* \in \{0,1\}^n$*, and outputs a signature* $\sigma_{\mathsf{one}} = \mathsf{SSig.Sign}(\mathsf{sgk}, m^*)$*, a one-message verification key* $\mathsf{vk}_{\mathsf{one}}$*, an all-but-one signing key* $\mathsf{sgk}_{\mathsf{abo}}$*, and an all-but-one verification key* $\mathsf{vk}_{\mathsf{abo}}$*.*
- $\mathsf{SSig.SignAbo}(\mathsf{sgk}_{\mathsf{abo}}, m)$ *is a deterministic algorithm that takes as input an all-but-one signing key* $\mathsf{sgk}_{\mathsf{abo}}$ *and a message* $m \in \{0,1\}^n$*, and outputs a signature* $\sigma \in \{0,1\}^{s(\lambda,n)}$*.*

SSig *must satisfy correctness and security as defined below.*

Remark 3.11. For convenience, we will sometimes combine $\mathsf{SSig.Sign}$ and $\mathsf{SSig.SignAbo}$ into one algorithm. This can be done by having the combined algorithm run $\mathsf{SSig.Sign}$ if it receives a regular signing key sgk and run $\mathsf{SSig.SignAbo}$ if it receives an all-but-one signing key $\mathsf{sgk}_{\mathsf{abo}}$ since the two types of signing keys are easily distinguishable in the construction from [KLW15]. When using the combined algorithm, we will overload notation and refer to it simply by $\mathsf{SSig.Sign}$.

Correctness. For any $\lambda, n \in \mathbb{N}$, let message $m^* \in \{0,1\}^n$, $(\mathsf{sgk}, \mathsf{vk}, \mathsf{vk}_{\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda, 1^n)$, and $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}, m^*)$. Then, we require the following correctness properties:

1. **Regular Correctness:** $\forall\, m \in \{0,1\}^n,\ \mathsf{SSig.Verify}(\mathsf{vk}, m, \mathsf{SSig.Sign}(\mathsf{sgk}, m)) = 1.$
2. **Correctness of Split Keys on Appropriate Values:**
   (a) $\sigma_{\mathsf{one}} = \mathsf{SSig.Sign}(\mathsf{sgk}, m^*).$
   (b) $\forall\, m \neq m^* \in \{0,1\}^n,\ \mathsf{SSig.SignAbo}(\mathsf{sgk}_{\mathsf{abo}}, m) = \mathsf{SSig.Sign}(\mathsf{sgk}, m).$
   (c) $\forall\, \sigma \in \{0,1\}^{s(\lambda,n)},\ \mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{one}}, m^*, \sigma) = \mathsf{SSig.Verify}(\mathsf{vk}, m^*, \sigma).$
   (d) $\forall\, m \neq m^* \in \{0,1\}^n, \sigma \in \{0,1\}^{s(\lambda,n)},\ \mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{abo}}, m, \sigma) = \mathsf{SSig.Verify}(\mathsf{vk}, m, \sigma).$
3. **Restrictions on Split Keys:**
   (a) $\forall\, m \neq m^* \in \{0,1\}^n, \sigma \in \{0,1\}^{s(\lambda,n)},\ \mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{one}}, m, \sigma) = 0.$
   (b) $\forall\, \sigma \in \{0,1\}^{s(\lambda,n)},\ \mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{abo}}, m^*, \sigma) = 0.$
4. $\mathsf{vk}_{\mathsf{rej}}$ **Always Rejects:** $\forall\, m \in \{0,1\}^n, \sigma \in \{0,1\}^{s(\lambda,n)},\ \mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{rej}}, m, \sigma) = 0.$

Security. SSig must satisfy $\mathsf{vk}_{\mathsf{rej}}$ indistinguishability, $\mathsf{vk}_{\mathsf{one}}$ indistintinguishability, $\mathsf{vk}_{\mathsf{abo}}$ indistinguishability, and splitting indistinguishability as defined below:

Definition 3.12 ($\mathsf{vk}_{\mathsf{rej}}$ Indistinguishability)**.** *There exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}REJ}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}REJ}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

$\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}REJ}}(1^\lambda, b)$

1. ***Parameters**: $\mathcal{A}$ takes as input 1^λ and outputs a message size 1^n.*
2. ***Compute Values**:*
   (a) $(\mathsf{sgk}, \mathsf{vk}, \mathsf{vk}_{\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda, 1^n)$.
   (b) *If $b = 0$, send vk to $\mathcal{A}$.*
   (c) *If $b = 1$, send $\mathsf{vk}_{\mathsf{rej}}$ to $\mathcal{A}$.*
3. ***Experiment Outcome**: $\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

Definition 3.13 ($\mathsf{vk}_{\mathsf{one}}$ Indistinguishability)**.** *There exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}ONE}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}ONE}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

$\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}ONE}}(1^\lambda, b)$

1. ***Parameters**: $\mathcal{A}$ takes as input 1^λ and outputs a message size 1^n and a message $m^* \in \{0,1\}^n$.*
2. ***Compute Values**:*
   (a) $(\mathsf{sgk}, \mathsf{vk}, \mathsf{vk}_{\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda, 1^n)$.
   (b) $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{sgk}_{\mathsf{one}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}, m^*)$.
   (c) *If $b = 0$, send $(\sigma_{\mathsf{one}}, \mathsf{vk})$ to $\mathcal{A}$.*
   (d) *If $b = 1$, send $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}})$ to $\mathcal{A}$.*
3. ***Experiment Outcome**: $\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

Definition 3.14 ($\mathsf{vk}_{\mathsf{abo}}$ Indistinguishability)**.** *There exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}ABO}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}ABO}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

$\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}ABO}}(1^\lambda, b)$

1. ***Parameters:*** *$\mathcal{A}$ takes as input 1^λ and outputs a message size 1^n and a message $m^* \in \{0,1\}^n$.*
2. ***Send Values:***
   (a) $(\mathsf{sgk}, \mathsf{vk}, \mathsf{vk}_{\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda, 1^n)$.
   (b) $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{sgk}_{\mathsf{one}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}, m^*)$.
   (c) *If $b = 0$, send $(\mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk})$ to $\mathcal{A}$.*
   (d) *If $b = 1$, send $(\mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{abo}})$ to $\mathcal{A}$.*
3. ***Experiment Outcome:*** *$\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

Definition 3.15 (Splitting Indistinguishability)**.** *There exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}Split}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}Split}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

$\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SSig\text{-}Split}}(1^\lambda, b)$

1. ***Parameters:*** *$\mathcal{A}$ takes as input 1^λ and outputs a message size 1^n and a message $m^* \in \{0,1\}^n$.*
2. ***Compute Values:***
   (a) $(\mathsf{sgk}, \mathsf{vk}, \mathsf{vk}_{\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda, 1^n)$.
   (b) $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{sgk}_{\mathsf{one}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}, m^*)$.
   (c) $(\mathsf{sgk}', \mathsf{vk}', \mathsf{vk}'_{\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda, 1^n)$.
   (d) $(\sigma'_{\mathsf{one}}, \mathsf{vk}'_{\mathsf{one}}, \mathsf{sgk}'_{\mathsf{abo}}, \mathsf{sgk}'_{\mathsf{one}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}', m^*)$.
   (e) *If $b = 0$, send $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{abo}})$ to $\mathcal{A}$.*
   (f) *If $b = 1$, send $(\sigma'_{\mathsf{one}}, \mathsf{vk}'_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{abo}})$ to $\mathcal{A}$.*
3. ***Experiment Outcome:*** *$\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

3.5 Functional Encryption

Here we give some fundamental definitions for functional encryption (FE) schemes. First, we define a class of functions parameterized by function size, input length, and output length.

Definition 3.16 (Function Class)**.** *The function class $\mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$ is the set of all functions f that have a description $\widehat{f} \in \{0,1\}^{\ell_{\mathcal{F}}}$, take inputs in $\{0,1\}^{\ell_{\mathcal{X}}}$, and output values in $\{0,1\}^{\ell_{\mathcal{Y}}}$.*

3.5.1 Public-Key Functional Encryption

Definition 3.17 (Public-Key Functional Encryption)**.** *A public-key functional encryption scheme for* $\mathsf{P/Poly}$ *is a tuple of PPT algorithms* $\mathsf{FE} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$ *defined as follows:*[12]

- $\mathsf{Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$*: takes as input the security parameter* λ*, a function size* $\ell_\mathcal{F}$*, an input size* $\ell_\mathcal{X}$*, and an output size* $\ell_\mathcal{Y}$*, and outputs the master public key* mpk *and the master secret key* msk*.*
- $\mathsf{Enc}(\mathsf{mpk}, x)$*: takes as input the master public key* mpk *and a message* $x \in \{0,1\}^{\ell_\mathcal{X}}$*, and outputs an encryption* ct *of* x*.*
- $\mathsf{KeyGen}(\mathsf{msk}, f)$*: takes as input the master secret key* msk *and a function* $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$*, and outputs a function key* sk_f*.*
- $\mathsf{Dec}(\mathsf{sk}_f, \mathsf{ct})$*: takes as input a function key* sk_f *and a ciphertext* ct*, and outputs a value* $y \in \{0,1\}^{\ell_\mathcal{Y}}$*.*

FE *satisfies* ***correctness*** *if for all polynomials* p*, there exists a negligible function* μ *such that for all* $\lambda \in \mathbb{N}$*, all* $\ell_\mathcal{F}, \ell_\mathcal{X}, \ell_\mathcal{Y} \leq p(\lambda)$*, all* $x \in \{0,1\}^{\ell_\mathcal{X}}$*, and all* $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$*,*

$$\Pr\left[\mathsf{Dec}(\mathsf{sk}_f, \mathsf{ct}_x) = f(x) : \begin{array}{c}(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}) \\ \mathsf{ct}_x \leftarrow \mathsf{Enc}(\mathsf{mpk}, x) \\ \mathsf{sk}_f \leftarrow \mathsf{KeyGen}(\mathsf{msk}, f)\end{array}\right] \geq 1 - \mu(\lambda).$$

We now define adaptive security.

Definition 3.18 (Adaptive Security for Public-Key FE)**.** *A public-key functional encryption scheme* FE *for* $\mathsf{P/Poly}$ *is adaptively secure if there exists a negligible function* μ *such that for all* $\lambda \in \mathbb{N}$ *and every PPT adversary* $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{Expt}_\mathcal{A}^{\mathsf{FE\text{-}Adaptive}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}_\mathcal{A}^{\mathsf{FE\text{-}Adaptive}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ *and* $\lambda \in \mathbb{N}$*, we define*

$\mathsf{Expt}_\mathcal{A}^{\mathsf{FE\text{-}Adaptive}}(1^\lambda, b)$

1. ***Parameters****:* $\mathcal{A}$ *takes as input* 1^λ*, and outputs a function size* $1^{\ell_\mathcal{F}}$*, an input size* $1^{\ell_\mathcal{X}}$*, and an output size* $1^{\ell_\mathcal{Y}}$*.*
2. ***Setup****: Compute* $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$*.*
3. ***Public Key****: Send* mpk *to* $\mathcal{A}$*.*
4. ***Function Queries Phase 1****: The following can be repeated any polynomial number of times:*
   (a) $\mathcal{A}$ *outputs a function query* $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$
   (b) $\mathsf{sk}_f \leftarrow \mathsf{FE.KeyGen}(\mathsf{msk}, f)$
   (c) *Send* sk_f *to* $\mathcal{A}$
5. ***Challenge Message Query****:*

[12] We also allow $\mathsf{Enc}, \mathsf{KeyGen}$, and Dec to additionally receive parameters $1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}$ as input, but omit them from our notation for convenience.

(a) $\mathcal{A}$ *outputs a challenge message pair* (x_0, x_1) *where* $x_0, x_1 \in \{0,1\}^{\ell_x}$.

(b) $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, x_b)$

(c) *Send* ct *to* $\mathcal{A}$.

6. ***Function Queries Phase 2****: This is identical to Function Queries Phase 1.*

7. ***Experiment Outcome****:* $\mathcal{A}$ *outputs a bit* b' *which is the output of the experiment.*

Additionally, when running the experiment, we immediately halt and output 0 *if the adversary ever aborts or if it at any point* $f(x_0) \neq f(x_1)$ *for some message query* (x_0, x_1) *and function query* f *submitted by the adversary.*

Definition 3.19 (Other Public-Key FE Security Definitions)**.** *There are many variations of the security definition. We list a few below:*

- ***Semi-Adaptive Security****: The adversary is required to make the message query before the function queries. This is identical to adaptive security, except that we remove* Function Queries Phase 1 *from the security game.*
- ***Function-Selective Semi-Adaptive Security:*** *The adversary is required to make all function queries before the message query. This is identical to adaptive security, except that we remove* Function Queries Phase 2 *from the security game.*
- ***Selective Security:*** *The adversary is required to make the message query at the beginning of the experiment before receiving the master public key. This is similar to adaptive security, except that in the security game, we move the* Challenge Message Query *step so that it now lies between the* Setup *step and the* Public Key *step. Note that the two function query phases are now adjacent and can thus be merged into one step.*
- ***Function-Selective Security:*** *The adversary is required to make the function queries at the beginning of the experiment before receiving the master public key. This is similar to adaptive security, except that in the security game, we move the two function query steps so that they now lie between the* Setup *step and the* Public Key *step. Note that the two function query phases are now adjacent and can thus be merged into one step.*

3.5.2 Secret-Key Functional Encryption

We can also define FE in the secret-key setting.

Definition 3.20 (Secret-Key Functional Encryption)**.** *Secret-key FE is the same as public-key FE except that* Setup *only outputs a master secret key and* Enc *requires the master secret key instead of the (non-existent) master public key. We formally define this in Appendix B.2.*

Remark 3.21. We can analogously define our public-key definitions of security in the secret-key setting. The only difference is that we do not give the (non-existent) master public key to the adversary and will therefore allow the adversary to submit multiple challenge message pairs. Note that in the secret-key setting, semi-adaptive security is equivalent to selective security, and function-selective semi-adaptive security is equivalent to function-selective security. We formally define these security definitions in Appendix B.2.

In the secret-key setting, we can also achieve function privacy. We define it now in the selective security setting.

Definition 3.22 (Function-Private-Selective-Security)**.** *A secret-key functional encryption scheme* FE *for* $\mathsf{P/Poly}$ *is function-private-selective-secure if there exists a negligible function* μ *such that for all* $\lambda \in \mathbb{N}$ *and every PPT adversary* $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{SKExpt}_{\mathcal{A}}^{\mathsf{FE\text{-}Func\text{-}Priv\text{-}Sel}}(1^\lambda, 0) = 1] - \Pr[\mathsf{SKExpt}_{\mathcal{A}}^{\mathsf{FE\text{-}Func\text{-}Priv\text{-}Sel}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ *and* $\lambda \in \mathbb{N}$*, we define*

$\mathsf{SKExpt}_{\mathcal{A}}^{\mathsf{FE\text{-}Func\text{-}Priv\text{-}Sel}}(1^\lambda, b)$

1. ***Parameters****:* $\mathcal{A}$ *takes as input* 1^λ*, and outputs a function size* $1^{\ell_\mathcal{F}}$*, an input size* $1^{\ell_\mathcal{X}}$*, and an output size* $1^{\ell_\mathcal{Y}}$*.*
2. ***Setup****:* $\mathsf{msk} \leftarrow \mathsf{FE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$*.*
3. ***Challenge Message Queries****:*
   (a) $\mathcal{A}$ *outputs challenge message pairs* $\{(x_{0,i}, x_{1,i})\}_{i\in[T]}$ *for some* T *chosen by the adversary where* $x_{0,i}, x_{1,i} \in \{0,1\}^{\ell_\mathcal{X}}$ *for all* $i \in [T]$*.*
   (b) *For* $i \in [T]$*, compute* $\mathsf{ct}_i \leftarrow \mathsf{FE.Enc}(\mathsf{msk}, x_{b,i})$ *and send* ct_i *to* $\mathcal{A}$*.*
4. ***Function Queries****: The following can be repeated any polynomial number of times:*
   (a) $\mathcal{A}$ *outputs a function query pair* (f_0, f_1) *where* $f_0, f_1 \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$
   (b) $\mathsf{sk}_f \leftarrow \mathsf{FE.KeyGen}(\mathsf{msk}, f_b)$
   (c) *Send* sk_f *to* $\mathcal{A}$
5. ***Experiment Outcome****:* $\mathcal{A}$ *outputs a bit* b' *which is the output of the experiment.*

Additionally, when running the experiment, we immediately halt and output 0 *if the adversary ever aborts or if it at any point* $f_0(x_0) \neq f_1(x_1)$ *for some message query* (x_0, x_1) *and function query* (f_0, f_1) *submitted by the adversary.*

3.5.3 Single-Key, Single-Ciphertext Security

Definition 3.23 (Single-Key, Single-Ciphertext Security)**.** *We can add the modifier "single-key. single-ciphertext" to any of our security definitions. This is a weakening of the security definition where we only require security against an adversary who is restricted to making only one function query and submitting only one challenge message pair in the relevant security game.*

3.5.4 Strong-Compactness

Additionally, we might also want our FE scheme to be *strongly-compact.*[13] Intuitively, this means that the sizes and running times of the setup and encryption algorithms are independent of the sizes of the circuits for which function keys are produced.

Definition 3.24 (Strong-Compactness)**.** *An FE scheme* $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.Enc}, \mathsf{FE.KeyGen}, \mathsf{FE.Dec})$ *for* $\mathsf{P/Poly}$ *is said to be* strongly-compact *if there exist PPT algorithms* $\mathsf{FE.Setup}^*, \mathsf{FE.Enc}^*$ *such*

[13] We call it strong-compactness since the usual notion of compactness found in the literature only requires the encryption algorithm to not grow with the function size.

that for all polynomials p, for all large enough $\lambda, \ell_{\mathcal{X}}$, we have that for all $\ell_{\mathcal{F}}, \ell_{\mathcal{Y}} \leq p(\lambda + \ell_{\mathcal{X}})$, the following holds:

- $\mathsf{FE.Setup}(1^\lambda, 1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}})$ *is identically distributed to* $\mathsf{FE.Setup}^*(1^\lambda, 1^{\ell_{\mathcal{X}}})$
- *For all* $\mathsf{mpk} \leftarrow \mathsf{FE.Setup}(1^\lambda, 1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}})$ *and all* $x \in \{0,1\}^{\ell_{\mathcal{X}}}$, $\mathsf{FE.Enc}(1^\lambda, 1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}}, \mathsf{mpk}, x)$ *is identically distributed to* $\mathsf{FE.Enc}^*(1^\lambda, 1^{\ell_{\mathcal{X}}}, \mathsf{mpk}, x)$

We will often abuse notation and write $\mathsf{FE.Setup}$ *to mean* $\mathsf{FE.Setup}^*$ *and write* $\mathsf{FE.Enc}$ *to mean* $\mathsf{FE.Enc}^*$.

3.6 Streaming Functional Encryption

Guan, Korb, and Sahai [GKS23] define streaming functional encryption (sFE) as functional encryption (FE) for a class of *streaming functions*.

3.6.1 Streaming Functions

Definition 3.25 (Streaming Function [GKS23])**.** *A streaming function with state space $\mathcal{S}$, input space $\mathcal{X}$, output space $\mathcal{Y}$, and starting state*[14] $\mathsf{st}_1 \in \mathcal{S}$ *is a function $f : \mathcal{X} \times \mathcal{S} \to \mathcal{Y} \times \mathcal{S}$.*

- *The* ***output*** *of f on $x = x_1 \dots x_n \in \mathcal{X}^n$ (denoted $f(x)$) is defined to be $y = y_1 \dots y_n \in \mathcal{Y}^n$ where*

$$\forall i \in [n],\ (y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$$

Definition 3.26 (Streaming Function Class [GKS23])**.** *The streaming function class $\mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$ is the set of all streaming functions f that have a description $\widehat{f} \in \{0,1\}^{\ell_{\mathcal{F}}}$, state space $\mathcal{S} = \{0,1\}^{\ell_{\mathcal{S}}}$, input space $\mathcal{X} = \{0,1\}^{\ell_{\mathcal{X}}}$, output space $\mathcal{Y} = \{0,1\}^{\ell_{\mathcal{Y}}}$, and starting state*[15] $\mathsf{st}_1 = \bot$.

When constructing $\mathsf{Pre\text{-}One\text{-}sFE}$, we will work with a specific streaming function class $\mathcal{F}_\bot$.

Definition 3.27 ($\mathcal{F}_\bot$)**.** *$\mathcal{F}_\bot$ is the set of all two-input functions in* $\mathsf{P/Poly}$ *such that if the first input is $\bot$, the function always outputs $\bot$ regardless of the second input, i.e.*

$$\mathcal{F}_\bot = \{\text{two-input } f \in \mathsf{P/Poly} : \forall s, f(\bot, s) = \bot\}.$$

If f is a streaming function, then $f \in \mathcal{F}_\bot$ means that $f(\bot, \mathsf{st}) = (\bot, \bot)$ for any state st.

Remark 3.28. Note that constructing a sFE scheme for the restricted function class $\mathcal{F}_\bot$ does not hinder the usability of the scheme since every one-input function can be interpreted as a two-input function, and for every two-input function $f \in \mathsf{P/Poly}$, we can construct a function $f' \in \mathsf{P/Poly}$ with essentially the same functionality by defining $f'(z, s) = \begin{cases} f(x, s) & \text{if } z = 1||x \text{ for some } x \\ \bot & \text{else} \end{cases}$.

[14] If not specified, we assume st_1 to be $\bot$ (or the all-zero string) by default.

[15] $\mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$ requires $\mathsf{st}_1 = \bot$. However, all of our results still hold even if we expand our function class to include functions with arbitrary starting states as we can simply include the starting state in the function description.

3.6.2 Public Key Streaming Function Encryption

Following the syntax of standard FE, we define public key sFE as follows.

Definition 3.29 (Public-Key Streaming FE [GKS23])**.** *A public-key streaming functional encryption scheme for* $\mathsf{P/Poly}$ *is a tuple of PPT algorithms* $\mathsf{sFE} = (\mathsf{Setup}, \mathsf{EncSetup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$ *defined as follows:*[16]

- $\mathsf{Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$*: takes as input the security parameter* λ*, a function size* $\ell_\mathcal{F}$*, a state size* $\ell_\mathcal{S}$*, an input size* $\ell_\mathcal{X}$*, and an output size* $\ell_\mathcal{Y}$*, and outputs the master public key* mpk *and the master secret key* msk*.*
- $\mathsf{EncSetup}(\mathsf{mpk})$*: takes as input the master public key* mpk *and outputs an encryption state* $\mathsf{Enc.st}$*.*
- $\mathsf{Enc}(\mathsf{mpk}, \mathsf{Enc.st}, i, x_i)$*: takes as input the master public key* mpk*, an encryption state* $\mathsf{Enc.st}$*, an index* i*, and a message* $x_i \in \{0,1\}^{\ell_\mathcal{X}}$ *and outputs an encryption* ct_i *of* x_i*.*
- $\mathsf{KeyGen}(\mathsf{msk}, f)$*: takes as input the master secret key* msk*, and a function* $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ *and outputs a function key* sk_f*.*
- $\mathsf{Dec}(\mathsf{sk}_f, \mathsf{Dec.st}_i, i, \mathsf{ct}_i)$*: where for each function key* sk_f*,* $\mathsf{Dec}(\mathsf{sk}_f, \cdot, \cdot, \cdot)$ *is a streaming function that takes as input a state* $\mathsf{Dec.st}_i$*, an index* i*, and an encryption* ct_i *and outputs a new state* $\mathsf{Dec.st}_{i+1}$ *and an output* $y_i \in \{0,1\}^{\ell_\mathcal{Y}}$*.*

sFE *must be* ***streaming efficient****, meaning that the size and runtime of all algorithms of* sFE *on security parameter* λ*, function size* $\ell_\mathcal{F}$*, state size* $\ell_\mathcal{S}$*, input size* $\ell_\mathcal{X}$*, and output size* $\ell_\mathcal{Y}$ *are* $\mathsf{poly}(\lambda, \ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y})$*.*

sFE *satisfies* ***correctness*** *if for all polynomials* p*, there exists a negligible function* μ *such that for all* $\lambda \in \mathbb{N}$*, all* $\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y} \le p(\lambda)$*, all* $n \in [2^\lambda]$*, all* $x = x_1 \dots x_n$ *where each* $x_i \in \{0,1\}^{\ell_\mathcal{X}}$*, and all* $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$*,*

$$\Pr\left[\overline{\mathsf{Dec}}(\mathsf{sk}_f, \mathsf{ct}_x) = f(x) : \begin{array}{c}(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}), \\ \mathsf{ct}_x \leftarrow \overline{\mathsf{Enc}}(\mathsf{mpk}, x) \\ \mathsf{sk}_f \leftarrow \mathsf{KeyGen}(\mathsf{msk}, \mathsf{Dec.st}_1, f)\end{array}\right] \ge 1 - \mu(\lambda)$$

where we define[17]

- $\overline{\mathsf{Enc}}(\mathsf{mpk}, x)$ *outputs* $\mathsf{ct}_x = (\mathsf{ct}_i)_{i\in[n]}$ *produced by sampling* $\mathsf{Enc.st} \leftarrow \mathsf{EncSetup}(\mathsf{mpk})$ *and then computing* $ct_i \leftarrow \mathsf{Enc}(\mathsf{mpk}, \mathsf{Enc.st}, i, x_i)$ *for* $i \in [n]$*.*
- $\overline{\mathsf{Dec}}(\mathsf{sk}_f, \mathsf{ct}_x)$ *outputs* $y = (y_i)_{i\in[n]}$ *where* $(y_i, \mathsf{Dec.st}_{i+1}) = \mathsf{Dec}(\mathsf{sk}_f, \mathsf{Dec.st}_i, i, \mathsf{ct}_i)$ *for* $i \in [n]$*.*

We now define adaptive security. Our definition of security is adaptive in a very strong sense, in that the adversary can not only adaptively pick each of the next values of its challenge streams based on the ciphertexts and function keys already received, but can also interweave function queries between the message queries.

[16] We also allow $\mathsf{Enc}, \mathsf{EncSetup}, \mathsf{KeyGen}$, and Dec to additionally receive parameters $1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}$ as input, but omit them from our notation for convenience.

[17] As with all streaming functions, we assume that $\mathsf{Dec.st}_1 = \bot$ if not otherwise specified.

Definition 3.30 (Adaptive Security for Public-Key sFE)**.** *A public-key streaming FE scheme* sFE *for* $\mathsf{P/Poly}$ *is adaptively secure if there exists a negligible function* μ *such that for all* $\lambda \in \mathbb{N}$ *and all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{sFE\text{-}Adaptive}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{sFE\text{-}Adaptive}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ *and* $\lambda \in \mathbb{N}$*, we define*

$\mathsf{Expt}_{\mathcal{A}}^{\mathsf{sFE\text{-}Adaptive}}(1^\lambda, b)$

1. ***Parameters****:* $\mathcal{A}$ *takes as input* 1^λ*, and outputs a function size* $1^{\ell_{\mathcal{F}}}$*, a state size* $1^{\ell_{\mathcal{S}}}$*, an input size* $1^{\ell_{\mathcal{X}}}$*, and an output size* $1^{\ell_{\mathcal{Y}}}$*.*
2. ***Setup****: Compute* $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{sFE.Setup}(1^\lambda, 1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}})$*.*
3. ***Public Key****: Send* mpk *to* $\mathcal{A}$*.*
4. *For a polynomial number of rounds, the adversary can do either one of the following in each round:*
   (a) ***Function Query****:*
      i. $\mathcal{A}$ *outputs a streaming function query* $f \in \mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$*.*
      ii. $\mathsf{sk}_f \leftarrow \mathsf{sFE.KeyGen}(\mathsf{msk}, f)$*.*
      iii. *Send* sk_f *to* $\mathcal{A}$*.*
   (b) ***Challenge Message Query****:*
      i. *If this is the first challenge message query, sample* $\mathsf{Enc.st} \leftarrow \mathsf{sFE.EncSetup}(\mathsf{mpk})$ *and initialize the index* $i = 1$*. Else, increment the index* i *by* 1*.*
      ii. $\mathcal{A}$ *outputs a challenge message pair* $(x_i^{(0)}, x_i^{(1)})$ *where* $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_{\mathcal{X}}}$*.*
      iii. $\mathsf{ct}_i \leftarrow \mathsf{sFE.Enc}(\mathsf{mpk}, \mathsf{Enc.st}, i, x_i^{(b)})$*.*
      iv. *Send* ct_i *to* $\mathcal{A}$*.*
5. ***Experiment Outcome****:* $\mathcal{A}$ *outputs a bit* b' *which is the output of the experiment.*

Additionally, when running the experiment, we immediately halt and output 0 *if the adversary ever aborts or if it at any point some function query* f *submitted by the adversary yields different outputs on the challenge message streams submitted so far (i.e. if* $f(x^{(0)}) \neq f(x^{(1)})$ *for some function query* f *submitted by the adversary where* $\{(x_i^{(0)}, x_i^{(1)})\}_{i \in [t]}$ *are the message queries submitted so far,* $x^{(0)} = x_1^{(0)} \dots x_t^{(0)}$*, and* $x^{(1)} = x_1^{(1)} \dots x_t^{(1)}$*).*

Definition 3.31 (Other Public-Key sFE Security Definitions)**.** *There are many variations of the security definition. We list a few below:*

- ***Semi-Adaptive Security****: The adversary is required to make all message queries before any function queries. This is identical to adaptive security, except that we do not allow the adversary to make a* Challenge Message Query *after it has made a* Function Query*.*
- ***Function-Selective Semi-Adaptive Security:*** *The adversary is required to make all function queries before any message queries. This is identical to adaptive security, except that we do not allow the adversary to make a* Function Query *after it has made a* Challenge Message Query*.*

- ***Selective Security:*** *The adversary is required to make the message query at the beginning of the experiment before receiving the master public key. This is similar to adaptive security, except that we allow the adversary to take a polynomial number of* Challenge Message Query *steps in between the* Setup *step and the* Public Key *step, but do not allow the adversary to take any* Challenge Message Query *steps afer the* Public Key *step.*

- ***Function-Selective Security:*** *The adversary is required to make the function queries at the beginning of the experiment before receiving the master public key. This is similar to adaptive security, except that we allow the adversary to take a polynomial number of* Function Query *steps in between the* Setup *step and the* Public Key *step, but do not allow the adversary to take any* Function Query *steps afer the* Public Key *step.*

3.6.3 Secret-Key Streaming Functional Encryption

We can also define sFE in the secret-key setting.

Definition 3.32 (Secret-Key Streaming Functional Encryption)**.** *Secret-key* sFE *is the same as public-key* sFE *except that* Setup *only outputs a master secret key and* $\mathsf{EncSetup}$ *and* Enc *require the master secret key instead of the (non-existent) master public key. We formally define this in Appendix B.3.*

Remark 3.33. We can analogously define our public-key definitions of security in the secret-key setting. The only difference is that we do not give the (non-existent) master public key to the adversary and will therefore allow the adversary to submit multiple pairs of challenge streams. Note that in the secret-key setting, semi-adaptive security is equivalent to selective security, and function-selective semi-adaptive security is equivalent to function-selective security. We formally define these security definitions in Appendix B.3.

3.6.4 Single-Key, Single-Ciphertext Security

Definition 3.34 (Single-Key, Single-Ciphertext Security)**.** *We can add the modifier "single-key. single-ciphertext" to any of our security definitions. This is a weakening of the security definition where we only require security against an adversary who is restricted to making only one function query and submitting only one pair of challenge message streams (though each stream may still consist of many elements) in the relevant security game.*

3.6.5 Notational Variations

Remark 3.35 (Providing the starting state st_1 as input to KeyGen)**.** When constructing our intermediate sFE schemes, we will sometimes define the KeyGen algorithm so that it additionally takes the starting state st_1 as input. This does not affect the scheme's ability to be a standalone sFE scheme since all functions $f \in \mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$ have starting state $\mathsf{st}_1 = \bot$, so we can simply define a new key generation algorithm with the proper amount of inputs by hardwiring $\mathsf{st}_1 = \bot$ into the old KeyGen algorithm.

Remark 3.36 (Modeling sFE decryption as a streaming function.)**.** We can easily change any sFE scheme sFE' into a new sFE scheme sFE^* with the same security but whose decryption algorithm is a streaming function in the standard format (i.e. takes only two inputs: a state and a value). This is achieved by modifying the KeyGen and Dec algorithms as below so that the decryption state also includes the function key for f and the index i.

Let $\mathsf{sFE}' = (\mathsf{sFE}'.\mathsf{Enc}, \mathsf{sFE}'.\mathsf{EncSetup}, \mathsf{sFE}'.\mathsf{KeyGen}, \mathsf{sFE}'.\mathsf{Dec})$ be a sFE scheme. We define sFE scheme $\mathsf{sFE}^* = (\mathsf{sFE}'.\mathsf{Enc}, \mathsf{sFE}'.\mathsf{EncSetup}, \mathsf{sFE}^*.\mathsf{KeyGen}, \mathsf{sFE}^*.\mathsf{Dec})$ where

- $\mathsf{sFE}^*.\mathsf{KeyGen}(\mathsf{msk}, f)$
  1. $\mathsf{sk}_f \leftarrow \mathsf{sFE}'.\mathsf{KeyGen}(\mathsf{msk}, f)$.
  2. Output $\mathsf{Dec.st}_1^* = (\mathsf{sk}_f, \mathsf{Dec.st}_1, 1)$.
- $\mathsf{sFE}^*.\mathsf{Dec}(\mathsf{Dec.st}_i^*, \mathsf{ct}_i)$
  1. Parse $\mathsf{Dec.st}_i^* = (\mathsf{sk}_f, \mathsf{Dec.st}_i, i)$.
  2. $(y_i, \mathsf{Dec.st}_{i+1}) = \mathsf{sFE}'.\mathsf{Dec}(\mathsf{sk}_f, \mathsf{Dec.st}_i, i, \mathsf{ct}_i)$.
  3. $\mathsf{Dec.st}_{i+1}^* = (\mathsf{sk}_f, \mathsf{Dec.st}_{i+1}, i+1)$.
  4. Output $(y_i, \mathsf{Dec.st}_{i+1}^*)$.

Note that in this case, KeyGen simply outputs the first decryption state $\mathsf{Dec.st}_1^*$. It is easy to see that sFE' and sFE^* have the same security.

4 Pre-One-sFE

We first build a single-key, single-ciphertext, *selectively* secure sFE scheme which we call Pre-One-sFE.

Theorem 4.1. *Assuming $i\mathcal{O}$ for* P/Poly *and injective* PRG*s, there exists a single-key, single-ciphertext, selectively secure, secret-key sFE scheme for the function class* $\mathcal{F}_\perp = \{\text{two-input } f \in \mathsf{P/Poly} : \forall s, f(\perp, s) = \perp\}$.

Please refer to the technical overview (Section 2) for a high level overview of our construction. To prove Theorem 4.1, we build an sFE scheme from the following tools, which as we show below, can each be instantiated using $i\mathcal{O}$ for P/Poly and OWFs.

Tools.

- $\mathsf{SKE} = (\mathsf{SKE.Setup}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$: A secure symmetric key encryption scheme.
- $\mathsf{PPRF} = (\mathsf{PPRF.Setup}, \mathsf{PPRF.Eval}, \mathsf{PPRF.Punc})$: A secure puncturable pseudorandom function family.[18]
- $\mathsf{Itr} = (\mathsf{Itr.Setup}, \mathsf{Itr.SetupEnforce}, \mathsf{Itr.Iterate})$: A cryptographic iterator.
- $\mathsf{SSig} = (\mathsf{SSig.Setup}, \mathsf{SSig.Sign}, \mathsf{SSig.Verify}, \mathsf{SSig.Split})$: A secure splittable signature scheme.[19]
- $i\mathcal{O}$: An indistinguishability obfuscator for P/Poly.

Instantiation of the Tools. Let $i\mathcal{O}$ be an indistinguishability obfuscator for P/Poly, and let PRG be an injective pseudorandom generator.

- We can build SKE and a one-way-function from PRGs using standard cryptographic techniques (e.g. [Gol01, Gol09]).
- We can build PPRF from any one-way-function as shown in [SW14, BW13, BGI14, KPTZ13].
- We can build SSig and Itr from $i\mathcal{O}$ and injective PRGs as shown in [KLW15].

4.1 Parameters

On security parameter λ, function size $L_\mathcal{F}$, state size $L_\mathcal{S}$, input size $L_\mathcal{X}$, and output size $L_\mathcal{Y}$, we will instantiate our primitives with the following parameters:

- SKE: We instantiate SKE with message size $L_{\mathsf{SKE}.m} = \mathsf{max}(L_\mathcal{S}, L_\mathcal{X})$. This means that we will use the following setup algorithm: $\mathsf{SKE.Setup}(1^\lambda, 1^{L_{\mathsf{SKE}.m}})$. When encrypting or decrypting messages of size less than this, we assume that we pad the messages accordingly.

  Observe that the algorithms and ciphertexts of SKE are of size $\mathsf{poly}(\lambda, L_\mathcal{S}, L_\mathcal{X})$.

- PPRF: We overload notation and instantiate PPRF in two different ways:

  1. With input size λ and output size λ. This means that we will use the following setup algorithm: $\mathsf{PPRF.Setup}(1^\lambda, 1^\lambda, 1^\lambda)$.

[18] As per Remark 3.4, we have combined PPRF.Eval and PPRF.EvalPunc into one algorithm.
[19] As per Remark 3.11, we have combined SSig.Sign and SSig.SignAbo into one algorithm.

2. With input size λ and output size 2λ. This means that we will use the following setup algorithm: $\mathsf{PPRF.Setup}(1^\lambda, 1^\lambda, 1^{2\lambda})$.

Which instantiation we are using can be determined by context as we will always use output length λ with keys $K_{\mathsf{inp}}, K_A, K_B$ and will always use output length 2λ with keys K_E.

Observe that the algorithms of both instantiations of PPRF are of size $\mathsf{poly}(\lambda)$.

- Itr: We instantiate Itr with message size $L_{\mathsf{Itr}.m} = \lambda + L_{\mathsf{SKE.ct}}$ and bound $B = 2^\lambda$ where $L_{\mathsf{SKE.ct}}$ is the size of ciphertexts of SKE. This means that we will use the following setup algorithm: $\mathsf{Itr.Setup}(1^\lambda, 1^{L_{\mathsf{Itr}.m}}, 2^\lambda)$.

  By properties of the iterator, the iterator state itr does not grow in size as we iterate more values into it. Thus, apart from $\mathsf{Itr.SetupEnforce}$ (which is used only in the security proof), the algorithms of Itr also remain the same size regardless of how many values we have iterated. Therefore, since $L_{\mathsf{Itr}.m} = \mathsf{poly}(\lambda, L_{\mathsf{SKE.ct}}) = \mathsf{poly}(\lambda, L_{\mathcal{S}}, L_{\mathcal{X}})$, then every occurrence of an algorithm of Itr in our construction is of size $\mathsf{poly}(\lambda, L_{\mathcal{S}}, L_{\mathcal{X}})$.

- SSig: We instantiate SSig with message size $L_{\mathsf{SSig}.m} = \lambda + L_{\mathsf{SKE.ct}} + L_{\mathsf{Itr.itr}}$ where $L_{\mathsf{SKE.ct}}$ is the size of ciphertexts of SKE and $\mathsf{Itr.itr}$ is the size of iterator states of Itr. This means that we will use the following setup algorithm: $\mathsf{SSig.Setup}(1^\lambda, 1^{L_{\mathsf{SSig}.m}})$.

  Therefore, since $L_{\mathsf{SSig}.m} = \mathsf{poly}(\lambda, L_{\mathsf{SKE.ct}}, L_{\mathsf{Itr.itr}}) = \mathsf{poly}(\lambda, L_{\mathcal{S}}, L_{\mathcal{X}})$, then the algorithms and signatures of SSig are of size $\mathsf{poly}(\lambda, L_{\mathcal{S}}, L_{\mathcal{X}})$.

- $i\mathcal{O}$: We instantiate $i\mathcal{O}$ for the set of all circuits in $\mathsf{P/Poly}$ with

  - circuit size L_{Prog} which is defined to be the maximum size of all programs which are obfuscated in the construction and security proof;
  - input size $L_{\mathsf{in}} = \lambda + 2L_{\mathsf{SKE.ct}} + 2L_{\mathsf{SSig}.\sigma} + L_{\mathsf{Itr.itr}}$;
  - output size $L_{\mathsf{out}} = L_{\mathcal{Y}} + L_{\mathsf{SKE.ct}} + L_{\mathsf{SSig}.\sigma} + L_{\mathsf{Itr.itr}}$;

  where $L_{\mathsf{SKE.ct}}$ is the size of ciphertexts of SKE, $L_{\mathsf{SSig}.\sigma}$ is the size of signatures of SSig, and $L_{\mathsf{Itr.itr}}$ is the size of iterator states of Itr. When signing or verifying messages of size less than this, we assume that we pad the messages accordingly.

  Observe that this means L_{in} and L_{out} are of size $\mathsf{poly}(\lambda, L_{\mathcal{S}}, L_{\mathcal{X}})$.

  It is tedious, but straightforward to check that each of the programs that are obfuscated in our construction and security proof are of size $\mathsf{poly}(\lambda, L_{\mathcal{F}}, L_{\mathcal{S}}, L_{\mathcal{X}}, L_{\mathcal{Y}})$ and do not have size dependent on the length of the stream. Therefore, the obfuscator $i\mathcal{O}$ and the obfuscated program $\mathcal{P}$ will be of size $\mathsf{poly}(L_{\mathsf{Prog}}) = \mathsf{poly}(\lambda, L_{\mathcal{F}}, L_{\mathcal{S}}, L_{\mathcal{X}}, L_{\mathcal{Y}})$.

Notation. For notational convenience, when the parameters are understood, we will often omit the security, input size, output size, message size, or state size parameters from each of the algorithms listed above.

Remark 4.2. We assume without loss of generality that for security parameter λ, all algorithms only require randomness of length λ. If the original algorithm requires additional randomness, we can replace it with a new algorithm that first expands the λ bits of randomness using a PRG of appropriate stretch and then runs the original algorithm. Note that this replacement does not affect the security of the above schemes (as long as $L_{\mathcal{F}}, L_{\mathcal{S}}, L_{\mathcal{X}}, L_{\mathcal{Y}}$ are polynomial in λ).

4.2 Construction

We now construct Pre-One-sFE. Recall that for notational convenience, we may omit the security, input size, output size, message size, function size, or state size parameters from our algorithms. For information on these parameters, please see the parameter section above.

For later use, we have defined our KeyGen algorithm so that it additionally takes the starting state st_1 as input. However, as per Remark 3.35, this does not affect the viability of Pre-One-sFE as a standalone scheme. We now describe our construction.

- $\mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$:
  1. $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
  * Throughout, for $i \in [2^\lambda]$, we will define

$$\begin{aligned} r_{\mathsf{inp},i} &= \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i) \\ (\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) &= \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},i}) \\ r_{A,i} &= \mathsf{PPRF.Eval}(K_A, i) \\ (\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) &= \mathsf{SSig.Setup}(1^\lambda; r_{A,i}) \\ (r_{E,i}, r_{\mathsf{Enc},i}) &= \mathsf{PPRF.Eval}(K_E, i) \\ k_{E,i} &= \mathsf{SKE.Setup}(1^\lambda; r_{E,i}) \end{aligned}$$

  2. Output $\mathsf{MSK} = (K_{\mathsf{inp}}, K_A, K_E)$.
- $\mathsf{Pre\text{-}One\text{-}sFE.EncSetup}(\mathsf{MSK})$: Output $\mathsf{Enc.st} = \bot$.
- $\mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, \mathsf{Enc.st}, i, x_i)$:
  1. Parse $\mathsf{MSK} = (K_{\mathsf{inp}}, K_A, K_E)$.
  2. **Compute** $\mathsf{ct}_{\mathsf{inp},i}$:
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}_{\mathsf{inp},i} \leftarrow \mathsf{SKE.Enc}(k_{E,i}, x_i)$.
  3. **Compute** $\sigma_{\mathsf{inp},i}$:
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     (b) $\sigma_{\mathsf{inp},i} \leftarrow \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.
  4. Output $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$.
- $\mathsf{Pre\text{-}One\text{-}sFE.KeyGen}(\mathsf{MSK}, f, \mathsf{st}_1)$:
  1. Parse $\mathsf{MSK} = (K_{\mathsf{inp}}, K_A, K_E)$.
  2. **Compute** $\mathsf{ct}_{\mathsf{st},1}$**:**
     (a) $(r_{E,1}, r_{\mathsf{Enc},1}) = \mathsf{PPRF.Eval}(K_E, 1)$.
     (b) $k_{E,1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,1})$.
     (c) $\mathsf{ct}_{\mathsf{st},1} = \mathsf{SKE.Enc}(k_{E,1}, \mathsf{st}_1; r_{\mathsf{Enc},1})$.
  3. **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda)$.
  4. **Compute** $\sigma_{\mathsf{st},1}$**:**

(a) $m_1 = (1, \mathsf{ct}_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$.

(b) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

(c) $\sigma_{\mathsf{st},1} \leftarrow \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1)$.

5. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}[f, K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}])$ where Prog is defined in Figure 2.
6. Output $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$.

- $\mathsf{Pre\text{-}One\text{-}sFE.Dec}(\mathsf{SK}_f, \mathsf{Dec.st}_i, i, \mathsf{CT}_i)$

1. Parse $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ and $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$.
2. If $i > 1$, parse $\mathsf{Dec.st}_i = (\mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
3. Compute $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i}) = \mathcal{P}(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
4. Set $\mathsf{Dec.st}_{i+1} = (\mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
5. Output $(y_i, \mathsf{Dec.st}_{i+1})$.

Program $\mathsf{Prog}[f, K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. **Verify state signature:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.
2. **Computation Step:**
   - i. **Decrypt input and state**:
     - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
     - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     - iii. $x_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
     - iv. $\mathsf{st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
   - ii. **Compute output value and next state:**
     - i. $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$.
   - iii. **Encrypt the new state**:
     - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
     - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
     - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   - i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   - ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   - iii. **Sign the new state:**
     - i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
     - ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Figure 2: Def of Prog.

4.3 Correctness and Efficiency

Efficiency. Using our discussion above on parameters, it is easy to see that the size and runtime of all algorithms of Pre-One-sFE on security parameter λ, function size $L_{\mathcal{F}}$, state size $L_{\mathcal{S}}$, input size $L_{\mathcal{X}}$, and output size $L_{\mathcal{Y}}$ are $\mathsf{poly}(\lambda, L_{\mathcal{F}}, L_{\mathcal{S}}, L_{\mathcal{X}}, L_{\mathcal{Y}})$.

Correctness Intuition. Given an encryption of x_i, an encryption of st_i, and signatures for both ciphertexts, then the obfuscated program $\mathcal{P}$ outputs y_i along with a ciphertext and signature for st_{i+1} where $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$.

The decryptor obtains the obfuscated program $\mathcal{P}$ from SK_f, and a ciphertext and signature for each x_i from CT_i. To get started, the decryptor also obtains a ciphertext and signature for the first state st_1 from SK_f. Decryption works by iteratively running $\mathcal{P}$ on the ciphertexts and signatures for x_i and st_i to get the output value y_i along with the ciphertext and signature of the next state st_{i+1}, which is needed for the next decryption step.

Correctness. While we can only prove security for functions $f \in \mathcal{F}_\perp$, we can prove correctness for all functions $f \in \mathsf{P/Poly}$. Furthermore, correctness holds even if we allow the function f to have an arbitrary starting state st_1 (as long as this state is provided as additional input to KeyGen as described in the construction).

More formally, let p be any polynomial and consider any $\lambda \in \mathbb{N}$ and any $L_\mathcal{F}, L_\mathcal{S}, L_\mathcal{X}, L_\mathcal{Y} \leq p(\lambda)$. Let SK_f be a function key for some function $f \in \mathcal{F}[L_\mathcal{F}, L_\mathcal{S}, L_\mathcal{X}, L_\mathcal{Y}]$ with starting state[20] st_1, and let $\{\mathsf{CT}_i\}_{i\in[n]}$ be a ciphertext for some x where $x = x_1 \dots x_n$ for some $n \in [2^\lambda]$ and where each $x_i \in \{0,1\}^{L_\mathcal{X}}$.

By correctness of SKE, SSig, and $i\mathcal{O}$, if

- $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$ and $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$,
  $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_E, i))$,
  $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$;
- $\mathcal{P} = i\mathcal{O}(\mathsf{Prog}[f, K_\mathsf{inp}, K_A, K_E, \mathsf{pp}_\mathsf{st}])$;
- $\mathsf{ct}_{\mathsf{inp},i}$ is an SKE encryption of x_i under key $k_{E,i}$;
- $\sigma_{\mathsf{inp},i}$ is a signature of $\mathsf{ct}_{\mathsf{inp},i}$ signed using $\mathsf{sgk}_{\mathsf{inp},i}$;
- $\mathsf{ct}_{\mathsf{st},i}$ is an SKE encryption of st_i under key $k_{E,i}$;
- $\sigma_{\mathsf{st},i}$ is a signature of $\mathsf{ct}_{\mathsf{st},i}$ signed using $\mathsf{sgk}_{A,i}$;
- $\mathsf{itr}_{\mathsf{st},i-1}$ is an iterator state associated with pp_st;

then

$$\begin{aligned}\mathcal{P}(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st}_i}) &= \mathsf{Prog}[f, K_\mathsf{inp}, K_A, K_E, \mathsf{pp}_\mathsf{st}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st}_i}) \\ &= (y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})\end{aligned}$$

where

- $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$;
- $\mathsf{ct}_{\mathsf{st},i+1}$ is an SKE encryption of st_{i+1} under key $k_{E,i+1}$;
- $\sigma_{\mathsf{st},i+1}$ is a signature of $\mathsf{ct}_{\mathsf{st},i+1}$ signed using $\mathsf{sgk}_{A,i+1}$;
- $\mathsf{itr}_{\mathsf{st},i}$ is an iterator state associated with pp_st.

[20] By definition, all functions $f \in \mathcal{F}[L_\mathcal{F}, L_\mathcal{S}, L_\mathcal{X}, L_\mathcal{Y}]$ have starting state $\mathsf{st}_1 = \perp$. Here, we are using an expanded definition of $\mathcal{F}[L_\mathcal{F}, L_\mathcal{S}, L_\mathcal{X}, L_\mathcal{Y}]$ which allows f to have an arbitrary starting state $\mathsf{st}_1 \in \{0,1\}^{L_\mathcal{S}}$.

Observe that for $i \in [n]$,

$$\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$$

where $\mathsf{ct}_{\mathsf{inp},i}$ is an SKE encryption of x_i under key $k_{E,i}$ and $\sigma_{\mathsf{inp},i}$ is a signature of $\mathsf{ct}_{\mathsf{inp},i}$ signed using $\mathsf{sgk}_{\mathsf{inp},i}$. Additionally,

$$\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$$

where $\mathcal{P} = i\mathcal{O}(\mathsf{Prog}[f, K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}])$, $\mathsf{ct}_{\mathsf{st},1}$ is an SKE encryption of st_1 under key $k_{E,1}$, $\sigma_{\mathsf{st},1}$ is a signature of $\mathsf{ct}_{\mathsf{st},1}$ signed using $\mathsf{sgk}_{A,1}$, and $\mathsf{itr}_{\mathsf{st},0}$ is an iterator state associated with $\mathsf{pp}_{\mathsf{st}}$.

Therefore, for $i = 1$,

$$\begin{aligned}
&\mathsf{Pre\text{-}One\text{-}sFE.Dec}(\mathsf{SK}_f, \mathsf{Dec.st}_1, 1, \mathsf{CT}_1) \\
&= \mathcal{P}(1, \mathsf{ct}_{\mathsf{inp},1}, \sigma_{\mathsf{inp},1}, \mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0}) \\
&= (y_1, \mathsf{ct}_{\mathsf{st},2}, \sigma_{\mathsf{st},2}, \mathsf{itr}_{\mathsf{st},1}) \\
&= (y_1, \mathsf{Dec.st}_2) \text{ for } \mathsf{Dec.st}_2 = (\mathsf{ct}_{\mathsf{st},2}, \sigma_{\mathsf{st},2}, \mathsf{itr}_{\mathsf{st},1})
\end{aligned}$$

where $(y_1, \mathsf{st}_2) = f(x_1, \mathsf{st}_1)$, $\mathsf{ct}_{\mathsf{st},2}$ is an SKE encryption of st_2 under key $k_{E,2}$, $\sigma_{\mathsf{st},2}$ is a signature of $\mathsf{ct}_{\mathsf{st},2}$ signed using $\mathsf{sgk}_{A,2}$, and $\mathsf{itr}_{\mathsf{st},1}$ is an iterator state associated with $\mathsf{pp}_{\mathsf{st}}$.

For $i > 1$, if $\mathsf{Dec.st}_i = (\mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$ where $\mathsf{ct}_{\mathsf{st},i}$ is an SKE encryption of st_i under key $k_{E,i}$, $\sigma_{\mathsf{st},i}$ is a signature of $\mathsf{ct}_{\mathsf{st},i}$ signed using $\mathsf{sgk}_{A,i}$, and $\mathsf{itr}_{\mathsf{st},i-1}$ is an iterator state associated with $\mathsf{pp}_{\mathsf{st}}$, then

$$\begin{aligned}
&\mathsf{Pre\text{-}One\text{-}sFE.Dec}(\mathsf{SK}_f, \mathsf{Dec.st}_i, i, \mathsf{CT}_i) \\
&= \mathcal{P}(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1}) \\
&= (y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i}) \\
&= (y_i, \mathsf{Dec.st}_{i+1}) \text{ for } \mathsf{Dec.st}_{i+1} = (\mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})
\end{aligned}$$

where $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$, $\mathsf{ct}_{\mathsf{st},i+1}$ is an SKE encryption of st_{i+1} under key $k_{E,i+1}$, $\sigma_{\mathsf{st},i+1}$ is a signature of $\mathsf{ct}_{\mathsf{st},i+1}$ signed using $\mathsf{sgk}_{A,i+1}$, and $\mathsf{itr}_{\mathsf{st},i-1}$ is an iterator state associated with $\mathsf{pp}_{\mathsf{st}}$.

Thus, by induction on i and the decryption state, the decryption algorithm correctly outputs $y = y_1 \dots y_n$ where $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$ for $i \in [n]$.

4.4 Additional Algorithms

We define the following algorithm which will be used in our security proof. It is similar to KeyGen except that it hardwires in the output values at steps $t-1$ and t for some chosen t and some chosen output values. We have highlighted the differences between this function and $\mathsf{Pre\text{-}One\text{-}sFE.KeyGen}$.

- $\mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{MSK}, f, \mathsf{st}_1, t, y^*_{t-1}, y^*_t, \mathsf{st}_{t+1})$
  1. Parse $\mathsf{MSK} = (K_{\mathsf{inp}}, K_A, K_E)$.
  2. **Compute $\mathsf{ct}_{\mathsf{st},1}$:**
     (a) $(r_{E,1}, r_{\mathsf{Enc},1}) = \mathsf{PPRF.Eval}(K_E, 1)$.
     (b) $k_{E,1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,1})$.
     (c) $\mathsf{ct}_{\mathsf{st},1} = \mathsf{SKE.Enc}(k_{E,1}, \mathsf{st}_1; r_{\mathsf{Enc},1})$.
  3. **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda)$.
  4. **Compute $\sigma_{\mathsf{st},1}$:**
     (a) $m_1 = (1, \mathsf{ct}_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$.

(b) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

(c) $\sigma_{\mathsf{st},1} \leftarrow \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1)$.

5. **Compute $\mathsf{ct}^*_{\mathsf{st},t}$ and $\mathsf{ct}^*_{\mathsf{st},t+1}$:**

(a) For $i \in \{t, t+1\}$,

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

(b) $\mathsf{ct}^*_{\mathsf{st},t} = \mathsf{SKE.Enc}(k_{E,t}, \bot; r_{\mathsf{Enc},t})$.

(c) $\mathsf{ct}^*_{\mathsf{st},t+1} = \mathsf{SKE.Enc}(k_{E,t+1}, \mathsf{st}_{t+1}; r_{\mathsf{Enc},t})$.

6. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{ProgHardwire}[f, K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

7. Output $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$.

Program $\mathsf{ProgHardwire}[f, K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

i. **Verify i is positive:** If $i \leq 0$, output $\bot$.

ii. **Verify input signature:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. **Verify state signature:**

i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\bot$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

iii. If $i \notin \{t-1, t\}$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $x_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. **Sign the new state:**

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

4.5 Security

We now prove that Pre-One-sFE is single-key, single-ciphertext, selectively secure for the function class $\mathcal{F}_\perp = \{\text{two-input } f \in \mathsf{P/Poly} : \forall s, f(\perp, s) = \perp\}$.

In this proof, we will use an alternate, but equivalent, definition of single-key, single-ciphertext, selective security where instead of needing to distinguish between an encryption of stream $x^{(0)}$ and an encryption of stream $x^{(1)}$, the adversary will receive an encryption of stream $x^{(b)}$ for a random bit b and will win if they correctly guess b.

Definition 4.3 (Single-Key, Single-Ciphertext, Selective Security for Secret-Key sFE, Equivalent Definition)**.** *A secret-key streaming FE scheme* sFE *for $\mathcal{F}_\perp$ is single-key, single-ciphertext, selectively secure if there exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all PPT adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{SKExptGuess}_{\mathcal{A}}^{\mathsf{sFE\text{-}1\text{-}Key\text{-}1\text{-}CT\text{-}Sel}}(1^\lambda) = 1]\right| \leq \frac{1}{2} + \mu(\lambda)$$

where for $\lambda \in \mathbb{N}$, we define

$\mathsf{SKExptGuess}_{\mathcal{A}}^{\mathsf{sFE\text{-}1\text{-}Key\text{-}1\text{-}CT\text{-}Sel}}(1^\lambda)$

1. ***Parameters**: $\mathcal{A}$ takes as input 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.*

2. ***Setup**: Compute $\mathsf{msk} \leftarrow \mathsf{sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.*

3. ***Challenge Bit**: Sample $b \leftarrow \{0,1\}$.*

4. *The following can be repeated any polynomial number of times:*

   (a) ***Challenge Message Query**:*

      i. *If this is the first challenge message query, sample $\mathsf{Enc.st} \leftarrow \mathsf{sFE.EncSetup}(\mathsf{msk})$ and initialize the index $i = 1$. Else, increment the index i by 1.*

      ii. *$\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.*

      iii. *$\mathsf{ct}_i \leftarrow \mathsf{sFE.Enc}(\mathsf{msk}, \mathsf{Enc.st}, i, x_i^{(b)})$.*

      iv. *Send ct_i to $\mathcal{A}$.*

5. ***Function Query**:*

   (a) *$\mathcal{A}$ outputs a streaming function query $f \in \mathcal{F}_\perp \cap \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$.*

   (b) *$\mathsf{sk}_f \leftarrow \mathsf{sFE.KeyGen}(\mathsf{msk}, f)$.*

   (c) *Send sk_f to $\mathcal{A}$.*

6. ***Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.*

Additionally, when running the experiment, we immediately halt and output 0 if the adversary ever aborts or if it at any point some function query f submitted by the adversary yields different outputs on the challenge message streams submitted so far (i.e. if $f(x^{(0)}) \neq f(x^{(1)})$ for some function query f submitted by the adversary where $\{(x_i^{(0)}, x_i^{(1)})\}_{i\in[t]}$ are the message queries submitted so far, $x^{(0)} = x_1^{(0)} \dots x_t^{(0)}$, and $x^{(1)} = x_1^{(1)} \dots x_t^{(1)}$).

Using standard techniques, it is easy to show that this is equivalent to the regular definition of single-key, single-ciphertext, selective security.

4.5.1 Formal Proof

Proof Overview. The first part of the proof of security of $\mathsf{One\text{-}sFE}$ in Section 6.4 contains a near-complete proof of the single-key, single-ciphertext, selective security of $\mathsf{Pre\text{-}One\text{-}sFE}$. As such, in this section, we will rely heavily on lemmas proven in Section 6.4, and will only show the small modifications needed to finish the security proof for $\mathsf{Pre\text{-}One\text{-}sFE}$. We refer the readers to Section 6.4 for the bulk of the security proof.

In more detail, the proof from Section 6.4 allows us to move from the original security game to a hybrid that is nearly independent of the challenge bit b. Unfortunately, this hybrid still needs to know the final output state $\mathsf{st}_{t^*+1}^{(b)}$ of challenge stream $x^{(b)} = x_1^{(b)} \dots x_{t^*}^{(b)}$ which depends on the bit b. To deal with this, we will need to weaken our function class from $\mathsf{P/Poly}$ to $\mathcal{F}_\perp$. This allows us to add a "dummy" query of $\perp$ to the end of both streams, which causes the final state of both streams to be $\perp$ (and thus independent of b). Then the hybrid becomes fully independent of b which implies security.

Hybrid Argument. We prove security via a hybrid argument starting with $\mathbf{Hybrid}_{\mathsf{Pre},0}^{\mathcal{A}}$ which represents the single-key, single-ciphertext, selective security game.

Remark 4.4. We require all of our unwrapped[21] hybrids to immediately halt and output 0 if the adversary ever aborts or if it at any point some function query f submitted by the adversary yields different outputs on the challenge message streams submitted so far (i.e. if $f(x^{(0)}) \neq f(x^{(1)})$ for some function query f submitted by the adversary where $\{(x_i^{(0)}, x_i^{(1)})\}_{i\in[t]}$ are the message queries submitted so far, $x^{(0)} = x_1^{(0)} \dots x_t^{(0)}$, and $x^{(1)} = x_1^{(1)} \dots x_t^{(1)}$). For notational simplicity, we omit this requirement from the description of our hybrids.

[21] Every hybrid named $\mathbf{Hybrid}_{\mathsf{sub}}^{\mathcal{A}}$ for some subscrip sub is considered an "unwrapped" hybrid. See the discussion before $\mathbf{Hybrid}_{\mathsf{Pre},2}^{\mathcal{A}}$ for details.

$\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},0}(1^\lambda)$: This is identical to $\mathsf{SKExptGuess}^{\mathsf{sFE\text{-}1\text{-}Key\text{-}1\text{-}CT\text{-}Sel}}_{\mathcal{A}}(1^\lambda)$.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{L_\mathcal{F}}$, a state size $1^{L_\mathcal{S}}$, an input size $1^{L_\mathcal{X}}$, and an output size $1^{L_\mathcal{Y}}$.
2. **Setup:** $\mathsf{MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
3. **Challenge Bit:** $b \leftarrow \{0,1\}$.
4. **Encryption:** For $i = 1, 2, \ldots$
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, i, x_i^{(b)})$.
   (c) Send CT_i to $\mathcal{A}$.
5. **KeyGen:**
   (a) $\mathcal{A}$ sends function f with starting state st_1 to the challenger.
   (b) $\mathsf{SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGen}(\mathsf{MSK}, f, \mathsf{st}_1)$.
   (c) Send SK_f to $\mathcal{A}$.
6. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

$\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},1}(1^\lambda)$: We now add a dummy query of $\perp$ to the end of both streams. Note that since $f \in \mathcal{F}_\perp$, then for all states s, $f(\perp, s) = (\perp, \perp)$. Thus, it will still be the case that $f(x^{(0)}) = f(x^{(1)})$. Additionally, this will make the final output states of both streams equal which will prove useful in a later hybrid.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{L_\mathcal{F}}$, a state size $1^{L_\mathcal{S}}$, an input size $1^{L_\mathcal{X}}$, and an output size $1^{L_\mathcal{Y}}$.
2. **Setup:** $\mathsf{MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
3. **Challenge Bit:** $b \leftarrow \{0, 1\}$.
4. **Encryption:** For $i = 1, 2, \ldots$
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, i, x_i^{(b)})$.
   (c) Send CT_i to $\mathcal{A}$.
5. **Dummy Query:**
   (a) Define $(x_{t^*}^{(0)}, x_{t^*}^{(1)}) = (\perp, \perp)$ where t^* is one more than the length of the challenge streams.
   (b) $\mathsf{CT}_{t^*} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, t^*, x_{t^*}^{(b)})$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends function f with starting state st_1 to the challenger.
   (b) $\mathsf{SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGen}(\mathsf{MSK}, f, \mathsf{st}_1)$.
   (c) Send SK_f to $\mathcal{A}$.
7. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma 4.5. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\left|\Pr[(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},0}(1^\lambda) = 1] - \Pr[\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},1})(1^\lambda) = 1]\right| = 0$$

Proof. The hybrids are identical. □

Using the security proof from Section 6.4. To move to the next hybrid, we will use a security proof that is very similar to the first part of the proof of security for $\mathsf{One\text{-}sFE}$ in Section 6.4. In this hybrid, we encrypt stream $x^{(0)}$ rather than stream $x^{(b)}$. However, we still compute the final state st_{t^*+1} using stream $x^{(b)}$.

We define the following wrapper function Wrap for our hybrids. We will call any hybrid named $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{sub}}$ for some subscript sub an *unwrapped* hybrid, and will call any hybrid named $\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{sub}})$ for some subscript sub a *wrapped* hybrid.

$\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},2})(1^\lambda)$:

1. For $\mathsf{Iteration} \in [T^3_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ:
   (a) **Run hybrid:** $v \leftarrow \mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},2}(1^\lambda)$.
   (b) **Check for correct guess:** If $v \neq \bot$, output v and halt.
2. Output 0.

$\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},2}(1^\lambda)$:

1. **Guess Stream Length:** $t^* - 1 \leftarrow [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{L_\mathcal{F}}$, a state size $1^{L_\mathcal{S}}$, an input size $1^{L_\mathcal{X}}$, and an output size $1^{L_\mathcal{Y}}$.
3. **Setup:** $\mathsf{MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption:** For $i = 1, 2, \ldots, t^* - 1$:
   If the adversary does not make exactly $t^* - 1$ queries during this phase, output $\bot$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, i, x_i^{(0)})$.
   (c) Send CT_i to $\mathcal{A}$.
6. **Dummy Query:**
   (a) Define $(x_{t^*}^{(0)}, x_{t^*}^{(1)}) = (\bot, \bot)$.
   (b) $\mathsf{CT}_{t^*} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, t^*, \bot)$.
7. **KeyGen:**
   (a) $\mathcal{A}$ sends function f with starting state st_1 to the challenger.
   (b) **Compute $(y_{t^*-1}, y_{t^*}, \mathsf{st}_{t^*+1})$:**
      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.
      ii. For $i \in [t^*]$,
         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i, \mathsf{st}_i^{(b)})$.

iii. $\mathsf{st}_{t^*+1} = \mathsf{st}^{(b)}_{t^*+1}$.

(c) $\mathsf{SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{MSK}, f, \mathsf{st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{st}_{t^*+1})$.

(d) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma 4.6. *If $i\mathcal{O}$ is a indistinguishability obfuscator,* PPRF *is a puncturable pseudorandom function,* SSig *is a splittable signature scheme,* Itr *is a cryptographic iterator, and* SKE *is a symmetric key encryption scheme, then for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\left|\Pr[(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},1}(1^\lambda) = 1] - \Pr[\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},2})(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. The proof is essentially the same as the proof of indistinguishability from Section 6.4 between $\mathbf{Hybrid}^{\mathcal{A}}_0$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},1}$) and $\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_6)$ (which is similar to $\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},2})$). The only differences in the hybrids and proofs are the following:

- We have removed **Encryption Phase 2** since we are in the selective security game. This only serves to make the proofs of indistinguishability even easier.
- The last query is a dummy query rather than one chosen by the adversary. The proof can be easily adapted to account for this change. Note that since $f \in \mathcal{F}_\perp$, it will still be the case that $f(x^{(0)}) = f(x^{(1)})$.
- We encrypt different streams and functions with Pre-One-sFE. In this proof, we use Pre-One-sFE to encrypt message streams

  $$x^{(0)} = x^{(0)}_1, x^{(0)}_2, x^{(0)}_3, \ldots$$
  $$x^{(1)} = x^{(1)}_1, x^{(1)}_2, x^{(1)}_3, \ldots$$

  and to create a function key for f with starting state st_1. In the proof of One-sFE, we use Pre-One-sFE to encrypt message streams

  $$\mathsf{Post.CT}^{(0)} = \mathsf{Post.CT}^{(0)}_1, \mathsf{Post.CT}^{(0)}_2, \mathsf{Post.CT}^{(0)}_3, \ldots$$
  $$\mathsf{Post.CT}^{(1)} = \mathsf{Post.CT}^{(1)}_1, \mathsf{Post.CT}^{(1)}_2, \mathsf{Post.CT}^{(1)}_3, \ldots$$

  and to create a function key for Post-One-sFE.Dec with starting state $\mathsf{Post\text{-}One\text{-}sFE.Dec.st}_1$ (where $\mathsf{Post.CT}^{(b)}_i$ is a Post-One-sFE encryption of $x^{(b)}_i$ and $\mathsf{Post\text{-}One\text{-}sFE.Dec.st}_1$ is the Post-One-sFE function key for f).

  Since we do not use any properties of Post-One-sFE until after $\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_6)$, exchanging the streams and functions can be considered more of a notational change. Thus, the correctness of the proof is unaffected.

Therefore, essentially the same proof as in Section 6.4 can be applied to prove this lemma. □

$\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},3}(1^\lambda)$: We now move to a hybrid that is independent of the bit b by replacing stream $x^{(b)}$ with stream $x^{(0)}$ when computing $(y_{t^*-1}, y_{t^*}, \mathsf{st}_{t^*+1})$. This relies on the fact that $f \in \mathcal{F}_\perp$.

1. **Guess Stream Length:** $t^* - 1 \leftarrow [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{L_\mathcal{F}}$, a state size $1^{L_\mathcal{S}}$, an input size $1^{L_\mathcal{X}}$, and an output size $1^{L_\mathcal{Y}}$.
3. **Setup:** $\mathsf{MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption:** For $i = 1, 2, \ldots, t^* - 1$:
   If the adversary does not make exactly $t^* - 1$ queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, i, x_i^{(0)})$.
   (c) Send CT_i to $\mathcal{A}$.
6. **Dummy Query:**
   (a) Let $(x_{t^*}^{(0)}, x_{t^*}^{(1)}) = (\perp, \perp)$.
   (b) $\mathsf{CT}_{t^*} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, t^*, \perp)$.
7. **KeyGen:**
   (a) $\mathcal{A}$ sends function f with starting state st_1 to the challenger.
   (b) **Compute** $(y_{t^*-1}, y_{t^*}, \mathsf{st}_{t^*+1})$**:**
      i. $\mathsf{st}_1^{(0)} = \mathsf{st}_1$.
      ii. For $i \in [t^*]$,
         A. $(y_i, \mathsf{st}_{i+1}^{(0)}) = f(x_i, \mathsf{st}_i^{(0)})$.
      iii. $\mathsf{st}_{t^*+1} = \mathsf{st}_{t^*+1}^{(0)}$.
   (c) $\mathsf{SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{MSK}, f, \mathsf{st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{st}_{t^*+1})$
   (d) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.
8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma 4.7. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},2})(1^\lambda) = 1] - \Pr[\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},3})(1^\lambda) = 1]\right| = 0$$

Proof. The hybrids are identical. Since the adversary is required to submit queries such that $f(x^{(0)}) = f(x^{(1)})$, then the value of y_{t^*-1} will be the same in both hybrids. Since $f \in \mathcal{F}_\perp$, then for all states s, $f(\perp, s) = (\perp, \perp)$. Thus,

$$f(x_{t^*}^{(b)}, \mathsf{st}_{t^*}^{(b)}) = f(\perp, \mathsf{st}_{t^*}^{(b)}) = (\perp, \perp) = f(\perp, \mathsf{st}_{t^*}^{(0)}) = f(x_{t^*}^{(0)}, \mathsf{st}_{t^*}^{(0)})$$

so the values of $y_{t^*}, \mathsf{st}_{t^*+1}$ will also be the same in both hybrids. Thus, we will compute identical values for $(y_{t^*-1}, y_{t^*}, \mathsf{st}_{t^*+1})$. □

Lemma 4.8. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Pr[\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},3})(1^\lambda) = 1] \leq \frac{1}{2}$$

Proof. The proof relies on the fact that the adversary's view in $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},3}$ is independent of the challenge bit b.

In each iteration of Wrap, we run an instance of $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},3}$. The output of Wrap is defined to be the output of the first instance of $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},3}$ where we did not output $\bot$, or 0 if all instances output $\bot$. Consider any instance of $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},3}$. Conditioned on not outputting $\bot$, we will only output 1 if the adversary correctly guesses $b = b'$. However, since the adversary's view in $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},3}$ is independent of the bit b, then conditioned on not outputting $\bot$, the probability that the adversary makes it to the end of the hybrid and guesses correctly b is at most $\frac{1}{2}$. Thus, the probability that $\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}}_{3})(1^\lambda)$ outputs 1 is at most $\frac{1}{2}$. □

Corollary 4.9. *If $i\mathcal{O}$ is a indistinguishability obfuscator,* PPRF *is a puncturable pseudorandom function,* SSig *is a splittable signature scheme,* Itr *is a cryptographic iterator, and* SKE *is a symmetric key encryption scheme, then for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Pr[\mathsf{SKExptGuess}^{\mathsf{sFE\text{-}1\text{-}Key\text{-}1\text{-}CT\text{-}Sel}}_{\mathcal{A}}(1^\lambda)] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

or, in other words, Pre-One-sFE *is single-key, single-ciphertext, selectively secure for $\mathcal{F}_\bot$.*

Proof. This follows by combining all of our intermediate lemmas since $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{Pre},0}$ is identical to $\mathsf{SKExptGuess}^{\mathsf{sFE\text{-}1\text{-}Key\text{-}1\text{-}CT\text{-}Sel}}_{\mathcal{A}}$. □

5 Post-One-sFE

Guan, Korb, and Sahai [GKS23] construct a single-key, single-ciphertext, function-selectively secure sFE scheme which we call Post-One-sFE.

Theorem 5.1 ([GKS23]). *Assuming a strongly-compact, selectively secure, secret-key FE scheme for* P/Poly*, there exists a single-key, single-ciphertext, function-selectively secure, secret-key sFE scheme for* P/Poly*.*

In this section, we provide the construction of Post-One-sFE and prove additional properties for it that will be useful in the security proof of our adaptive scheme One-sFE. For convenience, in the construction, we have made some minor, mostly notational changes, including merging the two PRFs in [GKS23] into one PRF. However, these changes do not affect any relevant properties of the construction.

Post-One-sFE is built from the following tools, which as we show below, can each be instantiated using a strongly-compact, selectively secure, secret-key FE scheme for P/Poly.

Tools.

- $\mathsf{PRF} = (\mathsf{PRF.Setup}, \mathsf{PRF.Eval})$: A secure pseudorandom function family.
- $\mathsf{SKE} = (\mathsf{SKE.Setup}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$: A secure symmetric key encryption scheme.
- $\mathsf{SKE}' = (\mathsf{SKE}'.\mathsf{Setup}, \mathsf{SKE}'.\mathsf{Enc}, \mathsf{SKE}'.\mathsf{Dec})$: A secure symmetric key encryption scheme.
- $\mathsf{OneCompFE} = (\mathsf{OneCompFE.Setup}, \mathsf{OneCompFE.Enc}, \mathsf{OneCompFE.KeyGen}, \mathsf{OneCompFE.Dec})$: A *strongly-compact*, single-key, single-ciphertext, *selectively* secure, secret-key FE scheme for P/Poly.
- $\mathsf{OneFSFE} = (\mathsf{OneFSFE.Setup}, \mathsf{OneFSFE.Enc}, \mathsf{OneFSFE.KeyGen}, \mathsf{OneFSFE.Dec})$: A single-key, single-ciphertext, *function-selectively* secure, secret-key FE scheme for P/Poly.

Instantiation of the Tools. Let SKFE be a strongly-compact, selectively secure, secret-key FE scheme for P/Poly.

- We can build $\mathsf{PRF}, \mathsf{SKE}, \mathsf{SKE}'$ from any one-way-function using standard cryptographic techniques (e.g. [Gol01, Gol09]). As FE implies one-way-functions, then we can build these from SKFE.
- SKFE already satisfies the compactness and security requirements needed for OneCompFE.
- We can first build a function-private, selectively secure, secret-key FE scheme FPFE for P/Poly by using the function-privacy transformation of [BS18] on SKFE. As observed in [BS18], a single-key, single-ciphertext, function-private, selectively secure, secret-key FE scheme for P/Poly is also a (non-compact) single-key, single-ciphertext, function-selectively secure, secret-key FE scheme for P/Poly as we can simply exchange the roles of the functions and messages using universal circuits. Thus, FPFE can be used to build OneFSFE.

5.1 Parameters

The parameters are identical to those in [GKS23] except that rather than using two PRFs, we use one PRF which has input size λ and output size $6\lambda + \ell_{\mathcal{S}}$. Thus, we do not redefine the parameters here. Additionally, for notational convenience, we will often omit the security, input size, output size, message size, function size, and state size parameters from our algorithms.

5.1.1 Post-One-sFE Construction

We now construct Post-One-sFE. This is identical to the construction from [GKS23] except for some minor, mostly notational changes.

For later use, we have applied a similar transformation as in Remark 3.36 to turn our decryption algorithm into a streaming function in the standard format (i.e. takes only two inputs: a decryption state and a ciphertext). In this case, KeyGen simply outputs the first decryption state $\mathsf{Dec.st}_1$. As this transformation only requires renaming sk_f to $\mathsf{Dec.st}_1$ and adding the index i to each $\mathsf{Dec.st}_i$, this change can be considered a notational variation, rather than a major change to the underlying algorithms.

We have also defined our KeyGen algorithm so that it additionally takes the starting state st_1 as input (rather than hardwiring $\mathsf{st}_1 = \bot$ as in [GKS23]).

- $\mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$:
  1. $K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$.
  * Throughout, for $i \in [2^\lambda]$, we will define
  $$K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i}) = \mathsf{PRF.Eval}(K, i)$$
  from which we can compute the following values defined below
  $$\begin{aligned} \mathsf{msk}_i &= \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i}) \\ \mathsf{msk}'_i &= \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i}) \\ k_i &= \mathsf{SKE.Setup}(1^\lambda; r_{k_i}) \\ k'_i &= \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i}) \end{aligned}$$
  2. Output $\mathsf{MSK} = K$.
- $\mathsf{Post\text{-}One\text{-}sFE.EncSetup}(\mathsf{MSK})$: Output $\mathsf{Enc.st} = \bot$.
- $\mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, \mathsf{Enc.st}, i, x_i)$:
  1. Parse $\mathsf{MSK} = K$.
  2. Compute $\mathsf{msk}_i, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, k_i, k'_i, \mathsf{msk}'_i$ from K.
  3. $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (x_i, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_\mathcal{Y}}))$.
  4. If $i = 1$, output $\mathsf{CT}_1 = \mathsf{ct}_1$.
  5. If $i > 1$,
     (a) $c_i \leftarrow \mathsf{SKE.Enc}(k_i, \bot)$.
     (b) $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \bot)$.
     (c) Let $h_i = h_{c_i, c'_i}$ as defined in Figure 4.
     (d) $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$.
     (e) Output $\mathsf{CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.
- $\mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{MSK}, f, \mathsf{st}_1)$:
  1. Parse $\mathsf{MSK} = K$.
  2. Compute $\mathsf{msk}_1, k_1, p_1, r_{\mathsf{KeyGen}_1}$ from K.

3. $c_1 \leftarrow \mathsf{SKE.Enc}(k_1, \bot)$.
4. $\widetilde{\mathsf{st}}_1 = \mathsf{st}_1 \oplus p_1$.
5. Let $g_1 = g_{f,\widetilde{\mathsf{st}}_1,c_1}$ as defined in Figure 3.
6. $\mathsf{sk}_{g_1} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_1, g_1; r_{\mathsf{KeyGen}_1})$.
7. Output $\mathsf{Dec.st}_1 = (1, \mathsf{sk}_{g_1})$.

$g_{f,\widetilde{\mathsf{st}}_i,c_i}(x_i, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, \alpha_i, r_{k_i}, \psi_i)$:

- If $\alpha_i = 0$,
  1. $\mathsf{st}_i = \widetilde{\mathsf{st}}_i \oplus p_i$.
  2. $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$.
  3. $\widetilde{\mathsf{st}}_{i+1} = \mathsf{st}_{i+1} \oplus p_{i+1}$.
  4. $\mathsf{msk}'_{i+1} = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_{i+1}})$.
  5. $\mathsf{ct}'_{i+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (f, \widetilde{\mathsf{st}}_{i+1}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{i+1}})$.
  6. Output (y_i, ct'_{i+1}).
- Else,
  1. $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$.
  2. $(\theta_i, \mathsf{ct}'_{i+1}) = \mathsf{SKE.Dec}(k_i, c_i)$.
  3. Output $(\theta_i \oplus \psi_i, \mathsf{ct}'_{i+1})$.

Figure 3: Definition of $g_{f,\widetilde{\mathsf{st}}_i,c_i}$.

$h_{c_i,c'_i}(f, \widetilde{\mathsf{st}}_i, r_{\mathsf{msk}_i}, r_{\mathsf{KeyGen}_i}, \alpha'_i, r'_{k_i})$:

- If $\alpha'_i = 0$,
  1. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$.
  2. Let $g_i = g_{f,\widetilde{\mathsf{st}}_i,c_i}$ as defined in Figure 3.
  3. $\mathsf{sk}_{g_i} = \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_i, g_i; r_{\mathsf{KeyGen}_i})$.
  4. Output sk_{g_i}.
- Else,
  1. $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.
  2. Output $\mathsf{sk}_{g_i} = \mathsf{SKE'.Dec}(k'_i, c'_i)$.

Figure 4: Definition of h_{c_i,c'_i}.

- $\mathsf{One\text{-}sFE.Dec}(\mathsf{Dec.st}_i, \mathsf{CT}_i)$:

1. Parse $\mathsf{Dec.st}_i$ into (i, val_i).
2. If $i = 1$, parse $\mathsf{val}_1 = \mathsf{sk}_{g_1}$ and $\mathsf{CT}_1 = \mathsf{ct}_1$.
3. If $i > 1$,
   (a) Parse $\mathsf{val}_i = \mathsf{ct}'_i$ and $\mathsf{CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.
   (b) $\mathsf{sk}_{g_i} = \mathsf{OneCompFE.Dec}(\mathsf{sk}'_{h_i}, \mathsf{ct}'_i)$.
4. $(y_i, \mathsf{ct}'_{i+1}) = \mathsf{OneFSFE.Dec}(\mathsf{sk}_{g_i}, \mathsf{ct}_i)$.
5. $\mathsf{Dec.st}_{i+1} = (i+1, \mathsf{ct}'_{i+1})$.
6. Output $(y_i, \mathsf{Dec.st}_{i+1})$.

5.2 Correctness, Efficiency, and Security

Correctness, efficiency, and single-key, single-ciphertext, function-selective security follow from the corresponding theorems in [GKS23] as our construction is identical to theirs except for some minor, mostly notational changes.

5.3 Additional Properties

Post-One-sFE has some useful properties that we will need for the security proof of our adaptive scheme One-sFE.

1. **The MSK K can be split up into individual parts: $K_1, K_2, K_3, \ldots$.**
   We define individual parts $K_i = \mathsf{PRF.Eval}(K, i)$ as shown in the construction.

2. **Encryption at index i only requires K_i and K_{i+1}, rather than all of K.**
   To show this we define the following local encryption function:

   - $\mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), \mathsf{Enc.st}, i, x_i)$.
     (a) Compute $\mathsf{msk}_i, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, k_i, k'_i, \mathsf{msk}'_i$ from (K_i, K_{i+1}).
     (b) $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (x_i, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_y}))$
     (c) If $i = 1$, output $\mathsf{Post.CT}_1 = \mathsf{ct}_1$.
     (d) If $i > 1$
        i. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, \bot)$
        ii. $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \bot)$
        iii. Let $h_i = h_{c_i, c'_i}$ as defined in Figure 4.
        iv. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$
        v. Output $\mathsf{CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$

   It is then easy to see that local encryption acts identically to regular encryption.

   Lemma 5.2. *For all* $K, \mathsf{Enc.st}, i, x_i$ *and randomness* rand,

   $$\mathsf{Post\text{-}One\text{-}sFE.Enc}(K, \mathsf{Enc.st}, i, x_i; \mathsf{rand}) = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), \mathsf{Enc.st}, i, x_i; \mathsf{rand})$$

   where $K_i = \mathsf{PRF.Eval}(K, i)$ *and* $K_{i+1} = \mathsf{PRF.Eval}(K, i+1)$.

   Proof. The proof follows immediately from the definitions of Enc and EncLocal. □

3. **Security holds even if we begin the two challenge streams at different (and potentially non-$\perp$) starting states, as long as their output y values are the same and the starting states are given to the challenger.**[22]
   We will not prove this here, but it is implicitly shown in the proof of security for $\mathsf{One\text{-}sFE}$ (which uses $\mathsf{Post\text{-}One\text{-}sFE}$ as a building block).

4. **We can generate intermediate decryption states $\mathsf{Dec.st}_i$ without running through the entire encryption and decryption process. In particular, we just need to know the intermediate state st_i along with K_i and f.**
   To generate $\mathsf{Dec.st}_1$, we can define the following local key generation function:

   - $\mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$
     (a) Compute $\mathsf{msk}_1, k_1, p_1, r_{\mathsf{KeyGen}_1}$ from K_1.
     (b) $c_1 \leftarrow \mathsf{SKE.Enc}(k_1, \perp)$.
     (c) $\widetilde{\mathsf{st}}_1 = p_1 \oplus \mathsf{st}_1$.
     (d) Let $g_1 = g_{f,\widetilde{\mathsf{st}}_1,c_1}$ as defined in Figure 3.
     (e) $\mathsf{sk}_{g_1} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_1, g_1; r_{\mathsf{KeyGen}_1})$
     (f) Output $\mathsf{Dec.st}_1 = (1, \mathsf{sk}_{g_1})$.

   It is then easy to see that local keygen acts identically to regular keygen.

   Lemma 5.3. *For all K, f, st_1 and randomness rand,,*

   $$\mathsf{Post\text{-}One\text{-}sFE.KeyGen}(K, f, \mathsf{st}_1; \mathsf{rand}) = \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1; \mathsf{rand})$$

   where $K_1 = \mathsf{PRF.Eval}(K, 1)$.

   Proof. The proof follows immediately from the definitions of KeyGen and $\mathsf{KeyGenLocal}$. □

   For $i > 1$, we can generate intermediate decryption states with the following function:

   - $\mathsf{Post\text{-}One\text{-}sFE.DecStGen}(i, K_i, f, \mathsf{st}_i)$.
     (a) Compute $p_i, \mathsf{msk}'_i, r_{\mathsf{msk}_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i}$ from K_i.
     (b) $\widetilde{\mathsf{st}}_i = \mathsf{st}_i \oplus p_i$.
     (c) $\mathsf{ct}'_i = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_i, (f, \widetilde{\mathsf{st}}_i, r_{\mathsf{msk}_i}, r_{\mathsf{KeyGen}_i}, 0, 0^\lambda); r'_{\mathsf{Enc}_i})$.
     (d) Output $\mathsf{Dec.st}_i = (i, \mathsf{ct}'_i)$.

   Lemma 5.4. *For any streaming function $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ with starting state*[23] st_1 *and any stream $x = x_1 x_2 \ldots x_{t^*}$ of length $t^* > 0$ where each $x_i \in \{0,1\}^{\ell_\mathcal{X}}$, then*

   $$\Pr[D_0(\mathsf{MSK}) \neq D_1(\mathsf{MSK}) : \mathsf{MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})] \leq \mathsf{negl}(\lambda)$$

   where we define

[22] Syntatically, if we want to begin the challenge streams at different starting states, then in the security game, if the adversary receives an encryption of stream $x^{(b)}$, then the starting state for stream $x^{(b)}$ is given as additional input to KeyGen (as in our construction) when generating the function key.

[23] By definition, all functions $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ have starting state $\mathsf{st}_1 = \perp$. Here, we are using an expanded definition of $\mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ which allows f to have an arbitrary starting state $\mathsf{st}_1 \in \{0,1\}^{\ell_\mathcal{S}}$.

$D_0(\mathsf{MSK})$

(a) $\mathsf{Enc.st} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.EncSetup}(\mathsf{MSK})$.

(b) $\mathsf{Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{MSK}, f, \mathsf{st}_1)$.

(c) For $i \in [t^*]$,

  i. $\mathsf{CT}_i \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, \mathsf{Enc.st}, i, x_i)$.

  ii. $(y_i, \mathsf{Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Dec.st}_i, \mathsf{CT}_i)$

(d) Output $\mathsf{Dec.st}_{t^*+1}$.

$D_1(\mathsf{MSK})$

(a) Parse $\mathsf{MSK} = K$.

(b) $K_{t^*+1} = \mathsf{PRF.Eval}(K, t^* + 1)$.

(c) For $i \in [t^*]$,

  i. $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$.

(d) Output $\mathsf{Dec.st}_{t^*+1} = \mathsf{Post\text{-}One\text{-}sFE.DecStGen}(t^* + 1, K_{t^*+1}, f, \mathsf{st}_{t^*+1})$.

Proof. Let f be any streaming function $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ with starting state st_1, and let $x = x_1 x_2 \dots x_{t^*}$ be any stream of length $t^* > 0$ where each $x_i \in \{0,1\}^{\ell_\mathcal{X}}$.

Let us first analyze D_0. Consider

(a) $\mathsf{MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda)$.

(b) $\mathsf{Enc.st} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.EncSetup}(\mathsf{MSK})$.

(c) $\mathsf{Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{MSK}, f, \mathsf{st}_1)$.

(d) For $i \in [t^*]$,

  i. $\mathsf{CT}_i \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{MSK}, \mathsf{Enc.st}, i, x_i)$.

For notational convenience, in the following calculations, we will allow $\mathsf{Dec.st}_i$ for $i > 1$ to be ordered or reordered as either (i, ct'_i) or (ct'_i, i).

When $i = 1$, by correctness of $\mathsf{OneFSFE}$, except with negligible probability,

$$\begin{aligned}
\mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Dec.st}_1, \mathsf{CT}_1) &= \mathsf{Post\text{-}One\text{-}sFE.Dec}((1, \mathsf{sk}_{g_1}), \mathsf{ct}_1) \\
&= (\mathsf{OneFSFE.Dec}(\mathsf{sk}_{g_1}, \mathsf{ct}_1), 2) \\
&= (g_{f, \mathsf{st}_1 \oplus p_1, c_1}(x_1, p_1, p_2, r'_{\mathsf{msk}_2}, r'_{\mathsf{Enc}_2}, r_{\mathsf{msk}_2}, r_{\mathsf{KeyGen}_2}, 0, 0^\lambda, 0^{\ell_\mathcal{Y}}), 2) \\
&= (y_1, \mathsf{ct}'_2, 2) \\
&= (y_1, \mathsf{Dec.st}_2) \text{ for } \mathsf{Dec.st}_2 = (2, \mathsf{ct}'_2)
\end{aligned}$$

where $(y_1, \mathsf{st}_2) = f(x_1, \mathsf{st}_1)$ and $\mathsf{ct}'_2 = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_2, (f, \mathsf{st}_2 \oplus p_2, r_{\mathsf{msk}_2}, r_{\mathsf{KeyGen}_2}, 0, 0^\lambda); r'_{\mathsf{Enc}_2})$.

When $i = 2$, by correctness of $\mathsf{OneCompFE}$ and $\mathsf{OneFSFE}$, except with negligible probablity,

$$\begin{aligned}
&\mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Dec.st}_2, \mathsf{CT}_2) \\
&= \mathsf{Post\text{-}One\text{-}sFE.Dec}((2, \mathsf{ct}'_2), (\mathsf{ct}_2, \mathsf{sk}'_{h_2})) \\
&= (\mathsf{OneFSFE.Dec}(\mathsf{OneCompFE.Dec}(\mathsf{sk}'_{h_2}, \mathsf{ct}'_2), \mathsf{ct}_2), 3) \\
&= (\mathsf{OneFSFE.Dec}(h_{c_2,c'_2}(f, \mathsf{st}_2 \oplus p_2, r_{\mathsf{msk}_2}, r_{\mathsf{KeyGen}_2}, 0, 0^\lambda), \mathsf{ct}_2), 3) \\
&= (\mathsf{OneFSFE.Dec}(\mathsf{OneFSFE.KeyGen}(\mathsf{msk}_2, g_{f,\mathsf{st}_2 \oplus p_2, c_2}; r_{\mathsf{KeyGen}_2}), \mathsf{ct}_2), 3) \\
&= (\mathsf{OneFSFE.Dec}(\mathsf{sk}_{g_2}, \mathsf{ct}_2), 3) \\
&= (g_{f,\mathsf{st}_2 \oplus p_2, c_2}(x_2, p_2, p_3, r'_{\mathsf{msk}_3}, r'_{\mathsf{Enc}_3}, r_{\mathsf{msk}_3}, r_{\mathsf{KeyGen}_3}, 0, 0^\lambda, 0^{\ell_{\mathcal{Y}}}), 3) \\
&= (y_2, \mathsf{ct}'_3, 3) \\
&= (y_2, \mathsf{Dec.st}_3) \text{ for } \mathsf{Dec.st}_3 = (3, \mathsf{ct}'_3)
\end{aligned}$$

where $(y_2, \mathsf{st}_3) = f(x_2, \mathsf{st}_2)$ and $\mathsf{ct}'_3 = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_3, (f, \mathsf{st}_3 \oplus p_3, r_{\mathsf{msk}_3}, r_{\mathsf{KeyGen}_3}, 0, 0^\lambda); r'_{\mathsf{Enc}_3})$. Similarly, by induction, for $i > 2$, except with negligible probability,

$$\mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Dec.st}_i, \mathsf{CT}_i) = (y_i, \mathsf{Dec.st}_{i+1}) \text{ for } \mathsf{Dec.st}_{i+1} = (i+1, \mathsf{ct}'_{i+1})$$

where $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$ and $\mathsf{ct}'_{i+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (f, \mathsf{st}_{i+1} \oplus p_{i+1}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{i+1}})$.

Thus, except with negligible probability, $D_0(\mathsf{MSK})$ outputs $(t^* + 1, \mathsf{ct}'_{t^*+1})$ where $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (f, \mathsf{st}_{t^*+1} \oplus p_{t^*+1}, r_{\mathsf{msk}_{t^*+1}}, r_{\mathsf{KeyGen}_{t^*+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{t^*+1}})$. But this is exactly what $D_1(\mathsf{MSK})$ outputs! Therefore the two distributions have identical output except with negligible probability. □

6 Combining Pre-One-sFE and Post-One-sFE to Build One-sFE

We now construct our main building block: a single-key, single-ciphertext, adaptively secure, secret-key sFE scheme which we call One-sFE. We prove the following:

Theorem 6.1. *Assuming $i\mathcal{O}$ for* P/Poly *and injective* PRG*s, there exists a single-key, single-ciphertext, adaptively secure, secret-key sFE scheme for* P/Poly*.*

We build our scheme by combining the following two schemes:

- Pre-One-sFE: the single-key, single-ciphertext, selectively secure sFE scheme for $\mathcal{F}_\perp$ defined in Section 4.
- Post-One-sFE: the single-key, single-ciphertext, function-selectively secure sFE scheme for P/Poly defined in Section 5.

This is not a general transformation as our security proof is very non-black-box and relies on properties of both schemes. Please refer to the technical overview (Section 2) for a high level overview of our construction.

By Theorem 4.1, we can build Pre-One-sFE from $i\mathcal{O}$ and injective PRGs. By Theorem 5.1, we can build Post-One-sFE from a strongly-compact, selectively secure, secret-key FE scheme for P/Poly which in turn can be built from $i\mathcal{O}$ for P/Poly and OWFs [Wat15]. As OWFs are implied by PRGs, then our final scheme One-sFE can also be built from $i\mathcal{O}$ for P/Poly and injective PRGs.

6.1 Parameters

On security parameter λ, function size $\ell_\mathcal{F}$, state size $\ell_\mathcal{S}$, input size $\ell_\mathcal{X}$, and output size $\ell_\mathcal{Y}$, we will instantiate our primitives with the following parameters:

- Post-One-sFE: We instantiate Post-One-sFE with function size $\ell_\mathcal{F}$, state size $\ell_\mathcal{S}$, input size $\ell_\mathcal{X}$, and output size $\ell_\mathcal{Y}$.
- Pre-One-sFE: We instantiate Pre-One-sFE with function size $L_\mathcal{F}$, state size $L_\mathcal{S}$, input size $L_\mathcal{X}$, and output size $L_\mathcal{Y}$ where
  - $L_\mathcal{F}$ is the size of Post-One-sFE's decryption function under the parameters listed above;
  - $L_\mathcal{S}$ is the size of Post-One-sFE's decryption states under the parameters listed above;
  - $L_\mathcal{X}$ is the size of Post-One-sFE's ciphertexts under the parameters listed above;
  - $L_\mathcal{Y} = \ell_\mathcal{Y}$.

6.2 Construction

We now construct One-sFE from Pre-One-sFE and Post-One-sFE. Here, we omit the encryption state from the encryption algorithms of both Pre-One-sFE and Post-One-sFE as their encryption states are always $\perp$ and are unused.

We also use the following notational variations when defining Pre-One-sFE and Post-One-sFE as was already made explicit in the constructions given in Section 4 and Section 5: We define the KeyGen algorithms of Pre-One-sFE and Post-One-sFE to additionally take the starting state as input. We also write Post-One-sFE so that its decryption algorithm is a streaming function in the standard format (i.e. takes only two inputs: a decryption state and a ciphertext). Note that in this case, Post-One-sFE.KeyGen simply outputs the first decryption state.

- $\mathsf{One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$:
  1. $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
  2. $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
  3. Output $\mathsf{MSK} = (\mathsf{Pre.MSK}, \mathsf{Post.MSK})$.
- $\mathsf{One\text{-}sFE.EncSetup}(\mathsf{MSK})$: Output $\mathsf{Enc.st} = \bot$.
- $\mathsf{One\text{-}sFE.Enc}(\mathsf{MSK}, \mathsf{Enc.st}, i, x_i)$:
  1. Parse $\mathsf{MSK} = (\mathsf{Pre.MSK}, \mathsf{Post.MSK})$.
  2. $\mathsf{Post.CT}_i \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i)$.
  3. $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.
  4. Output $\mathsf{CT}_i = \mathsf{Pre.CT}_i$.
- $\mathsf{One\text{-}sFE.KeyGen}(\mathsf{MSK}, f)$:
  1. Parse $\mathsf{MSK} = (\mathsf{Pre.MSK}, \mathsf{Post.MSK})$.
  2. $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$ where $\mathsf{st}_1 = \bot$.
  3. $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGen}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1)$.
  4. Output $\mathsf{SK}_f = \mathsf{Pre.SK}_f$.
- $\mathsf{One\text{-}sFE.Dec}(\mathsf{SK}_f, \mathsf{Dec.st}_i, i, \mathsf{CT}_i)$:
  1. Parse $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ and $\mathsf{CT}_i = \mathsf{Pre.CT}_i$.
  2. If $i > 0$, parse $\mathsf{Dec.st}_i = \mathsf{Pre.Dec.st}_i$. If $i = 1$, $\mathsf{Pre.Dec.st}_1 = \bot$.
  3. $(y_i, \mathsf{Pre.Dec.st}_{i+1}) = \mathsf{Pre\text{-}One\text{-}sFE.Dec}(\mathsf{Pre.SK}_f, \mathsf{Pre.Dec.st}_i, i, \mathsf{Pre.CT}_i)$.
  4. Output $(y_i, \mathsf{Dec.st}_{i+1} = \mathsf{Pre.Dec.st}_{i+1})$.

6.3 Correctness and Efficiency

Efficiency. Efficiency follows directly from the efficiency requirements of Pre-One-sFE and Post-One-sFE.

By the streaming efficiency of Post-One-sFE, the size and runtime of all algorithms of Post-One-sFE on security parameter λ, function size $\ell_\mathcal{F}$, state size $\ell_\mathcal{S}$, input size $\ell_\mathcal{X}$, and output size $\ell_\mathcal{Y}$ are $\mathsf{poly}(\lambda, \ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y})$.

By the streaming efficiency of Pre-One-sFE, the size and runtime of all algorithms of Pre-One-sFE on security parameter λ, function size $L_\mathcal{F}$, state size $L_\mathcal{S}$, input size $L_\mathcal{X}$, and output size $L_\mathcal{Y}$ are $\mathsf{poly}(\lambda, L_\mathcal{F}, L_\mathcal{S}, L_\mathcal{X}, L_\mathcal{Y})$ which are $\mathsf{poly}(\lambda, \ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y})$ by the streaming efficiency of Post-One-sFE.

Thus, the size and runtime of all algorithms of One-sFE on security parameter λ, function size $\ell_\mathcal{F}$, state size $\ell_\mathcal{S}$, input size $\ell_\mathcal{X}$, and output size $\ell_\mathcal{Y}$ are $\mathsf{poly}(\lambda, \ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y})$

Correctness Intuition. One-sFE composes Post-One-sFE and Pre-One-sFE in the following manner:

- Post-One-sFE encrypts the stream $x_1, x_2, \ldots, x_n$ and creates a function key for f.
- Pre-One-sFE encrypts the stream $\mathsf{Post.CT}_1, \mathsf{Post.CT}_2, \ldots, \mathsf{Post.CT}_n$ of Post-One-sFE ciphertexts, and creates a function key for Post-One-sFE.Dec.

The decryption algorithm for One-sFE simply runs Pre-One-sFE's decryption algorithm. This means we will compute the output of running Post-One-sFE.Dec on the stream $\mathsf{Post.CT}_1, \mathsf{Post.CT}_2, \ldots, \mathsf{Post.CT}_n$. But this means we will run Post-One-sFE's decryption algorithm and thus will compute the output of running f on our original stream $x_1, x_2, \ldots, x_n$. Thus, we will get the correct output values we desire.

Correctness. More formally, let p be any polynomial and consider any λ and any $\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}} \leq p(\lambda)$. Let SK_f be a function key for function $f \in \mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$, and let $\{\mathsf{CT}_i\}_{i\in[n]}$ be a ciphertext for x where $x = x_1 \ldots x_n$ for some $n \in [2^\lambda]$ and where each $x_i \in \{0,1\}^{\ell_{\mathcal{X}}}$.

For all $i \in [n]$, by correctness of Pre-One-sFE, except with negligible probability,

$$\begin{aligned}\mathsf{One\text{-}sFE.Dec}(\mathsf{SK}_f, \mathsf{Dec.st}_i, i, \mathsf{CT}_i) &= \mathsf{Pre\text{-}One\text{-}sFE.Dec}(\mathsf{Pre.SK}_f, \mathsf{Pre.Dec.st}_i, i, \mathsf{Pre.CT}_i)\\ &= (y_i, \mathsf{Pre.Dec.st}_{i+1})\\ &= (y_i, \mathsf{Dec.st}_{i+1})\end{aligned}$$

where y_i is the i^{th} output in the computation of the streaming function Post-One-sFE.Dec on the stream $\mathsf{Post.CT}_1, \mathsf{Post.CT}_2, \ldots, \mathsf{Post.CT}_n$, that is, we compute each y_i by computing $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$. For all $i \in [n]$, by correctness of Post-One-sFE, except with negligible probability,

$$\mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i) = (y_i, \mathsf{Post.Dec.st}_{i+1})$$

where $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$. Thus, we correctly output $y = y_1 \ldots y_n$ where $(y_i, \mathsf{st}_{i+1}) = f(x_i, \mathsf{st}_i)$.

6.4 Security

We now prove that One-sFE is single-key, single-ciphertext, adaptively secure for the function class P/Poly.

In this proof, we will use an alternate, but equivalent, definition of single-key, single-ciphertext, adaptive security where instead of needing to distinguish between an encryption of stream $x^{(0)}$ and an encryption of stream $x^{(1)}$, the adversary will receive an encryption of stream $x^{(b)}$ for a random bit b and will win if they correctly guess b.

Definition 6.2 (Single-Key, Single-Ciphertext, Adaptive Security for Secret-Key sFE, Equivalent Definition)**.** *A secret-key streaming FE scheme* sFE *for* $\mathcal{F}_\perp$ *is single-key, single-ciphertext,adaptively secure if there exists a negligible function* μ *such that for all* $\lambda \in \mathbb{N}$ *and all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{SKExptGuess}_{\mathcal{A}}^{\mathsf{sFE\text{-}1\text{-}Key\text{-}1\text{-}CT\text{-}Ad}}(1^\lambda) = 1]\right| \leq \frac{1}{2} + \mu(\lambda)$$

where for $\lambda \in \mathbb{N}$*, we define*

$\mathsf{SKExptGuess}_{\mathcal{A}}^{\mathsf{sFE\text{-}1\text{-}Key\text{-}1\text{-}CT\text{-}Ad}}(1^\lambda)$

1. ***Parameters**: $\mathcal{A}$ takes as input 1^λ, and outputs a function size $1^{\ell_{\mathcal{F}}}$, a state size $1^{\ell_{\mathcal{S}}}$, an input size $1^{\ell_{\mathcal{X}}}$, and an output size $1^{\ell_{\mathcal{Y}}}$.*
2. ***Setup**: Compute $\mathsf{msk} \leftarrow \mathsf{sFE.Setup}(1^\lambda, 1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}})$.*
3. ***Challenge Bit**: Sample $b \leftarrow \{0,1\}$.*
4. *The following can be repeated any polynomial number of times:*

(a) ***Challenge Message Query Phase 1:***

i. *If this is the first challenge message query, sample* $\mathsf{Enc.st} \leftarrow \mathsf{sFE.EncSetup}(\mathsf{msk})$ *and initialize the index* $i = 1$*. Else, increment the index* i *by* 1*.*

ii. $\mathcal{A}$ *outputs a challenge message pair* $(x_i^{(0)}, x_i^{(1)})$ *where* $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_{\mathcal{X}}}$*.*

iii. $\mathsf{ct}_i \leftarrow \mathsf{sFE.Enc}(\mathsf{msk}, \mathsf{Enc.st}, i, x_i^{(b)})$*.*

iv. *Send* ct_i *to* $\mathcal{A}$*.*

5. ***Function Query:***

(a) $\mathcal{A}$ *outputs a streaming function query* $f \in \mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$*.*

(b) $\mathsf{sk}_f \leftarrow \mathsf{sFE.KeyGen}(\mathsf{msk}, f)$*.*

(c) *Send* sk_f *to* $\mathcal{A}$*.*

6. *The following can be repeated any polynomial number of times:*

(a) ***Challenge Message Query Phase 2:***

i. *If this is the first challenge message query (in either Phase 1 or Phase 2), sample* $\mathsf{Enc.st} \leftarrow \mathsf{sFE.EncSetup}(\mathsf{msk})$ *and initialize the index* $i = 1$*. Else, increment the index* i *by* 1*.*

ii. $\mathcal{A}$ *outputs a challenge message pair* $(x_i^{(0)}, x_i^{(1)})$ *where* $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_{\mathcal{X}}}$*.*

iii. $\mathsf{ct}_i \leftarrow \mathsf{sFE.Enc}(\mathsf{msk}, \mathsf{Enc.st}, i, x_i^{(b)})$*.*

iv. *Send* ct_i *to* $\mathcal{A}$*.*

7. ***Experiment Outcome****:* $\mathcal{A}$ *outputs a bit* b'*. The output of the experiment is set to* 1 *if* $b = b'$*, and* 0 *otherwise.*

Additionally, when running the experiment, we immediately halt and output 0 *if the adversary ever aborts or if it at any point some function query* f *submitted by the adversary yields different outputs on the challenge message streams submitted so far (i.e. if* $f(x^{(0)}) \neq f(x^{(1)})$ *for some function query* f *submitted by the adversary where* $\{(x_i^{(0)}, x_i^{(1)})\}_{i \in [t]}$ *are the message queries submitted so far,* $x^{(0)} = x_1^{(0)} \dots x_t^{(0)}$*, and* $x^{(1)} = x_1^{(1)} \dots x_t^{(1)}$*).*

Using standard techniques, it is easy to show that this is equivalent to the regular definition of single-key, single-ciphertext, adaptive security.

6.4.1 Proof Overview

We strongly recommend that the reader first reads the technical overview for intuition on the proof. This section serves mainly as a structural overview of the proof which details the major hybrids.

The proof can be split into two main parts:

1. In Part 1, we use the security of Pre-One-sFE to deal with all the message queries given before the function query.
2. In Part 2, we use the security of Post-One-sFE to deal with all the message queries given after the function query.

Note that we use the security of these component schemes in a very non-black-box manner.

First, however, we move to a hybrid that allows us to know the stream length at the beginning of the hybrid. In our formal proof, we include the following two hybrids under Part 1.

Guessing the stream length. Define t^* to be the length of the challenge stream preceding the function query and define n to be the total length of the challenge stream. In many hybrids, it is useful to know these values at the beginning of the hybrid. However, (t^*, n) are adaptively (and implicitly) chosen by the adversary. Thus, to obtain these values earlier, we will use a guess and check method.

1. $\mathbf{Hybrid}_0^{\mathcal{A}}$: This is the real world security game where we encrypt stream $x^{(b)}$.
2. $\mathbf{Hybrid}_1^{\mathcal{A}}$: We guess the length of the challenge stream at the beginning of the experiment and output $\perp$ if our guess turns out to be incorrect.

Since we only guess the correct values of (t^*, n) with an inverse polynomial probability, the actual hybrid we will move to is $\mathsf{Wrap}(\mathbf{Hybrid}_1^{\mathcal{A}})$ which repeats the entire experiment of $\mathbf{Hybrid}_1^{\mathcal{A}}$ with fresh guesses whenever our guess turns out to be incorrect.

Part 1: Using the Security of Pre-One-sFE.
Our goal is to iteratively replace the encryption of each element of stream $x^{(b)}$ with a value which is independent of the challenge bit b. For message queries given before the function query, we can do this using the security of Pre-One-sFE. In particular, for $i < t^*$, we will replace each encryption of $x_i^{(b)}$ with an encryption of $x_i^{(0)}$. We will also replace the encryption of $x_{t^*}^{(b)}$ with an encryption of $\perp$.

Our Pre-One-sFE scheme encrypts each stream value x_i by first encrypting x_i using a secret key encryption scheme (SKE) and then signing the resulting ciphertext with a splittable signature scheme. The function key for a streaming function f consists of a signed encryption of the starting state along with an obfuscated program that at each step i, verifies and decrypts the incoming stream and state ciphertexts, computes the streaming function on these values, and encrypts and signs the new state.[24]

At a high level, for each index t, the idea is to replace the t^{th} input and state ciphertexts of stream $x^{(b)}$ for those of stream $x^{(0)}$ using the security of SKE. This works as long as the keys (and randomness) used to encrypt these ciphertexts are hidden. Unfortunately, the function key contains an obfuscated program which has these SKE keys hardwired into it since the program needs the keys to decrypt the incoming messages and to encrypt outgoing states. However, by using different SKE schemes for each index and by using standard punctured programming techniques, we can ensure that the SKE keys for the t^{th} ciphertexts are only used by the program at index t (to decrypt the input) and at index $t-1$ (to encrypt the new state). Then, we can modify the program so that it skips the computation phase (involving the aforementioned encryption and decryption) at steps $t-1$ and t and instead uses pre-computed hardwired values. This removes the t^{th} SKE keys from the program entirely which allows us to swap the t^{th} ciphertexts.

We describe the main hybrids involved in this part of the proof.

1. $\mathbf{Hybrid}_2^{\mathcal{A}}$: We unwrap the construction of Pre-One-sFE.
2. We proceed through a series of sub-hybrids for $t \in [t^* - 1]$, in which we iteratively replace the t^{th} encryption of stream $x^{(b)}$ with the corresponding encryption of stream $x^{(0)}$. For each index t, we do the following:

[24]The construction also involves a cryptographic iterator which we omit from this brief description for simplicity.

(a) We assume that step $t-1$ is hardwired into the program.

Note that we can hardwire step 0 *for free (i.e. without changing the behavior of the obfuscated program) since stream indexing starts at step* 1*.*

i. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0}$: For $i < t$, we encrypt stream $x^{(0)}$ instead of $x^{(b)}$. Additionally, we hardwire the output values at step $t-1$ into the program.

(b) We hardwire step t into the program.

This is a very complex and involved step. Please refer to the technical overview for further intuition.

i. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1}$: We introduce an alternative B-type signature scheme for verifying and signing state ciphertexts within our obfuscated program. In particular, we modify our program so that if the incoming state ciphertext does not verify using the (original) A-type signature scheme and $i \leq t-1$, rather than immediately rejecting, we instead try to verify it with the B-type signature scheme. If the incoming state ciphertext verifies under the B-type scheme, we proceed with our computation as usual except that we sign the outgoing state ciphertext with a B-type signature.

   We add in the B-type signature scheme one index at a time starting from index $t-1$ and going backwards to index 1. Hybrids $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,0}$ to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,6}$ detail how to add in this signature scheme at index j.

ii. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2}$: If our outgoing message at step $t-1$ is the correct pre-computed hardwired value corresponding to stream $x^{(0)}$, then we sign the message with a B-type signature. Additionally, we no longer verify using B type signatures.

   To reach this hybrid, we must trace the computation up through the program.We start by enforcing that at step 0 (which is unused), we will sign the outgoing message with a B-type signature if and only if the outgoing message is the correct pre-computed value corresponding to stream $x^{(0)}$. Then, for each index j from 1 to $t-1$, we remove the B-type verification at step j and move the enforcement up a step. (i.e. We instead enforce that at step j (rather than $j-1$), only the correct outgoing message corresponding to stream $x^{(0)}$ will be signed with the B-type scheme.)

   At a high level, for each index j this involves

   1. Hybrids $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,0}$ to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,6}$: Splitting the j^{th} A and B type signature schemes on the chosen values. Using further properties of splittable signatures, this lets us moves the enforcement from the outgoing message at step $j-1$ to the incoming message at step j.
   2. Hybrids $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,7}$ to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,10}$: Merging the $j-1^{th}$ A and B type signature schemes. This removes the B-type verification at step $j-1$.
   3. Hybrids $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,11}$ to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,19}$: Using the iterator to enforce the j^{th} step of computation. This moves the enforcement from the j^{th} incoming message to the j^{th} outgoing message.

iii. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,3}$: At index t, rather than computing $(y_t, \mathsf{ct}_{\mathsf{st},t+1})$ from the input values, we set these values to pre-computed and hardwired outputs corresponding to stream $x^{(b)}$. Hybrids $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,1}$ to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,9}$ detail how to hardwire these values.

iv. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4}$: We remove all references to B type signatures. To do this, we essentially perform the steps taken to bring us from $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0}$ to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,3}$ in reverse order.

(c) Now, that both steps $t-1$ and t are hardwired into the program, we can swap the t^{th} encryption of stream $x^{(b)}$ with an encryption of stream $x^{(0)}$ using the security of SKE.

i. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,5}$: At step t, we now encrypt stream $x^{(0)}$ instead of stream $x^{(b)}$. Hybrids $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,1}$ to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,5}$ detail how to swap the t^{th} ciphertext.

(d) We clean up by un-hardwiring step $t-1$ from the program. This returns us to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t+1,0}$. This is done in a similar fashion as when we originally hardwired the step.

i. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,6}$: At steps $i \leq t-2$, we introduce an alternative B-type signature scheme for verifying and signing state ciphertexts within our obfuscated program. This step is done in a similar fashion as in $\mathbf{Hybrid}^{\mathcal{A}}_{3,t-1,1}$.

ii. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7}$: If our outgoing message at step $t-2$ is the correct pre-computed hardwired value corresponding to stream $x^{(0)}$, then we sign the message with a B-type signature. Additionally, we no longer verify using B type signatures. This step is done in a similar fashion as in $\mathbf{Hybrid}^{\mathcal{A}}_{3,t-1,2}$.

iii. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,8}$: At index $t-1$, rather than setting $(y_t, \mathsf{ct}_{\mathsf{st},t+1})$ to pre-computed and hardwired outputs corresponding to stream $x^{(0)}$, we compute them from the input values. This is essentially the inverse of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t-1,3}$.

iv. $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,9}$: We remove all references to B type signatures. This step is done in a similar fashion as in $\mathbf{Hybrid}^{\mathcal{A}}_{3,t-1,4}$.

3. For the ciphertext at index t^*, rather than replacing it with an encryption of stream $x^{(0)}$, we replace it with an encryption of $\perp$.

   This breaks the chain of dependencies between the ciphertexts of message queries given before the function query and the ciphertexts of message queries given after the function query which is needed in Part 2 of the security proof. In particular, the ciphertexts of our adaptive scheme consist of an outer $\mathsf{Pre\text{-}One\text{-}sFE}$ encryption of an inner $\mathsf{Post\text{-}One\text{-}sFE}$ ciphertext of the stream value. Thus, this step replaces the inner t^{*th} $\mathsf{Post\text{-}One\text{-}sFE}$ ciphertext with $\perp$, removing it from the scheme entirely. This is required since the t^{*th} $\mathsf{Post\text{-}One\text{-}sFE}$ ciphertext contains secret values used in encrypting the $t^* + 1^{th}$ $\mathsf{Post\text{-}One\text{-}sFE}$ ciphertext.

   (a) $\mathbf{Hybrid}^{\mathcal{A}}_{4}$: This is identical to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t^*,4}$. In other words, for $i < t^*$, we encrypt stream $x^{(0)}$ instead of stream $x^{(b)}$. We also have steps $i = t^* - 1$ and $i = t^*$ hardwired into the program.

   (b) $\mathbf{Hybrid}^{\mathcal{A}}_{5}$: We replace the t^{*th} encryption of stream $x^{(b)}$ with an encryption of $\perp$. Hybrids $\mathbf{Hybrid}^{\mathcal{A}}_{4,1}$ to $\mathbf{Hybrid}^{\mathcal{A}}_{4,5}$ detail how to swap this ciphertext with $\perp$.

4. $\mathbf{Hybrid}^{\mathcal{A}}_{6}$: We re-wrap the parts of our construction corresponding to $\mathsf{Pre\text{-}One\text{-}sFE}$. Note that our hybrid no longer acts identically to the original construction of $\mathsf{Pre\text{-}One\text{-}sFE}$ since we have to maintain hardwired values in our hybrid in order to deal with the encryption of $\perp$ at step t^*. However, we have removed the dependency on the challenge bit b from the ciphertexts of message queries given before the function query.

Part 2: Using the Security of Post-One-sFE.
Recall that our goal is to iteratively replace the encryption of each element of stream $x^{(b)}$ with a value which is independent of the challenge bit b. Using our Pre-One-sFE scheme, we have already done this for all message queries given before the function query. For the remaining message queries, consisting of all message queries given after the function query, we will use the security of Post-One-sFE. In particular, we will replace these values with values that can be simulated given just the output of the streaming function on the challenge streams.

As this proof is essentially identical to the proof in [GKS23], we refer the reader to the proof overview given in Section 5.4 of [GKS23] and simply list which of their hybrids corresponds to each of our hybrids. Note that the last hybrid of Part 1 is $\mathbf{Hybrid}_6^{\mathcal{A}}$ which corresponds to the starting hybrid $\mathbf{Hybrid}_1^{\mathcal{A}}$ in [GKS23].

1. $\mathbf{Hybrid}_7^{\mathcal{A}}$: We unwrap the definition of Post-One-sFE and exchange some pseudorandom values for truly random values. This corresponds to $\mathbf{Hybrid}_2^{\mathcal{A}}$ in [GKS23].
2. $\mathbf{Hybrid}_8^{\mathcal{A}}$: This corresponds to $\mathbf{Hybrid}_3^{\mathcal{A}}$ in [GKS23].
3. $\mathbf{Hybrid}_9^{\mathcal{A}}$: This corresponds to $\mathbf{Hybrid}_4^{\mathcal{A}}$ in [GKS23].
4. $\mathbf{Hybrid}_{10}^{\mathcal{A}}$: This corresponds to $\mathbf{Hybrid}_5^{\mathcal{A}}$ in [GKS23].
5. We go through the following hybrids for $t \in [t^* + 1, n]$. Note that relative to the proof in [GKS23], we have modified the notational numbering of the following two hybrids. However, we have not changed the order in which the hybrids occur in the proof.
   - $\mathbf{Hybrid}_{11,t,0}^{\mathcal{A}}$: This corresponds to hybrid $\mathbf{Hybrid}_{6,k-1,2}^{\mathcal{A}}$ in [GKS23] for $k = t - t^*$.
   - $\mathbf{Hybrid}_{11,t,1}^{\mathcal{A}}$: This corresponds to hybrid $\mathbf{Hybrid}_{6,k,1}^{\mathcal{A}}$ in [GKS23] for $k = t - t^*$.
6. $\mathbf{Hybrid}_{12}^{\mathcal{A}}$: This corresponds to $\mathbf{Hybrid}_7^{\mathcal{A}}$ in [GKS23].
7. $\mathbf{Hybrid}_{13}^{\mathcal{A}}$: This corresponds to $\mathbf{Hybrid}_8^{\mathcal{A}}$ in [GKS23], which is their final hybrid.

After concluding Part 2, our final hybrid is independent of the challenge bit b. This implies the security of our scheme.

6.4.2 Formal Proof

As the proof is rather long and involved, we defer it to Appendix C (page 102).

7 Bootstrapping to an Adaptively Secure, Public-Key Streaming FE Scheme

To construct our adaptively secure, public-key sFE scheme, we use the following theorem which is implied by the work of [GKS23].

Theorem 7.1. *Assuming*

1. *a selectively secure, public-key FE scheme for* $\mathsf{P/Poly}$
2. *a single-key, single-ciphertext, adaptively secure, secret-key, sFE scheme for* $\mathsf{P/Poly}$

there exists an adaptively secure, public-key sFE scheme for $\mathsf{P/Poly}$.

Since [GKS23] only proves this theorem for function-selectively secure sFE schemes, for completeness, we provide a proof of Theorem 7.1 here.[25] Apart from the fact that $\mathsf{One\text{-}sFE}$ now represents a (single-key, single-ciphertext) adaptively secure scheme, rather than a (single-key, single-ciphertext) function-selectively secure scheme, our construction is identical to the construction in [GKS23]. Our proof of security is also essentially the same. We have merely reordered some of the steps in the hybrids so that they align with the adaptive security game. For a high level overview of the construction, we refer the reader to the Technical Overview of [GKS23].

To prove Theorem 7.1, we build an sFE scheme from the following tools. As we show below, apart from $\mathsf{One\text{-}sFE}$, all of the following tools can be instantiated using a selectively secure, public-key FE scheme for $\mathsf{P/Poly}$.

Tools.

- $\mathsf{One\text{-}sFE} = (\mathsf{One\text{-}sFE.Setup}, \mathsf{One\text{-}sFE.Enc}, \mathsf{One\text{-}sFE.KeyGen}, \mathsf{One\text{-}sFE.Dec})$: A single-key, single-ciphertext, adaptively secure, secret-key sFE scheme for $\mathsf{P/Poly}$.
- $\mathsf{PRF} = (\mathsf{PRF.Setup}, \mathsf{PRF.Eval})$: A secure pseudorandom function family.
- $\mathsf{PRF2} = (\mathsf{PRF2.Setup}, \mathsf{PRF2.Eval})$: A secure pseudorandom function family.
- $\mathsf{SKE} = (\mathsf{SKE.Setup}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$: A secure symmetric key encryption scheme with pseudorandom ciphertexts.
- $\mathsf{FPFE} = (\mathsf{FPFE.Setup}, \mathsf{FPFE.Enc}, \mathsf{FPFE.KeyGen}, \mathsf{FPFE.Dec})$: A function-private-selective-secure, secret-key FE scheme for $\mathsf{P/Poly}$
- $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.Enc}, \mathsf{FE.KeyGen}, \mathsf{FE.Dec})$: A selectively secure, public-key FE scheme for $\mathsf{P/Poly}$.

Instantiation of the Tools. Let FE' be a selectively secure, public-key FE scheme for $\mathsf{P/Poly}$.

- We can build $\mathsf{PRF}, \mathsf{PRF2}, \mathsf{SKE}$ from any one-way-function using standard cryptographic techniques (e.g. [Gol01, Gol09]). As FE' implies one-way-functions, then we can build these from FE'.
- FE' already satisfies the security requirements needed for FE.

[25] [GKS23] indeed remark, but do not formally prove, that their theorem should also hold for adaptively secure schemes.

- FE' immediately implies a selectively secure, *secret-key* FE scheme SKFE' for $\mathsf{P/Poly}$. We can then build our function-private-selective-secure, secret-key FE scheme FPFE for $\mathsf{P/Poly}$ by using the function-privacy transformation of [BS18] on SKFE'.

7.1 Parameters

The parameters are identical to those in [GKS23] since our construction is identical to [GKS23] except for the fact that $\mathsf{One\text{-}sFE}$ now represents a (single-key, single-ciphertext) adaptively secure scheme, rather than a (single-key, single-ciphertext) function-selectively secure scheme.

On security parameter λ, function size $\ell_{\mathcal{F}}$, state size $\ell_{\mathcal{S}}$, input size $\ell_{\mathcal{X}}$, and output size $\ell_{\mathcal{Y}}$, we will instantiate our primitives with the following parameters:

- $\mathsf{One\text{-}sFE}$: We instantiate $\mathsf{One\text{-}sFE}$ with function size $\ell_{\mathcal{F}}$, state size $\ell_{\mathcal{S}}$, input size $\ell_{\mathcal{X}}$, and output size $\ell_{\mathcal{Y}}$. This means that we will use the following setup algorithm: $\mathsf{One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}})$.
- PRF: We instantiate PRF with input size λ and output size 5λ. This means that we will use the following setup algorithm: $\mathsf{PRF.Setup}(1^\lambda, 1^\lambda, 1^{5\lambda})$.
- $\mathsf{PRF2}$: We instantiate $\mathsf{PRF2}$ with input size λ and output size λ. This means that we will use the following setup algorithm: $\mathsf{PRF2.Setup}(1^\lambda, 1^\lambda, 1^\lambda)$.
- FPFE: We instantiate FPFE with
  - **Input Size:** $\ell_{\mathsf{FPFE}.m_\lambda} = \ell_{\mathsf{One\text{-}sFE.msk}_\lambda} + \ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda} + \ell_{\mathsf{PRF2}.k_\lambda} + 2$ where $\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}$ is the size of master secret keys of $\mathsf{One\text{-}sFE}$, $\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}$ is the size of encryption states of $\mathsf{One\text{-}sFE}$, and $\ell_{\mathsf{PRF2}.k_\lambda}$ is the size of keys of $\mathsf{PRF2}$.
  - **Function Size:** ℓ_{H_λ} where ℓ_{H_λ} is the maximum of the size of H_{i,x_i,t_i} defined in Figure 5 and the size of H^*_{i,x_i,x'_i,t_i,v_i} defined in Figure 7 for any
    - $i, t_i \in \{0,1\}^\lambda$
    - $x_i, x'_i \in \{0,1\}^{\ell_{\mathcal{X}}}$
    - v_i of size $\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ where $\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ is the size of ciphertexts of $\mathsf{One\text{-}sFE}$

    Observe that the function size depends only on $\lambda, \ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}$ and the sizes of $\mathsf{PRF2}$, and $\mathsf{One\text{-}sFE}$.
  - **Output Size:** $\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ where $\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ is the size of ciphertexts of $\mathsf{One\text{-}sFE}$

  This means that we will use the following setup algorithm: $\mathsf{FPFE.Setup}(1^\lambda, 1^{\ell_{H_\lambda}}, 1^{\ell_{\mathsf{FPFE}.m_\lambda}}, 1^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}})$.
- SKE: We instantiate SKE with messages of length $\ell_{\mathsf{SKE}.m_\lambda} = \ell_{\mathsf{One\text{-}sFE.sk}_\lambda} + \ell_{\mathsf{FPFE.ct}_\lambda}$ where $\ell_{\mathsf{One\text{-}sFE.sk}_\lambda}$ is the size of function keys of $\mathsf{One\text{-}sFE}$ and $\ell_{\mathsf{FPFE.ct}_\lambda}$ is the size of ciphertexts of FPFE. This means that we will use the following setup algorithm: $\mathsf{SKE.Setup}(1^\lambda, 1^{\ell_{\mathsf{SKE}.m_\lambda}})$.
- FE: We instantiate FE with
  - **Input Size:** $\ell_{\mathsf{FE}.m_\lambda} = \ell_{\mathsf{FPFE.msk}_\lambda} + \ell_{\mathsf{PRF}.k_\lambda} + 1 + \ell_{\mathsf{SKE}.k_\lambda}$ where $\ell_{\mathsf{FPFE.msk}_\lambda}$ is the size of master secret keys of FPFE, $\ell_{\mathsf{PRF}.k_\lambda}$ is the size of keys of PRF, and $\ell_{\mathsf{SKE}.k_\lambda}$ is the size of keys of SKE.
  - **Function Size:** ℓ_{G_λ} where ℓ_{G_λ} is the maximum size of $G_{f,s,c}$ defined in Figure 6 for any
    - $f \in \mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$
    - $s \in \{0,1\}^\lambda$

* c of size $\ell_{\mathsf{SKE.ct}_\lambda}$ where $\ell_{\mathsf{SKE.ct}_\lambda}$ is the size of ciphertexts of SKE

Note that the function size depends only on $\lambda, \ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}$ and the sizes of PRF, $\mathsf{PRF2}$, $\mathsf{One\text{-}sFE}$, FPFE, and SKE.

– **Output Size:** $\ell_{\mathsf{FE}.out_\lambda} = \ell_{\mathsf{One\text{-}sFE.sk}_\lambda} + \ell_{\mathsf{FPFE}.ct_\lambda}$ where $\ell_{\mathsf{One\text{-}sFE.sk}_\lambda}$ is the size of secret keys of $\mathsf{One\text{-}sFE}$ and $\ell_{\mathsf{FPFE.ct}_\lambda}$ is the size of ciphertexts of FPFE

This means that we will use the following setup algorithm: $\mathsf{FE.Setup}(1^\lambda, 1^{\ell_{G_\lambda}}, 1^{\ell_{\mathsf{FE}.m_\lambda}}, 1^{\ell_{\mathsf{FE}.out_\lambda}})$.

Notation. For notational convenience, when the parameters are understood, we will often omit the security, input size, output size, message size, function size, or state size parameters from each of the algorithms listed above.

Remark 7.2. We assume without loss of generality that for security parameter λ, all algorithms only require randomness of length λ. If the original algorithm required additional randomness, we can replace it with a new algorithm that first expands the λ bits of randomness using a PRG of appropriate stretch and then runs the original algorithm. Note that this replacement does not affect the security of the above schemes (as long as $\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}$ are polynomial in λ).

7.2 Construction

We now construct our streaming FE scheme sFE. Our construction is identical to [GKS23] except for the fact that $\mathsf{One\text{-}sFE}$ now represents a (single-key, single-ciphertext) adaptively secure scheme, rather than a (single-key, single-ciphertext) function-selectively secure scheme. Recall that for notational convenience, we may omit the security, input size, output size, message size, function size, or state size parameters from our algorithms. For information on these parameters, please see the parameter section above.

- $\mathsf{sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$:
  1. $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
  2. Output $(\mathsf{MPK} = \mathsf{FE.mpk}, \mathsf{MSK} = \mathsf{FE.msk})$.
- $\mathsf{sFE.EncSetup}(\mathsf{MPK})$:
  1. Parse $\mathsf{MPK} = \mathsf{FE.mpk}$.
  2. $\mathsf{PRF}.K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$.
  3. $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
  4. $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}}))$.
  5. Output $\mathsf{Enc.ST} = (\mathsf{FPFE.msk}, \mathsf{FE.ct})$.
- $\mathsf{sFE.Enc}(\mathsf{MPK}, \mathsf{Enc.ST}, i, x_i)$:
  1. Parse $\mathsf{Enc.ST} = (\mathsf{FPFE.msk}, \mathsf{FE.ct})$.
  2. $t_i \leftarrow \{0,1\}^\lambda$.
  3. Let $H_i = H_{i,x_i,t_i}$ as defined in Figure 5.
  4. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$
  5. If $i = 1$, output $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$.
  6. Else, output $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$

$H_{i,x_i,t_i}(\mathsf{One\text{-}sFE.msk}, \mathsf{One\text{-}sFE.Enc.st}, \mathsf{PRF2}.k, \beta)$:

1. If $\beta = 0$
   (a) $r_i \leftarrow \mathsf{PRF2.Eval}(\mathsf{PRF2}.k, t_i)$
   (b) Output $\mathsf{One\text{-}sFE.ct}_i \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}, \mathsf{One\text{-}sFE.Enc.st}, i, x_i; r_i)$
2. Else, output $\perp$

Figure 5: Definition of H_{i,x_i,t_i}

- $\mathsf{sFE.KeyGen}(\mathsf{MSK}, f)$:
  1. Parse $\mathsf{MSK} = \mathsf{FE.msk}$.
  2. $s \leftarrow \{0,1\}^\lambda$.
  3. $c \leftarrow \{0,1\}^{\ell_{\mathsf{SKE.ct}_\lambda}}$.
  4. Let $G = G_{f,s,c}$ as defined in Figure 6.

5. $\mathsf{FE.sk}_G \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G)$.
6. Output $\mathsf{SK}_f = \mathsf{FE.sk}_G$.

$G_{f,s,c}(\mathsf{FPFE.msk}, \mathsf{PRF}.K, \alpha, \mathsf{SKE}.k)$:

1. If $\alpha = 0$
   (a) $(r_{\mathsf{Setup}}, r_{\mathsf{KeyGen}}, r_{\mathsf{EncSetup}}, r_{\mathsf{PRF2}}, r_{\mathsf{Enc}}) \leftarrow \mathsf{PRF.Eval}(\mathsf{PRF}.K, s)$
   (b) $\mathsf{One\text{-}sFE.msk} \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda; r_{\mathsf{Setup}})$
   (c) $\mathsf{One\text{-}sFE.Enc.st} \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}; r_{\mathsf{EncSetup}})$
   (d) $\mathsf{One\text{-}sFE.sk}_f \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}, f; r_{\mathsf{KeyGen}})$
   (e) $\mathsf{PRF2}.k \leftarrow \mathsf{PRF2.Setup}(1^\lambda; r_{\mathsf{PRF2}})$
   (f) $\mathsf{FPFE.ct} \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}, \mathsf{One\text{-}sFE.Enc.st}, \mathsf{PRF2}.k, 0); r_{\mathsf{Enc}})$
   (g) Output $(\mathsf{One\text{-}sFE.sk}_f, \mathsf{FPFE.ct})$
2. Else
   (a) Output $(\mathsf{One\text{-}sFE.sk}_f, \mathsf{FPFE.ct}) \leftarrow \mathsf{SKE.Dec}(\mathsf{SKE}.k, c)$

Figure 6: Definition of $G_{f,s,c}$

- $\mathsf{sFE.Dec}(\mathsf{SK}_f, \mathsf{Dec.ST}_i, i, \mathsf{CT}_i)$:
  1. If $i = 1$
     (a) Parse $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$ and $\mathsf{SK}_f = \mathsf{FE.sk}_G$.
     (b) $(\mathsf{One\text{-}sFE.sk}_f, \mathsf{FPFE.ct}) = \mathsf{FE.Dec}(\mathsf{FE.sk}_G, \mathsf{FE.ct})$
     (c) Set $\mathsf{One\text{-}sFE.Dec.st}_1 = \bot$.
  2. If $i > 1$
     (a) Parse $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$
     (b) Parse $\mathsf{Dec.ST}_i = (\mathsf{One\text{-}sFE.sk}_f, \mathsf{FPFE.ct}, \mathsf{One\text{-}sFE.Dec.st}_i)$
  3. $\mathsf{One\text{-}sFE.ct}_i = \mathsf{FPFE.Dec}(\mathsf{FPFE.sk}_{H_i}, \mathsf{FPFE.ct})$
  4. $(y_i, \mathsf{One\text{-}sFE.Dec.st}_{i+1}) = \mathsf{One\text{-}sFE.Dec}(\mathsf{One\text{-}sFE.sk}_f, \mathsf{One\text{-}sFE.Dec.st}_i, i, \mathsf{One\text{-}sFE.ct}_i)$
  5. Output $(y_i, \mathsf{Dec.ST}_{i+1} = (\mathsf{One\text{-}sFE.sk}_f, \mathsf{FPFE.ct}, \mathsf{One\text{-}sFE.Dec.st}_{i+1}))$

We also define the following function which will be used in our security proof.

$H^*_{i,x_i,x'_i,t_i,v_i}(\mathsf{One\text{-}sFE.msk}, \mathsf{One\text{-}sFE.Enc.st}, \mathsf{PRF2}.k, \beta)$:

- If $\beta = 0$
  1. $r_i \leftarrow \mathsf{PRF2.Eval}(\mathsf{PRF2}.k, t_i)$
  2. Output $\mathsf{One\text{-}sFE.ct}_i \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}, \mathsf{One\text{-}sFE.Enc.st}, i, x_i; r_i)$
- If $\beta = 1$
  1. $r_i \leftarrow \mathsf{PRF2.Eval}(\mathsf{PRF2}.k, t_i)$
  2. Output $\mathsf{One\text{-}sFE.ct}_i \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}, \mathsf{One\text{-}sFE.Enc.st}, i, x'_i; r_i)$
- Else, output v_i

Figure 7: Definition of H^*_{i,x_i,x'_i,t_i,v_i}

7.3 Correctness and Efficiency

Correctness and efficiency follow from the corresponding theorems in [GKS23] since our construction is identical to [GKS23] except for the fact that One-sFE now represents a (single-key, single-ciphertext) adaptively secure scheme, rather than a (single-key, single-ciphertext) function-selectively secure scheme. However, this difference only affects the security of One-sFE and thus only affects the proof of security.

7.4 Security

As the security proof is very similar to the one in [GKS23], we defer it to Appendix D (page 296).

8 Acknowledgements

This research was supported in part from a Simons Investigator Award, DARPA SIEVE award, NTT Research, BSF grant 2018393, a Xerox Faculty Research Award, a Google Faculty Research Award, and an Okawa Foundation Research Grant. This material is based upon work supported by the Defense Advanced Research Projects Agency through Award HR00112020024.

9 References

[AAB15] Benny Applebaum, Jonathan Avron, and Christina Brzuska. Arithmetic cryptography: Extended abstract. In Tim Roughgarden, editor, *ITCS 2015*, pages 143–151. ACM, January 2015.

[ABSV15] Prabhanjan Ananth, Zvika Brakerski, Gil Segev, and Vinod Vaikuntanathan. From selective to adaptive security in functional encryption. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 657–677. Springer, Heidelberg, August 2015.

[ACF+19] Shweta Agrawal, Michael Clear, Ophir Frieder, Sanjam Garg, Adam O'Neill, and Justin Thaler. Ad hoc multi-input functional encryption. Cryptology ePrint Archive, Paper 2019/356, 2019. `https://eprint.iacr.org/2019/356`.

[AGIS14] Prabhanjan Vijendra Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai. Optimizing obfuscation: Avoiding Barrington's theorem. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 646–658. ACM Press, November 2014.

[Agr19] Shweta Agrawal. Indistinguishability obfuscation without multilinear maps: New methods for bootstrapping and instantiation. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 191–225. Springer, Heidelberg, May 2019.

[AJ15] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 308–326. Springer, Heidelberg, August 2015.

[AJL+19] Prabhanjan Ananth, Aayush Jain, Huijia Lin, Christian Matt, and Amit Sahai. Indistinguishability obfuscation without multilinear maps: New paradigms via low degree weak pseudorandomness and security amplification. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 284–332. Springer, Heidelberg, August 2019.

[AJS15] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Indistinguishability obfuscation with constant size overhead. Cryptology ePrint Archive, Report 2015/1023, 2015. `https://eprint.iacr.org/2015/1023`.

[AJS17a] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Indistinguishability obfuscation for Turing machines: Constant overhead and amortization. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 252–279. Springer, Heidelberg, August 2017.

[AJS17b] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Patchable indistinguishability obfuscation: iO for evolving software. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 127–155. Springer, Heidelberg, April / May 2017.

[AJS18] Prabhanjan Ananth, Aayush Jain, and Amit Sahai. Indistinguishability obfuscation without multilinear maps: io from lwe, bilinear maps, and weak pseudorandomness.

Cryptology ePrint Archive, Paper 2018/615, 2018. https://eprint.iacr.org/2018/615.

[AP20] Shweta Agrawal and Alice Pellet-Mary. Indistinguishability obfuscation without maps: Attacks and fixes for noisy linear FE. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 110–140. Springer, Heidelberg, May 2020.

[AS16] Prabhanjan Vijendra Ananth and Amit Sahai. Functional encryption for Turing machines. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 125–153. Springer, Heidelberg, January 2016.

[AS17] Prabhanjan Ananth and Amit Sahai. Projective arithmetic functional encryption and indistinguishability obfuscation from degree-5 multilinear maps. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 152–181. Springer, Heidelberg, April / May 2017.

[BB11] Dan Boneh and Xavier Boyen. Efficient selective identity-based encryption without random oracles. *Journal of Cryptology*, 24(4):659–693, October 2011.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

[BDGM20] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Candidate iO from homomorphic encryption schemes. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 79–109. Springer, Heidelberg, May 2020.

[BFK+19] Saikrishna Badrinarayanan, Rex Fernando, Venkata Koppula, Amit Sahai, and Brent Waters. Output compression, MPC, and iO for Turing machines. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 342–370. Springer, Heidelberg, December 2019.

[BFKL94] Avrim Blum, Merrick L. Furst, Michael J. Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 278–291. Springer, Heidelberg, August 1994.

[BFM14] Christina Brzuska, Pooya Farshim, and Arno Mittelbach. Indistinguishability obfuscation and UCEs: The case of computationally unpredictable sources. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 188–205. Springer, Heidelberg, August 2014.

[BGG+14] Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 533–556. Springer, Heidelberg, May 2014.

[BGH+15] Zvika Brakerski, Craig Gentry, Shai Halevi, Tancrède Lepoint, Amit Sahai, and Mehdi Tibouchi. Cryptanalysis of the quadratic zero-testing of GGH. *IACR Cryptol. ePrint Arch.*, page 845, 2015.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014.

[BGK+14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 221–238. Springer, Heidelberg, May 2014.

[BIJ+20] James Bartusek, Yuval Ishai, Aayush Jain, Fermi Ma, Amit Sahai, and Mark Zhandry. Affine determinant programs: A framework for obfuscation and witness encryption. In Thomas Vidick, editor, *ITCS 2020*, volume 151, pages 82:1–82:39. LIPIcs, January 2020.

[BKS16] Zvika Brakerski, Ilan Komargodski, and Gil Segev. Multi-input functional encryption in the private-key setting: Stronger security from weaker assumptions. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 852–880. Springer, Heidelberg, May 2016.

[BMSZ16] Saikrishna Badrinarayanan, Eric Miles, Amit Sahai, and Mark Zhandry. Post-zeroizing obfuscation: New mathematical tools, and the case of evasive circuits. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 764–791. Springer, Heidelberg, May 2016.

[BPR15] Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a Nash equilibrium. In Venkatesan Guruswami, editor, *56th FOCS*, pages 1480–1498. IEEE Computer Society Press, October 2015.

[BR14] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 1–25. Springer, Heidelberg, February 2014.

[BS18] Zvika Brakerski and Gil Segev. Function-private functional encryption in the private-key setting. *Journal of Cryptology*, 31(1):202–225, January 2018.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 253–273. Springer, Heidelberg, March 2011.

[BV15] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. In Venkatesan Guruswami, editor, *56th FOCS*, pages 171–190. IEEE Computer Society Press, October 2015.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.

[BWZ14] Dan Boneh, David J. Wu, and Joe Zimmerman. Immunizing multilinear maps against zeroizing attacks. *IACR Cryptol. ePrint Arch.*, page 930, 2014.

[CCC+16] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In Madhu Sudan, editor, *ITCS 2016*, pages 179–190. ACM, January 2016.

[CCHR16] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled RAM or: How to delegate your database. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 61–90. Springer, Heidelberg, October / November 2016.

[CFL+16] Jung Hee Cheon, Pierre-Alain Fouque, Changmin Lee, Brice Minaud, and Hansol Ryu. Cryptanalysis of the new CLT multilinear map over the integers. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 509–536. Springer, Heidelberg, May 2016.

[CGH+15] Jean-Sébastien Coron, Craig Gentry, Shai Halevi, Tancrède Lepoint, Hemanta K. Maji, Eric Miles, Mariana Raykova, Amit Sahai, and Mehdi Tibouchi. Zeroizing without low-level zeroes: New MMAP attacks and their limitations. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 247–266. Springer, Heidelberg, August 2015.

[CHL+15] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 3–12. Springer, Heidelberg, April 2015.

[CHN+16] Aloni Cohen, Justin Holmgren, Ryo Nishimaki, Vinod Vaikuntanathan, and Daniel Wichs. Watermarking cryptographic capabilities. In Daniel Wichs and Yishay Mansour, editors, *48th ACM STOC*, pages 1115–1127. ACM Press, June 2016.

[CLT13] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 476–493. Springer, Heidelberg, August 2013.

[CLT15] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. New multilinear maps over the integers. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 267–286. Springer, Heidelberg, August 2015.

[DGG+18] Nico Döttling, Sanjam Garg, Divya Gupta, Peihan Miao, and Pratyay Mukherjee. Obfuscation from low noise multilinear maps. In Debrup Chakraborty and Tetsu Iwata, editors, *INDOCRYPT 2018*, volume 11356 of *LNCS*, pages 329–352. Springer, Heidelberg, December 2018.

[DKW16] Apoorvaa Deshpande, Venkata Koppula, and Brent Waters. Constrained pseudorandom functions for unconstrained inputs. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 124–153. Springer, Heidelberg, May 2016.

[GGG+14] Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 578–602. Springer, Heidelberg, May 2014.

[GGH12] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. Cryptology ePrint Archive, Report 2012/610, 2012. `https://eprint.iacr.org/2012/610`.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

[GGH15] Craig Gentry, Sergey Gorbunov, and Shai Halevi. Graph-induced multilinear maps from lattices. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 498–527. Springer, Heidelberg, March 2015.

[GGHZ16] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. Functional encryption without obfuscation. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 480–511. Springer, Heidelberg, January 2016.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[GJ18] Craig Gentry and Charanjit S. Jutla. Obfuscation using tensor products. *Electron. Colloquium Comput. Complex.*, TR18-149, 2018.

[GJO16] Vipul Goyal, Aayush Jain, and Adam O'Neill. Multi-input functional encryption with unbounded-message security. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 531–556. Springer, Heidelberg, December 2016.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 555–564. ACM Press, June 2013.

[GKS23] Jiaxin Guan, Alexis Korb, and Amit Sahai. Streaming functional encryption. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 433–463. Springer, Heidelberg, August 2023.

[Gol01] Oded Goldreich. *Foundations of Cryptography, Volume 1, Basic Tools*, volume 1. Cambridge university press, 2001.

[Gol09] Oded Goldreich. *Foundations of Cryptography, Volume 2, Basic Applications*, volume 2. Cambridge university press, 2009.

[GPS16] Sanjam Garg, Omkant Pandey, and Akshayaram Srinivasan. Revisiting the cryptographic hardness of finding a nash equilibrium. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 579–604. Springer, Heidelberg, August 2016.

[GPSZ17] Sanjam Garg, Omkant Pandey, Akshayaram Srinivasan, and Mark Zhandry. Breaking the sub-exponential barrier in obfustopia. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 156–181. Springer, Heidelberg, April / May 2017.

[GVW15] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Predicate encryption for circuits from LWE. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 503–523. Springer, Heidelberg, August 2015.

[Hal15] Shai Halevi. Graded encoding, variations on a scheme. *IACR Cryptol. ePrint Arch.*, page 866, 2015.

[HJ15] Yupu Hu and Huiwen Jia. Cryptanalysis of GGH map. *IACR Cryptol. ePrint Arch.*, page 301, 2015.

[HJK+16] Dennis Hofheinz, Tibor Jager, Dakshita Khurana, Amit Sahai, Brent Waters, and Mark Zhandry. How to generate and use universal samplers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 715–744. Springer, Heidelberg, December 2016.

[HSW13] Susan Hohenberger, Amit Sahai, and Brent Waters. Full domain hash from (leveled) multilinear maps and identity-based aggregate signatures. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 494–512. Springer, Heidelberg, August 2013.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, Heidelberg, August 2008.

[JKK+17] Zahra Jafargholi, Chethan Kamath, Karen Klein, Ilan Komargodski, Krzysztof Pietrzak, and Daniel Wichs. Be adaptive, avoid overcommitting. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 133–163. Springer, Heidelberg, August 2017.

[JLMS19] Aayush Jain, Huijia Lin, Christian Matt, and Amit Sahai. How to leverage hardness of constant-degree expanding polynomials over $\mathbb{R}$ to build $i\mathcal{O}$. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 251–281. Springer, Heidelberg, May 2019.

[JLS21] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. In Samir Khuller and Virginia Vassilevska Williams, editors, *53rd ACM STOC*, pages 60–73. ACM Press, June 2021.

[JLS22] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from LPN over $\mathbb{F}_p$, DLIN, and PRGs in NC^0. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 670–699. Springer, Heidelberg, May / June 2022.

[JW16] Zahra Jafargholi and Daniel Wichs. Adaptive security of yao's garbled circuits. Cryptology ePrint Archive, Report 2016/814, 2016. `https://eprint.iacr.org/2016/814`.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for Turing machines with unbounded memory. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 419–428. ACM Press, June 2015.

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.

[KW20] Lucas Kowalczyk and Hoeteck Wee. Compact adaptively secure ABE for NC^1 from k-Lin. *Journal of Cryptology*, 33(3):954–1002, July 2020.

[Lin16] Huijia Lin. Indistinguishability obfuscation from constant-degree graded encoding schemes. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 28–57. Springer, Heidelberg, May 2016.

[Lin17] Huijia Lin. Indistinguishability obfuscation from SXDH on 5-linear maps and locality-5 PRGs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 599–629. Springer, Heidelberg, August 2017.

[LM18] Huijia Lin and Christian Matt. Pseudo flawed-smudging generators and their application to indistinguishability obfuscation. *IACR Cryptol. ePrint Arch.*, page 646, 2018.

[LOS+10] Allison B. Lewko, Tatsuaki Okamoto, Amit Sahai, Katsuyuki Takashima, and Brent Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 62–91. Springer, Heidelberg, May / June 2010.

[LT17] Huijia Lin and Stefano Tessaro. Indistinguishability obfuscation from trilinear maps and block-wise local PRGs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 630–660. Springer, Heidelberg, August 2017.

[LV16] Huijia Lin and Vinod Vaikuntanathan. Indistinguishability obfuscation from DDH-like assumptions on constant-degree graded encodings. In Irit Dinur, editor, *57th FOCS*, pages 11–20. IEEE Computer Society Press, October 2016.

[LW10] Allison B. Lewko and Brent Waters. New techniques for dual system encryption and fully secure HIBE with short ciphertexts. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 455–479. Springer, Heidelberg, February 2010.

[MSZ16] Eric Miles, Amit Sahai, and Mark Zhandry. Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 629–658. Springer, Heidelberg, August 2016.

[O'N10] Adam O'Neill. Definitional issues in functional encryption. *IACR Cryptology ePrint Archive*, 2010:556, 2010.

[PST14] Rafael Pass, Karn Seth, and Sidharth Telang. Indistinguishability obfuscation from semantically-secure multilinear encodings. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 500–517. Springer, Heidelberg, August 2014.

[SW05] Amit Sahai and Brent R. Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 457–473. Springer, Heidelberg, May 2005.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In David B. Shmoys, editor, *46th ACM STOC*, pages 475–484. ACM Press, May / June 2014.

[Wat09] Brent Waters. Dual system encryption: Realizing fully secure IBE and HIBE under simple assumptions. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 619–636. Springer, Heidelberg, August 2009.

[Wat15] Brent Waters. A punctured programming approach to adaptively secure functional encryption. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 678–697. Springer, Heidelberg, August 2015.

A [JLS22] Assumptions

In this section, we detail the assumptions used in [JLS22] to build (subexponentially secure) $i\mathcal{O}$ for $\mathsf{P/Poly}$.

Definition A.1 (ϵ-Indistinguishability)**.** *We say that two ensembles $\mathcal{X} = \{\mathcal{X}_\lambda\}_{\lambda\in\mathbb{N}}$ and $\mathcal{Y} = \{\mathcal{Y}_\lambda\}_{\lambda\in\mathbb{N}}$ are ϵ-indistinguishable if for all PPT adversaries $\mathcal{A}$ and for all sufficiently large $\lambda \in \mathbb{N}$,*

$$\left|\Pr_{x\leftarrow\mathcal{X}_\lambda}[A(1^\lambda, x) = 1] - \Pr_{y\leftarrow\mathcal{Y}_\lambda}[\mathcal{A}(1^\lambda, y) = 1]\right| \leq \epsilon(\lambda)$$

We say that two ensembles are computationally indistinguishable if they are ϵ-indistinguishable for $\epsilon(\lambda) = \mathsf{negl}(\lambda)$ for some negligible negl, and that two ensembles are subexponentially indistinguishable if they are ϵ-indistinguishable for $\epsilon(\lambda) = 2^{-\lambda^c}$ for some positive real number c.

Definition A.2 (δ-LPN Assumption [BFKL94, IPS08, AAB15, BCGI18])**.** *Let $\delta \in (0, 1)$. We say that the δ-LPN Assumption is true if the following holds: For any constant $\eta_p > 0$, any function $p : \mathbb{N} \to \mathbb{N}$ such that for every $\ell \in \mathbb{N}$, $p(\ell)$ is a prime of ℓ^{η_p} bits, any constant $\eta_n > 0$, we set $p = p(\ell), n = n(\ell) = \ell^{\eta_n}$, and $r = r(\ell) = \ell^{-\delta}$, and we require that the following two distributions are computationally indistinguishable:*

$$\left\{(\boldsymbol{A}, \boldsymbol{b} = \boldsymbol{s}\cdot\boldsymbol{A} + \boldsymbol{e}) \mid \boldsymbol{A} \leftarrow \mathbb{Z}_p^{\ell\times n}, \boldsymbol{s} \leftarrow \mathbb{Z}_p^{1\times n}, \boldsymbol{e} \leftarrow \mathcal{D}_r^{1\times n}(p)\right\}_{\ell\in\mathbb{N}}$$

$$\left\{(\boldsymbol{A}, \boldsymbol{u}) \mid \boldsymbol{A} \leftarrow \mathbb{Z}_p^{\ell\times n}, \boldsymbol{u} \leftarrow \mathbb{Z}_p^{1\times n}\right\}_{\ell\in\mathbb{N}}$$

where $e \leftarrow \mathcal{D}_r(p)$ is a generalized Bernoulli distribution, i.e. e is sampled randomly from $\mathbb{Z}_p$ with probability $r = \ell^{-\delta}$ and set to be 0 with probability $1 - r$. We say that subexponential δ-LPN holds if the two distributions above are subexponentially indistinguishable.

Remark A.3. A PRG (see Definition B.1) is said to be in NC^0 if it is implementable by a uniformly efficiently generatable NC^0 circuit. We say a PRG with stretch $m(\cdot)$ is subexponentially secure if there exists a real positive constant c such that for all non-uniform PPT adversaries $\mathcal{A}$ and all sufficiently large $n \in \mathbb{N}$,

$$\left|\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(\mathsf{PRG}(r)) = 1] - \Pr_{z\leftarrow\{0,1\}^{m(n)}}[\mathcal{A}(z) = 1]\right| \leq 2^{-\lambda^c}$$

Definition A.4 (DLIN Assumption)**.** *The decision linear* (DLIN) *assumption over prime order symmetric bilinear groups is stated as follows: Given an appropriate prime p, two groups $\mathcal{G}, \mathcal{G}_T$ are chosen of order p such that there exists an efficiently computable nontrivial bilinear map $e : \mathcal{G} \times \mathcal{G} \to \mathcal{G}_T$. Canonical generators g for $\mathcal{G}$ and g_T for $\mathcal{G}_T$ are also computed. Then, the* DLIN *assumption requires that the following computational indistinguishability holds:*

$$\{(g^x, g^y, g^z, g^{xa}, g^{yb}, g^{z(a+b)}) : x, y, z, a, b \leftarrow \mathbb{Z}_p\} \approx_c \{(g^x, g^y, g^z, g^{xa}, g^{yb}, g^{zc}) : x, y, z, a, b, c \leftarrow \mathbb{Z}_p\}$$

We say that subexponential DLIN *holds if the two distributions above are subexponentially indistinguishable.*

B Preliminaries Continued

B.1 Standard Notions

Definition B.1 (Pseudorandom Generator (PRG))**.** *A pseudorandom generator with stretch $m(\cdot)$ is a Boolean function $\mathsf{PRG} : \{0,1\}^* \to \{0,1\}^*$ mapping n-bit inputs to $m(n)$-bit outputs that is computable by a uniform PPT machine. Security holds if there exists a negligible function μ such that for all $n \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\left| \Pr_{r \leftarrow \{0,1\}^n}[\mathcal{A}(\mathsf{PRG}(r)) = 1] - \Pr_{z \leftarrow \{0,1\}^{m(n)}}[\mathcal{A}(z) = 1] \right| \leq \mu(n)$$

Definition B.2 (Pseudorandom Function (PRF))**.** *A pseudorandom function family (PRF) with key space $\mathcal{K} = \{\mathcal{K}_{\lambda,n,m}\}_{\lambda,n,m \in \mathbb{N}}$ is a tuple of PPT algorithms $\mathsf{PRF} = (\mathsf{PRF.Setup}, \mathsf{PRF.Eval})$ where*

- $\mathsf{PRF.Setup}(1^\lambda, 1^n, 1^m)$ *is a randomized algorithm that takes as input the security parameter λ, an input length n, and an output length m, and outputs a key $K \in \mathcal{K}_{\lambda,n,m}$*
- $\mathsf{PRF.Eval}(K, x)$ *is a deterministic algorithm that takes as input a key $K \in \mathcal{K}_{\lambda,n,m}$ and an input $x \in \{0,1\}^n$, and outputs a value $y \in \{0,1\}^m$.*

Security requires that there exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all PPT *adversaries $\mathcal{A}$,*

$$\left| \Pr[\mathsf{Expt}^{\mathsf{PRF}}_{\mathcal{A}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}^{\mathsf{PRF}}_{\mathcal{A}}(1^\lambda, 1) = 1] \right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

$\mathsf{Expt}^{\mathsf{PRF}}_{\mathcal{A}}(1^\lambda, b)$

1. ***Parameters**: $\mathcal{A}$ takes as input 1^λ and outputs an input size 1^n and an output size 1^m.*
2. ***Setup**:*
   (a) *If $b = 0$, sample $K \leftarrow \mathsf{PRF.Setup}(1^\lambda, 1^n, 1^m)$.*
   (b) *If $b = 1$, sample $R \leftarrow \mathcal{R}_{n,m}$ where $\mathcal{R}_{n,m}$ is the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$.*
3. ***PRF Queries**: The following can be repeated any polynomial number of times:*
   (a) *$\mathcal{A}$ outputs a value $x \in \{0,1\}^n$.*
   (b) *If $b = 0$, send $y = \mathsf{PRF.Eval}(K, x)$ to $\mathcal{A}$.*

(c) If $b = 1$, send $y = R(x)$ to $\mathcal{A}$.

4. ***Experiment Outcome**: $\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

Definition B.3 (Symmetric Key Encryption (SKE))**.** *A symmetric key encryption scheme with key space $\mathcal{K} = \{\mathcal{K}_{\lambda,n}\}_{\lambda,n\in\mathbb{N}}$ and ciphertext size $m(\cdot)$ is a tuple of PPT algorithms* $\mathsf{SKE} = (\mathsf{SKE.Setup}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$ *where*

- $\mathsf{SKE.Setup}(1^\lambda, 1^n)$ *is a randomized algorithm that takes as input the security parameter λ and an input length n and outputs a secret key $k \in \mathcal{K}_{\lambda,n}$*
- $\mathsf{SKE.Enc}(k, x)$ *is a randomized algorithm that takes as input a secret key $k \in \mathcal{K}_{\lambda,n}$ and a message $x \in \{0,1\}^n$ and outputs an encryption $\mathsf{ct} \in \{0,1\}^{m(\lambda,n)}$ of x.*
- $\mathsf{SKE.Dec}(k, \mathsf{ct})$ *is a deterministic algorithm that takes as input a secret key $k \in \mathcal{K}_{\lambda,n}$ and a ciphertext $\mathsf{ct} \in \{0,1\}^{m(\lambda,n)}$ and outputs a value $y \in \{0,1\}^n$.*

Correctness requires that for all $\lambda, n \in \mathbb{N}$ and every $x \in \{0,1\}^n$,

$$\Pr\left[\mathsf{SKE.Dec}(k, \mathsf{SKE.Enc}(k, x)) = x : k \leftarrow \mathsf{SKE.Setup}(1^\lambda, 1^n)\right] = 1$$

Security requires that there exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all PPT *adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Expt}^{\mathsf{SKE}}_{\mathcal{A}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}^{\mathsf{SKE}}_{\mathcal{A}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

$\mathsf{Expt}^{\mathsf{SKE}}_{\mathcal{A}}(1^\lambda, b)$

1. ***Parameters**: $\mathcal{A}$ takes as input 1^λ and outputs an input size 1^n.*
2. ***Setup**: $k \leftarrow \mathsf{SKE.Setup}(1^\lambda, 1^n)$*
3. ***Challenge Message Queries**: The following can be repeated any polynomial number of times:*
   (a) $\mathcal{A}$ outputs a challenge message pair (x_0, x_1) where $x_0, x_1 \in \{0,1\}^n$.
   (b) $\mathsf{ct}_b \leftarrow \mathsf{SKE.Enc}(k, x_b)$
   (c) Sent ct_b to $\mathcal{A}$.
4. ***Experiment Outcome**: $\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

We will sometimes require that our symmetric key encryption scheme has pseudorandom ciphertexts. Intuitively, this means that ciphertexts should be indistinguishable from random strings of the same size.

Definition B.4 (Symmetric Key Encryption (SKE) with Pseudorandom Ciphertexts)**.** *A symmetric key encryption scheme* $\mathsf{SKE} = (\mathsf{SKE.Setup}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$ *with key space $\mathcal{K} = \{\mathcal{K}_{\lambda,n}\}_{\lambda,n\in\mathbb{N}}$ and ciphertext size $m(\cdot)$ has pseudorandom ciphertexts if there exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Expt}^{\mathsf{SKE\text{-}Pseudorandom\text{-}CT}}_{\mathcal{A}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}^{\mathsf{SKE\text{-}Pseudorandom\text{-}CT}}_{\mathcal{A}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

$\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SKE\text{-}Pseudorandom\text{-}CT}}(1^\lambda, b)$

1. ***Parameters***: *$\mathcal{A}$ takes as input 1^λ and outputs an input size 1^n.*
2. ***Setup***: *$k \leftarrow \mathsf{SKE.Setup}(1^\lambda, 1^n)$*
3. ***Challenge Message Queries***: *The following can be repeated any polynomial number of times:*
   (a) *$\mathcal{A}$ outputs a challenge message x where $x \in \{0,1\}^n$.*
   (b) *If $b = 0$, $\mathsf{ct} \leftarrow \mathsf{SKE.Enc}(k, x)$.*
   (c) *If $b = 1$, $\mathsf{ct} \leftarrow \{0,1\}^{m(\lambda,n)}$.*
   (d) *Send ct to $\mathcal{A}$.*
4. ***Experiment Outcome***: *$\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

B.2 Secret-Key Functional Encryption

In this section, we formally define secret-key functional encryption.

Definition B.5 (Secret-Key Functional Encryption)**.** *A secret-key functional encryption scheme for $\mathsf{P/Poly}$ is a tuple of PPT algorithms $\mathsf{FE} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$ defined as follows:*[26]

- $\mathsf{Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$*: takes as input the security parameter λ, a function size $\ell_\mathcal{F}$, an input size $\ell_\mathcal{X}$, and an output size $\ell_\mathcal{Y}$, and outputs the master secret key msk.*
- $\mathsf{Enc}(\mathsf{msk}, x)$*: takes as input the master secret key msk and a message $x \in \{0,1\}^{\ell_\mathcal{X}}$, and outputs an encryption ct of x.*
- $\mathsf{KeyGen}(\mathsf{msk}, f)$*: takes as input the master secret key msk and a function $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$, and outputs a function key sk_f.*
- $\mathsf{Dec}(\mathsf{sk}_f, \mathsf{ct})$*: takes as input a function key sk_f and a ciphertext ct, and outputs a value $y \in \{0,1\}^{\ell_\mathcal{Y}}$.*

FE *satisfies* ***correctness*** *if for all polynomials p, there exists a negligible function μ such that for all $\lambda \in \mathbb{N}$, all $\ell_\mathcal{F}, \ell_\mathcal{X}, \ell_\mathcal{Y} \leq p(\lambda)$, all $x \in \{0,1\}^{\ell_\mathcal{X}}$, and all $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$,*

$$\Pr\left[\mathsf{Dec}(\mathsf{sk}_f, \mathsf{ct}_x) = f(x) : \begin{array}{c} \mathsf{msk} \leftarrow \mathsf{Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}) \\ \mathsf{ct}_x \leftarrow \mathsf{Enc}(\mathsf{msk}, x) \\ \mathsf{sk}_f \leftarrow \mathsf{KeyGen}(\mathsf{msk}, f) \end{array}\right] \geq 1 - \mu(\lambda).$$

We now define adaptive security.

Definition B.6 (Adaptive Security for Secret-Key FE)**.** *A secret-key functional encryption scheme FE for $\mathsf{P/Poly}$ is adaptively secure if there exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and every PPT adversary $\mathcal{A}$,*

$$\left|\Pr[\mathsf{SKExpt}_{\mathcal{A}}^{\mathsf{FE\text{-}Adaptive}}(1^\lambda, 0) = 1] - \Pr[\mathsf{SKExpt}_{\mathcal{A}}^{\mathsf{FE\text{-}Adaptive}}(1^\lambda, 1) = 1]\right| \leq \mu(\lambda)$$

where for each $b \in \{0,1\}$ and $\lambda \in \mathbb{N}$, we define

[26] We also allow $\mathsf{Enc}, \mathsf{KeyGen}$, and Dec to additionally receive parameters $1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}$ as input, but omit them from our notation for convenience.

$\mathsf{SKExpt}_{\mathcal{A}}^{\mathsf{FE\text{-}Adaptive}}(1^\lambda, b)$

1. ***Parameters***: *$\mathcal{A}$ takes as input 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.*
2. ***Setup***: $\mathsf{msk} \leftarrow \mathsf{FE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
3. *For a polynomial number of rounds, the adversary can do either one of the following in each round:*
   (a) ***Function Query***:
      i. *$\mathcal{A}$ outputs a function query $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$.*
      ii. $\mathsf{sk}_f \leftarrow \mathsf{FE.KeyGen}(\mathsf{msk}, f)$.
      iii. *Send sk_f to $\mathcal{A}$.*
   (b) ***Challenge Message Query***:
      i. *$\mathcal{A}$ outputs a challenge message pair (x_0, x_1) where $x_0, x_1 \in \{0,1\}^{\ell_\mathcal{X}}$.*
      ii. $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{msk}, x_b)$.
      iii. *Send ct to $\mathcal{A}$.*
4. ***Experiment Outcome***: *$\mathcal{A}$ outputs a bit b' which is the output of the experiment.*

Additionally, when running the experiment, we immediately halt and output 0 if the adversary ever aborts or if it at any point $f(x_0) \neq f(x_1)$ for some message query (x_0, x_1) and function query f submitted by the adversary.

Definition B.7 (Other Secret-Key FE Security Definitions)**.** *There are many variations of the security definition. We list a few below:*

- ***Selective Security:*** *The adversary is required to make the message queries at the beginning of the experiment. This is similar to adaptive security, except that that we do not allow the adversary to make a* Challenge Message Query *after it has made a* Function Query*.*
- ***Function-Selective Security:*** *The adversary is required to make the function queries at the beginning of the experiment. This is similar to adaptive security, except that we do not allow the adversary to make a* Function Query *after it has made a* Challenge Message Query*.*

B.3 Secret-Key Streaming Functional Encryption

In this section, we formally define secret-key streaming functional encryption.

Definition B.8 (Secret-Key Streaming FE)**.** *A secret-key streaming functional encryption scheme for* P/Poly *is a tuple of PPT algorithms* $\mathsf{sFE} = (\mathsf{Setup}, \mathsf{EncSetup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$ *defined as follows:*[27]

1. $\mathsf{Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$*: takes as input the security parameter λ, a function size $\ell_\mathcal{F}$, a state size $\ell_\mathcal{S}$, an input size $\ell_\mathcal{X}$, and an output size $\ell_\mathcal{Y}$, and outputs the master secret key* msk*.*
2. $\mathsf{EncSetup}(\mathsf{msk})$*: takes as input the master secret key* msk *and outputs an encryption state* $\mathsf{Enc.st}$

[27] We also allow $\mathsf{Enc}, \mathsf{EncSetup}, \mathsf{KeyGen}$, and Dec to additionally receive parameters $1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}$ as input, but omit them from our notation for convenience.

3. $\mathsf{Enc}(\mathsf{msk}, \mathsf{Enc.st}, i, x_i)$*: takes as input the master secret key* msk*, an encryption state* $\mathsf{Enc.st}$*, an index* i*, and a message* $x_i \in \{0,1\}^{\ell_\mathcal{X}}$ *and outputs an encryption* ct_i *of* x_i*.*

4. $\mathsf{KeyGen}(\mathsf{msk}, f)$*: takes as input the master secret key* msk *and a function* $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ *and outputs a function key* sk_f*.*

5. $\mathsf{Dec}(\mathsf{sk}_f, \mathsf{Dec.st}_i, i, \mathsf{ct}_i)$*: where for each function key* sk_f*,* $\mathsf{Dec}(\mathsf{sk}_f, \cdot, \cdot, \cdot)$ *is a streaming function that takes as input a state* $\mathsf{Dec.st}_i$*, an index* i*, and an encryption* ct_i *and outputs a new state* $\mathsf{Dec.st}_{i+1}$ *and an output* $y_i \in \{0,1\}^{\ell_\mathcal{Y}}$*.*

sFE *must be* ***streaming efficient****, meaning that the size and runtime of all algorithms of* sFE *on security parameter* λ*, function size* $\ell_\mathcal{F}$*, state size* $\ell_\mathcal{S}$*, input size* $\ell_\mathcal{X}$*, and output size* $\ell_\mathcal{Y}$ *are* $\mathsf{poly}(\lambda, \ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y})$*.*

sFE *satisfies* ***correctness*** *if for all polynomials* p*, there exists a negligible function* μ *such that for all* $\lambda \in \mathbb{N}$*, all* $\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y} \le p(\lambda)$*, all* $n \in [2^\lambda]$*, all* $x = x_1 \dots x_n$ *where each* $x_i \in \{0,1\}^{\ell_\mathcal{X}}$*, and all* $f \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$*,*

$$\Pr\left[\overline{\mathsf{Dec}}(\mathsf{sk}_f, \mathsf{ct}_x) = f(x) : \begin{array}{c} \mathsf{msk} \leftarrow \mathsf{Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}), \\ \mathsf{ct}_x \leftarrow \overline{\mathsf{Enc}}(\mathsf{msk}, x), \\ \mathsf{sk}_f \leftarrow \mathsf{KeyGen}(\mathsf{msk}, f) \end{array}\right] \ge 1 - \mu(\lambda)$$

where we define[28]

- $\overline{\mathsf{Enc}}(\mathsf{msk}, x)$ *outputs* $\mathsf{ct}_x = (\mathsf{ct}_i)_{i \in [n]}$ *produced by sampling* $\mathsf{Enc.st} \leftarrow \mathsf{EncSetup}(\mathsf{msk})$ *and then computing* $ct_i \leftarrow \mathsf{Enc}(\mathsf{msk}, \mathsf{Enc.st}, i, x_i)$ *for* $i \in [n]$*.*
- $\overline{\mathsf{Dec}}(\mathsf{sk}_f, \mathsf{ct}_x)$ *outputs* $y = (y_i)_{i \in [n]}$ *where* $(y_i, \mathsf{Dec.st}_{i+1}) = \mathsf{Dec}(\mathsf{sk}_f, \mathsf{Dec.st}_i, i, \mathsf{ct}_i)$ *for* $i \in [n]$*.*

We now define adaptive security. Our definition of security is adaptive in a very strong sense, in that the adversary can not only adaptively pick each of the next values of each stream based on the ciphertexts and function keys already received, but can also interweave function queries with message queries of any stream. As each stream consists of multiple values, to avoid confusion, we require the adversary to specify which streams its challenge message queries are for by outputting a stream identity τ with its message queries.

Definition B.9 (Adaptive Security for Secret-Key sFE)**.** *A secret-key streaming FE scheme* sFE *for* $\mathsf{P/Poly}$ *is adaptively secure if there exists a negligible function* μ *such that for all* $\lambda \in \mathbb{N}$ *and all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{SKExpt}^{\mathsf{sFE\text{-}Adaptive}}_\mathcal{A}(1^\lambda, 0) = 1] - \Pr[\mathsf{SKExpt}^{\mathsf{sFE\text{-}Adaptive}}_\mathcal{A}(1^\lambda, 1) = 1]\right| \le \mu(\lambda)$$

where for each $b \in \{0,1\}$ *and* $\lambda \in \mathbb{N}$*, we define*

$\mathsf{SKExpt}^{\mathsf{sFE\text{-}Adaptive}}_\mathcal{A}(1^\lambda, b)$

1. ***Parameters****:* $\mathcal{A}$ *takes as input* 1^λ*, and outputs a function size* $1^{\ell_\mathcal{F}}$*, a state size* $1^{\ell_\mathcal{S}}$*, an input size* $1^{\ell_\mathcal{X}}$*, and an output size* $1^{\ell_\mathcal{Y}}$*.*
2. ***Setup****: Compute* $\mathsf{msk} \leftarrow \mathsf{sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$*.*
3. *For a polynomial number of rounds, the adversary can do either one of the following in*

[28] As with all streaming functions, we assume that $\mathsf{Dec.st}_1 = \bot$ if not otherwise specified.

each round:

(a) ***Function Query**:*

i. $\mathcal{A}$ *outputs a streaming function query* $f \in \mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$.

ii. $\mathsf{sk}_f \leftarrow \mathsf{sFE.KeyGen}(\mathsf{msk}, f)$.

iii. Send sk_f *to* $\mathcal{A}$.

(b) ***Challenge Message Query**:*

i. $\mathcal{A}$ *outputs a stream identity* τ.

ii. If this is the first challenge message query with stream identity τ*, sample* $\mathsf{Enc.st}_\tau \leftarrow \mathsf{sFE.EncSetup}(\mathsf{msk})$ *and initialize the index* $\mathsf{ind}_\tau = 1$*. Else, increment the index* ind_τ *by* 1.

iii. $\mathcal{A}$ *outputs a challenge message pair* $(x^{(0)}_{\mathsf{ind}_\tau}, x^{(1)}_{\mathsf{ind}_\tau})$ *for stream* τ *where* $x^{(0)}_{\mathsf{ind}_\tau}, x^{(1)}_{\mathsf{ind}_\tau} \in \{0,1\}^{\ell_{\mathcal{X}}}$.

iv. $\mathsf{ct}_{\mathsf{ind}_\tau} \leftarrow \mathsf{sFE.Enc}(\mathsf{msk}, \mathsf{Enc.st}_\tau, \mathsf{ind}_\tau, x^{(b)}_{\mathsf{ind}_\tau})$.

v. Send $\mathsf{ct}_{\mathsf{ind}_\tau}$ *to* $\mathcal{A}$.

4. ***Experiment Outcome**:* $\mathcal{A}$ *outputs a bit* b' *which is the output of the experiment.*

Additionally, when running the experiment, we immediately halt and output 0 *if the adversary ever aborts or if it at any point some function query* f *submitted by the adversary yields different outputs on any of the challenge message streams submitted so far (i.e. if* $f(x^{(0)}_\tau) \neq f(x^{(1)}_\tau)$ *for some function query* f *submitted by the adversary where* $\{(x^{(0)}_{\mathsf{ind}_\tau}, x^{(1)}_{\mathsf{ind}_\tau})\}_{\mathsf{ind}_\tau \in [t]}$ *are the message queries submitted so far under some stream identity* τ*,* $x^{(0)}_\tau = x^{(0)}_{1_\tau} \ldots x^{(0)}_{t_\tau}$*, and* $x^{(1)}_\tau = x^{(1)}_{1_\tau} \ldots x^{(1)}_{t_\tau}$*).*

Definition B.10 (Other Secret-Key sFE Security Definitions)**.** *There are many variations of the security definition. We list a few below:*

- ***Selective Security**: The adversary is required to make all message queries before any function queries. This is identical to adaptive security, except that we do not allow the adversary to make a* Challenge Message Query *after it has made a* Function Query*.*
- ***Function-Selective Security:** The adversary is required to make all function queries before any message queries. This is identical to adaptive security, except that we do not allow the adversary to make a* Function Query *after it has made a* Challenge Message Query*.*

C Security Proof from Section 6

In this section, we formally prove that $\mathsf{One\text{-}sFE}$ is single-key, single-ciphertext, adaptively secure for the function class $\mathsf{P/Poly}$.

Please refer to the technical overview (Section 2, page 9) for intuition on the proof and the proof overview (Section 6.4.1, page 77) for an overview of the hybrids.

The proof can be split into two main parts:

1. In Part 1, we use the security of $\mathsf{Pre\text{-}One\text{-}sFE}$ to deal with all the message queries given before the function query.

2. In Part 2, we use the security of $\mathsf{Post\text{-}One\text{-}sFE}$ to deal with all the message queries given after the function query.

Note that we use the security of these component schemes in a very non-black-box manner.

Remark C.1. Recall that for this proof, we will use an alternate, but equivalent, definition of single-key, single-ciphertext, adaptive security (Definition 6.2, page 76) where instead of needing to distinguish between an encryption of stream $x^{(0)}$ and an encryption of stream $x^{(1)}$, the adversary will receive an encryption of stream $x^{(b)}$ for a random bit b and will win if they correctly guess b.

Notation. For $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{sub}}$ labeled by subscript sub, we use $\mathsf{Prog}_{\mathsf{sub}}$ to denote the program that is obfuscated in that hybrid.

Remark C.2. We require all of our unwrapped[29] hybrids to immediately halt and output 0 if the adversary ever aborts or if it at any point some function query f submitted by the adversary yields different outputs on the challenge message streams submitted so far (i.e. if $f(x^{(0)}) \neq f(x^{(1)})$ for some function query f submitted by the adversary where $\{(x_i^{(0)}, x_i^{(1)})\}_{i\in[t]}$ are the message queries submitted so far, $x^{(0)} = x_1^{(0)} \dots x_t^{(0)}$, and $x^{(1)} = x_1^{(1)} \dots x_t^{(1)}$). For notational simplicitly, we omit this requirement from the description of our hybrids.

[29] Every hybrid named $\mathbf{Hybrid}^{\mathcal{A}}_{\mathsf{sub}}$ for some subscrip sub is considered an "unwrapped" hybrid. See the discussion betwen $\mathbf{Hybrid}^{\mathcal{A}}_0$ and $\mathbf{Hybrid}^{\mathcal{A}}_1$ for details.

C.1 Part 1: Using the Security of Pre-One-sFE

Hybrid$_0^{\mathcal{A}}$: Real world encryption of stream $x^{(b)}$. Here, we define t^* to be the length of the challenge stream after **Encryption Phase 1** and define n to be the total length of the challenge stream. Note that both t^* and n are adaptively (and implictly) chosen by the adversary.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
3. **Challenge Bit:** $b \leftarrow \{0,1\}$.
4. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(b)} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (c) $\mathsf{Pre.CT}_i^{(b)} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i^{(b)})$.
   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i^{(b)}$ to $\mathcal{A}$.
5. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \bot$.
   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.
   (c) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGen}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1)$.
   (d) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.
6. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(b)} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (c) $\mathsf{Pre.CT}_i^{(b)} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i^{(b)})$.
   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i^{(b)}$ to $\mathcal{A}$.
7. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Guessing the stream length. We defined t^* to be the length of the challenge stream after **Encryption Phase 1** and defined n to be the total length of the challenge stream. Note that both t^* and n were adapatively (and implicitly) chosen by the adversary.

In later hybrids, it is useful to know (t^*, n) at the beginning of the hybrid. To deal with this, we guess these values at the beginning of the experiment and abort (i.e. output $\perp$ and halt) if we notice that our guess is incorrect (i.e. the adversary asks for too few or too many queries at each stage). See $\mathbf{Hybrid}_1^{\mathcal{A}}$ below as an example.

As we only guess correctly with some low probability, we will repeat the whole experiment whenever our guess is incorrect. Thus, we define the following wrapper function Wrap for our hybrids. We will call any hybrid named $\mathbf{Hybrid}_{\mathsf{sub}}^{\mathcal{A}}$ for some subscript sub an *unwrapped* hybrid, and will call any hybrid named $\mathsf{Wrap}(\mathbf{Hybrid}_{\mathsf{sub}}^{\mathcal{A}})$ for some subscript sub a *wrapped* hybrid.

$\mathsf{Wrap}(\mathbf{Hybrid}^{\mathcal{A}})$:

1. For $\mathsf{Iteration} \in [T_{\mathcal{A},\lambda}^3]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ:
   (a) **Run hybrid:** $v \leftarrow \mathbf{Hybrid}^{\mathcal{A}}(1^\lambda)$.
   (b) **Check for correct guess:** If $v \neq \perp$, output v and halt.
2. Output 0.

Remark C.3. We overload notation and use (t^*, n) to additionally refer to the hybrid's guess for the length of the challenge stream after **Encryption Phase 1** and **Encryption Phase 2** respectively.

Remark C.4. By Lemma C.5 below, to show indistinguishability between two wrapped hybrids, it suffices to show indistinguishability between the unwrapped hybrids. Thus, for simplicity, for most of the remainder of this proof, we will omit this wrapper function from both our hybrids and our proofs of security. Note that the formal security proof would require all hybrids to be wrapped.

Lemma C.5. *Let* $\mathbf{Hybrid}_\alpha^{\mathcal{A}}$ *and* $\mathbf{Hybrid}_\beta^{\mathcal{A}}$ *be two* PPT *algorithms with outputs in* $\{0, 1, \perp\}$. *If for all* $\lambda \in \mathbb{N}$ *and any* PPT *adversary* $\mathcal{A}$,

$$\Delta(\mathbf{Hybrid}_\alpha^{\mathcal{A}}(1^\lambda), \mathbf{Hybrid}_\beta^{\mathcal{A}}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

then for all $\lambda \in \mathbb{N}$ *and any* PPT *adversary* $\mathcal{A}$,

$$\left|\Pr[\mathsf{Wrap}(\mathbf{Hybrid}_\alpha^{\mathcal{A}})(1^\lambda) = 1] - \Pr[\mathsf{Wrap}(\mathbf{Hybrid}_\beta^{\mathcal{A}})(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Let $\lambda \in \mathbb{N}$ and let $\mathcal{A}$ be a PPT adversary. We define intermediate hybrids $\mathsf{Wrap}_i(\mathbf{Hybrid}_\alpha^{\mathcal{A}}, \mathbf{Hybrid}_\beta^{\mathcal{A}})$ for $i \in [0, T_{\mathcal{A},\lambda}^3]$:

$\mathsf{Wrap}_i(\mathbf{Hybrid}_\alpha^{\mathcal{A}}, \mathbf{Hybrid}_\beta^{\mathcal{A}})$:

1. For $\mathsf{Iteration} \in [T_{\mathcal{A},\lambda}^3]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ:
   (a) **Run hybrid:**
      i. If $\mathsf{Iteration} \leq i$, $v \leftarrow \mathbf{Hybrid}_\beta^{\mathcal{A}}(1^\lambda)$.
      ii. If $\mathsf{Iteration} > i$, $v \leftarrow \mathbf{Hybrid}_\alpha^{\mathcal{A}}(1^\lambda)$.
   (b) **Check for correct guess:** If $v \neq \perp$, output v and halt.

2. Output 0.

Note that $\mathsf{Wrap}(\mathbf{Hybrid}_\alpha^{\mathcal{A}}) = \mathsf{Wrap}_0(\mathbf{Hybrid}_1^{\mathcal{A}}, \mathbf{Hybrid}_2^{\mathcal{A}})$ and $\mathsf{Wrap}(\mathbf{Hybrid}_\beta^{\mathcal{A}}) = \mathsf{Wrap}_{T^3_{\mathcal{A},\lambda}}(\mathbf{Hybrid}_\alpha^{\mathcal{A}}, \mathbf{Hybrid}_\beta^{\mathcal{A}})$.

Now, we claim that for all $i \in [T^3_{\mathcal{A},\lambda}]$, $\mathsf{Wrap}_{i-1}(\mathbf{Hybrid}_\alpha^{\mathcal{A}}, \mathbf{Hybrid}_\beta^{\mathcal{A}})$ is indistinguishable from $\mathsf{Wrap}_i(\mathbf{Hybrid}_\alpha^{\mathcal{A}}, \mathbf{Hybrid}_\beta^{\mathcal{A}})$. Consider the i^{th} execution (which is the only place they differ). In $\mathsf{Wrap}_{i-1}(\mathbf{Hybrid}_\alpha^{\mathcal{A}}, \mathbf{Hybrid}_\beta^{\mathcal{A}})$, we run $\mathbf{Hybrid}_\alpha^{\mathcal{A}}(1^\lambda)$, and in $\mathsf{Wrap}_i(\mathbf{Hybrid}_\alpha^{\mathcal{A}}, \mathbf{Hybrid}_\beta^{\mathcal{A}})$, we run $\mathbf{Hybrid}_\beta^{\mathcal{A}}(1^\lambda)$. Since the outputs of these two hybrids differ by only a negligible amount, then the outputs of $\mathsf{Wrap}_{i-1}(\mathbf{Hybrid}_\alpha^{\mathcal{A}}, \mathbf{Hybrid}_\beta^{\mathcal{A}})$ and $\mathsf{Wrap}_i(\mathbf{Hybrid}_\alpha^{\mathcal{A}}, \mathbf{Hybrid}_\beta^{\mathcal{A}})$ differ by only a negligible amount.

Thus, the output distributions of $\mathsf{Wrap}(\mathbf{Hybrid}_\alpha^{\mathcal{A}})(1^\lambda)$ and $\mathsf{Wrap}(\mathbf{Hybrid}_\alpha^{\mathcal{A}})(1^\lambda)$ can differ by only $T^3_{\mathcal{A},\lambda} \cdot \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$. □

$\mathbf{Hybrid}_1^{\mathcal{A}}$: In this hybrid, we guess the stream length at the beginning of the experiment. We abort the hybrid by outputting $\perp$ if we notice our guess is incorrect.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(b)} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (c) $\mathsf{Pre.CT}_i^{(b)} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i^{(b)})$.
   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i^{(b)}$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.
   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.
   (c) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGen}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1)$.
   (d) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.
7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
   If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(b)} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (c) $\mathsf{Pre.CT}_i^{(b)} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i^{(b)})$.
   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i^{(b)}$ to $\mathcal{A}$.
8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.6. *For all* PPT *adversaries* $\mathcal{A}$,

$$\left|\Pr[\mathbf{Hybrid}_0^{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathsf{Wrap}(\mathbf{Hybrid}_1^{\mathcal{A}})(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. As long as we correctly guess (t^*, n) in any iteration of $\mathsf{Wrap}(\mathbf{Hybrid}_1^{\mathcal{A}})$, then $\mathsf{Wrap}(\mathbf{Hybrid}_1^{\mathcal{A}})$ and $\mathbf{Hybrid}_1^{\mathcal{A}}$ are identically distributed. This is because $\mathsf{Wrap}(\mathbf{Hybrid}_1^{\mathcal{A}})$ throws out all executions of $\mathbf{Hybrid}_1^{\mathcal{A}}$ that are aborted, but conditioned on not aborting (i.e. correctly guessing (t^*, n)), then $\mathbf{Hybrid}_0^{\mathcal{A}}$ and $\mathbf{Hybrid}_1^{\mathcal{A}}$ are identical to the adversary. Note also that $T_{\mathcal{A},\lambda}$ is an upper bound on the number of queries that $\mathcal{A}$ can make on security parameter λ (and thus an upper bound on the actual values of (t^*, n)).

Since $\mathbf{Hybrid}_1^{\mathcal{A}}$ does not make use of (t^*, n) beyond its abort condition, then until or unless $\mathbf{Hybrid}_1^{\mathcal{A}}$ aborts, the distribution of the number of queries the adversary makes in $\mathbf{Hybrid}_1^{\mathcal{A}}$ is identical to that of $\mathbf{Hybrid}_0^{\mathcal{A}}$. Thus, we can argue that the probability that $\mathbf{Hybrid}_1^{\mathcal{A}}$ correctly guesses (t^*, n) is at least $\frac{1}{T_{\mathcal{A},\lambda}^2}$. In the up to $T_{\mathcal{A},\lambda}^3$ independent executions of $\mathbf{Hybrid}_1^{\mathcal{A}}$ in $\mathsf{Wrap}(\mathbf{Hybrid}_1^{\mathcal{A}})$, the probability that $\mathbf{Hybrid}_1^{\mathcal{A}}$ never guesses correctly is $\left(1 - \frac{1}{T_{\mathcal{A},\lambda}^2}\right)^{T_{\mathcal{A},\lambda}^3} \approx e^{-T_{\mathcal{A},\lambda}} = e^{-\mathsf{poly}(\lambda)} = \mathsf{negl}(\lambda)$. Thus, the outputs of $\mathbf{Hybrid}_0^{\mathcal{A}}$ and $\mathsf{Wrap}(\mathbf{Hybrid}_1^{\mathcal{A}})$ are negligibly close.

□

Hybrid$_2^{\mathcal{A}}$: This is identical to the previous hybrid except that we have unwrapped the definition of Pre-One-sFE.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{ct}_{\mathsf{inp},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.
   (d) **Compute $\sigma_{\mathsf{inp},i}$:**
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.
   (e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.
   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.
   (c) **Compute $\mathsf{ct}_{\mathsf{st},1}$:**
      i. $(r_{E,1}, r_{\mathsf{Enc},1}) = \mathsf{PPRF.Eval}(K_E, 1)$.
      ii. $k_{E,1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,1})$.
      iii. $\mathsf{ct}_{\mathsf{st},1} = \mathsf{SKE.Enc}(k_{E,1}, \mathsf{Post.Dec.st}_1; r_{\mathsf{Enc},1})$.
   (d) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.
   (e) **Compute $\sigma_{\mathsf{st},1}$:**
      i. $m_1 = (1, \mathsf{ct}_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$.
      ii. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.
      iii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1)$.
   (f) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}[\mathsf{Post\text{-}One\text{-}sFE.Dec}, K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}])$ where Prog is defined below.

(g) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute** $\mathsf{ct}_{\mathsf{inp},i}$:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute** $\sigma_{\mathsf{inp},i}$:

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}[\mathsf{Post\text{-}One\text{-}sFE.Dec}, K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

i. **Verify i is positive:** If $i \leq 0$, output $\perp$.

ii. **Verify input signature:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. **Verify state signature:**

i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.

2. **Computation Step:**

(a) **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

(b) **Compute output value and next state:**

i. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

(c) **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

   iii. **Sign the new state:**

      i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.7. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}_1^{\mathcal{A}}(1^\lambda), \mathbf{Hybrid}_2^{\mathcal{A}}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Replacing the encrypted stream values. Our goal is to iteratively replace the encryption of each element of stream $x^{(b)}$ with the corresponding value from $x^{(0)}$. We can replace the t^{th} ciphertext using the security of SKE as long as the keys and randomness used for encryption are hidden. However, these values are present in the obfuscated program since it needs them to decrypt the input and encrypt the new state. But, if we hardcode the output values for steps $t-1$ and t into the program, then the program will no longer need these secret keys and randomness so we can swap the t^{th} ciphertext.

Thus our first goal will be to hardwire two steps of the program. We start by hardwiring step 0 for free since the indexing starts at $t = 1$. In general, we proceed as follows:

1. Assume step $t-1$ is hardwired.
2. Hardwire step t.
3. Switch t^{th} ciphertext to stream $x^{(0)}$.
4. Un-hardwire step $t-1$.
5. Repeat process with $t = t+1$.

These define a series of hybrids for steps $t \in [t^*]$.

Hybrid$^{\mathcal{A}}_{3,t,0}$: For $i < t$, we now encrypt stream $x^{(0)}$ instead of stream $x^{(b)}$. Furthermore, inside Prog, at step $i = t-1$, rather than computing $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$ by decrypting, evaluating, and re-encrypting, we instead set them to hardwired values that we have pre-computed ahead of time. These hardwired values correspond to stream $x^{(0)}$ for $i < t$ and to stream $x^{(b)}$ for $i \geq t$.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.
   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (d) **Compute input ciphertexts**:
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.

iv. $\mathsf{ct}^{(b)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}^{(b)}_i)$.

(e) **Set hardwired values**:

i. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i \geq t \end{cases}$.

(f) **Compute input signature keys**: Do nothing. (Will be added in a later hybrid.)

(g) **Compute** $\sigma_{\mathsf{inp},i}$:

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

(h) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

6. **KeyGen:**

(a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \bot$.

(b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.

(c) $\mathsf{Post.Dec.st}^{(0)}_1 = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}^{(b)}_1 = \mathsf{Post.Dec.st}_1$.

(d) **Compute state and output values:**

i. For $i \in [t^*]$,

A. $(y_i, \mathsf{Post.Dec.st}^{(0)}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}^{(0)}_i, \mathsf{Post.CT}^{(0)}_i)$.

B. $(y_i, \mathsf{Post.Dec.st}^{(b)}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}^{(b)}_i, \mathsf{Post.CT}^{(b)}_i)$.[30]

(e) **Compute state ciphertexts:**

i. For $i \in [t^* + 1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}^{(0)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}^{(0)}_i; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}^{(b)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}^{(b)}_i; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y^*_i = y_i$.

ii. For $i \in [t^* + 1]$,

A. $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i < t \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i \geq t \end{cases}$

B. $\mathsf{itr}^*_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}^*_{\mathsf{st},i-1}, \mathsf{ct}^*_{\mathsf{st},i})$ where $\mathsf{itr}^*_{\mathsf{st},0} = \mathsf{itr}_{\mathsf{st},0}$.

C. $m^*_i = (i, \mathsf{ct}^*_{\mathsf{st},i}, \mathsf{itr}^*_{\mathsf{st},i-1})$.

(h) **Compute state signature keys**: Do nothing. (Will be added in a later hybrid.)

(i) **Compute** $\sigma_{\mathsf{st},1}$**:**

i. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

[30] Recall that if $\mathcal{A}$ submits a function or message query such that $f(x^{(0)}) \neq f(x^{(1)})$ for all stream values seen so far, then we immediately halt and output 0. Thus, if we reach this point in the hybrid, the i^{th} output value $y^{(0)}_i$ for stream $x^{(0)}$ is the same as the i^{th} output value $y^{(1)}_i$ for stream $x^{(1)}$ and we can denote both by $y_i = y^{(0)}_i = y^{(1)}_i$.

ii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1^*)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},0}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*])$.
(We can omit the function Post-One-sFE.Dec *from the parameters to the program as we always use the same function.)*

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}^*, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute** $\mathsf{ct}_{\mathsf{inp},i}$:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute** $\sigma_{\mathsf{inp},i}$:

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}_{3,\mathsf{t},0}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

i. **Verify i is positive:** If $i \leq 0$, output $\perp$.

ii. **Verify input signature:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. **Verify state signature:**

i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.

2. **Computation Step:**

i. If $i = t - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*)$.

ii. If $i \neq t - 1$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. **Sign the new state:**

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.8. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}_2^{\mathcal{A}}(1^\lambda), \mathbf{Hybrid}_{3,1,0}^{\mathcal{A}}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. Prog and $\mathsf{Prog}_{3,1,0}$ have the same input/output behavior since if $i = t - 1 = 0$, then both programs will output $\perp$ in the verification step before even reaching the computation step. Thus the $i = t - 1$ branch in $\mathsf{Prog}_{3,1,0}$ can never be reached.

Apart from the obfuscated program $\mathcal{P}$, the hybrids are identical since we encrypt stream $x^{(b)}$ for all values of $i \geq t = 1$.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Adding B type signatures. Our next goal is to hardwire step t into the obfuscated program. However, to do so, we first need to do some prepwork.

In the next series of hybrids, we will introduce an alternative signature scheme for the state, which we will call a B type signature scheme (since it uses K_B). The original state signature scheme will be called an A type signature scheme (since it use K_A). Our obfuscated program will now allow the state signature to verify under either an A type or B type signature and will sign the outgoing state with either an A type signature (if it verified with A type) or a B type signature (if it verified with B type).

Our eventual goal is to isolate a single computational path up to step t under the A type scheme and make it the only path that can be verified under the A type scheme. This will allow us to hardwire in step t since it lets us enforce the output at step t to be only one possible value (under A type signatures). However, we cannot simply abort on all other computational paths. Instead, we will use the B type signature scheme to provide an alternate path for all these other computational paths. Thus, since we wish to hardwire step t, we will have to add in B type signatures for each index $j \in [t-1]$. This defines a sequence of hybrids for each additional index j we want to add.

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j}$ = $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,0}$: For $i \in [j+1, t-1]$, if the state signature fails to verify under the original A type signature scheme, then rather than immediately outputting $\bot$, we check if it verifies under the B type signature scheme. If it verifies using a B type signature, then we will carry on the computation, and sign the outgoing message using the B type signature scheme.

This hybrid is the same as the previous hybrid except that

- We compute Setup as

  3. **Setup:**
     (a) $K_{\mathsf{inp}}, K_A, K_B, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
     (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}})$.

- During KeyGen, we make changes to the following steps:

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,0,j,0}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,0,j,0}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\bot$.
   ii. **Verify input signature:**
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \bot$.
   v. **Verify A type signatures:**
      i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. **Verify B type signatures:**

i. If in-type $\neq A$ and $j+1 \leq i \leq t-1$,
A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, in-type = out-type $= B$.

vii. If in-type $= \bot$, output $\bot$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
ii. If $i \neq t-1$,
i. **Decrypt input and state**:
i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
ii. **Compute output value and next state:**
A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
iii. **Encrypt the new state**:
i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
iii. **Sign A type messages:** If out-type $= A$,
i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
iv. **Sign B type messages:** If out-type $= B$,
i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.
ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.9. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,t-1}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. $\mathsf{Prog}_{3,\mathsf{t},0}$ and $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{t}-1}$ have identical input/output behavior since there is no i such that $j+1 = t \leq i \leq t-1$. Therefore, $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{t}-1}$ will never verify using B type signatures and thus will never use the B type branches.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,0,j,1}$: We introduce B type signatures at step j. However, we only verify at step j using $\mathsf{vk}_{B,j,\mathsf{rej}}$. Since this verification key always rejects, we do not change the behavior of the obfuscated program and can rely on the security of $i\mathcal{O}$.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,0,j,1}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,0,j,1}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\bot$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. $\mathsf{in\text{-}type} = \bot$.
   - v. **Verify A type signatures:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   - vi. **Verify B type signatures:**
     - i. If $\mathsf{in\text{-}type} \neq A$ and $i = j$,
       - A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i,\mathsf{rej}}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
     - ii. If $\mathsf{in\text{-}type} \neq A$ and $j + 1 \leq i \leq t - 1$,
       - A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
   - vii. If $\mathsf{in\text{-}type} = \bot$, output $\bot$.
2. **Computation Step:**
   - i. If $i = t - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   - ii. If $i \neq t - 1$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
iii. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,
   i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
   ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
iv. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,
   i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.
   ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.10. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,1}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. The only difference between $\mathsf{Prog}_{\mathsf{3,t,0,j,0}}$ and $\mathsf{Prog}_{\mathsf{3,t,0,j,1}}$ is that at step $i = j$, the latter will check if the state signature verifies under $\mathsf{vk}_{B,j,\mathsf{rej}}$. However, since verifying with $\mathsf{vk}_{B,j,\mathsf{rej}}$ which always results in a rejected signature, then the two programs have identical input/output behavior.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,0,j,2}$: We puncture key K_B at j. We verify B type signatures at step j using a hardwired verification key. Note that we do not need to sign any B type messages at step j.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

6h. **Compute state signature keys:**
  (a) $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
  (b) $r_{B,j} = \mathsf{PPRF.Eval}(K_B, j)$.
  (c) $(\mathsf{sgk}_{B,j}, \mathsf{vk}_{B,j}, \mathsf{vk}_{B,j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{B,j})$.
  (d) $\mathsf{vk}^*_{B,j} = \mathsf{vk}_{B,j,\mathsf{rej}}$.

6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,0,j,2}[K_{\mathsf{inp}}, K_A, K_B[j], \mathsf{vk}^*_{B,j}, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program
$\mathsf{Prog}_{3,t,0,j,2}[K_{\mathsf{inp}}, K_A, K_B[j], \mathsf{vk}^*_{B,j}, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \perp$.
   v. **Verify A type signatures:**
      i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. **Verify B type signatures:**
      i. If $\mathsf{in\text{-}type} \neq A$ and $i = j$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{B,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
      ii. If $\mathsf{in\text{-}type} \neq A$ and $j + 1 \leq i \leq t - 1$,
         A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
   vii. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.
2. **Computation Step:**
   i. If $i = t - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   ii. If $i \neq t - 1$,
      i. **Decrypt input and state**:
         i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
         ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
         iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

iv. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,
//Observe that this branch can only be reached if $j \le i \le t-1$.

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.11. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,1}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,2}(1^\lambda)) \le \mathsf{negl}(\lambda)$$

Proof. $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j},2}$ only evaluates $K_B[j]$ in two places:

- During verification, it evaluates $K_B[j]$ at index i for $j < i \le t-1$.
- During authentication, it evaluates $K_B[j]$ at index $i+1$ when $\mathsf{out\text{-}type} = B$. This can only occur if we have verified a B type signature, which can only happen when $j \le i \le t-1$.

Thus, $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j},2}$ only evaluates $K_B[j]$ at points $v \ne j$. By correctness of puncturing,

$$\mathsf{PPRF.Eval}(K_B[j], v) = \mathsf{PPRF.Eval}(K_B, v) \text{ for any } v \ne j.$$

Additionally the hardwired value $\mathsf{vk}^*_{B,j} = \mathsf{vk}_{B,j,\mathsf{rej}}$ is the same as what would have been computed in the previous hybrid.

Thus, $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j},1}$ and $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j},2}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,3}$: We compute $r_{B,j}$ as a random value. This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**
  (a) $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
  (b) $r_{B,j} \leftarrow \{0,1\}^\lambda$.
  (c) $(\mathsf{sgk}_{B,j}, \mathsf{vk}_{B,j}, \mathsf{vk}_{B,j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Sign}(1^\lambda; r_{B,j})$.
  (d) $\mathsf{vk}^*_{B,j} = \mathsf{vk}_{B,j,\mathsf{rej}}$.

Lemma C.12. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$, *all* $t \in [t^*]$, *all* $j \in [t-1]$, *and all* PPT *adversaries* $\mathcal{A}$,

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,2}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,3}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the selective pseudorandomness at punctured points property of our PPRF.

Observe that we can run both hybrids without knowing K_B as long as we are given $(K_B[j], r_{B,j})$. In the reduction, without computing K_B, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,3}$ up to just before step 6h of KeyGen. Then, we receive $(K_B[j], r_{B,j})$ from the PPRF challenger where $r_{B,j}$ is either a random value or equal to $\mathsf{PPRF.Eval}(K_B, j)$. We then use these values to run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,3}$ starting from step 6h(c) of KeyGen. Observe that if $r_{B,j}$ was a random value then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,3}$, and if $r_{B,j}$ was equal to $\mathsf{PPRF.Eval}(K_B, j)$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,2}$. Thus, by PPRF security, the outputs of these hybrids must be indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,4}$: We set vk^* to be $\mathsf{vk}_{B,j}$. This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**
  (a) $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
  (b) $r_{B,j} \leftarrow \{0,1\}^\lambda$.
  (c) $(\mathsf{sgk}_{B,j}, \mathsf{vk}_{B,j}, \mathsf{vk}_{B,j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Sign}(1^\lambda; r_{B,j})$.
  (d) $\mathsf{vk}^*_{B,j} = \mathsf{vk}_{B,j}$.

Lemma C.13. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$, *all* $t \in [t^*]$, *all* $j \in [t-1]$, *and all* PPT *adversaries* $\mathcal{A}$,

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,3}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,4}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the $\mathsf{vk}_{\mathsf{rej}}$ indistinguishabilty of SSig.

Observe that we can run both hybrids without knowing $(r_{B,j}, \mathsf{sgk}_{B,j}, \mathsf{vk}_{B,j}, \mathsf{vk}_{B,j,\mathsf{rej}})$ as long as we are given $\mathsf{vk}^*_{B,j}$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,4}$ up to just before step 6h(b) of KeyGen. Then, we receive vk^* from the SSig challenger where vk^* is either a regular SSig verification key vk or is a rejecting SSig verification key $\mathsf{vk}_{\mathsf{rej}}$. We then set $\mathsf{vk}^*_{B,j,\mathsf{rej}} = \mathsf{vk}^*$ and run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,4}$ starting from step 6i of KeyGen. Observe that if vk^* was a regular verification key we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,4}$, and if vk^* was a rejecting verification key we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,3}$. Thus, by the $\mathsf{vk}_{\mathsf{rej}}$ indistinguishabilty of SSig security, the outputs of these hybrids must be indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,5}$: We change $r_{B,j}$ back to a PPRF evaluation. This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6h **Compute state signature keys:**
  (a) $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
  (b) $r_{B,j} \leftarrow \mathsf{PPRF.Eval}(K_B, j)$.
  (c) $(\mathsf{sgk}_{B,j}, \mathsf{vk}_{B,j}, \mathsf{vk}_{B,j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Sign}(1^\lambda; r_{B,j})$.
  (d) $\mathsf{vk}^*_{B,j} = \mathsf{vk}_{B,j}$.

Lemma C.14. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$, *all* $t \in [t^*]$, *all* $j \in [t-1]$, *and all* PPT *adversaries* $\mathcal{A}$,

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,4}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,5}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the selective pseudorandomness at punctured points property of our PPRF.

Observe that we can run both hybrids without knowing K_B as long as we are given $(K_B[j], r_{B,j})$. In the reduction, without computing K_B, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,5}$ up to just before step 6h of KeyGen. Then, we receive $(K_B[j], r_{B,j})$ from the PPRF challenger where $r_{B,j}$ is either a random value or equal to $\mathsf{PPRF.Eval}(K_B, j)$. We then use these values to run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,5}$ starting from step 6h(c) of KeyGen. Observe that if $r_{B,j}$ was a random value then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,4}$, and if $r_{B,j}$ was equal to $\mathsf{PPRF.Eval}(K_B, j)$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,5}$. Thus, by PPRF security, the outputs of these hybrids must be indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,0,j,6}$: This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:** Do nothing.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,0,j,6}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,0,j,6}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\bot$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. $\mathsf{in\text{-}type} = \bot$.
   - v. **Verify A type signatures:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   - vi. **Verify B type signatures:**
     - i. If $\mathsf{in\text{-}type} \neq A$ and $i = j$,
       - A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
     - ii. If $\mathsf{in\text{-}type} \neq A$ and $j + 1 \leq i \leq t - 1$,
       - A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
   - vii. If $\mathsf{in\text{-}type} = \bot$, output $\bot$.
2. **Computation Step:**
   - i. If $i = t - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   - ii. If $i \neq t - 1$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i + 1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

   iii. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

      i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

   iv. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,
   //Observe that this branch can only be reached if $j \le i \le t-1$.

      i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.15. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,5}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,6}(1^\lambda)) \le \mathsf{negl}(\lambda)$$

Proof. The previous hybrid $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j},5}$ only evaluates $K_B[j]$ in two places:

- During verification, it evaluates $K_B[j]$ at index i for $j < i \le t-1$.
- During authentication, it evaluates $K_B[j]$ at index $i+1$ when $\mathsf{out\text{-}type} = B$. This can only occur if we have verified a B type signature, which can only happen when $j \le i \le t-1$.

Thus, $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j},5}$ only evaluates $K_B[j]$ at points $v \ne j$. By correctness of puncturing,

$$\mathsf{PPRF.Eval}(K_B[j], v) = \mathsf{PPRF.Eval}(K_B, v) \text{ for any } v \ne j.$$

Additionally the hardwired value $\mathsf{vk}^*_{B,j} = \mathsf{vk}_{B,j}$ of the previous hybrid is the same as the value that is computed in the current hybrid.

Thus, $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j},5}$ and $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j},6}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Lemma C.16. *If $i\mathcal{O}$ is a indistinguishability obfuscation, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j,5}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,j-1,0}(1^\lambda)) \le \mathsf{negl}(\lambda)$$

Proof. The hybrids are identical except that one hybrid obfuscates $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j},5}$ and the other hybrid obfuscates $\mathsf{Prog}_{3,\mathsf{t},0,\mathsf{j}-1,0}$. Observe that though the two programs have mild notational differences, they behave identically and have the same input/output behavior. Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1}$: This hybrid is identical to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,0,0}$. We have rewritten it here to make the hybrids easier to follow. We have highlighted the differences from $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0}$. Observe that we now have B type signatures for all $i \in [t-1]$.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $K_{\mathsf{inp}}, K_A, K_B, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.
   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (d) **Compute input ciphertexts**:
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.
   (e) **Set hardwired values**:
      i. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i \geq t \end{cases}$.
   (f) **Compute input signature keys**: Do nothing.
   (g) **Compute** $\sigma_{\mathsf{inp},i}$:
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.
   (h) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.
   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.
   (c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.
   (d) **Compute state and output values:**
      i. For $i \in [t^*]$,

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^* + 1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^* + 1]$,

A. $\mathsf{ct}_{\mathsf{st},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{st},i}^{(0)} & \text{if } i < t \\ \mathsf{ct}_{\mathsf{st},i}^{(b)} & \text{if } i \geq t \end{cases}$

B. $\mathsf{itr}_{\mathsf{st},i}^* = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}^*, \mathsf{ct}_{\mathsf{st},i}^*)$ where $\mathsf{itr}_{\mathsf{st},0}^* = \mathsf{itr}_{\mathsf{st},0}$.

C. $m_i^* = (i, \mathsf{ct}_{\mathsf{st},i}^*, \mathsf{itr}_{\mathsf{st},i-1}^*)$.

(h) **Compute state signature keys**: Do nothing.

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

ii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1^*)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},1}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}^*, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}_{3,t,1}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\bot$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. in-type $= \bot$.
   - v. **Verify A type signatures:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, in-type $=$ out-type $= A$.
   - vi. **Verify B type signatures:**
     - i. If in-type $\neq A$ and $1 \leq i \leq t-1$,
       - A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, in-type $=$ out-type $= B$.
   - vii. If in-type $= \bot$, output $\bot$.
2. **Computation Step:**
   - i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   - ii. If $i \neq t-1$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   - i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   - ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   - iii. **Sign A type messages:** If out-type $= A$,
     - i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
     - ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

iv. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.17. *For all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0,0,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Tracing the computation through the program. Again, our next goal is to hardwire step t into the obfuscated program. Since we are using $i\mathcal{O}$, we can only hardwire in this step if the hardwired values are the only possible values that could be computed in that portion of the program. Thus, we want to enforce that only one set of incoming messages at step t can pass the verification step. This can be accomplished as long as we only ever sign one message set with the t^{th} signature scheme since then the properties of SSig allow us to replace the verification key with one that only verifies this single value.

Unfortunately, the key K_A for the state signatures is embedded in the obfuscated program since we need to sign outgoing states. Somehow, we will need to modify our program so that the t^{th} state signature scheme can only sign one possible state. This is the focus of our next set of hybrids.

At index $t = 1$, this is easy to do since the program does not need $\mathsf{sgk}_{A,1}$ since it only signs *outgoing* states. For indices $t > 1$, it is more complicated. At a high level, we will need to trace our computation path up from step 1 to t. At each step, we will move every state message other than the chosen one to the B type scheme. Then, we can use the single state isolated in the A type scheme to enforce the next step of computation. Once we make it up to step t, we can finally hardwire the computation.

In more detail. First, we will add a conditional statement to our program (described below) at step 0. This can be added for free at step 0 since our indexing starts at $i = 1$. In general, we will proceed as follows:

1. We assume that at step $i = j - 1$, we have a conditional statement that sets $\mathsf{out\text{-}type} = A$ if and only if the outgoing message m_{i+1} is our chosen message m^*_j.

2. Our conditional statement ensures that we only use the A type scheme at j to sign one message m^*_j. Thus, we can use properties of SSig to change the verification key of our A type scheme at j to only verify this one message. (We can also ensure that our B type scheme at j cannot verify this one message.)

3. Since our A type scheme can only verify m^*_j (and the B type scheme cannot), we can add a conditional statement at step $i = j$ that sets the $\mathsf{out\text{-}type}$ to A if and only if the *incoming* message m_i is equal to m^*_j.

4. To clean up the conditional statement at step $i = j - 1$, we will merge the A and B type signature schemes at index j using the properties of SSig. Thus, the $\mathsf{out\text{-}type}$ at $i = j - 1$ will always be A and the conditional will not be needed.

5. Finally, we will change the conditional at step j so that it sets the $\mathsf{out\text{-}type}$ to A if and only if the *outgoing* message m_{i+1} is our chosen message m^*_{j+1}. This will require using our iterator and signature schemes to enforce that the incoming message is m^*_j if and only if the outgoing message is m^*_{j+1}.

6. We then repeat the process with $j = j + 1$.

This defines a series of hybrids for $j \in [t]$.

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j} = \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,0}$: We now only verify with B type signatures for steps $i \in [j, t-1]$. Furthermore, at step $i = j-1$, if the outgoing message is not our chosen value m^*_j, then we sign it with a B type signature.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,1,j,0}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,1,j,0}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. $\mathsf{in\text{-}type} = \perp$.
   - v. **Verify A type signatures:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   - vi. **Verify B type signatures:**
     - i. If $\mathsf{in\text{-}type} \neq A$ and $j \leq i \leq t-1$,
       - A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
   - vii. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.
2. **Computation Step:**
   - i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   - ii. If $i \neq t-1$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

   iii. If $i = j-1$,

      i. If $m_{i+1} = m_j^*$, out-type $= A$. Else, out-type $= B$.

   iv. **Sign A type messages:** If out-type $= A$,

      i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

   v. **Sign B type messages:** If out-type $= B$,

      i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.18. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,1}(1^\lambda)) \leq \mathsf{negl}(1^\lambda)$$

Proof. $\mathsf{Prog}_{3,t,0}$ and $\mathsf{Prog}_{3,t,1,1}$ have the same input/output behavior since if $i = j-1 = 0$, then both programs will output $\perp$ in the verification step before even reaching the authentication step. Thus the $i = j-1$ branch in $\mathsf{Prog}_{3,t,1,1}$ can never be reached.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,1,j,1}$: We puncture both K_A and K_B at index j. We sign and verify A and B type signatures at step j using hardwired keys.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

6h. **Compute state signature keys:**
1. $K_A[j] = \mathsf{PPRF.Punc}(K_A, j)$.
2. $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
3. For $\alpha \in \{A, B\}$,
   (a) $r_{\alpha,j} \leftarrow \mathsf{PPRF.Eval}(K_\alpha, j)$.
   (b) $(\mathsf{sgk}_{\alpha,j}, \mathsf{vk}_{\alpha,j}, \mathsf{vk}_{\alpha,j,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; r_{\alpha,j})$.
4. $(\sigma^*_{\mathsf{st},j}, \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\bot, \mathsf{sgk}_{A,j}, \mathsf{vk}_{A,j})$.
5. $(\mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}) = (\mathsf{sgk}_{B,j}, \mathsf{vk}_{B,j})$.

6i. **Compute $\sigma_{\mathsf{st},1}$:**
1. If $1 = j$, $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,1}, m^*_1)$.
2. If $1 \neq j$,
   (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], 1))$.
   (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,1,j,1}[K_{\mathsf{inp}}, K_A[j], \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}, K_B[j], \mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,1,j,1}[K_{\mathsf{inp}}, K_A[j], \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}, K_B[j], \mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\bot$.
   ii. **Verify input signature:**
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \bot$.
   v. **Verify A type signatures:**
      i. If $i = j$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
      ii. If $i \neq j$,
         A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. **Verify B type signatures:**
      i. If $\mathsf{in\text{-}type} \neq A$ and $i = j$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{B,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
      ii. If $\mathsf{in\text{-}type} \neq A$ and $j + 1 \leq i \leq t - 1$,

A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.

vii. If $\mathsf{in\text{-}type} = \bot$, output $\bot$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i \neq t-1$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = j-1$,

i. If $m_{i+1} = m^*_j$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. If $i = j-1$,

A. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,i+1}, m_{i+1})$.

ii. If $i \neq j-1$,

A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. If $i = j-1$,

A. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{B,j}, m_{i+1})$.

ii. If $i \neq j-1$,

A. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.19. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all*

$t \in [t^*]$, *all* $j \in [t-1]$, *and all* PPT *adversaries* $\mathcal{A}$,

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,1}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing, for $\alpha \in \{A, B\}$,

$$\mathsf{PPRF.Eval}(K_\alpha[j], v) = \mathsf{PPRF.Eval}(K_\alpha, v) \text{ for any } v \neq j.$$

Observe that we never evaluate punctured keys on their punctured points. Furthermore, the hard-wired signing and verification keys are set to what they would have been computed to be in the previous hybrid. Thus, $\mathsf{Prog}_{3,t,1,j,0}$ and $\mathsf{Prog}_{3,t,1,j,1}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,1,j,2}$: We split both the A and B type signatures at index j on our chosen message m^*_j. We replace $\mathsf{sgk}^*_{B,j}$ with $\mathsf{sgk}_{B,\mathsf{abo},j}$ and replace signatures using $\mathsf{sgk}^*_{A,j}$ with $\sigma_{A,\mathsf{one},j}$.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

6h. **Compute state signature keys:**

1. $K_A[j] = \mathsf{PPRF.Punc}(K_A, j)$.
2. $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
3. For $\alpha \in \{A, B\}$,
   (a) $r_{\alpha,j} \leftarrow \mathsf{PPRF.Eval}(K_\alpha, j)$.
   (b) $(\mathsf{sgk}_{\alpha,j}, \mathsf{vk}_{\alpha,j}, \mathsf{vk}_{\alpha,j,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; r_{\alpha,j})$.
   (c) $(\sigma_{\alpha,\mathsf{one},j}, \mathsf{vk}_{\alpha,\mathsf{one},j}, \mathsf{sgk}_{\alpha,\mathsf{abo},j}, \mathsf{vk}_{\alpha,\mathsf{abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\alpha,j}, m^*_j)$.
4. $(\sigma^*_{\mathsf{st},j}, \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\sigma_{A,\mathsf{one},j}, \bot, \mathsf{vk}_{A,j})$.
5. $(\mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}) = (\mathsf{sgk}_{B,\mathsf{abo},j}, \mathsf{vk}_{B,j})$.

6i. **Compute $\sigma_{\mathsf{st},1}$:**

1. If $1 = j$, $\sigma_{\mathsf{st},1} = \sigma^*_{\mathsf{st},j}$.
2. If $1 \neq j$,
   (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], 1))$.
   (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,1,j,2}[K_{\mathsf{inp}}, K_A[j], \sigma^*_{A,j}, \mathsf{vk}^*_{A,j}, K_B[j], \mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,1,j,2}[K_{\mathsf{inp}}, K_A[j], \sigma^*_{A,j}, \mathsf{vk}^*_{A,j}, K_B[j], \mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

i. **Verify i is positive:** If $i \leq 0$, output $\bot$.

ii. **Verify input signature:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. $\mathsf{in\text{-}type} = \bot$.

v. **Verify A type signatures:**

i. If $i = j$,
   A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
ii. If $i \neq j$,
   A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], i))$.
   B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. **Verify B type signatures:**

i. If $\mathsf{in\text{-}type} \neq A$ and $i = j$,
   A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{B,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.

ii. If $\mathsf{in\text{-}type} \neq A$ and $j+1 \leq i \leq t-1$,
   A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i))$.
   B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.

vii. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i \neq t-1$,

   i. **Decrypt input and state**:
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
      iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
   ii. **Compute output value and next state:**
      A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
   iii. **Encrypt the new state**:
      i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
      ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
      iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = j-1$,
   i. If $m_{i+1} = m^*_j$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,
   i. If $i = j-1$,
      A. $\sigma_{\mathsf{st},i+1} = \sigma^*_{\mathsf{st},j}$.
   ii. If $i \neq j-1$,
      A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], i+1))$.
      B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,
   i. If $i = j-1$,
      A. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{B,j}, m_{i+1})$.
   ii. If $i \neq j-1$,
      A. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i+1))$.
      B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.20. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all*

$t \in [t^*]$, *all* $j \in [t-1]$, *and all* PPT *adversaries* $\mathcal{A}$,

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,1}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,2}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of SSig, if $(\sigma_{A,\mathsf{one},j}, \mathsf{vk}_{A,\mathsf{one},j}, \mathsf{sgk}_{A,\mathsf{abo},j}, \mathsf{vk}_{A,\mathsf{abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,j}, m^*_j)$ and $(\sigma_{B,\mathsf{one},j}, \mathsf{vk}_{B,\mathsf{one},j}, \mathsf{sgk}_{B,\mathsf{abo},j}, \mathsf{vk}_{B,\mathsf{abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{B,j}, m^*_j)$, then

$$\sigma_{A,\mathsf{one},j} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,j}, m^*_j).$$

$$\forall m \neq m^*_j,\ \mathsf{SSig.Sign}(\mathsf{sgk}_{B,j}, m) = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,\mathsf{abo},j}, m).$$

Thus, apart from the obfuscated programs, the hybrids act identically since $\sigma_{A,\mathsf{one},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$ so step 6i of KeyGen results in the same signature.

Now, in the previous hybrid, $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},1}$ only used $\mathsf{sgk}^*_{A,j} = \mathsf{sgk}_{A,j}$ and $\mathsf{sgk}^*_{B,j} = \mathsf{sgk}_{B,j}$ in two places:

- If $\mathsf{out\text{-}type} = A$ and $i = j-1$, then it signed m_{i+1} with $\mathsf{sgk}_{A,j}$. However, we can only have $\mathsf{out\text{-}type} = A$ at $i = j-1$ if $m_{i+1} = m^*_j$. Thus, replacing the signature with $\sigma_{A,\mathsf{one},j}$ does not change the behavior of the program.
- If $\mathsf{out\text{-}type} = B$ and $i = j-1$, then it signed m_{i+1} with $\mathsf{sgk}_{B,j}$. However, we can only have $\mathsf{out\text{-}type} = B$ at $i = j-1$ if $m_{i+1} \neq m^*_j$. Thus, signing instead with $\mathsf{sgk}_{B,\mathsf{abo},j}$ does not change the behavior of the program.

Therefore, $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},1}$ and $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},2}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,3}$: We change $r_{A,j}$ and $r_{B,j}$ to random values. This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[j] = \mathsf{PPRF.Punc}(K_A, j)$.
  2. $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
  3. For $\alpha \in \{A, B\}$,
     (a) $r_{\alpha,j} \leftarrow \{0,1\}^\lambda$.
     (b) $(\mathsf{sgk}_{\alpha,j}, \mathsf{vk}_{\alpha,j}, \mathsf{vk}_{\alpha,j,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; r_{\alpha,j})$.
     (c) $(\sigma_{\alpha,\mathsf{one},j}, \mathsf{vk}_{\alpha,\mathsf{one},j}, \mathsf{sgk}_{\alpha,\mathsf{abo},j}, \mathsf{vk}_{\alpha,\mathsf{abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\alpha,j}, m^*_j)$.
  4. $(\sigma^*_{\mathsf{st},j}, \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\sigma_{A,\mathsf{one},j}, \bot, \mathsf{vk}_{A,j})$.
  5. $(\mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}) = (\mathsf{sgk}_{B,\mathsf{abo},j}, \mathsf{vk}_{B,j})$.

Lemma C.21. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, all* $j \in [t-1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,2}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,3}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by two reductions to the selective pseudorandomness at punctured points property of our PPRF.

We first swap out only $r_{A,j}$ for a random value and leave $r_{B,j}$ as a PPRF evaluation. We call this intermediate hybrid $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,2.5}$.

Observe that we can run both the previous hybrid and the intermediate hybrid without knowing K_A as long as we are given $(K_A[j], r_{A,j})$. In the reduction, without computing K_A, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,2.5}$ up to just before step 6h of KeyGen. We also run the parts of step 6h that only deal with B type signatures. Then, we receive $(K_A[j], r_{A,j})$ from the PPRF challenger where $r_{A,j}$ is either a random value or equal to $\mathsf{PPRF.Eval}(K_A, j)$. We use this randomness to compute $(\sigma^*_{\mathsf{st},j}, \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j})$. We then run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,2.5}$ starting from step 6i of KeyGen. Observe that if $r_{A,j}$ was a random value then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,2.5}$, and if $r_{A,j}$ was equal to $\mathsf{PPRF.Eval}(K_A, j)$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,2}$. Thus, by PPRF security, the outputs of these hybrids must be indistinguishable.

By a similar reduction on the B type signature scheme, $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,2.5}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,3}$ are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,4}$: We set $\mathsf{vk}^*_{A,j}$ to $\mathsf{vk}_{A,\mathsf{one},j}$ which will only verify m^*_j. This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[j] = \mathsf{PPRF.Punc}(K_A, j)$.
  2. $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
  3. For $\alpha \in \{A, B\}$,
     (a) $r_{\alpha,j} \leftarrow \{0,1\}^\lambda$.
     (b) $(\mathsf{sgk}_{\alpha,j}, \mathsf{vk}_{\alpha,j}, \mathsf{vk}_{\alpha,j,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; r_{\alpha,j})$.
     (c) $(\sigma_{\alpha,\mathsf{one},j}, \mathsf{vk}_{\alpha,\mathsf{one},j}, \mathsf{sgk}_{\alpha,\mathsf{abo},j}, \mathsf{vk}_{\alpha,\mathsf{abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\alpha,j}, m^*_j)$.
  4. $(\sigma^*_{\mathsf{st},j}, \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\sigma_{A,\mathsf{one},j}, \bot, \mathsf{vk}_{A,\mathsf{one},j})$.
  5. $(\mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}) = (\mathsf{sgk}_{B,\mathsf{abo},j}, \mathsf{vk}_{B,j})$.

Lemma C.22. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, all* $j \in [t-1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,3}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,4}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig.

Observe that we can run both hybrids without knowing $(r_{A,j}, \mathsf{sgk}_{A,j}, \mathsf{vk}_{A,j}, \mathsf{vk}_{A,j,\mathsf{rej}})$ as long as we are given $(\sigma^*_{\mathsf{st},j}, \mathsf{vk}^*_{A,j})$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,4}$ up to just before step 6h.3. of KeyGen. We also run the parts of step 6h that only deal with B type signatures. We send m^*_j to the SSig challenger and receive $(\sigma^*, \mathsf{vk}^*)$ from the SSig challenger where σ^* is a signature of m^*_j and vk^* is either a verification key $\mathsf{vk}_{\mathsf{one}}$ that only verifies m^*_j or is a regular verification key vk. We then set $(\sigma^*_{\mathsf{st},j}, \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\sigma^*, \bot, \mathsf{vk}^*)$ and run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,4}$ starting from step 6i of KeyGen. Observe that if vk^* was a regular verification key vk we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,3}$, and if vk^* was $\mathsf{vk}_{\mathsf{one}}$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,4}$. Thus, by the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig security, the outputs of these hybrids must be indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,5}$: We set $\mathsf{vk}^*_{B,j}$ to $\mathsf{vk}_{B,\mathsf{abo},j}$ which cannot verify m^*_j. This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[j] = \mathsf{PPRF.Punc}(K_A, j)$.
  2. $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
  3. For $\alpha \in \{A, B\}$,
     (a) $r_{\alpha,j} \leftarrow \{0,1\}^\lambda$.
     (b) $(\mathsf{sgk}_{\alpha,j}, \mathsf{vk}_{\alpha,j}, \mathsf{vk}_{\alpha,j,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; r_{\alpha,j})$.
     (c) $(\sigma_{\alpha,\mathsf{one},j}, \mathsf{vk}_{\alpha,\mathsf{one},j}, \mathsf{sgk}_{\alpha,\mathsf{abo},j}, \mathsf{vk}_{\alpha,\mathsf{abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\alpha,j}, m^*_j)$.
  4. $(\sigma^*_{\mathsf{st},j}, \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\sigma_{A,\mathsf{one},j}, \bot, \mathsf{vk}_{A,\mathsf{one},j})$.
  5. $(\mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}) = (\mathsf{sgk}_{B,\mathsf{abo},j}, \mathsf{vk}_{B,\mathsf{abo},j})$.

Lemma C.23. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, all* $j \in [t-1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,4}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,5}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the $\mathsf{vk}_{\mathsf{abo}}$ indistinguishabilty of SSig.

Observe that we can run both hybrids without knowing $(r_{B,j}, \mathsf{sgk}_{B,j}, \mathsf{vk}_{B,j}, \mathsf{vk}_{B,j,\mathsf{rej}})$ as long as we are given $(\mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j})$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,5}$ up to just before step 6h.3. of KeyGen. We also run the parts of step 6h that only deal with A type signatures. We send m^*_j to the SSig challenger and receive $(\mathsf{sgk}^*, \mathsf{vk}^*)$ from the SSig challenger where sgk^* is a signing key $\mathsf{sgk}_{\mathsf{abo}}$ for signing every message except m^*_j and vk^* is either a verification key $\mathsf{vk}_{\mathsf{abo}}$ that verifies every mesage except m^*_j or is a regular verification key vk. We then set $(\mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}) = (\mathsf{sgk}^*, \mathsf{vk}^*)$ and run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,5}$ starting from step 6i of KeyGen. Observe that if vk^* was a regular verification key vk we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,4}$, and if vk^* was $\mathsf{vk}_{\mathsf{abo}}$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,5}$. Thus, by the $\mathsf{vk}_{\mathsf{abo}}$ indistinguishabilty of SSig security, the outputs of these hybrids must be indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,1,j,6}$: We add a conditional statement at step $i = j$ based on the incoming message. At $i = j$, if the incoming message m_i is equal to m^*_j, we will set $\mathsf{out\text{-}type} = A$. Otherwise, we will set $\mathsf{out\text{-}type} = B$.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,1,j,6}[K_{\mathsf{inp}}, K_A[j], \sigma^*_{A,j}, \mathsf{vk}^*_{A,j}, K_B[j], \mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,1,j,6}[K_{\mathsf{inp}}, K_A[j], \sigma^*_{A,j}, \mathsf{vk}^*_{A,j}, K_B[j], \mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. $\mathsf{in\text{-}type} = \perp$.
   - v. **Verify A type signatures:**
     - i. If $i = j$,
       - A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
     - ii. If $i \neq j$,
       - A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   - vi. **Verify B type signatures:**
     - i. If $\mathsf{in\text{-}type} \neq A$ and $i = j$,
       - A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{B,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
     - ii. If $\mathsf{in\text{-}type} \neq A$ and $j + 1 \leq i \leq t - 1$,
       - A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
   - vii. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.
2. **Computation Step:**
   - i. If $i = t - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   - ii. If $i \neq t - 1$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**
   A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:
   i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
   ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
   iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

   iii. If $i = j-1$,
      i. If $m_{i+1} = m^*_j$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

   iv. If $i = j$,
      i. If $m_i = m^*_j$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

   v. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,
      i. If $i = j-1$,
         A. $\sigma_{\mathsf{st},i+1} = \sigma^*_{\mathsf{st},j}$.
      ii. If $i \neq j-1$,
         A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], i+1))$.
         B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

   vi. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,
      i. If $i = j-1$,
         A. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{B,j}, m_{i+1})$.
      ii. If $i \neq j-1$,
         A. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i+1))$.
         B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.24. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,5}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,6}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},5}$ and $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},6}$ can only differ when $i = j$. If $i = j$ and $m_i = m^*_j$, both programs must have $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$ since they verify B type signatures at $i = j$ with $\mathsf{vk}^*_{B,j} = \mathsf{vk}_{B,\mathsf{abo},j}$ which always rejects $m_i = m^*_j$. If $i = j$ and $m_i \neq m^*_j$, then they must have $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$ since they verify A type signatures at $i = j$ with $\mathsf{vk}^*_{A,j} = \mathsf{vk}_{A,\mathsf{one},j}$ which always rejects $m_i \neq m^*_j$. Thus, the additional code we have added into $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},6}$ does not change its behavior relative to the previous program.

Therefore, the programs have identical input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,7}$: We begin the process of merging the two signature schemes at index j by replacing the B type keys at j with the corresponding A type keys.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[j] = \mathsf{PPRF.Punc}(K_A, j)$.
  2. $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
  3. For $\alpha = A$,
     (a) $r_{\alpha,j} \leftarrow \{0,1\}^\lambda$.
     (b) $(\mathsf{sgk}_{\alpha,j}, \mathsf{vk}_{\alpha,j}, \mathsf{vk}_{\alpha,j,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; r_{\alpha,j})$.
     (c) $(\sigma_{\alpha,\mathsf{one},j}, \mathsf{vk}_{\alpha,\mathsf{one},j}, \mathsf{sgk}_{\alpha,\mathsf{abo},j}, \mathsf{vk}_{\alpha,\mathsf{abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\alpha,j}, m^*_j)$.
  4. $(\sigma^*_{\mathsf{st},j}, \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\sigma_{A,\mathsf{one},j}, \bot, \mathsf{vk}_{A,\mathsf{one},j})$.
  5. $(\mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}) = (\mathsf{sgk}_{A,\mathsf{abo},j}, \mathsf{vk}_{A,\mathsf{abo},j})$.

Lemma C.25. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$, *all* $t \in [t^*]$, *all* $j \in [t-1]$, *and all* PPT *adversaries* $\mathcal{A}$,

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,6}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,7}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the splitting indistinguishabilty of SSig.

Observe that we can run both hybrids without knowing $(r_{\alpha,j}, \mathsf{sgk}_{\alpha,j}, \mathsf{vk}_{\alpha,j}, \mathsf{vk}_{\alpha,j,\mathsf{rej}})$ for $\alpha \in \{A, B\}$ as long as we are given $(\sigma^*_{\mathsf{st},j}, \mathsf{vk}^*_{A,j}, \mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j})$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,7}$ up to just before step 6h.3. of KeyGen. We send m^*_j to the SSig challenger and receive $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}'_{\mathsf{abo}}, \mathsf{vk}'_{\mathsf{abo}})$ where $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}})$ and $(\mathsf{sgk}'_{\mathsf{abo}}, \mathsf{vk}'_{\mathsf{abo}})$ are either from the same signature scheme or two independent ones and were computed by splitting on m^*_j. We set $(\sigma^*_{\mathsf{st},j}, \mathsf{vk}^*_{A,j}, \mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}) = (\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}'_{\mathsf{abo}}, \mathsf{vk}'_{\mathsf{abo}})$ and run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,7}$ starting from step 6i of KeyGen. Observe that if the keys were from the same signature scheme, then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,7}$, and if they were from two independent schemes, then we we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,6}$. Thus, by the splitting indistinguishabilty of SSig security, the outputs of these hybrids must be indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,1,j,8}$: We complete the merge of the signature schemes at j. For index j, we no longer split the signature on m^*_j and we remove all references to B type signatures.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

6h. **Compute state signature keys:**

1. $K_A[j] = \mathsf{PPRF.Punc}(K_A, j)$.
2. $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
3. For $\alpha = A$,
   (a) $r_{\alpha,j} \leftarrow \{0,1\}^\lambda$.
   (b) $(\mathsf{sgk}_{\alpha,j}, \mathsf{vk}_{\alpha,j}, \mathsf{vk}_{\alpha,j,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; r_{\alpha,j})$.
   (c) ~~$(\sigma_{\alpha,\mathsf{one},j}, \mathsf{vk}_{\alpha,\mathsf{one},j}, \mathsf{sgk}_{\alpha,\mathsf{abo},j}, \mathsf{vk}_{\alpha,\mathsf{abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\alpha,j}, m^*_j)$.~~
4. ~~$(\sigma^*_{\mathsf{st},j}, \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\sigma_{A,\mathsf{one},j}, \perp, \mathsf{vk}_{A,\mathsf{one},j})$.~~
5. ~~$(\mathsf{sgk}^*_{B,j}, \mathsf{vk}^*_{B,j}) = (\mathsf{sgk}_{A,\mathsf{abo},j}, \mathsf{vk}_{A,\mathsf{abo},j})$.~~
6. $(\mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\mathsf{sgk}_{A,j}, \mathsf{vk}_{A,j})$.

6i. **Compute $\sigma_{\mathsf{st},1}$:**

1. If $1 = j$, $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,1}, m^*_1)$.
2. If $1 \neq j$,
   (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], 1))$.
   (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,1,j,8}[K_{\mathsf{inp}}, K_A[j], \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}, K_B[j], K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,1,j,8}[K_{\mathsf{inp}}, K_A[j], \mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}, K_B[j], K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \perp$.
   v. **Verify A type signatures:**
      i. If $i = j$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
      ii. If $i \neq j$,
         A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. **Verify B type signatures:**
      i. ~~If $\mathsf{in\text{-}type} \neq A$ and $i = j$,~~

A. ~~If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{B,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, in-type = out-type = B.~~

ii. If in-type $\neq A$ and $j+1 \leq i \leq t-1$,

A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, in-type = out-type = B.

vii. If in-type = $\perp$, output $\perp$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i \neq t-1$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. ~~If $i = j-1$,~~

i. ~~If $m_{i+1} = m^*_j$, out-type = A. Else, out-type = B.~~

iv. If $i = j$,

i. If $m_i = m^*_j$, out-type = A. Else, out-type = B.

v. **Sign A type messages:** If out-type = A,

i. If $i = j-1$,

A. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,j}, m_{i+1})$.

ii. If $i \neq j-1$,

A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[j], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

vi. **Sign B type messages:** If out-type = B,
//Observe that this branch can only be reached if $j \leq i \leq t-1$.

i. ~~If $i = j-1$,~~

A. ~~$\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{B,j}, m_{i+1})$.~~

ii. ~~If $i \neq j-1$,~~

A. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B[j], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.26. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,7}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,8}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of SSig, if $(\mathsf{sgk}_{A,j}, \mathsf{vk}_{A,j}, \mathsf{vk}_{A,j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda)$ and $(\sigma_{A,\mathsf{one},j}, \mathsf{vk}_{A,\mathsf{one},j}, \mathsf{sgk}_{A,\mathsf{abo},j}, \mathsf{vk}_{A,\mathsf{abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,j}, m^*_j)$, then

$$\sigma_{A,\mathsf{one},j} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,j}, m^*_j).$$

$$\forall m \neq m^*_j,\ \mathsf{SSig.Sign}(\mathsf{sgk}_{A,\mathsf{abo},j}, m) = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,j}, m).$$

$$\forall \sigma,\ \mathsf{SSig.Verify}(\mathsf{vk}_{A,\mathsf{one},j}, m^*, \sigma) = \mathsf{SSig.Verify}(\mathsf{vk}_{A,j}, m^*, \sigma).$$

$$\forall m \neq m^* \text{ and } \sigma,\ \mathsf{SSig.Verify}(\mathsf{vk}_{A,\mathsf{abo},j}, m, \sigma) = \mathsf{SSig.Verify}(\mathsf{vk}_{A,j}, m, \sigma).$$

Thus, apart from the obfuscated programs, the hybrids act identically since $\sigma_{A,\mathsf{one},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$ so step 6i of KeyGen results in the same signature.

Now we need to argue that $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},7}$ and $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},8}$ have the same input/output behavior. We will show this by carefully going through each of the relevant cases where they could differ. Note that the computation steps of both programs are unaffected by these differences, so we will omit the computation step from the analysis below.

- **Case 1:** $i = j-1$ and $m_{i+1} = m^*_j$.

	Previous Hybrid	**Current Hybrid**
Verification Step	Same behavior as the other hybrid since $i < j$.	
Authentication Step	Will have $\mathsf{out\text{-}type} = A$ since $i = j-1$ and $m_{i+1} = m^*_j$, and thus will output $\sigma^*_{\mathsf{st},j} = \sigma_{A,\mathsf{one},j}$.	Will have $\mathsf{out\text{-}type} = A$ since there is no B type verification branch for $i = j-1$, and thus will sign the outgoing message using $\mathsf{sgk}^*_{A,j} = \mathsf{sgk}_{A,j}$. This results in the same signature as $\sigma_{A,\mathsf{one},j}$ since $m_{i+1} = m^*_j$.

- **Case 2:** $i = j-1$ and $m_{i+1} \neq m^*_{j-1}$.

	Previous Hybrid	**Current Hybrid**
Verification Step	Same behavior as the other hybrid since $i < j$.	
Authentication Step	Will have $\mathsf{out\text{-}type} = B$ since $i = j-1$ but $m_{i+1} \neq m^*_j$, and thus will sign the outgoing message using $\mathsf{sgk}^*_{B,j} = \mathsf{sgk}_{A,\mathsf{abo},j}$.	Will have $\mathsf{out\text{-}type} = A$ since there is no B type verification branch for $i = j-1$, and thus will sign the outgoing message using $\mathsf{sgk}^*_{A,j} = \mathsf{sgk}_{A,j}$. This signature is the same as signing with $\mathsf{sgk}_{A,\mathsf{abo},j}$ since $m_{i+1} \neq m^*_j$.

- **Case 3:** $i = j$ and $m_i = m_j^*$.

	Previous Hybrid	**Current Hybrid**
Verification Step	Must verify using an A type signature since the B type verification uses $\mathsf{vk}^*_{B,j} = \mathsf{vk}_{A,\mathsf{abo},j}$ which always rejects on m_j^*. In the A branch, tries to verify the incoming message using $\mathsf{vk}_{A,\mathsf{one},j}$.	Must verify using an A type signature since the B type verification branch for $i = j$ has been removed. In the A branch, tries to verify the incoming message using $\mathsf{vk}^*_{A,j} = \mathsf{vk}_{A,j}$ which gives the same verification output on m_j^* as $\mathsf{vk}_{A,\mathsf{one},j}$
Authentication Step	Will have $\mathsf{out\text{-}type} = A$ since $i = j$ and $m_i = m_j^*$, and thus will sign the outgoing message using $\mathsf{sgk}_{A,j}$.	Will have $\mathsf{out\text{-}type} = A$ since $i = j$ and $m_i = m_j^*$, and thus will sign the outgoing message using $\mathsf{sgk}_{A,j}$.

- **Case 4:** $i = j$ and $m_i \neq m_j^*$.

	Previous Hybrid	**Current Hybrid**
Verification Step	Must verify using a B type signature since the A type verification uses $\mathsf{vk}^*_{A,j} = \mathsf{vk}_{A,\mathsf{one},j}$ which always rejects on m_j^*. In the B branch, tries to verify the incoming message using $\mathsf{vk}^*_{B,j} = \mathsf{vk}_{A,\mathsf{abo},j}$.	Must verify using an A type signature since the B type verification branch for $i = j$ has been removed. In the A branch, tries to verify the incoming message using $\mathsf{vk}^*_{A,j} = \mathsf{vk}_{A,j}$ which gives the same verification output on $m_i \neq m_j^*$ as $\mathsf{vk}_{A,\mathsf{abo},j}$.
Authentication Step	Will have $\mathsf{out\text{-}type} = A$ (even though $\mathsf{in\text{-}type} = B$) since $i = j$ and $m_i = m_j^*$, and thus will sign the outgoing message using $\mathsf{sgk}_{A,j}$.	Will have $\mathsf{out\text{-}type} = A$ since $i = j$ and $m_i = m_j^*$, and thus will sign the outgoing message using $\mathsf{sgk}_{A,j}$.

Therefore, $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},7}$ and $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},8}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,9}$: We change $r_{A,j}$ back to a PPRF evaluation. This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[j] = \mathsf{PPRF.Punc}(K_A, j)$.
  2. $K_B[j] = \mathsf{PPRF.Punc}(K_B, j)$.
  3. For $\alpha = A$,
     (a) $r_{\alpha,j} \leftarrow \mathsf{PPRF.Eval}(K_\alpha, j)$.
     (b) $(\mathsf{sgk}_{\alpha,j}, \mathsf{vk}_{\alpha,j}, \mathsf{vk}_{\alpha,j,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; r_{\alpha,j})$.
  4. $(\mathsf{sgk}^*_{A,j}, \mathsf{vk}^*_{A,j}) = (\mathsf{sgk}_{A,j}, \mathsf{vk}_{A,j})$.

Lemma C.27. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$, *all* $t \in [t^*]$, *all* $j \in [t-1]$, *and all* PPT *adversaries* $\mathcal{A}$,

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,8}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,9}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the selective pseudorandomness at punctured points property of our PPRF.

Observe that we can run both hybrids without knowing K_A as long as we are given $(K_A[j], r_{A,j})$. In the reduction, without computing K_A, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,9}$ up to just before step 6h.3. of KeyGen. Then, we receive $(K_A[j], r_{A,j})$ from the PPRF challenger where $r_{A,j}$ is either a random value or equal to $\mathsf{PPRF.Eval}(K_A, j)$. We use this randomness to run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,9}$ starting from step 6h.3(b) of KeyGen. Observe that if $r_{A,j}$ was a random value then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,9}$, and if $r_{A,j}$ was equal to $\mathsf{PPRF.Eval}(K_A, j)$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,8}$. Thus, by PPRF security, the outputs of these hybrids must be indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,1,j,10}$: We no longer puncture either K_A or K_B at j. We sign and verify A and B type signatures at step j using keys computed from K_A and K_B rather than hardwired values.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

6h. **Compute state signature keys:** Do nothing.

6i. **Compute $\sigma_{\mathsf{st},1}$:**

1. ~~If $1 = j$, $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,1}, m^*_1)$.~~
2. ~~If $1 \neq j$,~~
   (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.
   (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,1,j,10}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,1,j,10}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \perp$.
   v. **Verify A type signatures:**
      i. ~~If $i = j$,~~
         A. ~~If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,j}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.~~
      ii. ~~If $i \neq j$,~~
         A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. **Verify B type signatures:**
      i. If $\mathsf{in\text{-}type} \neq A$ and $j + 1 \leq i \leq t - 1$,
         A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
   vii. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.
2. **Computation Step:**
   i. If $i = t - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   ii. If $i \neq t - 1$,
      i. **Decrypt input and state**:
         i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**
A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:
i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
iii. If $i = j$,
i. If $m_i = m^*_j$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.
iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,
i. ~~If $i = j-1$,~~
A. ~~$\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,j}, m_{i+1})$.~~
ii. ~~If $i \neq j-1$,~~
A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,
//Observe that this branch can only be reached if $j \leq i \leq t-1$.
i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.
ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.28. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,9}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,10}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing, for $\alpha \in \{A, B\}$,

$$\mathsf{PPRF.Eval}(K_\alpha[j], v) = \mathsf{PPRF.Eval}(K_\alpha, v) \text{ for any } v \neq j$$

. Observe that the previous hybrid explicitly prevents the evaluation of punctured keys on their punctured points everywhere except for during the authentication step where it evaluates $K_B[j]$ at index $i+1$ if $\mathsf{out\text{-}type} = B$. However, we can only have $\mathsf{out\text{-}type} = B$ if we either verified a B type signature or had $i = j$. Either way, this means we would have $j \leq i \leq t-1$. Thus, we never evaluate punctured keys on punctured points.

Additionally, the hardwired signing and verification keys of the previous hybrid are the same as what is computed in the current hybrid. Thus, $\mathsf{Prog}_{3,t,1,j,9}$ and $\mathsf{Prog}_{3,t,1,j,10}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. $\square$

Enforcing the input. We want to change the conditional at step $i = j$ so that it sets the out-type based on the *outgoing* message rather than the *incoming* message. Thus, we will need to enforce that the incoming message is m^*_j if and only if the outgoing message is m^*_{j+1}.

Since the value of m^*_{j+1} depends on the $j+1^{th}$ state which depends on the j^{th} value of the input stream, we will need to ensure that the incoming input ciphertext $\mathsf{ct}_{\mathsf{inp},j}$ is equal to our chosen value $\mathsf{ct}^*_{\mathsf{inp},j}$. Since we only ever sign one input ciphertext at j, we can use properties of SSig to change the verification key of K_{inp} to only verify this one ciphertext. This will allow us to enforce that the input stream value at j must be our chosen value.

Hybrid$^{\mathcal{A}}_{3,t,1,j,11}$: We puncture K_{inp} at j. We sign and verify inputs at j using hardwired signatures and verification keys. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:
  1. $K_{\mathsf{inp}}[j] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, j)$.
  2. $r_{\mathsf{inp},j} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, j)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},j})$.
  4. $(\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{one}}, \mathsf{sgk}_{\mathsf{inp},j,\mathsf{abo}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{inp},j})$.
  5. $(\sigma^*_{\mathsf{inp},j}, \mathsf{vk}^*_{\mathsf{inp},j}) = (\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j})$.

  5g. **Compute** $\sigma_{\mathsf{inp},i}$:
  1. If $i = j$, $\sigma_{\mathsf{inp},i} = \sigma^*_{\mathsf{inp},j}$.
  2. If $i \neq j$,
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[j], i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,1,j,11}[K_{\mathsf{inp}}[j], \mathsf{vk}^*_{\mathsf{inp},j}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

  7d. **Compute** $\sigma_{\mathsf{inp},i}$:
  1. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[j], i))$.
  2. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

Program $\mathsf{Prog}_{3,t,1,j,11}[K_{\mathsf{inp}}[j], \mathsf{vk}^*_{\mathsf{inp},j}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. If $i = j$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
      ii. If $i \neq j$,
         A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[j], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. $\mathsf{in\text{-}type} = \perp$.

v. **Verify A type signatures:**

i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. **Verify B type signatures:**

i. If $\mathsf{in\text{-}type} \neq A$ and $j + 1 \leq i \leq t - 1$,

A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.

vii. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**

i. If $i = t - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i \neq t - 1$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i + 1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i + 1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = j$,

i. If $m_i = m^*_j$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i + 1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i + 1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.29. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,10}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,11}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing,

$$\mathsf{PPRF.Eval}(K_{\mathsf{inp}}[j], v) = \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, v) \text{ for any } v \neq j.$$

By correctness of SSig, if $(\sigma_{\mathsf{inp,one},j}, \mathsf{vk}_{\mathsf{inp,one},j}, \mathsf{sgk}_{\mathsf{inp,abo},j}, \mathsf{vk}_{\mathsf{inp,abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{inp},j})$, then

$$\sigma_{\mathsf{inp,one},j} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{inp},j}).$$

Observe that we never evaluate punctured keys on their punctured points. This is true even in **Encryption Phase 2** since in that phase, we have $i \geq t^* + 1$ which is not equal to j since $j \leq t - 1 \leq t^* - 1$ by assumption. Additionally $\sigma_{\mathsf{inp,one},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},1}, \mathsf{ct}^*_{\mathsf{inp},1})$ so step 5g of **Encryption Phase 1** results in the same signature. Thus, apart from the obfuscated programs, the hybrids act identically.

Furthermore, the hardwired verification key $\mathsf{vk}^*_{\mathsf{inp},j}$ is set to what it would have been computed to be in program of the previous hybrid. Thus, $\mathsf{Prog}_{3,t,1,j,10}$ and $\mathsf{Prog}_{3,t,1,j,11}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,12}$: We change $r_{\mathsf{inp},j}$ to a random value. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[j] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, j)$.
  2. $r_{\mathsf{inp},j} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},j})$.
  4. $(\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{one}}, \mathsf{sgk}_{\mathsf{inp},j,\mathsf{abo}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{inp},j})$.
  5. $(\sigma^*_{\mathsf{inp},j}, \mathsf{vk}^*_{\mathsf{inp},j}) = (\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j})$.

Lemma C.30. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, all* $j \in [t-1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,11}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,12}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the selective pseudorandomness at punctured points property of our PPRF.

Observe that we can run both hybrids without knowing K_{inp} as long as we are given $(K_{\mathsf{inp}}[j], r_{\mathsf{inp},j})$. In the reduction, without computing K_{inp}, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,12}$ up to just before step 5f of **Encryption Phase 1**. Then, we receive $(K_{\mathsf{inp}}[j], r_{\mathsf{inp},j})$ from the PPRF challenger where $r_{\mathsf{inp},j}$ is either a random value or equal to $\mathsf{PPRF.Eval}(K_{\mathsf{inp}}, j)$. We use this randomness to run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,12}$ starting from step 5f.3 of **Encryption Phase 1**. Observe that if $r_{\mathsf{inp},j}$ was a random value then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,12}$, and if $r_{\mathsf{inp},j}$ was equal to $\mathsf{PPRF.Eval}(K_{\mathsf{inp}}, j)$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,11}$. Thus, by PPRF security, the outputs of these hybrids must be indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,13}$: We change $\mathsf{vk}^*_{\mathsf{inp},j}$ to $\mathsf{vk}_{\mathsf{inp},j,\mathsf{one}}$ which can only verify $\mathsf{ct}^*_{\mathsf{inp},j}$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[j] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, j)$.
  2. $r_{\mathsf{inp},j} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},j})$.
  4. $(\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{one}}, \mathsf{sgk}_{\mathsf{inp},j,\mathsf{abo}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{inp},j})$.
  5. $(\sigma^*_{\mathsf{inp},j}, \mathsf{vk}^*_{\mathsf{inp},j}) = (\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{one}})$.

Lemma C.31. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, all* $j \in [t-1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,12}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,13}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig.

Observe that we can run both hybrids without knowing $(r_{\mathsf{inp},j}, \mathsf{sgk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{rej}})$ as long as we are given $(\sigma^*_{\mathsf{inp},j}, \mathsf{vk}^*_{\mathsf{inp},j})$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,13}$ up to just before step 5f.2 of **Encryption Phase 1**. We send $\mathsf{ct}^*_{\mathsf{inp},j}$ to the SSig challenger and receive $(\sigma^*, \mathsf{vk}^*)$ from the SSig challenger where σ^* is a signature of $\mathsf{ct}^*_{\mathsf{inp},j}$ and vk^* is either a verification key $\mathsf{vk}_{\mathsf{one}}$ that only verifies $\mathsf{ct}^*_{\mathsf{inp},j}$ or is a regular verification key vk. We then set $(\sigma^*_{\mathsf{inp},j}, \mathsf{vk}^*_{\mathsf{inp},j}) = (\sigma^*, \mathsf{vk}^*)$ and run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,13}$ starting from step 5g of **Encryption Phase 1**. Observe that if vk^* was a regular verification key vk we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,12}$, and if vk^* was $\mathsf{vk}_{\mathsf{one}}$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,13}$. Thus, by the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig security, the outputs of these hybrids must be indistinguishable. □

Using the iterator. Again, we want to enforce that the incoming message is m_j^* if and only if the outgoing message is m_{j+1}^*.

Intuitively, the forward direction now follows. If the incoming message is m_j^*, then this means that the incoming state must be our chosen value. We have also just enforced that the incoming stream input must be our chosen value. Thus, the program will compute our chosen output state, corresponding to m_{j+1}^*.

However, the backwards direction does not work since it is not true that some specific output state implies a unique input state. To solve this, we will use the iterator in m_{j+1}^* to enforce that the values iterated into it are correct. Since we iterate the input state into m_{j+1}^*'s iterator, we can then enforce the input state (and thus m_j^*) to be correct.

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,14}$: We enforce the iterator on $\mathsf{ct}^*_{\mathsf{st},i}$ for $i \in [j]$. This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6f. **Setup iterator**: $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.SetupEnforce}(1^\lambda, (1, \mathsf{ct}^*_{\mathsf{st},1}), \ldots, (j, \mathsf{ct}^*_{\mathsf{st},j}))$.[31]

Lemma C.32. *If* Itr *is a cryptographic iterator, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, all* $j \in [t-1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,13}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,14}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by a straightforward reduction to the indistinguishabilty of setup of the iterator. □

[31] Technically the values $\mathsf{ct}^*_{\mathsf{st},j}$ are not defined until the next step. For completeness, we can define these values here as $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i < t \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i \geq t \end{cases}$ which is identical to how they are defined in the next step.

Hybrid$^{\mathcal{A}}_{3,t,1,j,15}$: We change the conditional at step $i = j$ so that it sets the $\mathsf{out\text{-}type}$ based on the *outgoing* message rather than the *incoming* message. Recall that in the previous hybrid, we set the following values in **Encryption Phase 1** and KeyGen:

5e. **Set hardwired values**:

1. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i \geq t \end{cases}$.

6g. **Set hardwired values:**

1. For $i \in [t^*]$, $y^*_i = y_i$.
2. For $i \in [t^* + 1]$,
   (a) $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i < t \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i \geq t \end{cases}$
   (b) $\mathsf{itr}^*_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}^*_{\mathsf{st},i-1}, \mathsf{ct}^*_{\mathsf{st},i})$ where $\mathsf{itr}^*_{\mathsf{st},0} = \mathsf{itr}_{\mathsf{st},0}$.
   (c) $m^*_i = (i, \mathsf{ct}^*_{\mathsf{st},i}, \mathsf{itr}^*_{\mathsf{st},i-1})$.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,1,j,15}[K_{\mathsf{inp}}[j], \mathsf{vk}^*_{\mathsf{inp},j}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{j+1}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,t,1,j,15}[K_{\mathsf{inp}}[j], \mathsf{vk}^*_{\mathsf{inp},j}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{j+1}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. If $i = j$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
      ii. If $i \neq j$,
         A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[j], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \perp$.
   v. **Verify A type signatures:**
      i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. **Verify B type signatures:**
      i. If $\mathsf{in\text{-}type} \neq A$ and $j + 1 \leq i \leq t - 1$,
         A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
   vii. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**

   i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

   ii. If $i \neq t-1$,

      i. **Decrypt input and state**:

         i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

         ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

         iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

         iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

      ii. **Compute output value and next state:**

         A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

      iii. **Encrypt the new state**:

         i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

         ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

         iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

   iii. If $i = j$,

      i. If $m_{i+1} = m^*_{j+1}$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

   iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

      i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

   v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

      i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.33. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,14}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,15}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. To show that $\mathsf{Prog}_{3,t,1,j,14}$ and $\mathsf{Prog}_{3,t,1,j,15}$ have the same input/output behavior, we need to show that the following two conditions in the authentication steps of the programs are equivalent:

1. $i = j$ and $m_i = m^*_j$.
2. $i = j$ and $m_{i+1} = m^*_{j+1}$.

We show this in two steps:

- (1) $\implies$ (2). Let $i = j$. Since we verify input ciphertexts with $\mathsf{vk}^*_{\mathsf{inp},j} = \mathsf{vk}_{\mathsf{inp},j,\mathsf{one}}$, the only way to pass the verification step is to have

$$\mathsf{ct}_{\mathsf{inp},i} = \mathsf{ct}^*_{\mathsf{inp},j}$$

If $m_i = m^*_j$, then we must have

$$m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1}) = (j, \mathsf{ct}^*_{\mathsf{st},j}, \mathsf{itr}^*_{\mathsf{st},j-1}) = m^*_j$$

which means that since $j < t$,

$$(\mathsf{ct}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}) = (\mathsf{ct}^*_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{st},j}) = (\mathsf{ct}^{(0)}_{\mathsf{inp},j}, \mathsf{ct}^{(0)}_{\mathsf{st},j})$$

In the computation step,

- If $i = j < t - 1$, by correctness of decryption of SKE, since our incoming ciphertexts are $(\mathsf{ct}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}) = (\mathsf{ct}^{(0)}_{\mathsf{inp},j}, \mathsf{ct}^{(0)}_{\mathsf{st},j})$, we will compute the outgoing ciphertext as $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{ct}^{(0)}_{\mathsf{st},j+1}$. Thus, since $j + 1 < t$, then we must compute

$$\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{ct}^{(0)}_{\mathsf{st},j+1} = \mathsf{ct}^*_{\mathsf{st},j+1}$$

- If $i = j = t - 1$, then we set

$$\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{ct}^*_{\mathsf{st},t} = \mathsf{ct}^*_{\mathsf{st},j+1}$$

Since $\mathsf{ltr.Iterate}$ is a deterministic function, then

$$\mathsf{itr}_{\mathsf{st},i} = \mathsf{ltr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}) = \mathsf{ltr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}^*_{\mathsf{st},j-1}, (j, \mathsf{ct}^*_{\mathsf{st},j})) = \mathsf{itr}^*_{\mathsf{st},j}$$

But this means that

$$m_{i+1} = (i + 1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i}) = (j + 1, \mathsf{ct}^*_{\mathsf{st},j+1}, \mathsf{itr}^*_{\mathsf{st},j}) = m^*_{j+1}$$

which means that (2) holds.

- (2) $\implies$ (1). Let $i = j$. If $m_{i+1} = m^*_{j+1}$, then we must have

$$m_{i+1} = (i + 1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i}) = (j + 1, \mathsf{ct}^*_{\mathsf{st},j+1}, \mathsf{itr}^*_{\mathsf{st},j}) = m^*_{j+1}$$

Since the iterator is in enforcing mode and

$$\mathsf{itr}^*_{\mathsf{st},j} = \mathsf{itr}_{\mathsf{st},i} = \mathsf{ltr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i})$$

it must be the case that

$$(\mathsf{itr}_{\mathsf{st},i-1}, \mathsf{ct}_{\mathsf{st},i}) = (\mathsf{itr}^*_{\mathsf{st},j-1}, \mathsf{ct}^*_{\mathsf{st},j})$$

But this means that

$$m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i}) = (j, \mathsf{ct}^*_{\mathsf{st},j}, \mathsf{itr}^*_{\mathsf{st},j-1}) = m^*_j$$

which means that (1) holds.

Thus, $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},14}$ and $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},15}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,16}$: We undo the enforcing on the iterator. This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6f. **Setup iterator**: $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$

Lemma C.34. *If* Itr *is a cryptographic iterator, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, all* $j \in [t-1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,15}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,16}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by a straightforward reduction to the indistinguishabilty of setup of the iterator. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,17}$: We change $\mathsf{vk}^*_{\mathsf{inp},j}$ back to $\mathsf{vk}_{\mathsf{inp},j}$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:
  1. $K_{\mathsf{inp}}[j] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, j)$.
  2. $r_{\mathsf{inp},j} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},j})$.
  4. $(\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{one}}, \mathsf{sgk}_{\mathsf{inp},j,\mathsf{abo}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{inp},j})$.
  5. $(\sigma^*_{\mathsf{inp},j}, \mathsf{vk}^*_{\mathsf{inp},j}) = (\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j})$.

Lemma C.35. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, all* $j \in [t-1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,16}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,17}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig. The reduction is essentially the same as that of Lemma C.31 as it is still the case that we can run both hybrids without knowing $(r_{\mathsf{inp},j}, \mathsf{sgk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{rej}})$ as long as we are given $(\sigma^*_{\mathsf{inp},j}, \mathsf{vk}^*_{\mathsf{inp},j})$. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,18}$: We change $r_{\mathsf{inp},j}$ back to a PPRF value. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:
  1. $K_{\mathsf{inp}}[j] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, j)$.
  2. $r_{\mathsf{inp},j} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, j)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},j})$.
  4. $(\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{one}}, \mathsf{sgk}_{\mathsf{inp},j,\mathsf{abo}}, \mathsf{vk}_{\mathsf{inp},j,\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{inp},j})$.
  5. $(\sigma^*_{\mathsf{inp},j}, \mathsf{vk}^*_{\mathsf{inp},j}) = (\sigma_{\mathsf{inp},j,\mathsf{one}}, \mathsf{vk}_{\mathsf{inp},j})$.

Lemma C.36. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, all* $j \in [t-1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,17}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,18}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by a reduction to the selective pseudorandomness at punctured points property of our PPRF. The reduction is essentially the same as that of Lemma C.30 as it is still the case that we can run both hybrids without knowing K_{inp} as long as we are given $(K_{\mathsf{inp}}[j], r_{\mathsf{inp},j})$. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,19}$: We no longer puncture K_{inp} at j and no longer use hardwired signatures and verification keys. Note that this hybrid is identical to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j+1,0}$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**: Do nothing.

  5g. **Compute** $\sigma_{\mathsf{inp},i}$:

  1. ~~If $i = j$, $\sigma_{\mathsf{inp},i} = \mathsf{SSig}.\sigma^*_{\mathsf{inp},j}$.~~
  2. ~~If $i \neq j$,~~
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,1,j,19}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_j, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

  7d. **Compute** $\sigma_{\mathsf{inp},i}$:

  1. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
  2. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

Program $\mathsf{Prog}_{3,t,1,j,19}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{j+1}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. ~~If $i = j$,~~
         A. ~~If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.~~
      ii. ~~If $i \neq j$,~~
         A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \perp$.
   v. **Verify A type signatures:**
      i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. **Verify B type signatures:**
      i. If $\mathsf{in\text{-}type} \neq A$ and $j + 1 \leq i \leq t - 1$,
         A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
   vii. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**

   i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

   ii. If $i \neq t-1$,

      i. **Decrypt input and state**:

         i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

         ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

         iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

         iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

      ii. **Compute output value and next state:**

         A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

      iii. **Encrypt the new state**:

         i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

         ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

         iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

   iii. If $i = j$,

      i. If $m_{i+1} = m^*_{j+1}$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

   iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

      i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

   v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

      i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.37. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,18}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,19}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing,

$$\mathsf{PPRF.Eval}(K_{\mathsf{inp}}[j], v) = \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, v) \text{ for any } v \neq j.$$

By correctness of SSig, if $(\sigma_{\mathsf{inp,one},j}, \mathsf{vk}_{\mathsf{inp,one},j}, \mathsf{sgk}_{\mathsf{inp,abo},j}, \mathsf{vk}_{\mathsf{inp,abo},j}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{inp},j})$, then

$$\sigma_{\mathsf{inp,one},j} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},j}, \mathsf{ct}^*_{\mathsf{inp},j}).$$

Observe that the previous hybrid never evaluated punctured keys on their punctured points. This was true even in **Encryption Phase 2** since in that phase, we had $i \geq t^* + 1$ which is not equal to j since $j \leq t-1 \leq t^* - 1$ by assumption. Additionally $\sigma_{\mathsf{inp,one},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},1}, \mathsf{ct}^*_{\mathsf{inp},1})$ so

step 5g of **Encryption Phase 1** results in the same signature. Thus, apart from the obfuscated programs, the hybrids act identically.

Furthermore, the hardwired verification key $\mathsf{vk}^*_{\mathsf{inp},j}$ of the previous hybrid is set to the same value that is computed in the program of the current hybrid. Thus, $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},18}$ and $\mathsf{Prog}_{3,\mathsf{t},1,\mathsf{j},19}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Lemma C.38. *For all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, all $j \in [t-1]$, and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j,18}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,j+1,0}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2} = \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,0}$: This hybrid is the same as $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,t,0}$ except that we have removed the B type verification branch in the obfuscated program. Except for this removal, we have highlighted all other differences between this hybrid and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1}$

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $K_{\mathsf{inp}}, K_A, K_B, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.

   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.

   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

   (d) **Compute input ciphertexts**:

      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.

      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

   (e) **Set hardwired values**:

      i. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i \geq t \end{cases}$.

   (f) **Compute input signature keys**: Do nothing.

   (g) **Compute $\sigma_{\mathsf{inp},i}$**:

      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

   (h) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.

   (c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.

   (d) **Compute state and output values:**

      i. For $i \in [t^*]$,

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^* + 1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^* + 1]$,

A. $\mathsf{ct}_{\mathsf{st},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{st},i}^{(0)} & \text{if } i < t \\ \mathsf{ct}_{\mathsf{st},i}^{(b)} & \text{if } i \geq t \end{cases}$

B. $\mathsf{itr}_{\mathsf{st},i}^* = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}^*, \mathsf{ct}_{\mathsf{st},i}^*)$ where $\mathsf{itr}_{\mathsf{st},0}^* = \mathsf{itr}_{\mathsf{st},0}$.

C. $m_i^* = (i, \mathsf{ct}_{\mathsf{st},i}^*, \mathsf{itr}_{\mathsf{st},i-1}^*)$.

(h) **Compute state signature keys**: Do nothing.

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

ii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1^*)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},2}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m_t^*, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}^*, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}_{3,t,2}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m_t^*, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. $\mathsf{in\text{-}type} = \perp$.
   - v. **Verify A type signatures:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   - vi. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.
2. **Computation Step:**
   - i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*)$.
   - ii. If $i \neq t-1$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   - i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   - ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   - iii. If $i = t-1$,
     - i. If $m_{i+1} = m_t^*$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.
   - iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,
     - i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
     - ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
   - v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.39. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all PPT adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1,t,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2}(1^\lambda)) \leq \mathsf{negl}(1^\lambda)$$

Proof. The only difference between the hybrids is that $\mathsf{Prog}_{3,t,2}$ has entirely removed the B type verification branch. However, for $j = t$, since there is no i such that $j = t \leq i \leq t-1$, then $\mathsf{Prog}_{3,t,1,t,0}$ will never use the B type verification branch anyway. Thus $\mathsf{Prog}_{3,t,2}$ and $\mathsf{Prog}_{3,t,1,t,0}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hardcoding step t. It is now the case that at index t, we only have one input ciphertext $\mathsf{ct}^*_{\mathsf{st},t}$ signed using K_{inp} and only one state message m^*_t signed using K_A. Therefore, we can use SSig to enforce that these are the only input and state values that can be verified. Thus, there can be only one possible output in the computation step, which means we can finally hardcode in step $i = t$.

Hybrid$^{\mathcal{A}}_{3,t,2,1}$: We start by puncturing both K_{inp} and K_A at t. We sign and verify at index t using hardcoded values.

This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:
  1. $K_{\mathsf{inp}}[t] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t)$.
  2. $r_{\mathsf{inp},t} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t})$.
  4. $(\sigma^*_{\mathsf{inp},t}, \mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t}) = (\bot, \mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t})$.

  5g. **Compute** $\sigma_{\mathsf{inp},i}$:
  1. If $i = t$, $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{inp},t})$.
  2. If $i \neq t$,
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t], i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**
  1. $K_A[t] = \mathsf{PPRF.Punc}(K_A, t)$.
  2. $r_{A,t} \leftarrow \mathsf{PPRF.Eval}(K_A, t)$.
  3. $(\mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t}, \mathsf{vk}_{A,t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t})$.
  4. $(\sigma^*_{\mathsf{st},t}, \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}) = (\bot, \mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t})$.

  6i. **Compute** $\sigma_{\mathsf{st},1}$**:**
  1. If $1 = t$, $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,1}, m^*_1)$.
  2. If $1 \neq t$,
     (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], 1))$.
     (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},2,1}[K_{\mathsf{inp}}[t], \mathsf{vk}^*_{\mathsf{inp},t}, K_A[t], \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

  7d. **Compute** $\sigma_{\mathsf{inp},i}$:
  1. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t], i))$.
  2. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

Program $\mathsf{Prog}_{3,t,2,1}[K_{\mathsf{inp}}[t], \mathsf{vk}^*_{\mathsf{inp},t}, K_A[t], \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. If $i = t$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
      ii. If $i \neq t$,
         A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \perp$.
   v. **Verify A type signatures:**
      i. If $i = t$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
      ii. If $i \neq t$,
         A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.
2. **Computation Step:**
   i. If $i = t - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   ii. If $i \neq t - 1$,
      i. **Decrypt input and state**:
         i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
         ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
         iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
         iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
      ii. **Compute output value and next state:**
         A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
      iii. **Encrypt the new state**:
         i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i + 1)$.
         ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
         iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   ii. $m_{i+1} = (i + 1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = t - 1$,

i. If $m_{i+1} = m_t^*$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. If $i = t - 1$,

A. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,t}, m_{i+1})$.

ii. If $i \neq t - 1$,

A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.40. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,1}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing, for $\alpha \in \{A, \mathsf{inp}\}$,

$$\mathsf{PPRF.Eval}(K_\alpha[t], v) = \mathsf{PPRF.Eval}(K_\alpha, v) \text{ for any } v \neq t$$

. Observe that we never evaluate punctured keys on their punctured points. This is true even in **Encryption Phase 2** since in that phase, we have $i \geq t^* + 1 > t$.

Furthermore, the hardwired signing and verification keys are set to what they would have been computed to be in the previous hybrid. Thus, $\mathsf{Prog}_{3,\mathsf{t},2,0}$ and $\mathsf{Prog}_{3,\mathsf{t},2,1}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,2,2}$: We split our input signature and our A type signature at t on $\mathsf{ct}^*_{\mathsf{inp},t}$ and m^*_t respectively. We replace signatures using $\mathsf{sgk}^*_{\mathsf{inp},t}$ or $\mathsf{sgk}^*_{A,t}$ with $\sigma_{\mathsf{inp,one},t}$ and $\sigma_{A,\mathsf{one},t}$ respectively.

This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:
  1. $K_{\mathsf{inp}}[t] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t)$.
  2. $r_{\mathsf{inp},t} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t})$.
  4. $(\sigma_{\mathsf{inp,one},t}, \mathsf{vk}_{\mathsf{inp,one},t}, \mathsf{sgk}_{\mathsf{inp,abo},t}, \mathsf{vk}_{\mathsf{inp,abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{inp},t})$.
  5. $(\sigma^*_{\mathsf{inp},t}, \mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t}) = (\sigma_{\mathsf{inp,one},t}, \bot, \mathsf{vk}_{\mathsf{inp},t})$.

  5g. **Compute** $\sigma_{\mathsf{inp},i}$:
  1. If $i = t$, $\sigma_{\mathsf{inp},i} = \sigma^*_{\mathsf{inp},t}$.
  2. If $i \neq t$,
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t], i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**
  1. $K_A[t] = \mathsf{PPRF.Punc}(K_A, t)$.
  2. $r_{A,t} \leftarrow \mathsf{PPRF.Eval}(K_A, t)$.
  3. $(\mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t}, \mathsf{vk}_{A,t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t})$.
  4. $(\sigma_{A,\mathsf{one},t}, \mathsf{vk}_{A,\mathsf{one},t}, \mathsf{sgk}_{A,\mathsf{abo},t}, \mathsf{vk}_{A,\mathsf{abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t}, m^*_t)$.
  5. $(\sigma^*_{\mathsf{st},t}, \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}) = (\sigma_{A,\mathsf{one},t}, \bot, \mathsf{vk}_{A,t})$.

  6i. **Compute** $\sigma_{\mathsf{st},1}$**:**
  1. If $1 = t$, $\sigma_{\mathsf{st},i} = \sigma^*_{\mathsf{st},t}$.
  2. If $1 \neq t$,
     (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], 1))$.
     (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},2,2}[K_{\mathsf{inp}}[t], \mathsf{vk}^*_{\mathsf{inp},t}, K_A[t], \sigma^*_{\mathsf{st},t}, \mathsf{vk}^*_{A,t}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}])$.

Program $\mathsf{Prog}_{3,\mathsf{t},2,2}[K_{\mathsf{inp}}[t], \mathsf{vk}^*_{\mathsf{inp},t}, K_A[t], \sigma^*_{\mathsf{st},t}, \mathsf{vk}^*_{A,t}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

   i. **Verify** i **is positive:** If $i \leq 0$, output $\bot$.

   ii. **Verify input signature:**

      i. If $i = t$,

         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.

      ii. If $i \neq t$,

         A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. $\mathsf{in\text{-}type} = \perp$.

v. **Verify A type signatures:**

i. If $i = t$,

A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

ii. If $i \neq t$,

A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i \neq t-1$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = t-1$,

i. If $m_{i+1} = m^*_t$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. If $i = t-1$,

A. $\sigma_{\mathsf{st},i+1} = \sigma^*_{\mathsf{st},t}$.

ii. If $i \neq t-1$,

A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.41. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,1}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,2}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of SSig, if $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}, v^*)$, then

$$\sigma_{\mathsf{one}} = \mathsf{SSig.Sign}(\mathsf{sgk}, v^*).$$

Thus, apart from the obfuscated programs, the hybrids act identically since $\sigma_{\mathsf{inp},\mathsf{one},t} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{st},t})$ and $\sigma_{A,\mathsf{one},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$ so step 5g of **Encryption Phase 1** and step 6i of KeyGen result in the same signatures.

Now, in the previous hybrid, $\mathsf{Prog}_{3,\mathsf{t},2,1}$ only used $\mathsf{sgk}^*_{A,t} = \mathsf{sgk}_{A,t}$ in one place: If $\mathsf{out\text{-}type} = A$ and $i = t-1$, then it signed m_{i+1} with $\mathsf{sgk}_{A,t}$. However, we can only have $\mathsf{out\text{-}type} = A$ at $i = t-1$ if $m_{i+1} = m^*_t$. Thus, replacing the signature with $\sigma_{A,\mathsf{one},t}$ does not change the behavior of the program.

Therefore, $\mathsf{Prog}_{3,\mathsf{t},2,1}$ and $\mathsf{Prog}_{3,\mathsf{t},2,2}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,3}$: We change $r_{\mathsf{inp},t}$ and $r_{A,t}$ to random values. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[t] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t)$.
  2. $r_{\mathsf{inp},t} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t})$.
  4. $(\sigma_{\mathsf{inp,one},t}, \mathsf{vk}_{\mathsf{inp,one},t}, \mathsf{sgk}_{\mathsf{inp,abo},t}, \mathsf{vk}_{\mathsf{inp,abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{inp},t})$.
  5. $(\sigma^*_{\mathsf{inp},t}, \mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t}) = (\sigma_{\mathsf{inp,one},t}, \bot, \mathsf{vk}_{\mathsf{inp},t})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[t] = \mathsf{PPRF.Punc}(K_A, t)$.
  2. $r_{A,t} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t}, \mathsf{vk}_{A,t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t})$.
  4. $(\sigma_{A,\mathsf{one},t}, \mathsf{vk}_{A,\mathsf{one},t}, \mathsf{sgk}_{A,\mathsf{abo},t}, \mathsf{vk}_{A,\mathsf{abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t}, m^*_t)$.
  5. $(\sigma^*_{\mathsf{st},t}, \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}) = (\sigma_{A,\mathsf{one},t}, \bot, \mathsf{vk}_{A,t})$.

Lemma C.42. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,2}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,3}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by two reductions to the selective pseudorandomness at punctured points property of our PPRF.

We first swap out only $r_{\mathsf{inp},t}$ for a random value and leave $r_{A,t}$ as a PPRF evaluation. We call this intermediate hybrid $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,2.5}$.

Observe that we can run both the previous hybrid and the intermediate hybrid without knowing K_{inp} as long as we are given $(K_{\mathsf{inp}}[t], r_{\mathsf{inp},t})$. In the reduction, without computing K_{inp}, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,2.5}$ up to just before step 5f of **Encryption Phase 1**. Then, we receive $(K_{\mathsf{inp}}[t], r_{\mathsf{inp},t})$ from the PPRF challenger where $r_{\mathsf{inp},t}$ is either a random value or equal to $\mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t)$. We use this randomness to compute $(\sigma^*_{\mathsf{inp},t}, \mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t})$. We then run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,j,2.5}$ starting from step 5g of **Encryption Phase 1**. Observe that if $r_{\mathsf{inp},t}$ was a random value then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,2.5}$, and if $r_{\mathsf{inp},t}$ was equal to $\mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t)$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,2}$. Thus, by PPRF security, the outputs of these hybrids must be indistinguishable.

By a similar reduction on the A type signature scheme, $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,2.5}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,3}$ are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,4}$: We set $\mathsf{vk}^*_{\mathsf{inp},t}$ to $\mathsf{vk}_{\mathsf{inp},\mathsf{one},t}$ which will only verify $\mathsf{ct}^*_{\mathsf{inp},t}$, and set $\mathsf{vk}^*_{A,t}$ to $\mathsf{vk}_{A,\mathsf{one},t}$ which will only verify m^*_t. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[t] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t)$.
  2. $r_{\mathsf{inp},t} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t})$.
  4. $(\sigma_{\mathsf{inp},\mathsf{one},t}, \mathsf{vk}_{\mathsf{inp},\mathsf{one},t}, \mathsf{sgk}_{\mathsf{inp},\mathsf{abo},t}, \mathsf{vk}_{\mathsf{inp},\mathsf{abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{inp},t})$.
  5. $(\sigma^*_{\mathsf{inp},t}, \mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t}) = (\sigma_{\mathsf{inp},\mathsf{one},t}, \bot, \mathsf{vk}_{\mathsf{inp},\mathsf{one},t})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[t] = \mathsf{PPRF.Punc}(K_A, t)$.
  2. $r_{A,t} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t}, \mathsf{vk}_{A,t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t})$.
  4. $(\sigma_{A,\mathsf{one},t}, \mathsf{vk}_{A,\mathsf{one},t}, \mathsf{sgk}_{A,\mathsf{abo},t}, \mathsf{vk}_{A,\mathsf{abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t}, m^*_t)$.
  5. $(\sigma^*_{\mathsf{st},t}, \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}) = (\sigma_{A,\mathsf{one},t}, \bot, \mathsf{vk}_{A,\mathsf{one},t})$.

Lemma C.43. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,3}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,4}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by two reductions to the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig.

We first swap out only $\mathsf{vk}^*_{\mathsf{inp},t}$ for $\mathsf{vk}_{\mathsf{inp},\mathsf{one},t}$ and leave $\mathsf{vk}^*_{A,t}$ as $\mathsf{vk}_{A,t}$. We call this intermediate hybrid $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,3.5}$.

Observe that we can run both hybrids without knowing $(r_{\mathsf{inp},t}, \mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t,\mathsf{rej}})$ as long as we are given $(\sigma^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t})$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,4}$ up to just before step 5f.2 of **Encryption Phase 1**. We send $\mathsf{ct}^*_{\mathsf{inp},t}$ to the SSig challenger and receive $(\sigma^*, \mathsf{vk}^*)$ from the SSig challenger where σ^* is a signature of $\mathsf{ct}^*_{\mathsf{inp},t}$ and vk^* is either a verification key $\mathsf{vk}_{\mathsf{one}}$ that only verifies $\mathsf{ct}^*_{\mathsf{inp},t}$ or is a regular verification key vk. We then set $(\sigma^*_{\mathsf{inp},t}, \mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t}) = (\sigma^*, \bot, \mathsf{vk}^*)$ and run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,4}$ starting from step 5g of **Encryption Phase 1**. Observe that if vk^* was a regular verification key vk we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,3}$, and if vk^* was $\mathsf{vk}_{\mathsf{one}}$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,3.5}$. Thus, by the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig security, the outputs of these hybrids must be indistinguishable.

By a similar reduction on the A type signature scheme, $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,3.5}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,4}$ are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,2,5}$: We now hardwire step t into the program. Recall that in the previous hybrids, we set the following values in **Encryption Phase 1** and KeyGen:

5e. **Set hardwired values**:

1. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i \geq t \end{cases}$.

6g. **Set hardwired values:**

1. For $i \in [t^*]$, $y^*_i = y_i$.
2. For $i \in [t^* + 1]$,
   (a) $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i < t \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i \geq t \end{cases}$
   (b) $\mathsf{itr}^*_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}^*_{\mathsf{st},i-1}, \mathsf{ct}^*_{\mathsf{st},i})$ where $\mathsf{itr}^*_{\mathsf{st},0} = \mathsf{itr}_{\mathsf{st},0}$.
   (c) $m^*_i = (i, \mathsf{ct}^*_{\mathsf{st},i}, \mathsf{itr}^*_{\mathsf{st},i-1})$.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,2,5}[K_{\mathsf{inp}}[t], \mathsf{vk}^*_{\mathsf{inp},t}, K_A[t], \sigma^*_{\mathsf{st},t}, \mathsf{vk}^*_{A,t}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

Program $\mathsf{Prog}_{3,t,2,5}[K_{\mathsf{inp}}[t], \mathsf{vk}^*_{\mathsf{inp},t}, K_A[t], \sigma^*_{\mathsf{st},t}, \mathsf{vk}^*_{A,t}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. If $i = t$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
      ii. If $i \neq t$,
         A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \perp$.
   v. **Verify A type signatures:**
      i. If $i = t$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
      ii. If $i \neq t$,
         A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**

   i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

   ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

   iii. If $i \notin \{t-1, t\}$,

      i. **Decrypt input and state**:

         i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

         ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

         iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

         iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

      ii. **Compute output value and next state:**

         A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

      iii. **Encrypt the new state**:

         i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

         ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

         iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

   iii. If $i = t-1$,

      i. If $m_{i+1} = m^*_t$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

   iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

      i. If $i = t-1$,

         A. $\sigma_{\mathsf{st},i+1} = \sigma^*_{\mathsf{st},t}$.

      ii. If $i \neq t-1$,

         A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], i+1))$.

         B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

   v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

      i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.44. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,4}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,5}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. We will show that $\mathsf{Prog}_{3,t,2}$ and $\mathsf{Prog}_{3,t,2,t-1}$ have identical input/output behavior. The programs can only differ when $i = t$. Thus, we will restrict ourselves to this setting.

Since $\mathsf{vk}^*_{\mathsf{inp},t} = \mathsf{vk}_{\mathsf{inp},\mathsf{one},t}$, then in order to pass the verification step, it must be the case that

$$\mathsf{ct}_{\mathsf{inp},i} = \mathsf{ct}^*_{\mathsf{inp},t} = \mathsf{ct}^{(b)}_{\mathsf{inp},t}$$

Similarly, since $\mathsf{vk}^*_{A,t} = \mathsf{vk}_{A,\mathsf{one},t}$, in order to pass the verification step, we must have

$$m_i = m^*_t = (t, \mathsf{ct}^{(b)}_{\mathsf{st},t}, \mathsf{itr}^*_{\mathsf{st},t-1})$$

Therefore, by correctness of decryption of SKE and $\mathsf{Post\text{-}One\text{-}sFE}$, in the previous hybrid, we would have computed

$$(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y_t, \mathsf{ct}^{(b)}_{\mathsf{st},t+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$$

But these are exactly the values that we have hardwired them to be in the current hybrids. Thus the input/output behavior of the two programs is identical, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,2,6}$: We change $\mathsf{vk}^*_{\mathsf{inp},t}$ and $\mathsf{vk}^*_{A,t}$ back to $\mathsf{vk}_{\mathsf{inp},t}$ and $\mathsf{vk}_{A,t}$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[t] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t)$.
  2. $r_{\mathsf{inp},t} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t})$.
  4. $(\sigma_{\mathsf{inp,one},t}, \mathsf{vk}_{\mathsf{inp,one},t}, \mathsf{sgk}_{\mathsf{inp,abo},t}, \mathsf{vk}_{\mathsf{inp,abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{inp},t})$.
  5. $(\sigma^*_{\mathsf{inp},t}, \mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t}) = (\sigma_{\mathsf{inp,one},t}, \bot, \mathsf{vk}_{\mathsf{inp},t})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[t] = \mathsf{PPRF.Punc}(K_A, t)$.
  2. $r_{A,t} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t}, \mathsf{vk}_{A,t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t})$.
  4. $(\sigma_{A,\mathsf{one},t}, \mathsf{vk}_{A,\mathsf{one},t}, \mathsf{sgk}_{A,\mathsf{abo},t}, \mathsf{vk}_{A,\mathsf{abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t}, m^*_t)$.
  5. $(\sigma^*_{\mathsf{st},t}, \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}) = (\sigma_{A,\mathsf{one},t}, \bot, \mathsf{vk}_{A,t})$.

Lemma C.45. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,5}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,6}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by two reductions to the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig. The reductions are essentially the same as in Lemma C.43 as it is still the case that we can run both hybrids without knowing $(r_{\mathsf{inp},t}, \mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t,\mathsf{rej}})$ or $(r_{A,t}, \mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t}, \mathsf{vk}_{A,t,\mathsf{rej}})$ as long as we are given $(\sigma^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t})$ and $(\sigma^*_{A,t}, \mathsf{vk}^*_{A,t})$. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,7}$: We change $r_{\mathsf{inp},t}$ and $r_{A,t}$ back to PPRF values. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[t] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t)$.
  2. $r_{\mathsf{inp},t} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t})$.
  4. $(\sigma_{\mathsf{inp},\mathsf{one},t}, \mathsf{vk}_{\mathsf{inp},\mathsf{one},t}, \mathsf{sgk}_{\mathsf{inp},\mathsf{abo},t}, \mathsf{vk}_{\mathsf{inp},\mathsf{abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{inp},t})$.
  5. $(\sigma^*_{\mathsf{inp},t}, \mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t}) = (\sigma_{\mathsf{inp},\mathsf{one},t}, \bot, \mathsf{vk}_{\mathsf{inp},t})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[t] = \mathsf{PPRF.Punc}(K_A, t)$.
  2. $r_{A,t} \leftarrow \mathsf{PPRF.Eval}(K_A, t)$.
  3. $(\mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t}, \mathsf{vk}_{A,t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t})$.
  4. $(\sigma_{A,\mathsf{one},t}, \mathsf{vk}_{A,\mathsf{one},t}, \mathsf{sgk}_{A,\mathsf{abo},t}, \mathsf{vk}_{A,\mathsf{abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t}, m^*_t)$.
  5. $(\sigma^*_{\mathsf{st},t}, \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}) = (\sigma_{A,\mathsf{one},t}, \bot, \mathsf{vk}_{A,t})$.

Lemma C.46. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,6}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,7}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by two reductions to the selective pseudorandomness at punctured points property of our PPRF. The reductions are essentially the same as that of Lemma C.42 as it is still the case that we can run both hybrids without knowing K_{inp} or K_A as long as we are given $(K_{\mathsf{inp}}[t], r_{\mathsf{inp},t})$ and $(K_A[t], r_{A,t})$. □

Hybrid$^{\mathcal{A}}_{3,t,2,8}$: We no longer split our input signature and our A type signature at t on $\mathsf{ct}^*_{\mathsf{inp},t}$ and m^*_t respectively. We replace signatures $\sigma_{\mathsf{inp,one},t}$ and $\sigma_{A,\mathsf{one},t}$ with signatures using $\mathsf{sgk}^*_{\mathsf{inp},t}$ and $\mathsf{sgk}^*_{A,t}$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:
  1. $K_{\mathsf{inp}}[t] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t)$.
  2. $r_{\mathsf{inp},t} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t})$.
  4. ~~$(\sigma_{\mathsf{inp,one},t}, \mathsf{vk}_{\mathsf{inp,one},t}, \mathsf{sgk}_{\mathsf{inp,abo},t}, \mathsf{vk}_{\mathsf{inp,abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{inp},t})$.~~
  5. $(\sigma^*_{\mathsf{inp},t}, \mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{vk}^*_{\mathsf{inp},t}) = (\bot, \mathsf{sgk}_{\mathsf{inp},t}, \mathsf{vk}_{\mathsf{inp},t})$.

  5g. **Compute** $\sigma_{\mathsf{inp},i}$:
  1. If $i = t$, $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.
  2. If $i \neq t$,
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t], i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**
  1. $K_A[t] = \mathsf{PPRF.Punc}(K_A, t)$.
  2. $r_{A,t} \leftarrow \mathsf{PPRF.Eval}(K_A, t)$.
  3. $(\mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t}, \mathsf{vk}_{A,t,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t})$.
  4. ~~$(\sigma_{A,\mathsf{one},t}, \mathsf{vk}_{A,\mathsf{one},t}, \mathsf{sgk}_{A,\mathsf{abo},t}, \mathsf{vk}_{A,\mathsf{abo},t}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t}, m^*_t)$.~~
  5. $(\sigma^*_{\mathsf{st},t}, \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}) = (\bot, \mathsf{sgk}_{A,t}, \mathsf{vk}_{A,t})$.

  6i. **Compute** $\sigma_{\mathsf{st},1}$**:**
  1. If $1 = t$, $\sigma_{\mathsf{st},i} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,t}, m^*_1)$.
  2. If $1 \neq t$,
     (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], 1))$.
     (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},2,8}[K_{\mathsf{inp}}[t], \mathsf{vk}^*_{\mathsf{inp},t}, K_A[t], \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

Program $\mathsf{Prog}_{3,\mathsf{t},2,8}[K_{\mathsf{inp}}[t], \mathsf{vk}^*_{\mathsf{inp},t}, K_A[t], \mathsf{sgk}^*_{A,t}, \mathsf{vk}^*_{A,t}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

   i. **Verify** i **is positive:** If $i \leq 0$, output $\bot$.

   ii. **Verify input signature:**

      i. If $i = t$,

         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.

      ii. If $i \neq t$,

         A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. $\mathsf{in\text{-}type} = \perp$.

v. **Verify A type signatures:**

i. If $i = t$,

A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

ii. If $i \neq t$,

A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

iii. If $i \notin \{t-1, t\}$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = t-1$,

i. If $m_{i+1} = m^*_t$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. If $i = t-1$,

A. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,t}, m_{i+1})$.

ii. If $i \neq t-1$,

A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.47. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,7}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,8}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of SSig, if $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}, v^*)$, then

$$\sigma_{\mathsf{one}} = \mathsf{SSig.Sign}(\mathsf{sgk}, v^*).$$

Thus, apart from the obfuscated programs, the hybrids act identically since $\sigma_{\mathsf{inp},\mathsf{one},t} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{st},t})$ and $\sigma_{A,\mathsf{one},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$ so step 5g of **Encryption Phase 1** and step 6i of KeyGen result in the same signatures.

The current hybrid $\mathsf{Prog}_{3,\mathsf{t},2,8}$ only uses $\mathsf{sgk}^*_{A,t} = \mathsf{sgk}_{A,t}$ in one place: If $\mathsf{out\text{-}type} = A$ and $i = t-1$, then it signs m_{i+1} with $\mathsf{sgk}_{A,t}$. However, we can only have $\mathsf{out\text{-}type} = A$ at $i = t-1$ if $m_{i+1} = m^*_t$. Thus, the signature it generates is the same as the signature $\sigma^*_{\mathsf{st},t} = \sigma_{A,\mathsf{one},t}$ used in the previous hybrid.

Therefore, $\mathsf{Prog}_{3,\mathsf{t},2,7}$ and $\mathsf{Prog}_{3,\mathsf{t},2,8}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,2,9}$: We no longer puncture K_{inp} at t and no longer use hardwired signatures and verification keys. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**: Do nothing.

  5g. **Compute** $\sigma_{\mathsf{inp},i}$:

  1. ~~If $i = t$, $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{\mathsf{inp},t}, \mathsf{ct}^*_{\mathsf{inp},t})$.~~
  2. ~~If $i \neq t$,~~
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:** Do nothing.

  6i. **Compute** $\sigma_{\mathsf{st},1}$**:**

  1. ~~If $1 = t$, $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,1}, m^*_1)$.~~
  2. ~~If $1 \neq t$,~~
     (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.
     (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,2,9}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

  7d. **Compute** $\sigma_{\mathsf{inp},i}$:

  1. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
  2. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

Program $\mathsf{Prog}_{3,t,2,9}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_t, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

   i. **Verify** i **is positive:** If $i \leq 0$, output $\perp$.

   ii. **Verify input signature:**

   i. ~~If $i = t$,~~
   A. ~~If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.~~

   ii. ~~If $i \neq t$,~~
   A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
   B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

   iv. $\mathsf{in\text{-}type} = \perp$.

   v. **Verify A type signatures:**

   i. ~~If $i = t$,~~
   A. ~~If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.~~

ii. ~~If $i \neq t$,~~
   A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
   B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. If $\mathsf{in\text{-}type} = \bot$, output $\bot$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.
iii. If $i \notin \{t-1, t\}$,
   i. **Decrypt input and state**:
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
      iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
   ii. **Compute output value and next state:**
      A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
   iii. **Encrypt the new state**:
      i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
      ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
      iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
iii. If $i = t-1$,
   i. If $m_{i+1} = m^*_t$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.
iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,
   i. ~~If $i = t-1$,~~
      A. ~~$\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,t}, m_{i+1})$.~~
   ii. ~~If $i \neq t-1$,~~
      A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
      B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,
   i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.
   ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.48. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,8}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,9}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing, for $\alpha \in \{A, \mathsf{inp}\}$,

$$\mathsf{PPRF.Eval}(K_\alpha[t], v) = \mathsf{PPRF.Eval}(K_\alpha, v) \text{ for any } v \neq t$$

. Observe that our previous hybrid never evaluated punctured keys on their punctured points. This was true even in **Encryption Phase 2** since in that phase, we have $i \geq t^* + 1 > t$.

Additionally, the hardwired signing and verification keys of the previous hybrid are the same as what is computed in the current hybrid. Thus, $\mathsf{Prog}_{3,t,2,8}$ and $\mathsf{Prog}_{3,t,2,9}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,3}$: This hybrid is identical to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,9}$. Note that we now have two steps hardwired into the program. We have highlighted the differences between this hybrid and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2}$ below:

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $K_{\mathsf{inp}}, K_A, K_B, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.

   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.

   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

   (d) **Compute input ciphertexts**:

      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.

      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

   (e) **Set hardwired values**:

      i. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}_{\mathsf{inp},i}^{(0)} & \text{if } i < t \\ \mathsf{ct}_{\mathsf{inp},i}^{(b)} & \text{if } i \geq t \end{cases}$.

   (f) **Compute input signature keys**: Do nothing.

   (g) **Compute** $\sigma_{\mathsf{inp},i}$:

      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

   (h) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.

   (c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.

   (d) **Compute state and output values:**

      i. For $i \in [t^*]$,

         A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^*+1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^*+1]$,

A. $\mathsf{ct}_{\mathsf{st},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{st},i}^{(0)} & \text{if } i < t \\ \mathsf{ct}_{\mathsf{st},i}^{(b)} & \text{if } i \geq t \end{cases}$

B. $\mathsf{itr}_{\mathsf{st},i}^* = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}^*, \mathsf{ct}_{\mathsf{st},i}^*)$ where $\mathsf{itr}_{\mathsf{st},0}^* = \mathsf{itr}_{\mathsf{st},0}$.

C. $m_i^* = (i, \mathsf{ct}_{\mathsf{st},i}^*, \mathsf{itr}_{\mathsf{st},i-1}^*)$.

(h) **Compute state signature keys**: Do nothing.

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

ii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1^*)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},3,}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m_t^*, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*, y_t^*, \mathsf{ct}_{\mathsf{st},t+1}^*])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}^*, \sigma_{\mathsf{st},1}, \mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}_{3,\mathsf{t},3}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m_t^*, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*, y_t^*, \mathsf{ct}_{\mathsf{st},t+1}^*]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

i. **Verify i is positive:** If $i \leq 0$, output $\bot$.

ii. **Verify input signature:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. $\mathsf{in\text{-}type} = \bot$.

v. **Verify A type signatures:**

i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. If $\mathsf{in\text{-}type} = \bot$, output $\bot$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

iii. If $i \notin \{t-1, t\}$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = t-1$,

i. If $m_{i+1} = m^*_t$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.49. *For all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all PPT adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2,9}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,3}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Cleaning up the conditional statement. Hardwiring step t took a lot of prepwork and required us to iterate up through the entire computation. The end result was a conditional statement at $i = t-1$ that sets the $\mathsf{out\text{-}type} = A$ if and only if the outgoing message m_{i+1} is equal to our chosen message m_t^*. We now wish to remove this conditional statement, which we can do by going through all the prepwork hybrids in reverse order.

It would have been nice to reuse this prepwork for hardwiring and un-hardwiring other steps later in the proof. Unfortunately, it is unclear how to reuse this work as which step it allows you to hardwire is very dependent on how many initial B type signatures you set up, which is determined in the beginning of the prepwork. Thus, we simply remove all prepwork after each time we hardwire or unhardwire a step.

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4} = \mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,0}$: We remove the conditional statement at $i = t-1$ from the obfuscated program and remove all references to B type signatures. We have highlighted the differences between this hybrid and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0}$ below:

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.
   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (d) **Compute input ciphertexts**:
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.
   (e) **Set hardwired values**:
      i. $\mathsf{ct}_{\mathsf{inp},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{inp},i}^{(0)} & \text{if } i < t \\ \mathsf{ct}_{\mathsf{inp},i}^{(b)} & \text{if } i \geq t \end{cases}$.
   (f) **Compute input signature keys**: Do nothing.
   (g) **Compute** $\sigma_{\mathsf{inp},i}$:
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}^*)$.

(h) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

6. **KeyGen:**

(a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \bot$.

(b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.

(c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.

(d) **Compute state and output values:**

i. For $i \in [t^*]$,

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^* + 1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^* + 1]$,

A. $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}_{\mathsf{st},i}^{(0)} & \text{if } i < t \\ \mathsf{ct}_{\mathsf{st},i}^{(b)} & \text{if } i \geq t \end{cases}$

B. $\mathsf{itr}^*_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}^*_{\mathsf{st},i-1}, \mathsf{ct}^*_{\mathsf{st},i})$ where $\mathsf{itr}^*_{\mathsf{st},0} = \mathsf{itr}_{\mathsf{st},0}$.

C. $m_i^* = (i, \mathsf{ct}^*_{\mathsf{st},i}, \mathsf{itr}^*_{\mathsf{st},i-1})$.

(h) **Compute state signature keys**: Do nothing.

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

ii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1^*)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},4}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}^*_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\bot$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute** $\sigma_{\mathsf{inp},i}$:

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}_{3,\mathsf{t},4}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

i. **Verify i is positive:** If $i \leq 0$, output $\perp$.

ii. **Verify input signature:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. **Verify state signature:**

i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

iii. If $i \notin \{t-1, t\}$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. **Sign the new state:**

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.50. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme,* PPRF *is a puncturable pseudorandom function,* SSig *is a splittable signature scheme, and* Itr *is a cryptographic iterator, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,3}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,4}(1^\lambda)) \leq \mathsf{negl}(1^\lambda)$$

Proof. The proof is essentially the same as the proof of indistinguishability between $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4}$) and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,3}$).

The only difference between hybrids $(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0}, \mathbf{Hybrid}^{\mathcal{A}}_{3,t,2})$ and hybrids $(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4}, \mathbf{Hybrid}^{\mathcal{A}}_{3,t,3})$ is in the way they compute $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$ for $i = t$ in the computation step of the obfuscated program. In the former hybrids, they compute $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$ for $i = t$ by decrypting, computing the function, and re-encrypting. In the latter hybrids, they set $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$ for $i = t$ to the hardwired values $(y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

However, this difference does not affect the progression of the proof. If we simply change all the intermediate hybrids of the previous proof so that they compute $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$ for $i = t$ as in hybrids $(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4}, \mathbf{Hybrid}^{\mathcal{A}}_{3,t,3})$, then we get a proof for this lemma. We do not even have to modify any of the proofs of indistinguishability between the intermediate hybrids.

□

Replacing the t^{th} encrypted stream values. Now that the obfuscated program has the output values for steps $t-1$ and t hardwired into the program, it no longer needs to know the SKE keys and randomness at index t. Thus, we can swap the t^{th} input ciphertext and t^{th} state ciphertext of $x^{(b)}$ for those of $x^{(0)}$.

Hybrid$^{\mathcal{A}}_{3,t,4,1}$: We puncture K_E at t. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:

  1. $K_E[t] = \mathsf{PPRF.Punc}(K_E, t)$.
  2. If $i = t$,
     (a) $(r_{E,t}, r_{\mathsf{Enc},t}) = \mathsf{PPRF.Eval}(K_E, t)$.
     (b) $k_{E,t} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t})$.
     (c) $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,t}, \mathsf{Post.CT}_i^{(b)})$.
  3. If $i \neq t$,
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t], i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
     (d) $\mathsf{ct}^{(b)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

  5e. **Set hardwired values**:

  1. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t \\ \mathsf{ct}_{\mathsf{inp},i} & \text{if } i = t \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i > t \end{cases}$.

- During KeyGen, we make changes to the following steps:

  6e. **Compute state ciphertexts:**

  1. For $i \in [t^* + 1]$,
     (a) If $i = t$,
        i. $\mathsf{ct}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,t}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},t})$.
     (b) If $i \neq t$,
        i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t], i)$.
        ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
        iii. $\mathsf{ct}^{(0)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.
        iv. $\mathsf{ct}^{(b)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

  6g. **Set hardwired values:**

  1. For $i \in [t^*]$, $y^*_i = y_i$.
  2. For $i \in [t^* + 1]$,
     (a) $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i < t \\ \mathsf{ct}_{\mathsf{st},i} & \text{if } i = t \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i > t \end{cases}$

(b) $\mathsf{itr}^*_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}^*_{\mathsf{st},i-1}, \mathsf{ct}^*_{\mathsf{st},i})$ where $\mathsf{itr}^*_{\mathsf{st},0} = \mathsf{itr}_{\mathsf{st},0}$.

(c) $m^*_i = (i, \mathsf{ct}^*_{\mathsf{st},i}, \mathsf{itr}^*_{\mathsf{st},i-1})$.

6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},4,1}[K_{\mathsf{inp}}, K_A, K_E[t], \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

7c. **Compute** $\mathsf{ct}_{\mathsf{inp},i}$:

1. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t], i)$.
2. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
3. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}^{(b)}_i)$.

Program $\mathsf{Prog}_{3,\mathsf{t},4}[K_{\mathsf{inp}}, K_A, K_E[t], \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. **Verify state signature:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.
2. **Computation Step:**
   - i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   - ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.
   - iii. If $i \notin \{t-1, t\}$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t], i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E[t], i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   - i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. **Sign the new state:**

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.51. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,1}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing,

$$\mathsf{PPRF.Eval}(K_E[t], v) = \mathsf{PPRF.Eval}(K_E, v) \text{ for any } v \neq t.$$

Observe that we never evaluate punctured keys on their punctured points. This is true even in **Encryption Phase 2** since in that phase, we have $i \geq t^* + 1 > t$. It is also true within the obfuscated program since we only need to encrypt or decrypt using $K_E[t]$ on steps $i \notin \{t-1, t\}$. Furthermore, $\mathsf{ct}^*_{\mathsf{inp},t}$ and $\mathsf{ct}^*_{\mathsf{st},t}$ are set to the same values as in the previous hybrid.

Thus, $\mathsf{Prog}_{3,t,4,0}$ and $\mathsf{Prog}_{3,t,4,1}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,2}$: We change $r_{E,t}$ and $r_{\mathsf{Enc},t}$ to random values. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:

  1. $K_E[t] = \mathsf{PPRF.Punc}(K_E, t)$.
  2. If $i = t$,
     (a) $(r_{E,t}, r_{\mathsf{Enc},t}) \leftarrow \{0,1\}^\lambda \times \{0,1\}^\lambda$.
     (b) $k_{E,t} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t})$.
     (c) $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.
  3. If $i \neq t$,
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t], i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
     (d) $\mathsf{ct}^{(b)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

Lemma C.52. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$, *all* $t \in [t^*]$ *and all* PPT *adversaries* $\mathcal{A}$,

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,1}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,2}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by a reduction to the selective pseudorandomness at punctured points property of our PPRF.

Observe that we can run both hybrids without knowing K_E as long as we are given $(K_E[t], r_{E,t}, r_{\mathsf{Enc},t})$. In the reduction, without computing K_E, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,2}$ up to just before step 5d.2 of **Encryption Phase 1**. Then, we receive $(K_E[t], r_{E,t}, r_{\mathsf{Enc},t})$ from the PPRF challenger where $(r_{E,t}, r_{\mathsf{Enc},t})$ is either a random value or equal to $\mathsf{PPRF.Eval}(K_E, t)$. We use this randomness to compute $\mathsf{ct}_{\mathsf{inp},t}$ now (and to compute $\mathsf{ct}_{\mathsf{st},t}$ later in the hybrid). We then run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,2}$ starting from step 5d.3 of **Encryption Phase 1**. Observe that if $(r_{E,t}, r_{\mathsf{Enc},t})$ was a random value then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,2}$, and if $(r_{E,t}, r_{\mathsf{Enc},t})$ was equal to $\mathsf{PPRF.Eval}(K_E, t)$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,1}$. Thus, by PPRF security, the outputs of these hybrids must be indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,3}$: We change the t^{th} ciphertext to an encryption of stream $x^{(0)}$ instead of stream $x^{(b)}$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:
  1. $K_E[t] = \mathsf{PPRF.Punc}(K_E, t)$.
  2. If $i = t$,
     (a) $(r_{E,t}, r_{\mathsf{Enc},t}) \leftarrow \{0,1\}^\lambda \times \{0,1\}^\lambda$.
     (b) $k_{E,t} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t})$.
     (c) $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
  3. If $i \neq t$,
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t], i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
     (d) $\mathsf{ct}^{(b)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

- During KeyGen, we make changes to the following steps:

  6e. **Compute state ciphertexts:**
  1. For $i \in [t^* + 1]$,
     (a) If $i = t$,
        i. $\mathsf{ct}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,t}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},t})$.
     (b) If $i \neq t$,
        i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t], i)$.
        ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
        iii. $\mathsf{ct}^{(0)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.
        iv. $\mathsf{ct}^{(b)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

Lemma C.53. *If* SKE *is a secure encryption scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,2}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,3}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by a reduction to the security of SKE.

Observe that we can run both hybrids without knowing $(r_{E,t}, r_{\mathsf{Enc},t}, k_{E,t})$ as long as we are given $(\mathsf{ct}_{\mathsf{inp},t}, \mathsf{ct}_{\mathsf{st},t})$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,3}$ up to just before step 5d.2 of **Encryption Phase 1**. Then, we send challenge message pair $(\mathsf{Post.CT}_i^{(b)}, \mathsf{Post.CT}_i^{(0)})$ to the SKE challenger and receive an encryption ct_1 of one of the two messages. We set $\mathsf{ct}_{\mathsf{inp},t} = \mathsf{ct}_1$. Next, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,3}$ starting from step 5d.3 of **Encryption Phase 1** to just before step 6e of KeyGen. We then send another challenge message pair $(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.Dec.st}_i^{(0)})$ to the SKE challenger and receive an encryption ct_2 of one of the two messages. We set $\mathsf{ct}_{\mathsf{st},t} = \mathsf{ct}_2$. We also run the parts of step 6e of KeyGen that are not contained within the $i = t$ branch. We then run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,3}$ starting from step 6f of KeyGen. Observe that if $(\mathsf{ct}_1, \mathsf{ct}_2)$ were encryptions of the first message of each set (i.e. $(\mathsf{Post.CT}_i^{(b)}, \mathsf{Post.Dec.st}_i^{(b)})$) then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,2}$, and if $(\mathsf{ct}_1, \mathsf{ct}_2)$ were encryptions of the second message of each set (i.e. $(\mathsf{Post.CT}_i^{(0)}, \mathsf{Post.Dec.st}_i^{(0)})$) then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,3}$. Thus, by SKE security, the outputs of these hybrids must be indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,4,4}$: We change $(r_{E,t}, r_{\mathsf{Enc},t})$ back to PPRF evaluations. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:

  1. $K_E[t] = \mathsf{PPRF.Punc}(K_E, t)$.
  2. If $i = t$,
     (a) $(r_{E,t}, r_{\mathsf{Enc},t}) \leftarrow \mathsf{PPRF.Eval}(K_E, t)$.
     (b) $k_{E,t} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t})$.
     (c) $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
  3. If $i \neq t$,
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t], i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
     (d) $\mathsf{ct}^{(b)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

Lemma C.54. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,3}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,4}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by a reduction to the selective pseudorandomness at punctured points property of our PPRF. The reduction is essentially the same as that of Lemma C.52 as it is still the case that we can run both hybrids without knowing K_E as long as we are given $(K_E[t], r_{E,t}, r_{\mathsf{Enc},t})$. □

Hybrid$^{\mathcal{A}}_{3,t,4,5}$: We no longer puncture K_E at t. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:

  1. ~~$K_E[t] = \mathsf{PPRF.Punc}(K_E, t)$.~~
  2. ~~If $i = t$,~~
     (a) ~~$(r_{E,t}, r_{\mathsf{Enc},t}) = \mathsf{PPRF.Eval}(K_E, t)$.~~
     (b) ~~$k_{E,t} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t})$.~~
     (c) ~~$\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,t}, \mathsf{Post.CT}^{(0)}_i)$.~~
  3. ~~If $i \neq t$,~~
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}^{(0)}_i)$.
     (d) $\mathsf{ct}^{(b)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}^{(b)}_i)$.

  5e. **Set hardwired values**:

  1. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t+1 \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i \geq t+1 \end{cases}$.

- During KeyGen, we make changes to the following steps:

  6e. **Compute state ciphertexts:**

  1. For $i \in [t^* + 1]$,
     (a) ~~If $i = t$,~~
        i. ~~$\mathsf{ct}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,t}, \mathsf{Post.Dec.st}^{(0)}_i; r_{\mathsf{Enc},t})$.~~
     (b) ~~If $i \neq t$,~~
        i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
        ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
        iii. $\mathsf{ct}^{(0)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}^{(0)}_i; r_{\mathsf{Enc},i})$.
        iv. $\mathsf{ct}^{(b)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}^{(b)}_i; r_{\mathsf{Enc},i})$.

  6g. **Set hardwired values:**

  1. For $i \in [t^*]$, $y^*_i = y_i$.
  2. For $i \in [t^* + 1]$,
     (a) $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i < t+1 \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i \geq t+1 \end{cases}$
     (b) $\mathsf{itr}^*_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}^*_{\mathsf{st},i-1}, \mathsf{ct}^*_{\mathsf{st},i})$ where $\mathsf{itr}^*_{\mathsf{st},0} = \mathsf{itr}_{\mathsf{st},0}$.
     (c) $m^*_i = (i, \mathsf{ct}^*_{\mathsf{st},i}, \mathsf{itr}^*_{\mathsf{st},i-1})$.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,4,5}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

  7c. **Compute** $\mathsf{ct}_{\mathsf{inp},i}$:

1. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
2. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
3. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

Program $\mathsf{Prog}_{3,\mathsf{t},4}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. **Verify state signature:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.
2. **Computation Step:**
   - i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   - ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.
   - iii. If $i \notin \{t-1, t\}$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   - i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   - ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   - iii. **Sign the new state:**
     - i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
     - ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.55. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all PPT adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,4}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,5}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing,

$$\mathsf{PPRF.Eval}(K_E[t], v) = \mathsf{PPRF.Eval}(K_E, v) \text{ for any } v \neq t.$$

Observe that the previous hybrid never evaluate punctured keys on their punctured points. This was true even in **Encryption Phase 2** since in that phase, we have $i \geq t^* + 1 > t$. It was also true within the obfuscated program since we only needed to encrypt or decrypt using $K_E[t]$ on steps $i \notin \{t-1, t\}$. Furthermore, $\mathsf{ct}^*_{\mathsf{inp},t}$ and $\mathsf{ct}^*_{\mathsf{st},t}$ from the previous hybrid are set to the same values as in the current hybrid.

Thus, $\mathsf{Prog}_{3,t,4,4}$ and $\mathsf{Prog}_{3,t,4,5}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,5}$: This hybrid is identical to **Hybrid**$^{\mathcal{A}}_{3,t,4,5}$. Observe that we now encrypt stream $x^{(0)}$ for $i < t+1$. We have highlighted the differences between this hybrid and **Hybrid**$^{\mathcal{A}}_{3,t,0}$ below:

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.

   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.

   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

   (d) **Compute input ciphertexts**:

      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.

      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

   (e) **Set hardwired values**:

      i. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}_{\mathsf{inp},i}^{(0)} & \text{if } i < t+1 \\ \mathsf{ct}_{\mathsf{inp},i}^{(b)} & \text{if } i \geq t+1 \end{cases}$.

   (f) **Compute input signature keys**: Do nothing. (Will be added in a later hybrid.)

   (g) **Compute** $\sigma_{\mathsf{inp},i}$:

      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

   (h) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.

   (c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.

   (d) **Compute state and output values:**

      i. For $i \in [t^*]$,

         A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^*+1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^*+1]$,

A. $\mathsf{ct}_{\mathsf{st},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{st},i}^{(0)} & \text{if } i < t+1 \\ \mathsf{ct}_{\mathsf{st},i}^{(b)} & \text{if } i \geq t+1 \end{cases}$

B. $\mathsf{itr}_{\mathsf{st},i}^* = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}^*, \mathsf{ct}_{\mathsf{st},i}^*)$ where $\mathsf{itr}_{\mathsf{st},0}^* = \mathsf{itr}_{\mathsf{st},0}$.

C. $m_i^* = (i, \mathsf{ct}_{\mathsf{st},i}^*, \mathsf{itr}_{\mathsf{st},i-1}^*)$.

(h) **Compute state signature keys**: Do nothing. (Will be added in a later hybrid.)

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

ii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1^*)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},5}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*, y_t^*, \mathsf{ct}_{\mathsf{st},t+1}^*])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}^*, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}_{3,\mathsf{t},5}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*, y_t^*, \mathsf{ct}_{\mathsf{st},t+1}^*](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

i. **Verify i is positive:** If $i \leq 0$, output $\perp$.

ii. **Verify input signature:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. **Verify state signature:**

i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

iii. If $i \notin \{t-1, t\}$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. **Sign the new state:**

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.56. *For all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,4,5}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,5}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Preparing to un-hardwire step $t-1$. We now want to remove the hardwiring of step $t-1$ within the obfuscated program. To do so, we will use the same tools we used to hardwire step t. First, we will add in B type signatures up to step $t-2$. Then, we will iterate up through the computation to get a conditional statement at step $t-2$ that sets the $\mathsf{out\text{-}type} = A$ if and only if the outgoing message m_{i+1} is equal to our chosen message m^*_{t-1}.

Hybrid$^{\mathcal{A}}_{3,t,6}$: We add in B type signatures up to step $t-2$. We have highlighted the differences between this hybrid and **Hybrid**$^{\mathcal{A}}_{3,t,1}$ below. This hybrid is the same as the previous hybrid except that

- We compute Setup as

  3. **Setup:**
     (a) $K_{\mathsf{inp}}, K_A, K_B, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
     (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.

- During KeyGen, we make changes to the following steps:

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,6}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

Program
$\mathsf{Prog}_{3,t,6}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \perp$.
   v. **Verify A type signatures:**
      i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. **Verify B type signatures:**
      i. If $\mathsf{in\text{-}type} \neq A$ and $1 \leq i \leq t-2$,
         A. $(\mathsf{sgk}_{B,i}, \mathsf{vk}_{B,i}, \mathsf{vk}_{B,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{B,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = B$.
   vii. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**
   i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.
   iii. If $i \notin \{t-1, t\}$,

i. **Decrypt input and state**:
   i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
   ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
   iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
   iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
ii. **Compute output value and next state:**
   A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
iii. **Encrypt the new state**:
   i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
   ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
   iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:
   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   iii. **Sign A type messages:** If out-type $= A$,
      i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
   iv. **Sign B type messages:** If out-type $= B$,
      i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.
      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.57. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme,* PPRF *is a puncturable pseudorandom function, and* SSig *is a splittable signature scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,5}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,6}(1^\lambda)) \leq \mathsf{negl}(1^\lambda)$$

Proof. The proof is almost the same as the proof of indistinguishability between $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,0}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,5}$) and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,6}$).

The only differences are that

1. We only add in B type signatures up to step $t-2$ (as opposed to $t-1$). The proof can be easily modified to account for this change. We just do one fewer iteration.

2. We compute different values at step $i = t$ both within and outside of the obfuscated program. This difference does not affect the proof as the necessary hybrids are only concerned with values up to $i \leq t-1$.

Thus, essentially the same proof can be used for this lemma. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7} = \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,0}$: We have removed the B type verification branch in the obfuscated program. Furthermore, at step $t-2$, we set $\mathsf{out\text{-}type} = A$ if and only if the outgoing message m_{i+1} is equal to our chosen message m^*_{t-1}. We have highlighted the differences between this hybrid and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2}$.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $K_{\mathsf{inp}}, K_A, K_B, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.
   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (d) **Compute input ciphertexts**:
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
      iv. $\mathsf{ct}^{(b)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.
   (e) **Set hardwired values**:
      i. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t+1 \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i \geq t+1 \end{cases}$.
   (f) **Compute input signature keys**: Do nothing.
   (g) **Compute $\sigma_{\mathsf{inp},i}$**:
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.
   (h) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.
   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.
   (c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.
   (d) **Compute state and output values:**
      i. For $i \in [t^*]$,

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^* + 1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^* + 1]$,

A. $\mathsf{ct}_{\mathsf{st},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{st},i}^{(0)} & \text{if } i < t+1 \\ \mathsf{ct}_{\mathsf{st},i}^{(b)} & \text{if } i \geq t+1 \end{cases}$

B. $\mathsf{itr}_{\mathsf{st},i}^* = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}^*, \mathsf{ct}_{\mathsf{st},i}^*)$ where $\mathsf{itr}_{\mathsf{st},0}^* = \mathsf{itr}_{\mathsf{st},0}$.

C. $m_i^* = (i, \mathsf{ct}_{\mathsf{st},i}^*, \mathsf{itr}_{\mathsf{st},i-1}^*)$.

(h) **Compute state signature keys**: Do nothing.

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

ii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1^*)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},7}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m_{t-1}^*, y_{t-1}^*, \mathsf{ct}_{\mathsf{st},t}^*, y_t^*, \mathsf{ct}_{\mathsf{st},t+1}^*])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}^*, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}_{3,t,7}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\bot$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. $\mathsf{in\text{-}type} = \bot$.
   - v. **Verify A type signatures:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   - vi. If $\mathsf{in\text{-}type} = \bot$, output $\bot$.
2. **Computation Step:**
   - i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   - ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.
   - iii. If $i \notin \{t-1, t\}$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   - i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   - ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   - iii. If $i = t-2$,
     - i. If $m_{i+1} = m^*_{t-1}$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.
   - iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,
     - i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
     - ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.58. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme,* PPRF *is a puncturable pseudorandom function,* SSig *is a splittable signature scheme, and* Itr *is a cryptographic iterator, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,6}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7}(1^\lambda)) \leq \mathsf{negl}(1^\lambda)$$

Proof. The proof is almost the same as the proof of indistinguishability between $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,1}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,6}$) and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7}$).

The only differences are that

1. We only iterate the computation up to step $t-2$ with outgoing message m^*_{t-1} (as opposed to step $t-1$ with outgoing message m^*_t). The proof can be easily modified to account for this change since our previous hybrid only added in B type signatures up to step $t-1$. We just do one fewer iteration.

2. We compute different values at step $i = t$ both within and outside of the obfuscated program. This difference does not affect the proof as the necessary hybrids are only concerned with values up to $i \leq t-1$.

Thus, essentially the same proof can be used for this lemma. □

Un-hardwiring step $t-1$. We can now remove the hardwiring of step $t-1$ within the obfuscated program. We remove the hardwiring by doing essentially the reverse of what we did to add in hardwiring. Thus the proof of indistinguishability between $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,8}$ is very similar to the proof of indistinguishability between $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,2}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,8}$) and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,3}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7}$).

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,1}$: We start by puncturing both K_{inp} and K_A at $t-1$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:
  1. $K_{\mathsf{inp}}[t-1] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t-1)$.
  2. $r_{\mathsf{inp},t-1} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t-1)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t-1})$.
  4. $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1}) = (\bot, \mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1})$.

  5g. **Compute $\sigma_{\mathsf{inp},i}$**:
  1. If $i = t-1$, $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{inp},t-1})$.
  2. If $i \neq t-1$,
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t-1], i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**
  1. $K_A[t-1] = \mathsf{PPRF.Punc}(K_A, t)$.
  2. $r_{A,t-1} \leftarrow \mathsf{PPRF.Eval}(K_A, t-1)$.
  3. $(\mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1}, \mathsf{vk}_{A,t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t-1})$.
  4. $(\sigma^*_{\mathsf{st},t-1}, \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}) = (\bot, \mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1})$.

  6i. **Compute $\sigma_{\mathsf{st},1}$:**
  1. If $1 = t-1$, $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,1}, m^*_1)$.
  2. If $1 \neq t-1$,
     (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], 1))$.
     (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,7,1}[K_{\mathsf{inp}}[t-1], \mathsf{vk}^*_{\mathsf{inp},t-1}, K_A[t-1], \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

  7d. **Compute $\sigma_{\mathsf{inp},i}$**:
  1. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t-1], i))$.
  2. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

Program

$\mathsf{Prog}_{3,t,7,1}[K_{\mathsf{inp}}[t-1], \mathsf{vk}^*_{\mathsf{inp},t-1}, K_A[t-1], \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}]$

$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   - ii. **Verify input signature:**
     - i. If $i = t-1$,
       - A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t-1}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
     - ii. If $i \neq t-1$,
       - A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t-1], i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. $\mathsf{in\text{-}type} = \perp$.
   - v. **Verify A type signatures:**
     - i. If $i = t-1$,
       - A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t-1}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
     - ii. If $i \neq t-1$,
       - A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], i))$.
       - B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   - vi. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.
2. **Computation Step:**
   - i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.
   - ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.
   - iii. If $i \notin \{t-1, t\}$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = t-2$,

i. If $m_{i+1} = m^*_{t-1}$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. If $i = t-2$,

A. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,t-1}, m_{i+1})$.

ii. If $i \neq t-2$,

A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.59. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,1}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing, for $\alpha \in \{\mathsf{inp}, A\}$,

$$\mathsf{PPRF.Eval}(K_\alpha[t-1], v) = \mathsf{PPRF.Eval}(K_\alpha, v) \text{ for any } v \neq t-1$$

. Observe that we never evaluate punctured keys on their punctured points. This is true even in **Encryption Phase 2** since in that phase, we have $i \geq t^* + 1 > t - 1$.

Furthermore, the hardwired signing and verification keys are set to what they would have been computed to be in the previous hybrid. Thus, $\mathsf{Prog}_{3,t,7,0}$ and $\mathsf{Prog}_{3,t,7,1}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,7,2}$: We split our input signature and our A type signature at $t-1$ on $\mathsf{ct}^*_{\mathsf{inp},t-1}$ and m^*_{t-1} respectively. We replace signatures using $\mathsf{sgk}^*_{\mathsf{inp},t-1}$ or $\mathsf{sgk}^*_{A,t-1}$ with $\sigma_{\mathsf{inp},\mathsf{one},t-1}$ and $\sigma_{A,\mathsf{one},t-1}$ respectively.

This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:
  1. $K_{\mathsf{inp}}[t-1] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t-1)$.
  2. $r_{\mathsf{inp},t-1} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t-1)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t-1})$.
  4. $(\sigma_{\mathsf{inp},\mathsf{one},t-1}, \mathsf{vk}_{\mathsf{inp},\mathsf{one},t-1}, \mathsf{sgk}_{\mathsf{inp},\mathsf{abo},t-1}, \mathsf{vk}_{\mathsf{inp},\mathsf{abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{inp},t-1})$.
  5. $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1}) = (\sigma_{\mathsf{inp},\mathsf{one},t-1}, \bot, \mathsf{vk}_{\mathsf{inp},t-1})$.

  5g. **Compute** $\sigma_{\mathsf{inp},i}$:
  1. If $i = t-1$, $\sigma_{\mathsf{inp},i} = \sigma^*_{\mathsf{inp},t-1}$.
  2. If $i \neq t-1$,
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t-1], i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**
  1. $K_A[t-1] = \mathsf{PPRF.Punc}(K_A, t-1)$.
  2. $r_{A,t-1} \leftarrow \mathsf{PPRF.Eval}(K_A, t-1)$.
  3. $(\mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1}, \mathsf{vk}_{A,t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t-1})$.
  4. $(\sigma_{A,\mathsf{one},t-1}, \mathsf{vk}_{A,\mathsf{one},t-1}, \mathsf{sgk}_{A,\mathsf{abo},t-1}, \mathsf{vk}_{A,\mathsf{abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t-1}, m^*_{t-1})$.
  5. $(\sigma^*_{\mathsf{st},t-1}, \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}) = (\sigma_{A,\mathsf{one},t-1}, \bot, \mathsf{vk}_{A,t-1})$.

  6i. **Compute** $\sigma_{\mathsf{st},1}$**:**
  1. If $1 = t-1$, $\sigma_{\mathsf{st},i} = \sigma^*_{\mathsf{st},t-1}$.
  2. If $1 \neq t-1$,
     (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], 1))$.
     (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,7,2}[K_{\mathsf{inp}}[t-1], \mathsf{vk}^*_{\mathsf{inp},t-1}, K_A[t-1], \sigma^*_{\mathsf{st},t-1}, \mathsf{vk}^*_{A,t-1}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

Program $\mathsf{Prog}_{3,t,7,2}[K_{\mathsf{inp}}[t-1], \mathsf{vk}^*_{\mathsf{inp},t-1}, K_A[t-1], \sigma^*_{\mathsf{st},t-1}, \mathsf{vk}^*_{A,t-1}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify** i **is positive:** If $i \leq 0$, output $\bot$.
   ii. **Verify input signature:**
      i. If $i = t-1$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t-1}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
      ii. If $i \neq t-1$,

A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t-1], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. $\mathsf{in\text{-}type} = \bot$.

v. **Verify A type signatures:**

i. If $i = t-1$,

A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t-1}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

ii. If $i \neq t-1$,

A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. If $\mathsf{in\text{-}type} = \bot$, output $\bot$.

2. **Computation Step:**

i. If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

iii. If $i \notin \{t-1, t\}$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = t-2$,

i. If $m_{i+1} = m^*_{t-1}$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. If $i = t-2$,

A. $\sigma_{\mathsf{st},i+1} = \sigma^*_{\mathsf{st},t-1}$.

ii. If $i \neq t-2$,

A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.60. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,1}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,2}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of SSig, if $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}, v^*)$, then

$$\sigma_{\mathsf{one}} = \mathsf{SSig.Sign}(\mathsf{sgk}, v^*).$$

Thus, apart from the obfuscated programs, the hybrids act identically since $\sigma_{\mathsf{inp},\mathsf{one},t-1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{st},t-1})$ and $\sigma_{A,\mathsf{one},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$ so step 5g of **Encryption Phase 1** and step 6i of KeyGen result in the same signatures.

Now, in the previous hybrid, $\mathsf{Prog}_{3,t,7,1}$ only used $\mathsf{sgk}^*_{A,t-1} = \mathsf{sgk}_{A,t-1}$ in one place: If $\mathsf{out\text{-}type} = A$ and $i = t-2$, then it signed m_{i+1} with $\mathsf{sgk}_{A,t-1}$. However, we can only have $\mathsf{out\text{-}type} = A$ at $i = t-2$ if $m_{i+1} = m^*_{t-1}$. Thus, replacing the signature with $\sigma_{A,\mathsf{one},t-1}$ does not change the behavior of the program.

Therefore, $\mathsf{Prog}_{3,t,7,1}$ and $\mathsf{Prog}_{3,t,7,2}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,3}$: We change $r_{\mathsf{inp},t-1}$ and $r_{A,t-1}$ to random values. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[t-1] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t-1)$.
  2. $r_{\mathsf{inp},t-1} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t-1})$.
  4. $(\sigma_{\mathsf{inp,one},t-1}, \mathsf{vk}_{\mathsf{inp,one},t-1}, \mathsf{sgk}_{\mathsf{inp,abo},t-1}, \mathsf{vk}_{\mathsf{inp,abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{inp},t-1})$.
  5. $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1}) = (\sigma_{\mathsf{inp,one},t-1}, \bot, \mathsf{vk}_{\mathsf{inp},t-1})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[t-1] = \mathsf{PPRF.Punc}(K_A, t-1)$.
  2. $r_{A,t-1} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1}, \mathsf{vk}_{A,t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t-1})$.
  4. $(\sigma_{A,\mathsf{one},t-1}, \mathsf{vk}_{A,\mathsf{one},t-1}, \mathsf{sgk}_{A,\mathsf{abo},t-1}, \mathsf{vk}_{A,\mathsf{abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t-1}, m^*_{t-1})$.
  5. $(\sigma^*_{\mathsf{st},t-1}, \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}) = (\sigma_{A,\mathsf{one},t-1}, \bot, \mathsf{vk}_{A,t-1})$.

Lemma C.61. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,2}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,3}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by two reductions to the selective pseudorandomness at punctured points property of our PPRF.

We first swap out only $r_{\mathsf{inp},t-1}$ for a random value and leave $r_{A,t-1}$ as a PPRF evaluation. We call this intermediate hybrid $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,2.5}$.

Observe that we can run both the previous hybrid and the intermediate hybrid without knowing K_{inp} as long as we are given $(K_{\mathsf{inp}}[t-1], r_{\mathsf{inp},t-1})$. In the reduction, without computing K_{inp}, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,2.5}$ up to just before step 5f of **Encryption Phase 1**. Then, we receive $(K_{\mathsf{inp}}[t-1], r_{\mathsf{inp},t-1})$ from the PPRF challenger where $r_{\mathsf{inp},t-1}$ is either a random value or equal to $\mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t-1)$. We use this randomness to compute $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1})$. We then run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,2.5}$ starting from step 5g of **Encryption Phase 1**. Observe that if $r_{\mathsf{inp},t-1}$ was a random value then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,2.5}$, and if $r_{\mathsf{inp},t-1}$ was equal to $\mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t-1)$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,2}$. Thus, by PPRF security, the outputs of these hybrids must be indistinguishable.

By a similar reduction on the A type signature scheme, $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,2.5}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,3}$ are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,4}$: We set $\mathsf{vk}^*_{\mathsf{inp},t-1}$ to $\mathsf{vk}_{\mathsf{inp},\mathsf{one},t-1}$ which will only verify $\mathsf{ct}^*_{\mathsf{inp},t-1}$, and set $\mathsf{vk}^*_{A,t-1}$ to $\mathsf{vk}_{A,\mathsf{one},t-1}$ which will only verify m^*_{t-1}. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[t-1] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t-1)$.
  2. $r_{\mathsf{inp},t-1} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t-1})$.
  4. $(\sigma_{\mathsf{inp},\mathsf{one},t-1}, \mathsf{vk}_{\mathsf{inp},\mathsf{one},t-1}, \mathsf{sgk}_{\mathsf{inp},\mathsf{abo},t-1}, \mathsf{vk}_{\mathsf{inp},\mathsf{abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{inp},t-1})$.
  5. $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1}) = (\sigma_{\mathsf{inp},\mathsf{one},t-1}, \bot, \mathsf{vk}_{\mathsf{inp},\mathsf{one},t-1})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[t-1] = \mathsf{PPRF.Punc}(K_A, t-1)$.
  2. $r_{A,t-1} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1}, \mathsf{vk}_{A,t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t-1})$.
  4. $(\sigma_{A,\mathsf{one},t-1}, \mathsf{vk}_{A,\mathsf{one},t-1}, \mathsf{sgk}_{A,\mathsf{abo},t-1}, \mathsf{vk}_{A,\mathsf{abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t-1}, m^*_{t-1})$.
  5. $(\sigma^*_{\mathsf{st},t-1}, \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}) = (\sigma_{A,\mathsf{one},t-1}, \bot, \mathsf{vk}_{A,\mathsf{one},t-1})$.

Lemma C.62. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,3}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,4}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follow by two reductions to the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig.

We first swap out only $\mathsf{vk}^*_{\mathsf{inp},t-1}$ for $\mathsf{vk}_{\mathsf{inp},\mathsf{one},t-1}$ and leave $\mathsf{vk}^*_{A,t-1}$ as $\mathsf{vk}_{A,t-1}$. We call this intermediate hybrid $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,3.5}$.

Observe that we can run both hybrids without knowing $(r_{\mathsf{inp},t-1}, \mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1,\mathsf{rej}})$ as long as we are given $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1})$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,4}$ up to just before step 5f.2 of **Encryption Phase 1**. We send $\mathsf{ct}^*_{\mathsf{inp},t-1}$ to the SSig challenger and receive $(\sigma^*, \mathsf{vk}^*)$ from the SSig challenger where σ^* is a signature of $\mathsf{ct}^*_{\mathsf{inp},t-1}$ and vk^* is either a verification key $\mathsf{vk}_{\mathsf{one}}$ that only verifies $\mathsf{ct}^*_{\mathsf{inp},t-1}$ or is a regular verification key vk. We then set $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1}) = (\sigma^*, \bot, \mathsf{vk}^*)$ and run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,4}$ starting from step 5g of **Encryption Phase 1**. Observe that if vk^* was a regular verification key vk we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,3}$, and if vk^* was $\mathsf{vk}_{\mathsf{one}}$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,3.5}$. Thus, by the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig security, the outputs of these hybrids must be indistinguishable.

By a similar reduction on the A type signature scheme, $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,3.5}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,4}$ are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,7,5}$: We now remove the hardwiring of step $t-1$ from the program. Recall that in the previous hybrids, we set the following values in **Encryption Phase 1** and KeyGen:

5e. **Set hardwired values**:

1. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t+1 \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i \geq t+1 \end{cases}$.

6g. **Set hardwired values:**

1. For $i \in [t^*]$, $y^*_i = y_i$.
2. For $i \in [t^*+1]$,
   (a) $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i < t+1 \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i \geq t+1 \end{cases}$
   (b) $\mathsf{itr}^*_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}^*_{\mathsf{st},i-1}, \mathsf{ct}^*_{\mathsf{st},i})$ where $\mathsf{itr}^*_{\mathsf{st},0} = \mathsf{itr}_{\mathsf{st},0}$.
   (c) $m^*_i = (i, \mathsf{ct}^*_{\mathsf{st},i}, \mathsf{itr}^*_{\mathsf{st},i-1})$.

This hybrid is the same as the previous hybrid except that

- During KeyGen, we make changes to the following steps:

6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,7,5}[K_{\mathsf{inp}}[t-1], \mathsf{vk}^*_{\mathsf{inp},t-1}, K_A[t-1], \sigma^*_{\mathsf{st},t-1}, \mathsf{vk}^*_{A,t-1}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

Program
$\mathsf{Prog}_{3,t,7,5}[K_{\mathsf{inp}}[t-1], \mathsf{vk}^*_{\mathsf{inp},t-1}, K_A[t-1], \sigma^*_{\mathsf{st},t-1}, \mathsf{vk}^*_{A,t-1}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   ii. **Verify input signature:**
      i. If $i = t-1$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t-1}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
      ii. If $i \neq t-1$,
         A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t-1], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. $\mathsf{in\text{-}type} = \perp$.
   v. **Verify A type signatures:**
      i. If $i = t-1$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t-1}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
      ii. If $i \neq t-1$,
         A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], i))$.
         B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.
   vi. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**

   i. ~~If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.~~

   ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

   iii. If $i \neq t$,

      i. **Decrypt input and state**:

         i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

         ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

         iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

         iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

      ii. **Compute output value and next state:**

         A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

      iii. **Encrypt the new state**:

         i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

         ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

         iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

   iii. If $i = t-2$,

      i. If $m_{i+1} = m^*_{t-1}$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

   iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

      i. If $i = t-2$,

         A. $\sigma_{\mathsf{st},i+1} = \sigma^*_{\mathsf{st},t-1}$.

      ii. If $i \neq t-2$,

         A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], i+1))$.

         B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

   v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

      i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.63. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,4}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,5}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. We will show that $\mathsf{Prog}_{3,\mathsf{t},7}$ and $\mathsf{Prog}_{3,\mathsf{t},7,\mathsf{t}-1}$ have identical input/output behavior. The programs can only differ when $i = t-1$. Thus, we will restrict ourselves to this setting.

Since $\mathsf{vk}^*_{\mathsf{inp},t-1} = \mathsf{vk}_{\mathsf{inp},\mathsf{one},t-1}$, then in order to pass the verification step, it must be the case that

$$\mathsf{ct}_{\mathsf{inp},i} = \mathsf{ct}^*_{\mathsf{inp},t-1} = \mathsf{ct}^{(0)}_{\mathsf{inp},t-1}$$

Similarly, since $\mathsf{vk}^*_{A,t-1} = \mathsf{vk}_{A,\mathsf{one},t-1}$, in order to pass the verification step, we must have

$$m_i = m^*_{t-1} = (t-1, \mathsf{ct}^{(0)}_{\mathsf{st},t-1}, \mathsf{itr}^*_{\mathsf{st},t-2})$$

Therefore, by correctness of decryption of SKE and $\mathsf{Post\text{-}One\text{-}sFE}$, in the current hybrid, we compute

$$(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y_{t-1}, \mathsf{ct}^{(0)}_{\mathsf{st},t}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$$

But these are exactly the values that we had hardwired them to be in the previous hybrid. Thus the input/output behavior of the two programs is identical, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,6}$: We change $\mathsf{vk}^*_{\mathsf{inp},t-1}$ and $\mathsf{vk}^*_{A,t-1}$ back to $\mathsf{vk}_{\mathsf{inp},t-1}$ and $\mathsf{vk}_{A,t-1}$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[t-1] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t-1)$.
  2. $r_{\mathsf{inp},t-1} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t-1})$.
  4. $(\sigma_{\mathsf{inp,one},t-1}, \mathsf{vk}_{\mathsf{inp,one},t-1}, \mathsf{sgk}_{\mathsf{inp,abo},t-1}, \mathsf{vk}_{\mathsf{inp,abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{inp},t-1})$.
  5. $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1}) = (\sigma_{\mathsf{inp,one},t-1}, \bot, \mathsf{vk}_{\mathsf{inp},t-1})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[t-1] = \mathsf{PPRF.Punc}(K_A, t-1)$.
  2. $r_{A,t-1} \leftarrow \{0,1\}^\lambda$.
  3. $(\mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1}, \mathsf{vk}_{A,t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t-1})$.
  4. $(\sigma_{A,\mathsf{one},t-1}, \mathsf{vk}_{A,\mathsf{one},t-1}, \mathsf{sgk}_{A,\mathsf{abo},t-1}, \mathsf{vk}_{A,\mathsf{abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t-1}, m^*_{t-1})$.
  5. $(\sigma^*_{\mathsf{st},t-1}, \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}) = (\sigma_{A,\mathsf{one},t-1}, \bot, \mathsf{vk}_{A,t-1})$.

Lemma C.64. *If* SSig *is a splittable signature scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,5}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,6}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by two reductions to the $\mathsf{vk}_{\mathsf{one}}$ indistinguishabilty of SSig. The reductions are essentially the same as in Lemma C.62 as it is still the case that we can run both hybrids without knowing $(r_{\mathsf{inp},t-1}, \mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1,\mathsf{rej}})$ or $(r_{A,t-1}, \mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1}, \mathsf{vk}_{A,t-1,\mathsf{rej}})$ as long as we are given $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1})$ and $(\sigma^*_{A,t-1}, \mathsf{vk}^*_{A,t-1})$. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,7}$: We change $r_{\mathsf{inp},t-1}$ and $r_{A,t-1}$ back to PPRF values. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:

  1. $K_{\mathsf{inp}}[t-1] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t-1)$.
  2. $r_{\mathsf{inp},t-1} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t-1)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t-1})$.
  4. $(\sigma_{\mathsf{inp,one},t-1}, \mathsf{vk}_{\mathsf{inp,one},t-1}, \mathsf{sgk}_{\mathsf{inp,abo},t-1}, \mathsf{vk}_{\mathsf{inp,abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{inp},t-1})$.
  5. $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1}) = (\sigma_{\mathsf{inp,one},t-1}, \bot, \mathsf{vk}_{\mathsf{inp},t-1})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**

  1. $K_A[t-1] = \mathsf{PPRF.Punc}(K_A, t-1)$.
  2. $r_{A,t-1} \leftarrow \mathsf{PPRF.Eval}(K_A, t-1)$.
  3. $(\mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1}, \mathsf{vk}_{A,t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t-1})$.
  4. $(\sigma_{A,\mathsf{one},t-1}, \mathsf{vk}_{A,\mathsf{one},t-1}, \mathsf{sgk}_{A,\mathsf{abo},t-1}, \mathsf{vk}_{A,\mathsf{abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t-1}, m^*_{t-1})$.
  5. $(\sigma^*_{\mathsf{st},t-1}, \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}) = (\sigma_{A,\mathsf{one},t-1}, \bot, \mathsf{vk}_{A,t-1})$.

Lemma C.65. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^*]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,6}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,7}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by two reductions to the selective pseudorandomness at punctured points property of our PPRF. The reductions are essentially the same as that of Lemma C.61 as it is still the case that we can run both hybrids without knowing K_{inp} or K_A as long as we are given $(K_{\mathsf{inp}}[t-1], r_{\mathsf{inp},t-1})$ and $(K_A[t-1], r_{A,t-1})$. □

Hybrid$^{\mathcal{A}}_{3,t,7,8}$: We no longer split our input signature and our A type signature at $t-1$ on $\mathsf{ct}^*_{\mathsf{inp},t-1}$ and $m^*_t - 1$ respectively. We replace signatures $\sigma_{\mathsf{inp},\mathsf{one},t-1}$ and $\sigma_{A,\mathsf{one},t-1}$ with signatures using $\mathsf{sgk}^*_{\mathsf{inp},t-1}$ and $\mathsf{sgk}^*_{A,t-1}$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**:
  1. $K_{\mathsf{inp}}[t-1] \leftarrow \mathsf{PPRF.Punc}(K_{\mathsf{inp}}, t-1)$.
  2. $r_{\mathsf{inp},t-1} \leftarrow \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, t-1)$.
  3. $(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{\mathsf{inp},t-1})$.
  4. ~~$(\sigma_{\mathsf{inp},\mathsf{one},t-1}, \mathsf{vk}_{\mathsf{inp},\mathsf{one},t-1}, \mathsf{sgk}_{\mathsf{inp},\mathsf{abo},t-1}, \mathsf{vk}_{\mathsf{inp},\mathsf{abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{inp},t-1})$.~~
  5. $(\sigma^*_{\mathsf{inp},t-1}, \mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{vk}^*_{\mathsf{inp},t-1}) = (\bot, \mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{vk}_{\mathsf{inp},t-1})$.

  5g. **Compute** $\sigma_{\mathsf{inp},i}$:
  1. If $i = t-1$, $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.
  2. If $i \neq t-1$,
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t-1], i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:**
  1. $K_A[t-1] = \mathsf{PPRF.Punc}(K_A, t-1)$.
  2. $r_{A,t-1} \leftarrow \mathsf{PPRF.Eval}(K_A, t-1)$.
  3. $(\mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1}, \mathsf{vk}_{A,t-1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; r_{A,t-1})$.
  4. ~~$(\sigma_{A,\mathsf{one},t-1}, \mathsf{vk}_{A,\mathsf{one},t-1}, \mathsf{sgk}_{A,\mathsf{abo},t-1}, \mathsf{vk}_{A,\mathsf{abo},t-1}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}_{A,t-1}, m^*_{t-1})$.~~
  5. $(\sigma^*_{\mathsf{st},t-1}, \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}) = (\bot, \mathsf{sgk}_{A,t-1}, \mathsf{vk}_{A,t-1})$.

  6i. **Compute** $\sigma_{\mathsf{st},1}$**:**
  1. If $1 = t-1$, $\sigma_{\mathsf{st},i} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,t-1}, m^*_1)$.
  2. If $1 \neq t-1$,
     (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], 1))$.
     (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,7,j,8}[K_{\mathsf{inp}}[t-1], \mathsf{vk}^*_{\mathsf{inp},t-1}, K_A[t-1], \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

Program
$\mathsf{Prog}_{3,t,7,8}[K_{\mathsf{inp}}[t-1], \mathsf{vk}^*_{\mathsf{inp},t-1}, K_A[t-1], \mathsf{sgk}^*_{A,t-1}, \mathsf{vk}^*_{A,t-1}, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify** i **is positive:** If $i \leq 0$, output $\bot$.
   ii. **Verify input signature:**
      i. If $i = t-1$,
         A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t-1}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
      ii. If $i \neq t-1$,

A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}[t-1], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. $\mathsf{in\text{-}type} = \bot$.

v. **Verify A type signatures:**

i. If $i = t-1$,

A. If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t-1}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

ii. If $i \neq t-1$,

A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], i))$.

B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. If $\mathsf{in\text{-}type} = \bot$, output $\bot$.

2. **Computation Step:**

i. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

ii. If $i \neq t$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = t-2$,

i. If $m_{i+1} = m^*_{t-1}$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. If $i = t-2$,

A. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,t-1}, m_{i+1})$.

ii. If $i \neq t-2$,

A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A[t-1], i+1))$.

B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.66. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,7}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,8}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of SSig, if $(\sigma_{\mathsf{one}}, \mathsf{vk}_{\mathsf{one}}, \mathsf{sgk}_{\mathsf{abo}}, \mathsf{vk}_{\mathsf{abo}}) \leftarrow \mathsf{SSig.Split}(\mathsf{sgk}, v^*)$, then

$$\sigma_{\mathsf{one}} = \mathsf{SSig.Sign}(\mathsf{sgk}, v^*).$$

Thus, apart from the obfuscated programs, the hybrids act identically since $\sigma_{\mathsf{inp},\mathsf{one},t-1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{st},t-1})$ and $\sigma_{A,\mathsf{one},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$ so step 5g of **Encryption Phase 1** and step 6i of KeyGen result in the same signatures.

The current hybrid $\mathsf{Prog}_{3,t,7,8}$ only uses $\mathsf{sgk}^*_{A,t-1} = \mathsf{sgk}_{A,t-1}$ in one place: If $\mathsf{out\text{-}type} = A$ and $i = t-2$, then it signs m_{i+1} with $\mathsf{sgk}_{A,t-1}$. However, we can only have $\mathsf{out\text{-}type} = A$ at $i = t-2$ if $m_{i+1} = m^*_{t-1}$. Thus, the signature it generates is the same as the signature $\sigma^*_{\mathsf{st},t-1} = \sigma_{A,\mathsf{one},t-1}$ used in the previous hybrid.

Therefore, $\mathsf{Prog}_{3,t,7,7}$ and $\mathsf{Prog}_{3,t,7,8}$ have the same input/output behavior, so by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Hybrid$^{\mathcal{A}}_{3,t,7,9}$: We no longer puncture K_{inp} at $t-1$ and no longer use hardwired signatures and verification keys. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5f. **Compute input signature keys**: Do nothing.

  5g. **Compute** $\sigma_{\mathsf{inp},i}$:

  1. ~~If $i = t-1$, $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{inp},t-1})$.~~
  2. ~~If $i \neq t-1$,~~
     (a) $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     (b) $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

- During KeyGen, we make changes to the following steps:

  6h. **Compute state signature keys:** Do nothing.

  6i. **Compute** $\sigma_{\mathsf{st},1}$**:**

  1. ~~If $1 = t-1$, $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,1}, m^*_1)$.~~
  2. ~~If $1 \neq t-1$,~~
     (a) $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.
     (b) $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m^*_1)$.

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,t,7,j,9}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

  7d. **Compute** $\sigma_{\mathsf{inp},i}$:

  1. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
  2. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

Program $\mathsf{Prog}_{3,t,7,9}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m^*_{t-1}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}]$
$(i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

   i. **Verify** i **is positive:** If $i \leq 0$, output $\perp$.

   ii. **Verify input signature:**

   i. ~~If $i = t-1$,~~
      A. ~~If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{\mathsf{inp},t-1}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.~~

   ii. ~~If $i \neq t-1$,~~
      A. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

   iv. $\mathsf{in\text{-}type} = \perp$.

   v. **Verify A type signatures:**

   i. ~~If $i = t-1$,~~
      A. ~~If $\mathsf{SSig.Verify}(\mathsf{vk}^*_{A,t-1}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.~~

ii. ~~If $i \neq t-1$,~~
   A. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
   B. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. If $\mathsf{in\text{-}type} = \bot$, output $\bot$.

2. **Computation Step:**

i. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y_t^*, \mathsf{ct}^*_{\mathsf{st},t+1})$.

ii. If $i \neq t$,

   i. **Decrypt input and state**:
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
      iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
   ii. **Compute output value and next state:**
      A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
   iii. **Encrypt the new state**:
      i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
      ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
      iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = t-2$,

   i. If $m_{i+1} = m^*_{t-1}$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

   i. ~~If $i = t-2$,~~
      A. ~~$\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}^*_{A,t-1}, m_{i+1})$.~~
   ii. ~~If $i \neq t-2$,~~
      A. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
      B. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

   i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.
   ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.67. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,8}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,9}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing, for $\alpha \in \{\mathsf{inp}, A\}$,

$$\mathsf{PPRF.Eval}(K_\alpha[t-1], v) = \mathsf{PPRF.Eval}(K_\alpha, v) \text{ for any } v \neq t-1$$

. Observe that our previous hybrid never evaluated punctured keys on their punctured points. This was true even in **Encryption Phase 2** since in that phase, we have $i \geq t^* + 1 > t - 1$.

Additionally, the hardwired signing and verification keys of the previous hybrid are the same as what is computed in the current hybrid. Thus, $\mathsf{Prog}_{3,\mathsf{t},7,8}$ and $\mathsf{Prog}_{3,\mathsf{t},7,9}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{3,t,8}$: This hybrid is identical to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,9}$. Note that we now have two steps hardwired into the program. We have highlighted the differences between this hybrid and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7}$ below:

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $K_{\mathsf{inp}}, K_A, K_B, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.

   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.

   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

   (d) **Compute input ciphertexts**:

      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.

      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

   (e) **Set hardwired values**:

      i. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}_{\mathsf{inp},i}^{(0)} & \text{if } i < t+1 \\ \mathsf{ct}_{\mathsf{inp},i}^{(b)} & \text{if } i \geq t+1 \end{cases}$.

   (f) **Compute input signature keys**: Do nothing.

   (g) **Compute** $\sigma_{\mathsf{inp},i}$:

      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

   (h) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.

   (c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.

   (d) **Compute state and output values:**

      i. For $i \in [t^*]$,

         A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^*+1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^*+1]$,

A. $\mathsf{ct}_{\mathsf{st},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{st},i}^{(0)} & \text{if } i < t+1 \\ \mathsf{ct}_{\mathsf{st},i}^{(b)} & \text{if } i \geq t+1 \end{cases}$

B. $\mathsf{itr}_{\mathsf{st},i}^* = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}^*, \mathsf{ct}_{\mathsf{st},i}^*)$ where $\mathsf{itr}_{\mathsf{st},0}^* = \mathsf{itr}_{\mathsf{st},0}$.

C. $m_i^* = (i, \mathsf{ct}_{\mathsf{st},i}^*, \mathsf{itr}_{\mathsf{st},i-1}^*)$.

(h) **Compute state signature keys**: Do nothing.

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

ii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1^*)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},8}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m_{t-1}^*, y_t^*, \mathsf{ct}_{\mathsf{st},t+1}^*])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}^*, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}_{3,\mathsf{t},8}[K_{\mathsf{inp}}, K_A, K_B, K_E, \mathsf{pp}_{\mathsf{st}}, m_{t-1}^*, y_t^*, \mathsf{ct}_{\mathsf{st},t+1}^*](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

i. **Verify i is positive:** If $i \leq 0$, output $\perp$.

ii. **Verify input signature:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. $\mathsf{in\text{-}type} = \perp$.

v. **Verify A type signatures:**

i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 1$, $\mathsf{in\text{-}type} = \mathsf{out\text{-}type} = A$.

vi. If $\mathsf{in\text{-}type} = \perp$, output $\perp$.

2. **Computation Step:**

i. ~~If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.~~

ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.

iii. If $i = t$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. If $i = t-2$,

i. If $m_{i+1} = m^*_{t-1}$, $\mathsf{out\text{-}type} = A$. Else, $\mathsf{out\text{-}type} = B$.

iv. **Sign A type messages:** If $\mathsf{out\text{-}type} = A$,

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

v. **Sign B type messages:** If $\mathsf{out\text{-}type} = B$,

i. $(\mathsf{sgk}_{B,i+1}, \mathsf{vk}_{B,i+1}, \mathsf{vk}_{B,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_B, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{B,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.68. *For all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7,9}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,8}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Hybrid$^{\mathcal{A}}_{3,t,9}$: We remove the conditional statement at $i = t - 1$ from the obfuscated program and remove all references to B type signatures. We have highlighted the differences between this hybrid and **Hybrid**$^{\mathcal{A}}_{3,t,5}$ below:

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.

   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.

4. **Challenge Bit:** $b \leftarrow \{0, 1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.

   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

   (d) **Compute input ciphertexts**:

      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.

      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

   (e) **Set hardwired values**:

      i. $\mathsf{ct}_{\mathsf{inp},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{inp},i}^{(0)} & \text{if } i < t+1 \\ \mathsf{ct}_{\mathsf{inp},i}^{(b)} & \text{if } i \geq t+1 \end{cases}$.

   (f) **Compute input signature keys**: Do nothing. (Will be added in a later hybrid.)

   (g) **Compute** $\sigma_{\mathsf{inp},i}$:

      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}^*)$.

   (h) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}^*, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.

   (c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.

   (d) **Compute state and output values:**

      i. For $i \in [t^*]$,

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^* + 1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^* + 1]$,

A. $\mathsf{ct}_{\mathsf{st},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{st},i}^{(0)} & \text{if } i < t+1 \\ \mathsf{ct}_{\mathsf{st},i}^{(b)} & \text{if } i \geq t+1 \end{cases}$

B. $\mathsf{itr}_{\mathsf{st},i}^* = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}^*, \mathsf{ct}_{\mathsf{st},i}^*)$ where $\mathsf{itr}_{\mathsf{st},0}^* = \mathsf{itr}_{\mathsf{st},0}$.

C. $m_i^* = (i, \mathsf{ct}_{\mathsf{st},i}^*, \mathsf{itr}_{\mathsf{st},i-1}^*)$.

(h) **Compute state signature keys**: Do nothing. (Will be added in a later hybrid.)

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

ii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1^*)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{3,\mathsf{t},9}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y_t^*, \mathsf{ct}_{\mathsf{st},t+1}^*])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}^*, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program $\mathsf{Prog}_{3,t,9}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   i. **Verify i is positive:** If $i \leq 0$, output $\bot$.
   ii. **Verify input signature:**
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
   iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   iv. **Verify state signature:**
      i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
      ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\bot$.
2. **Computation Step:**
   i. ~~If $i = t-1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.~~
   ii. If $i = t$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_t, \mathsf{ct}^*_{\mathsf{st},t+1})$.
   iii. If $i \neq t$,
      i. **Decrypt input and state**:
         i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
         ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
         iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
         iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
      ii. **Compute output value and next state:**
         A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
      iii. **Encrypt the new state**:
         i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
         ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
         iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   iii. **Sign the new state:**
      i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
      ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.69. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, PPRF is a puncturable pseudorandom function, SSig is a splittable signature scheme, and Itr is a cryptographic iterator, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*]$, and all PPT adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,8}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t,9}(1^\lambda)) \leq \mathsf{negl}(1^\lambda)$$

Proof. The proof is similar to the proof of indistinguishability between $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,5}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,9}$) and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,7}$ (which is similar to $\mathbf{Hybrid}^{\mathcal{A}}_{3,t,8}$).

The only difference between hybrids ($\mathbf{Hybrid}^{\mathcal{A}}_{3,t,5}, \mathbf{Hybrid}^{\mathcal{A}}_{3,t,7}$) and hybrids ($\mathbf{Hybrid}^{\mathcal{A}}_{3,t,9}, \mathbf{Hybrid}^{\mathcal{A}}_{3,t,8}$) is in the way they compute $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$ for $i = t-1$ in the computation step of the obfuscated program. In the latter hybrids, they compute $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$ for $i = t-1$ by decrypting, computing the function, and re-encrypting. In the former hybrids, they set $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$ for $i = t-1$ to the hardwired values $(y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t})$.

However, the proof can be easily modified to accommodate these changes. First, we change all the intermediate hybrids of the previous proof so that they compute $(y_i, \mathsf{ct}_{\mathsf{st},i+1})$ for $i = t-1$ as in hybrids ($\mathbf{Hybrid}^{\mathcal{A}}_{3,t,9}, \mathbf{Hybrid}^{\mathcal{A}}_{3,t,8}$). Then, the only lemmas affected by these changes are the following:

- Lemma corresponding to Lemma C.26:
  Observe that the proof relies on the adjacent hybrids having the same behavior during the computation step regardless of whether $\mathsf{in\text{-}type} = A$ or $\mathsf{in\text{-}type} = B$. Although we have changed the behavior of the computation step so that it does not hardcode values at $i = t-1$, the computation step is still oblivious of whether $\mathsf{in\text{-}type} = A$ or $\mathsf{in\text{-}type} = B$. Thus, the proof still follows.

- Lemma corresponding to Lemma C.33:
  This proof relies on the fact that if $(\mathsf{ct}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}) = (\mathsf{ct}^*_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{st},i})$, then we will compute $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_i, \mathsf{ct}^*_{\mathsf{st},i+1})$. For $i = t-1$, since $(\mathsf{ct}^*_{\mathsf{inp},t-1}, \mathsf{ct}^*_{\mathsf{st},t-1}) = (\mathsf{ct}^{(0)}_{\mathsf{inp},t-1}, \mathsf{ct}^{(0)}_{\mathsf{st},t-1})$ and $(y^*_{t-1}, \mathsf{ct}^*_{\mathsf{st},t}) = (y^*_{t-1}, \mathsf{ct}^{(0)}_{\mathsf{st},t})$, then we will compute the correct values regardless of whether they are hardcoded (as in the previous proof) or computed by decrypting, evaluating, and re-encrypting (as in the current proof). Thus, the proof still follows.

□

Lemma C.70. *For all $\lambda \in \mathbb{N}$, all $t \in [t^*]$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t,9}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{3,t+1,0}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Erasing the streams from ciphertext t^*. We could continue advancing through our hybrids until we have reached $\mathbf{Hybrid}^{\mathcal{A}}_{3,t^*+1,0}$. At this point, we would encrypt stream $x^{(0)}$ for $i \leq t^*$ and would encrypt stream $x^{(b)}$ for $i > t^*$.

For $i > t^*$ we want to rely on the security of Post-One-sFE. However, in Post-One-sFE, the security of each ciphertext is correlated with the security of adjacent ciphertexts. Thus, to break this chain of dependencies, we actually want to remove both streams entirely from ciphertext t^*.

Therefore, rather than starting with $\mathbf{Hybrid}^{\mathcal{A}}_{3,t^*+1,0}$, we will actually start with $\mathbf{Hybrid}^{\mathcal{A}}_{3,t^*,4}$. At this stage, we have both steps $t^* - 1$ and t^* hardwired into the program. Then, we will replace the encryption of $x^{(b)}$ at t^* with an encryption of $\bot$. The following sequence of hybrids is very similar to the proof of indistinguishabilty between $\mathbf{Hybrid}^{\mathcal{A}}_{3,t^*,4}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{3,t^*,5}$.

$\mathbf{Hybrid}^{\mathcal{A}}_{4} = \mathbf{Hybrid}^{\mathcal{A}}_{4,0}$: This is the same as hybrid $\mathbf{Hybrid}^{\mathcal{A}}_{3,t^*,4}$. We have marked the minor notational differences below.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.

   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.

4. **Challenge Bit:** $b \leftarrow \{0, 1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\bot$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.

   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

   (d) **Compute input ciphertexts**:

      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.

      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

   (e) **Set hardwired values**:

      i. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t^* \\ \mathsf{ct}^{(b)}_{\mathsf{inp},i} & \text{if } i = t^* \end{cases}$.

   (f) **Compute input signature keys**: Do nothing. (Will be added in a later hybrid.)

   (g) **Compute $\sigma_{\mathsf{inp},i}$**:

      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

(h) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

6. **KeyGen:**

(a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \bot$.

(b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.

(c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.

(d) **Compute state and output values:**

i. For $i \in [t^*]$,

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^* + 1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^* + 1]$,

A. $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i = 1 \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i \in \{t^*, t^* + 1\} \end{cases}$

B. ~~$\mathsf{itr}^*_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}^*_{\mathsf{st},i-1}, \mathsf{ct}^*_{\mathsf{st},i})$ where $\mathsf{itr}^*_{\mathsf{st},0} = \mathsf{itr}_{\mathsf{st},0}$.~~

C. ~~$m_i^* = (i, \mathsf{ct}^*_{\mathsf{st},i}, \mathsf{itr}^*_{\mathsf{st},i-1})$.~~

(h) **Compute state signature keys**: Do nothing. (Will be added in a later hybrid.)

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $m_1 = (1, \mathsf{ct}^*_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$.

ii. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

iii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_4[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t^*, y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*}, y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1}])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}^*_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\bot$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute** $\sigma_{\mathsf{inp},i}$:

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program

$\mathsf{Prog}_4[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t^*, y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*}, y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:

i. **Verify i is positive:** If $i \leq 0$, output $\perp$.

ii. **Verify input signature:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.

iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.

iv. **Verify state signature:**

i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.

ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.

2. **Computation Step:**

i. If $i = t^* - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*})$.

ii. If $i = t^*$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1})$.

iii. If $i \notin \{t^* - 1, t^*\}$,

i. **Decrypt input and state**:

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.

iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.

ii. **Compute output value and next state:**

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.

iii. **Encrypt the new state**:

i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.

ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.

iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.

3. **Authentication Step**:

i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.

ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

iii. **Sign the new state:**

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.71. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{3,t^*,4}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{4}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Hybrid$^{\mathcal{A}}_{4,1}$: We puncture K_E at t. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:
  1. $K_E[t^*] = \mathsf{PPRF.Punc}(K_E, t^*)$.
  2. If $i = t^*$,
     (a) $(r_{E,t^*}, r_{\mathsf{Enc},t^*}) = \mathsf{PPRF.Eval}(K_E, t^*)$.
     (b) $k_{E,t^*} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t^*})$.
     (c) $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,t^*}, \mathsf{Post.CT}_i^{(b)})$.
  3. If $i \neq t^*$,
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t^*], i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
     (d) ~~$\mathsf{ct}^{(b)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.~~

  5e. **Set hardwired values**:
  1. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t^* \\ \mathsf{ct}_{\mathsf{inp},i} & \text{if } i = t^* \end{cases}$.

- During KeyGen, we make changes to the following steps:

  6e. **Compute state ciphertexts:**
  1. For $i \in [t^* + 1]$,
     (a) If $i = t^*$,
        i. $\mathsf{ct}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,t^*}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},t^*})$.
     (b) If $i \neq t^*$,
        i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t^*], i)$.
        ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
        iii. $\mathsf{ct}^{(0)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.
        iv. $\mathsf{ct}^{(b)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

  6g. **Set hardwired values:**
  1. For $i \in [t^*]$, $y^*_i = y_i$.
  2. For $i \in [t^* + 1]$,
     (a) $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i = 1 \\ \mathsf{ct}_{\mathsf{st},i} & \text{if } i = t^* \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i = t^* + 1 \end{cases}$

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{4,1}[K_{\mathsf{inp}}, K_A, K_E[t^*], \mathsf{pp}_{\mathsf{st}}, t^*, y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*}, y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

  7c. **Compute** $\mathsf{ct}_{\mathsf{inp},i}$:
  1. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t^*], i)$.
  2. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
  3. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

Program

$\mathsf{Prog}_{4,1}[K_{\mathsf{inp}}, K_A, K_E[t^*], \mathsf{pp}_{\mathsf{st}}, t^*, y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*}, y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. **Verify state signature:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.
2. **Computation Step:**
   - i. If $i = t^* - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*})$.
   - ii. If $i = t^*$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1})$.
   - iii. If $i \notin \{t^* - 1, t^*\}$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t^*], i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E[t^*], i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   - i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   - ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   - iii. **Sign the new state:**
     - i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
     - ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.72. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{4,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{4,1}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing,

$$\mathsf{PPRF.Eval}(K_E[t^*], v) = \mathsf{PPRF.Eval}(K_E, v) \text{ for any } v \neq t^*.$$

Observe that we never evaluate punctured keys on their punctured points. This is true even in **Encryption Phase 2** since in that phase, we have $i \geq t^* + 1 > t^*$. It is also true within the obfuscated program since we only need to encrypt or decrypt using $K_E[t^*]$ on steps $i \notin \{t^* - 1, t^*\}$. Furthermore, $\mathsf{ct}^*_{\mathsf{inp},t^*}$ and $\mathsf{ct}^*_{\mathsf{st},t^*}$ are set to the same values as in the previous hybrid.

Thus, $\mathsf{Prog}_{4,0}$ and $\mathsf{Prog}_{4,1}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{4,2}$: We change r_{E,t^*} and r_{Enc,t^*} to random values. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:

  1. $K_E[t^*] = \mathsf{PPRF.Punc}(K_E, t^*)$.
  2. If $i = t^*$,
     (a) $(r_{E,t^*}, r_{\mathsf{Enc},t^*}) \leftarrow \{0,1\}^\lambda \times \{0,1\}^\lambda$.
     (b) $k_{E,t^*} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t^*})$.
     (c) $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.
  3. If $i \neq t^*$,
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t^*], i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.

Lemma C.73. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{4,1}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{4,2}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by a reduction to the selective pseudorandomness at punctured points property of our PPRF.

Observe that we can run both hybrids without knowing K_E as long as we are given $(K_E[t^*], r_{E,t^*}, r_{\mathsf{Enc},t^*})$. In the reduction, without computing K_E, we run $\mathbf{Hybrid}^{\mathcal{A}}_{4,2}$ up to just before step 5d.2 of **Encryption Phase 1**. Then, we receive $(K_E[t^*], r_{E,t^*}, r_{\mathsf{Enc},t^*})$ from the PPRF challenger where $(r_{E,t^*}, r_{\mathsf{Enc},t^*})$ is either a random value or equal to $\mathsf{PPRF.Eval}(K_E, t^*)$. We use this randomness to compute $\mathsf{ct}_{\mathsf{inp},t^*}$ now (and to compute $\mathsf{ct}_{\mathsf{st},t^*}$ later in the hybrid). We then run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{4,2}$ starting from step 5d.3 of **Encryption Phase 1**. Observe that if $(r_{E,t^*}, r_{\mathsf{Enc},t^*})$ was a random value then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{4,2}$, and if $(r_{E,t^*}, r_{\mathsf{Enc},t^*})$ was equal to $\mathsf{PPRF.Eval}(K_E, t^*)$ we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{4,1}$. Thus, by PPRF security, the outputs of these hybrids must be indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{4,3}$: We change ciphertext t^* so that it encrypts $\bot$ instead of stream $x^{(b)}$. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:
  1. $K_E[t^*] = \mathsf{PPRF.Punc}(K_E, t^*)$.
  2. If $i = t^*$,
     (a) $(r_{E,t^*}, r_{\mathsf{Enc},t^*}) \leftarrow \{0,1\}^\lambda \times \{0,1\}^\lambda$.
     (b) $k_{E,t^*} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t^*})$.
     (c) $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \bot)$.
  3. If $i \neq t^*$,
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t^*], i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}^{(0)}_i)$.

- During KeyGen, we make changes to the following steps:

  6e. **Compute state ciphertexts:**
  1. For $i \in [t^* + 1]$,
     (a) If $i = t^*$,
        i. $\mathsf{ct}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,t^*}, \bot; r_{\mathsf{Enc},t^*})$.
     (b) If $i \neq t^*$,
        i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t^*], i)$.
        ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
        iii. $\mathsf{ct}^{(0)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}^{(0)}_i; r_{\mathsf{Enc},i})$.
        iv. $\mathsf{ct}^{(b)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}^{(b)}_i; r_{\mathsf{Enc},i})$.

Lemma C.74. *If* SKE *is a secure encryption scheme, then for all* $\lambda \in \mathbb{N}$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{4,2}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{4,3}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by a reduction to the security of SKE.

Observe that we can run both hybrids without knowing $(r_{E,t^*}, r_{\mathsf{Enc},t^*}, k_{E,t^*})$ as long as we are given $(\mathsf{ct}_{\mathsf{inp},t^*}, \mathsf{ct}_{\mathsf{st},t^*})$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{4,3}$ up to just before step 5d.2 of **Encryption Phase 1**. Then, we send challenge message pair $(\mathsf{Post.CT}^{(b)}_i, \bot)$ to the SKE challenger and receive an encryption ct_1 of one of the two messages. We set $\mathsf{ct}_{\mathsf{inp},t^*} = \mathsf{ct}_1$. Next, we run $\mathbf{Hybrid}^{\mathcal{A}}_{4,3}$ starting from step 5d.3 of **Encryption Phase 1** to just before step 6e of KeyGen. We then send another challenge message pair $(\mathsf{Post.Dec.st}^{(b)}_i, \bot)$ to the SKE challenger and receive an encryption ct_2 of one of the two messages. We set $\mathsf{ct}_{\mathsf{st},t^*} = \mathsf{ct}_2$. We also run the parts of step 6e of KeyGen that are not contained within the $i = t^*$ branch. We then run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{4,3}$ starting from step 6f of KeyGen. Observe that if $(\mathsf{ct}_1, \mathsf{ct}_2)$ were encryptions of the first message of each set (i.e. $(\mathsf{Post.CT}^{(b)}_i, \mathsf{Post.Dec.st}^{(b)}_i)$) then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{4,2}$, and if $(\mathsf{ct}_1, \mathsf{ct}_2)$ were encryptions of the second message of each set (i.e. $(\bot, \bot)$) then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{4,3}$. Thus, by SKE security, the outputs of these hybrids must be indistinguishable. □

$\mathbf{Hybrid}^{\mathcal{A}}_{4,4}$: We change $(r_{E,t^*}, r_{\mathsf{Enc},t^*})$ back to PPRF evaluations. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:

  1. $K_E[t^*] = \mathsf{PPRF.Punc}(K_E, t^*)$.
  2. If $i = t^*$,
     (a) $(r_{E,t^*}, r_{\mathsf{Enc},t^*}) \leftarrow \mathsf{PPRF.Eval}(K_E, t^*)$.
     (b) $k_{E,t^*} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t^*})$.
     (c) $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \bot)$.
  3. If $i \neq t^*$,
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E[t^*], i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}^{(0)}_i)$.

Lemma C.75. *If* PPRF *is a puncturable pseudorandom function, then for all* $\lambda \in \mathbb{N}$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{4,3}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{4,4}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by a reduction to the selective pseudorandomness at punctured points property of our PPRF. The reduction is essentially the same as that of Lemma C.73 as it is still the case that we can run both hybrids without knowing K_E as long as we are given $(K_E[t^*], r_{E,t^*}, r_{\mathsf{Enc},t^*})$. □

Hybrid$^{\mathcal{A}}_{4,5}$: We no longer puncture K_E at t^*. This hybrid is the same as the previous hybrid except that

- During **Encryption Phase 1**, we make changes to the following steps:

  5d. **Compute input ciphertexts**:

  1. ~~$K_E[t^*] = \mathsf{PPRF.Punc}(K_E, t^*)$.~~
  2. ~~If $i = t^*$,~~
     (a) ~~$(r_{E,t^*}, r_{\mathsf{Enc},t^*}) = \mathsf{PPRF.Eval}(K_E, t^*)$.~~
     (b) ~~$k_{E,t^*} = \mathsf{SKE.Setup}(1^\lambda; r_{E,t^*})$.~~
     (c) ~~$\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,t^*}, \bot)$.~~
  3. ~~If $i \neq t^*$,~~
     (a) $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
     (b) $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
     (c) $\mathsf{ct}^{(0)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}^{(0)}_i)$.
     (d) $\mathsf{ct}^{(\bot)}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \bot)$.

  5e. **Set hardwired values**:

  1. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t^* \\ \mathsf{ct}^{(\bot)}_{\mathsf{inp},i} & \text{if } i = t^* \end{cases}$.

- During KeyGen, we make changes to the following steps:

  6e. **Compute state ciphertexts:**

  1. For $i \in [t^* + 1]$,
     (a) ~~If $i = t^*$,~~
        i. ~~$\mathsf{ct}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,t^*}, \bot; r_{\mathsf{Enc},t^*})$.~~
     (b) ~~If $i \neq t^*$,~~
        i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
        ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
        iii. $\mathsf{ct}^{(0)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}^{(0)}_i; r_{\mathsf{Enc},i})$.
        iv. $\mathsf{ct}^{(\bot)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \bot; r_{\mathsf{Enc},i})$.
        v. $\mathsf{ct}^{(b)}_{\mathsf{st},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}^{(b)}_i; r_{\mathsf{Enc},i})$.

  6g. **Set hardwired values:**

  1. For $i \in [t^*]$, $y^*_i = y_i$.
  2. For $i \in [t^* + 1]$,
     (a) $\mathsf{ct}^*_{\mathsf{st},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{st},i} & \text{if } i = 1 \\ \mathsf{ct}^{(\bot)}_{\mathsf{st},i} & \text{if } i = t^* \\ \mathsf{ct}^{(b)}_{\mathsf{st},i} & \text{if } i = t^* + 1 \end{cases}$

  6j. **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_{4,5}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t^*, y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*}, y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1}])$.

- During **Encryption Phase 2**, we make changes to the following steps:

  7c. **Compute** $\mathsf{ct}_{\mathsf{inp},i}$:

1. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
2. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
3. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

Program

$\mathsf{Prog}_{4,5}[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t^*, y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*}, y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\perp$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\perp$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. **Verify state signature:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\perp$.
2. **Computation Step:**
   - i. If $i = t^* - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*})$.
   - ii. If $i = t^*$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1})$.
   - iii. If $i \notin \{t^* - 1, t^*\}$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   - i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   - ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   - iii. **Sign the new state:**

i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.

ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.

4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.76. *If $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme, then for all $\lambda \in \mathbb{N}$, all $t^* \in [t^*]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{4,4}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{4,5}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By correctness of puncturing,

$$\mathsf{PPRF.Eval}(K_E[t^*], v) = \mathsf{PPRF.Eval}(K_E, v) \text{ for any } v \neq t^*.$$

Observe that the previous hybrid never evaluate punctured keys on their punctured points. This was true even in **Encryption Phase 2** since in that phase, we have $i \geq t^* + 1 > t^*$. It was also true within the obfuscated program since we only needed to encrypt or decrypt using $K_E[t^*]$ on steps $i \notin \{t^* - 1, t^*\}$. Furthermore, $\mathsf{ct}^*_{\mathsf{inp},t^*}$ and $\mathsf{ct}^*_{\mathsf{st},t^*}$ from the previous hybrid are set to the same values as in the current hybrid.

Thus, $\mathsf{Prog}_{4,4}$ and $\mathsf{Prog}_{4,5}$ have the same input/output behavior. For the same reasons, apart from the obfuscated programs, the hybrids are identical.

Thus, by a straightforward reduction to the security of $i\mathcal{O}$, the hybrids are indistinguishable. □

Cleaning up the hybrid. We now want to write our hybrids in terms of Pre-One-sFE's algorithms. First, however, we will make some notational changes to make the hybrid easier to understand.

$\mathbf{Hybrid}_5^{\mathcal{A}} = \mathbf{Hybrid}_{5,0}^{\mathcal{A}}$: This is the same as $\mathbf{Hybrid}_{4,5}^{\mathcal{A}}$. We have marked the changes between this hybrid and $\mathbf{Hybrid}_4^{\mathcal{A}}$.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.
   (c) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (d) **Compute input ciphertexts**:
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.
      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(\perp)} = \mathsf{SKE.Enc}(k_{E,i}, \perp)$.
   (e) **Set hardwired values**:
      i. $\mathsf{ct}_{\mathsf{inp},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{inp},i}^{(0)} & \text{if } i < t^* \\ \mathsf{ct}_{\mathsf{inp},i}^{(\perp)} & \text{if } i = t^* \end{cases}$.
   (f) **Compute $\sigma_{\mathsf{inp},i}$**:
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}^*)$.
   (g) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}^*, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.
   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.
   (c) $\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.
   (d) **Compute state and output values:**

i. For $i \in [t^*]$,

A. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.

B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute state ciphertexts:**

i. For $i \in [t^* + 1]$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. $\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.

D. $\mathsf{ct}_{\mathsf{st},i}^{(\bot)} = \mathsf{SKE.Enc}(k_{E,i}, \bot; r_{\mathsf{Enc},i})$.

E. $\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Set hardwired values:**

i. For $i \in [t^*]$, $y_i^* = y_i$.

ii. For $i \in [t^* + 1]$,

A. $\mathsf{ct}_{\mathsf{st},i}^* = \begin{cases} \mathsf{ct}_{\mathsf{st},i}^{(0)} & \text{if } i = 1 \\ \mathsf{ct}_{\mathsf{st},i}^{(\bot)} & \text{if } i = t^* \\ \mathsf{ct}_{\mathsf{st},i}^{(b)} & \text{if } i = t^* + 1 \end{cases}$

(h) **Compute $\sigma_{\mathsf{st},1}$:**

i. $m_1 = (1, \mathsf{ct}_{\mathsf{st},1}^*, \mathsf{itr}_{\mathsf{st},0})$.

ii. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

iii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1)$.

(i) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_5[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t^*, y_{t^*-1}^*, \mathsf{ct}_{\mathsf{st},t^*}^*, y_{t^*}^*, \mathsf{ct}_{\mathsf{st},t^*+1}^*])$.

(j) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}^*, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\bot$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Program

$\mathsf{Prog}_5[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t^*, y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*}, y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1}](i, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{st},i}, \sigma_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$

1. **Verification Step**:
   - i. **Verify i is positive:** If $i \leq 0$, output $\bot$.
   - ii. **Verify input signature:**
     - i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i}) = 0$, output $\bot$.
   - iii. $m_i = (i, \mathsf{ct}_{\mathsf{st},i}, \mathsf{itr}_{\mathsf{st},i-1})$.
   - iv. **Verify state signature:**
     - i. $(\mathsf{sgk}_{A,i}, \mathsf{vk}_{A,i}, \mathsf{vk}_{A,i,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i))$.
     - ii. If $\mathsf{SSig.Verify}(\mathsf{vk}_{A,i}, m_i, \sigma_{\mathsf{st},i}) = 0$, output $\bot$.
2. **Computation Step:**
   - i. If $i = t^* - 1$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*})$.
   - ii. If $i = t^*$, $(y_i, \mathsf{ct}_{\mathsf{st},i+1}) = (y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1})$.
   - iii. If $i \notin \{t^* - 1, t^*\}$,
     - i. **Decrypt input and state**:
       - i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
       - ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
       - iii. $\mathsf{Post.CT}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{inp},i})$.
       - iv. $\mathsf{Post.Dec.st}_i = \mathsf{SKE.Dec}(k_{E,i}, \mathsf{ct}_{\mathsf{st},i})$.
     - ii. **Compute output value and next state:**
       - A. $(y_i, \mathsf{Post.Dec.st}_{i+1}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i, \mathsf{Post.CT}_i)$.
     - iii. **Encrypt the new state**:
       - i. $(r_{E,i+1}, r_{\mathsf{Enc},i+1}) = \mathsf{PPRF.Eval}(K_E, i+1)$.
       - ii. $k_{E,i+1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i+1})$.
       - iii. $\mathsf{ct}_{\mathsf{st},i+1} = \mathsf{SKE.Enc}(k_{E,i+1}, \mathsf{Post.Dec.st}_{i+1}; r_{\mathsf{Enc},i+1})$.
3. **Authentication Step**:
   - i. $\mathsf{itr}_{\mathsf{st},i} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},i-1}, (i, \mathsf{ct}_{\mathsf{st},i}))$.
   - ii. $m_{i+1} = (i+1, \mathsf{ct}_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.
   - iii. **Sign the new state:**
     - i. $(\mathsf{sgk}_{A,i+1}, \mathsf{vk}_{A,i+1}, \mathsf{vk}_{A,i+1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, i+1))$.
     - ii. $\sigma_{\mathsf{st},i+1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,i+1}, m_{i+1})$.
4. Output $(y_i, \mathsf{ct}_{\mathsf{st},i+1}, \sigma_{\mathsf{st},i+1}, \mathsf{itr}_{\mathsf{st},i})$.

Lemma C.77. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{4,5}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{5}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Hybrid$^{\mathcal{A}}_{5,1}$: This is the same as the previous hybrid except that we have rearranged some steps and changed some notation.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.

   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) $\mathsf{Post.CT}_i^{(0)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(0)})$.

   (c) ~~$\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.~~

   (d) **Compute input ciphertexts**:

      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

      iii. $\mathsf{ct}_{\mathsf{inp},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(0)})$.

      iv. $\mathsf{ct}_{\mathsf{inp},i}^{(\perp)} = \mathsf{SKE.Enc}(k_{E,i}, \perp)$.

   (e) **Set hardwired values**:

      i. $\mathsf{ct}^*_{\mathsf{inp},i} = \begin{cases} \mathsf{ct}^{(0)}_{\mathsf{inp},i} & \text{if } i < t^* \\ \mathsf{ct}^{(\perp)}_{\mathsf{inp},i} & \text{if } i = t^* \end{cases}$.

   (f) **Compute $\sigma_{\mathsf{inp},i}$**:

      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}^*_{\mathsf{inp},i})$.

   (g) Send $\mathsf{CT}_i = (\mathsf{ct}^*_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.

   (c) ~~$\mathsf{Post.Dec.st}_1^{(0)} = \mathsf{Post.Dec.st}_1$ and~~ $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.

   (d) **Compute state and output values:**

      i. For $i \in [t^*]$,

         A. ~~$(y_i, \mathsf{Post.Dec.st}_{i+1}^{(0)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(0)}, \mathsf{Post.CT}_i^{(0)})$.~~

         B. $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

C. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.

(e) **Compute $\mathsf{ct}_{\mathsf{st},1}$:**

i. $(r_{E,1}, r_{\mathsf{Enc},1}) = \mathsf{PPRF.Eval}(K_E, 1)$.

ii. $k_{E,1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,1})$.

iii. $\mathsf{ct}_{\mathsf{st},1} = \mathsf{SKE.Enc}(k_{E,1}, \mathsf{Post.Dec.st}_1; r_{\mathsf{Enc},1})$.

(f) **Compute state ciphertexts:**

i. For $i \in \{t^*, t^*+1\}$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

C. ~~$\mathsf{ct}_{\mathsf{st},i}^{(0)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(0)}; r_{\mathsf{Enc},i})$.~~

D. ~~$\mathsf{ct}_{\mathsf{st},i}^{(\bot)} = \mathsf{SKE.Enc}(k_{E,i}, \bot; r_{\mathsf{Enc},i})$.~~

E. ~~$\mathsf{ct}_{\mathsf{st},i}^{(b)} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.Dec.st}_i^{(b)}; r_{\mathsf{Enc},i})$.~~

ii. $\mathsf{ct}_{\mathsf{st},t^*}^{(\bot)} = \mathsf{SKE.Enc}(k_{E,t^*}, \bot; r_{\mathsf{Enc},t^*})$.

iii. $\mathsf{ct}_{\mathsf{st},t^*+1}^{(b)} = \mathsf{SKE.Enc}(k_{E,t^*+1}, \mathsf{Post.Dec.st}_{t^*+1}^{(b)}; r_{\mathsf{Enc},t^*})$.

(g) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(h) **Set hardwired values:**

i. $(y_{t^*-1}^*, y_{t^*}^*) = (y_{t^*-1}, y_{t^*})$.

ii. $(\mathsf{ct}_{\mathsf{st},t^*}^*, \mathsf{ct}_{\mathsf{st},t^*+1}^*) = (\mathsf{ct}_{\mathsf{st},t^*}^{(\bot)}, \mathsf{ct}_{\mathsf{st},t^*+1}^{(b)})$.

(i) **Compute $\sigma_{\mathsf{st},1}$:**

i. $m_1 = (1, \mathsf{ct}_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$.

ii. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

iii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1)$.

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_5[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t^*, y_{t^*-1}^*, \mathsf{ct}_{\mathsf{st},t^*}^*, y_{t^*}^*, \mathsf{ct}_{\mathsf{st},t^*+1}^*])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\bot$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i^{(b)})$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.78. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{5,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{5,1}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Hybrid$^{\mathcal{A}}_{5,2}$: This is the same as the previous hybrid except that we have rearranged some steps and changed some notation.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $K_{\mathsf{inp}}, K_A, K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\bot$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) **Compute phase 1 ciphertexts:**
      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(0)})$.
      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \bot$.
   (c) **Compute input ciphertexts**:
      i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.
      ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.
      iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i)$.
   (d) ~~**Set hardwired values:**~~
   (e) **Compute** $\sigma_{\mathsf{inp},i}$:
      i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.
      ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.
   (f) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \bot$.
   (b) **Compute** $\mathsf{Post.Dec.st}_1$**:** $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.
   (c) **Compute** $(y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$**:**
      i. $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.
      ii. For $i \in [t^*]$,
         A. $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
         B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.
      iii. If $t^* + 1 > 1$, $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{Post.Dec.st}_{t^*+1}^{(b)}$.
   (d) $(y^*_{t^*-1}, y^*_{t^*}) = (y_{t^*-1}, y_{t^*})$.
   (e) **Compute** $\mathsf{ct}_{\mathsf{st},1}$**:**

i. $(r_{E,1}, r_{\mathsf{Enc},1}) = \mathsf{PPRF.Eval}(K_E, 1)$.

ii. $k_{E,1} = \mathsf{SKE.Setup}(1^\lambda; r_{E,1})$.

iii. $\mathsf{ct}_{\mathsf{st},1} = \mathsf{SKE.Enc}(k_{E,1}, \mathsf{Post.Dec.st}_1; r_{\mathsf{Enc},1})$.

(f) **Setup iterator:** $(\mathsf{pp}_{\mathsf{st}}, \mathsf{itr}_{\mathsf{st},0}) \leftarrow \mathsf{Itr.Setup}(1^\lambda, 2^\lambda)$.

(g) **Compute $\sigma_{\mathsf{st},1}$:**

i. $m_1 = (1, \mathsf{ct}_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$.

ii. $(\mathsf{sgk}_{A,1}, \mathsf{vk}_{A,1}, \mathsf{vk}_{A,1,\mathsf{rej}}) \leftarrow \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_A, 1))$.

iii. $\sigma_{\mathsf{st},1} = \mathsf{SSig.Sign}(\mathsf{sgk}_{A,1}, m_1)$.

(h) **Compute state ciphertexts:**

i. For $i \in \{t^*, t^*+1\}$,

A. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

B. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

ii. $\mathsf{ct}^*_{\mathsf{st},t^*} = \mathsf{ct}^{(\bot)}_{\mathsf{st},t^*} = \mathsf{SKE.Enc}(k_{E,t^*}, \bot; r_{\mathsf{Enc},t^*})$.

iii. $\mathsf{ct}^*_{\mathsf{st},t^*+1} = \mathsf{ct}^{(b)}_{\mathsf{st},t^*+1} = \mathsf{SKE.Enc}(k_{E,t^*+1}, \mathsf{Post.Dec.st}_{t^*+1}; r_{\mathsf{Enc},t^*+1})$.

(i) ~~**Set hardwired values:**~~

(j) **Compute program:** $\mathcal{P} \leftarrow i\mathcal{O}(\mathsf{Prog}_5[K_{\mathsf{inp}}, K_A, K_E, \mathsf{pp}_{\mathsf{st}}, t^*, y^*_{t^*-1}, \mathsf{ct}^*_{\mathsf{st},t^*}, y^*_{t^*}, \mathsf{ct}^*_{\mathsf{st},t^*+1}])$.

(k) Send $\mathsf{SK}_f = (\mathcal{P}, \mathsf{ct}_{\mathsf{st},1}, \sigma_{\mathsf{st},1}, \mathsf{itr}_{\mathsf{st},0})$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\bot$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute phase 2 ciphertexts:**

i. $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.

(c) **Compute $\mathsf{ct}_{\mathsf{inp},i}$:**

i. $(r_{E,i}, r_{\mathsf{Enc},i}) = \mathsf{PPRF.Eval}(K_E, i)$.

ii. $k_{E,i} = \mathsf{SKE.Setup}(1^\lambda; r_{E,i})$.

iii. $\mathsf{ct}_{\mathsf{inp},i} = \mathsf{SKE.Enc}(k_{E,i}, \mathsf{Post.CT}_i)$.

(d) **Compute $\sigma_{\mathsf{inp},i}$:**

i. $(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i}, \mathsf{vk}_{\mathsf{inp},i,\mathsf{rej}}) = \mathsf{SSig.Setup}(1^\lambda; \mathsf{PPRF.Eval}(K_{\mathsf{inp}}, i))$.

ii. $\sigma_{\mathsf{inp},i} = \mathsf{SSig.Sign}(\mathsf{sgk}_{\mathsf{inp},i}, \mathsf{ct}_{\mathsf{inp},i})$.

(e) Send $\mathsf{CT}_i = (\mathsf{ct}_{\mathsf{inp},i}, \sigma_{\mathsf{inp},i})$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.79. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{5,1}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{5,2}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Hybrid$_6^{\mathcal{A}}$: We now rewrite our hybrid in terms of Pre-One-sFE.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0, 1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) **Compute phase 1 ciphertexts:**
      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(0)})$.
      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.
   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.
   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.
   (b) **Compute** $\mathsf{Post.Dec.st}_1$**:** $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.
   (c) **Compute** $(y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$**:**
      i. $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.
      ii. For $i \in [t^*]$,
         A. $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
         B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.
      iii. If $t^* + 1 > 1$, $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{Post.Dec.st}_{t^*+1}^{(b)}$.
   (d) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$.
   (e) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.
7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
   If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) **Compute phase 2 ciphertexts:**
      i. $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.80. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{5,2}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{6}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

C.2 Part 2: Using the Security of Post-One-sFE

Using the security of Post-One-sFE. We will now rely on the security of Post-One-sFE for indices $i > t$. First, we unwrap the algorithms of Post-One-sFE and reorganize our hybrid. We make use of the fact that many of the algorithms of Post-One-sFE can be computed with only a portion of the MSK.

$\mathbf{Hybrid}_6^{\mathcal{A}} = \mathbf{Hybrid}_{6,0}^{\mathcal{A}}$: Our starting hybrid has already swapped out encryptions of $x^{(b)}$ for encryptions of $x^{(0)}$ for $i < t^*$ and for encryptions of $\bot$ for $i = t^*$.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
   (b) $\mathsf{Post.MSK} \leftarrow \mathsf{Post\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\bot$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) **Compute phase 1 ciphertexts:**
      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(0)})$.
      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \bot$.
   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.
   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \bot$.
   (b) **Compute** $\mathsf{Post.Dec.st}_1$**:** $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGen}(\mathsf{Post.MSK}, f, \mathsf{st}_1)$.
   (c) **Compute** $(y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$**:**
      i. $\mathsf{Post.Dec.st}_1^{(b)} = \mathsf{Post.Dec.st}_1$.
      ii. For $i \in [t^*]$,
         A. $\mathsf{Post.CT}_i^{(b)} = \mathsf{Post\text{-}One\text{-}sFE.Enc}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
         B. $(y_i, \mathsf{Post.Dec.st}_{i+1}^{(b)}) = \mathsf{Post\text{-}One\text{-}sFE.Dec}(\mathsf{Post.Dec.st}_i^{(b)}, \mathsf{Post.CT}_i^{(b)})$.
      iii. If $t^* + 1 > 1$, $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{Post.Dec.st}_{t^*+1}^{(b)}$.
   (d) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$
   (e) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
   If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) **Compute phase 2 ciphertexts:**
      i. $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE}(\mathsf{Post.MSK}, i, x_i^{(b)})$.
   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.
   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.
8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

$\mathbf{Hybrid}^{\mathcal{A}}_{6,1}$: We now make use of the additional properties of Post-One-sFE as defined in Section 5.3. We unwrap Post-One-sFE.Setup and rewrite our hybrid in terms of the algorithms Post-One-sFE.EncLocal, Post-One-sFE.KeyGenLocal, and Post-One-sFE.DecStGen defined in Section 5.3.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.

   (b) $K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$.

   (c) For $i \in [n+1]$, $K_i \leftarrow \mathsf{PRF.Eval}(K, i)$.

4. **Challenge Bit:** $b \leftarrow \{0, 1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) **Compute phase 1 ciphertexts:**

      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.

      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.

   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) **Compute** $\mathsf{Post.Dec.st}_1$**:** $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.

   (c) **Compute** $(y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$**:**

      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.

      ii. For $i \in [t^*]$,

         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

      iii. If $t^* + 1 > 1$, $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{Post\text{-}One\text{-}sFE.DecStGen}(t^* + 1, K_{t^*+1}, f, \mathsf{st}_{t^*+1}^{(b)})$.

   (d) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

   (e) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
   If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) **Compute phase 2 ciphertexts:**

i. $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(b)})$.

(c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.81. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}_{6,0}^{\mathcal{A}}(1^\lambda), \mathbf{Hybrid}_{6,1}^{\mathcal{A}}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. By Lemmas 5.2,5.3,5.4, except with negligible probability the two hybrids are identical. □

$\mathbf{Hybrid}^{\mathcal{A}}_{6,2}$: We compute the partial keys K_i as random values. This hybrid is the same as the previous hybrid except that

- We compute Setup as

  3. **Setup:**
     (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
     (b) ~~$K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$.~~
     (c) For $i \in [n+1]$, $K_i \leftarrow \{0,1\}^{6\lambda+\ell_\mathcal{S}}$

Lemma C.82. *If* PRF *is a pseudorandom function, then for all* $\lambda \in \mathbb{N}$ *and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{6,1}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{6,2}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. This follows by a reduction to the security of our PRF.

Observe that we can run both hybrids without knowing K as long as we are given $\{K_i\}_{i\in[n+1]}$. In the reduction, we run $\mathbf{Hybrid}^{\mathcal{A}}_{6,2}$ up to just before the **Setup** step. For each $i \in [n+1]$, we query our PRF challenger on input i and receive back K_i which is either a random value or is equal to $\mathsf{PRF.Eval}(K, i)$ for some PRF key K. We then compute $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$ and run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{6,2}$ starting from **Encryption Phase 1**. Observe that if we received random values, then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{6,2}$, and if we received PRF evaluations, then we emulate $\mathbf{Hybrid}^{\mathcal{A}}_{6,1}$. Thus, by PRF security, the outputs of these hybrids must be indistinguishable. □

Hybrid$^{\mathcal{A}}_{6,3}$: We now unwrap more of Post-One-sFE.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
   (b) For $i \in [n+1]$,
      i. $K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i}) \leftarrow \{0,1\}^{6\lambda+\ell_\mathcal{S}}$.
      ii. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$, $\mathsf{msk}'_i = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i})$, $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$, $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) **Compute phase 1 ciphertexts:**
      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.
      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.
   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.
   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.
   (b) **Compute** $\mathsf{Post.Dec.st}_1$**:**
      i. If $t^* + 1 = 1$,
         A. $c_1 \leftarrow \mathsf{SKE.Enc}(k_1, \perp)$.
         B. $\widetilde{\mathsf{st}}_1 = \mathsf{st}_1 \oplus p_1$.
         C. $\mathsf{sk}_{g_1} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_1, g_1; r_{\mathsf{KeyGen}_1})$ for $g_1 = g_{f,\widetilde{\mathsf{st}}_1,c_1}$.
         D. $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$.
      ii. If $t^* + 1 > 1$, $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.
   (c) **Compute** $(y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$**:**
      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.
      ii. For $i \in [t^*]$,
         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.
      iii. If $t^* + 1 > 1$,
         A. $\widetilde{\mathsf{st}}_{t^*+1} = \mathsf{st}_{t^*+1}^{(b)} \oplus p_{t^*+1}$.
         B. $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (f, \widetilde{\mathsf{st}}_{t^*+1}, r_{\mathsf{msk}_{t^*+1}}, r_{\mathsf{KeyGen}_{t^*+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{t^*+1}})$.

C. $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$.

(d) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

(e) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute phase 2 ciphertexts:**

i. $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (x_i^{(b)}, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_\mathcal{Y}}))$.

ii. If $i = 1$, $\mathsf{Post.CT}_1 = \mathsf{ct}_1$.

iii. If $i > 1$,

A. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, \perp)$.

B. $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \perp)$.

C. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$ for $h_i = h_{c_i, c'_i}$.

D. $\mathsf{Post.CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.

(c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.83. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{6,2}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{6,3}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Hybrid$_7^{\mathcal{A}}$: This is the same as the previous hybrid except that we have rearranged some steps and changed some notation.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.

   (b) For $i \in [n+1]$,

      i. $r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i} \leftarrow \{0,1\}^\lambda$.

      ii. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$, $\mathsf{msk}'_i = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i})$, $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$, $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.

      iii. $p_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

      iv. $K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i})$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) **Compute phase 1 ciphertexts:**

      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.

      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.

   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) **Compute phase 1 state and output values:**

      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.

      ii. For $i \in [t^*]$,

         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

      iii. $\widetilde{\mathsf{st}}_{t^*+1} = \mathsf{st}_{t^*+1}^{(b)} \oplus p_{t^*+1}$.

   (c) **Compute sk'_{h_i}:** For $i \in [t^*+1, n]$,

      i. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, \perp)$.

      ii. If $i > 1$,

         A. $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \perp)$.

         B. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$ for $h_i = h_{c_i, c'_i}$.

   (d) **Compute hardcoded values:**

i. If $t^*+1=1$,

A. $\mathsf{sk}_{g_1} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_1, g_1; r_{\mathsf{KeyGen}_1})$ for $g_1 = g_{f,\widetilde{\mathsf{st}}_1,c_1}$. (Note that in this case $(\widetilde{\mathsf{st}}_1, c_1) = (\widetilde{\mathsf{st}}_{t^*+1}, c_{t^*+1})$).

B. $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$.

ii. If $t^*+1>1$,

A. $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.

B. $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (f, \widetilde{\mathsf{st}}_{t^*+1}, r_{\mathsf{msk}_{t^*+1}}, r_{\mathsf{KeyGen}_{t^*+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{t^*+1}})$.

C. $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$.

(e) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

(f) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute phase 2 ciphertexts:**

i. $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (x_i^{(b)}, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_{\mathcal{Y}}}))$.

ii. If $i=1$, $\mathsf{Post.CT}_1 = \mathsf{ct}_1$. Else $\mathsf{Post.CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.

(c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.84. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{6,3}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{7}(1^\lambda)) = 0$$

Proof. The hybrids are identical. □

Finishing the proof. We now use the security of Post-One-sFE to finish our proof. This proof is essentially the same as the proof in [GKS23].

Hybrid$_8^{\mathcal{A}}$: For each $i \geq t^*+1$, rather than sampling p_i at random and later computing $\widetilde{\mathsf{st}}_i = \mathsf{st}_i^{(b)} \oplus p_i$, we instead sample $\widetilde{\mathsf{st}}_i$ at random and later compute $p_i = \mathsf{st}_i^{(b)} \oplus \widetilde{\mathsf{st}}_i$. Our hybrid can change this ordering since for $i \geq t^*+1$, we don't need to compute p_i until after we have received $x_{i-1}^{(b)}$ and can compute $\mathsf{st}_i^{(b)}$. Note that for $i \geq t^*+1$ we do not need to compute K_i.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.
2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
3. **Setup:**
   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.
   (b) For $i \in [n+1]$,
      i. $r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i} \leftarrow \{0,1\}^\lambda$.
      ii. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$, $\mathsf{msk}'_i = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i})$, $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$, $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.
      iii. If $i < t^*+1$,
         A. $p_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.
         B. $K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i})$.
      iv. If $i \geq t^*+1$,
         A. $\widetilde{\mathsf{st}}_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.
4. **Challenge Bit:** $b \leftarrow \{0,1\}$.
5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.
   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.
   (b) **Compute phase 1 ciphertexts:**
      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.
      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.
   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.
   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.
6. **KeyGen:**
   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.
   (b) **Compute phase 1 state and output values:**
      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.
      ii. For $i \in [t^*]$,
         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

iii. $p_{t^*+1} = \mathsf{st}^{(b)}_{t^*+1} \oplus \widetilde{\mathsf{st}}_{t^*+1}$.

(c) **Compute sk'_{h_i}:** For $i \in [t^*+1, n]$,

i. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, \bot)$.

ii. If $i > 1$,

A. $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \bot)$.

B. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$ for $h_i = h_{c_i, c'_i}$.

(d) **Compute hardcoded values:**

i. If $t^*+1 = 1$,

A. $\mathsf{sk}_{g_1} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_1, g_1; r_{\mathsf{KeyGen}_1})$ for $g_1 = g_{f, \widetilde{\mathsf{st}}_1, c_1}$. (Note that in this case $(\widetilde{\mathsf{st}}_1, c_1) = (\widetilde{\mathsf{st}}_{t^*+1}, c_{t^*+1})$).

B. $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$.

ii. If $t^*+1 > 1$,

A. $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.

B. $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (f, \widetilde{\mathsf{st}}_{t^*+1}, r_{\mathsf{msk}_{t^*+1}}, r_{\mathsf{KeyGen}_{t^*+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{t^*+1}})$.

C. $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$.

(e) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

(f) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\bot$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute p_i:**

i. $(y_i, \mathsf{st}^{(b)}_{i+1}) = f(x_i^{(b)}, \mathsf{st}^{(b)}_i)$.

ii. $p_{i+1} = \mathsf{st}^{(b)}_{i+1} \oplus \widetilde{\mathsf{st}}_{i+1}$.

(c) **Compute phase 2 ciphertexts:**

i. $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (x_i^{(b)}, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_{\mathcal{Y}}}))$.

ii. If $i = 1$, $\mathsf{Post.CT}_1 = \mathsf{ct}_1$. Else $\mathsf{Post.CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.

(d) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(e) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.85. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}_7^{\mathcal{A}}(1^\lambda), \mathbf{Hybrid}_8^{\mathcal{A}}(1^\lambda) = 1) = 0$$

Proof. The hybrids are identically distributed. This follow from the fact that for any value of $\mathsf{st}_i^{(b)}$, the following two distributions are identically distributed:

- $D_{1,i}$:

1. $\widetilde{\mathsf{st}}_i \leftarrow \{0,1\}^{\ell_{\mathcal{S}}}$.

2. $p_i = \mathsf{st}_i^{(b)} \oplus \widetilde{\mathsf{st}}_i$.
3. Output $(\mathsf{st}_i^{(b)}, p_i, \widetilde{\mathsf{st}}_i)$.

- $D_{2,i}$:

1. $p_i \leftarrow \{0,1\}^{\ell_{\mathcal{S}}}$.
2. $\widetilde{\mathsf{st}}_i = \mathsf{st}_i^{(b)} \oplus p_i$.
3. Output $(\mathsf{st}_i^{(b)}, p_i, \widetilde{\mathsf{st}}_i)$.

Thus, it does not matter whether we compute $\widetilde{\mathsf{st}}_i$ first and then set $p_i = \mathsf{st}_i^{(b)} \oplus \widetilde{\mathsf{st}}_i$ (as in the previous hybrid), or compute p_i first and then set $\widetilde{\mathsf{st}}_i = \mathsf{st}_i^{(b)} \oplus p_i$ (as in the current hybrid). □

Hybrid$_9^{\mathcal{A}}$: For each $i \geq t^* + 1$, we hardwire into c_i the values $(y_i, \mathsf{ct}'_{i+1})$ that are output by $g_i = g_{f,\widetilde{\mathsf{st}},c_i}$ on the $\alpha_i = 0$ branch if we run it on the input generated by the challenge stream $x^{(b)}$. This will allow us to later switch to the $\alpha_i = 1$ branch in $g_i = g_{f,\widetilde{\mathsf{st}}_i,c_i}$ using the security of $\mathsf{OneFSFE}$. Observe that the values being hardcoded into c_i can be determined before knowing $x^{(b)}$.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.

   (b) For $i \in [n+1]$,

      i. $r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i} \leftarrow \{0,1\}^\lambda$.

      ii. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$, $\mathsf{msk}'_i = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i})$, $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$, $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.

      iii. If $i < t^* + 1$,

         A. $p_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

         B. $K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i})$.

      iv. If $i \geq t^* + 1$,

         A. $\widetilde{\mathsf{st}}_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) **Compute phase 1 ciphertexts:**

      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.

      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.

   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) **Compute phase 1 state and output values:**

      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.

      ii. For $i \in [t^*]$,

         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

      iii. $p_{t^*+1} = \mathsf{st}_{t^*+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{t^*+1}$.

   (c) **Compute** sk'_{h_i}: For $i \in [t^* + 1, n]$,

i. $\theta_i \leftarrow \{0,1\}^{\ell_{\mathcal{Y}}}$.

ii. $\mathsf{ct}'_{i+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (f, \widetilde{\mathsf{st}}_{i+1}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{i+1}})$

iii. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, (\theta_i, \mathsf{ct}'_{i+1}))$

iv. If $i > 1$,

A. $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \bot)$.

B. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$ for $h_i = h_{c_i, c'_i}$.

(d) **Compute hardcoded values:**

i. If $t^* + 1 = 1$,

A. $\mathsf{sk}_{g_1} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_1, g_1; r_{\mathsf{KeyGen}_1})$ for $g_1 = g_{f, \widetilde{\mathsf{st}}_1, c_1}$. (Note that in this case $(\widetilde{\mathsf{st}}_1, c_1) = (\widetilde{\mathsf{st}}_{t^*+1}, c_{t^*+1})$).

B. $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$.

ii. If $t^* + 1 > 1$,

A. $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.

B. $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (f, \widetilde{\mathsf{st}}_{t^*+1}, r_{\mathsf{msk}_{t^*+1}}, r_{\mathsf{KeyGen}_{t^*+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{t^*+1}})$.

C. $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$.

(e) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

(f) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\bot$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute p_i and ψ_i:**

i. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

ii. $\psi_i = y_i \oplus \theta_i$.

iii. $p_{i+1} = \mathsf{st}_{i+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{i+1}$.

(c) **Compute phase 2 ciphertexts:**

i. $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (x_i^{(b)}, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_{\mathcal{Y}}}))$.

ii. If $i = 1$, $\mathsf{Post.CT}_1 = \mathsf{ct}_1$. Else $\mathsf{Post.CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.

(d) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(e) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.86. *If* SKE *is a secure encryption scheme, then for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}_8^{\mathcal{A}}(1^\lambda), \mathbf{Hybrid}_9^{\mathcal{A}}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. The lemma follows by the security of SKE since we only use the keys k_i to encrypt c_i.

For $t \in [t^*, n]$, define $\mathbf{Hybrid}_{8,t}^{\mathcal{A}}$ be the same as $\mathbf{Hybrid}_9^{\mathcal{A}}$ except that in step 6c of KeyGen, if $i \leq t$, we compute $c_i \leftarrow \mathsf{SKE.Enc}(k_i, (\theta_i, \mathsf{ct}'_{i+1}))$ as in $\mathbf{Hybrid}_9^{\mathcal{A}}$ and if $i > t$, we compute $c_i \leftarrow \mathsf{SKE.Enc}(k_i, \bot)$ as in $\mathbf{Hybrid}_8^{\mathcal{A}}$. Observe that $\mathbf{Hybrid}_8^{\mathcal{A}} = \mathbf{Hybrid}_{8,t^*}^{\mathcal{A}}$ since for $i \geq t^* + 1$, ct'_{i+1}, θ_i and ψ_i are not used in $\mathbf{Hybrid}_{8,t^*}^{\mathcal{A}}$. Additionally, $\mathbf{Hybrid}_9^{\mathcal{A}} = \mathbf{Hybrid}_{8,n}^{\mathcal{A}}$.

By a reduction to the security of SKE, we show that for all $t \in [t^*+1, n]$, $\mathbf{Hybrid}^{\mathcal{A}}_{8,t-1}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{8,t}$ are indistinguishable. This implies our lemma. In the reduction, without sampling r_{k_t} or computing k_t, we run $\mathbf{Hybrid}^{\mathcal{A}}_{8,t-1}$ up to just before step 6c of KeyGen. We sample $\theta_t \leftarrow \{0,1\}^{\ell_{\mathcal{Y}}}$ and compute $\mathsf{ct}'_{t+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t+1}, (f, \widetilde{\mathsf{st}}_{t+1}, r_{\mathsf{msk}_{t+1}}, r_{\mathsf{KeyGen}_{t+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{t+1}})$. We then send $(\bot, (\theta_t, \mathsf{ct}'_{t+1}))$ to the SKE challenger and receive an encryption c^* of one of the two messages. We set $c_t = c^*$. Next, we compute c_i for $i \in [t^*+1, n] \backslash \{t\}$ as in step 6c of KeyGen of $\mathbf{Hybrid}^{\mathcal{A}}_{8,t-1}$. We then run the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{8,t-1}$ starting from step 6c.iv of KeyGen. Observe that if c^* was an encryption of $\bot$ then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{8,t-1}$, and if c^* was an encryption of $(\theta_t, \mathsf{ct}'_{t+1})$ then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{8,t}$. Thus, by SKE security, the outputs of these hybrids must be indistinguishable. $\square$

Hybrid$_{10}^{\mathcal{A}}$: For each $i \geq t^*+1$, we hardwire into c'_i the value sk_{g_i} that would be output by $h_i = h_{c_i,c'_i}$ in the $\alpha'_i = 0$ branch if we were to run it on the input generated by the challenge stream $x^{(b)}$. This will allow us to later switch to the $\alpha'_i = 1$ branch in $h_i = h_{c_i,c'_i}$ using the security of $\mathsf{OneCompFE}$. Observe that the values being hardcoded into c'_i can be determined before knowing $x^{(b)}$.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.

   (b) For $i \in [n+1]$,

      i. $r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i} \leftarrow \{0,1\}^\lambda$.

      ii. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$, $\mathsf{msk}'_i = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i})$, $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$, $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.

      iii. If $i < t^* + 1$,

         A. $p_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

         B. $K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i})$.

      iv. If $i \geq t^* + 1$,

         A. $\widetilde{\mathsf{st}}_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) **Compute phase 1 ciphertexts:**

      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.

      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.

   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) **Compute phase 1 state and output values:**

      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.

      ii. For $i \in [t^*]$,

         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

      iii. $p_{t^*+1} = \mathsf{st}_{t^*+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{t^*+1}$.

   (c) **Compute** sk'_{h_i}: For $i \in [t^*+1, n]$,

i. $\theta_i \leftarrow \{0,1\}^{\ell_{\mathcal{Y}}}$.

ii. $\mathsf{ct}'_{i+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (f, \widetilde{\mathsf{st}}_{i+1}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{i+1}})$.

iii. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, (\theta_i, \mathsf{ct}'_{i+1}))$.

iv. $\mathsf{sk}_{g_i} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_i, g_i; r_{\mathsf{KeyGen}_i})$ for $g_i = g_{f,\widetilde{\mathsf{st}}_i,c_i}$.

v. If $i > 1$,

A. $c'_i \leftarrow \mathsf{SKE}'.\mathsf{Enc}(k'_i, \mathsf{sk}_{g_i})$.

B. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$ for $h_i = h_{c_i,c'_i}$.

(d) **Compute hardcoded values:**

i. If $t^* + 1 = 1$,

A. ~~$\mathsf{sk}_{g_1} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_1, g_1; r_{\mathsf{KeyGen}_1})$ for $g_1 = g_{f,\widetilde{\mathsf{st}}_1,c_1}$.~~
~~(Note that in this case $(\widetilde{\mathsf{st}}_1, c_1) = (\widetilde{\mathsf{st}}_{t^*+1}, c_{t^*+1})$).~~

B. $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$.

ii. If $t^* + 1 > 1$,

A. $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.

B. $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (f, \widetilde{\mathsf{st}}_{t^*+1}, r_{\mathsf{msk}_{t^*+1}}, r_{\mathsf{KeyGen}_{t^*+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{t^*+1}})$.

C. $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$.

(e) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

(f) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute p_i and ψ_i:**

i. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

ii. $\psi_i = y_i \oplus \theta_i$.

iii. $p_{i+1} = \mathsf{st}_{i+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{i+1}$.

(c) **Compute phase 2 ciphertexts:**

i. $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (x_i^{(b)}, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_{\mathcal{Y}}}))$.

ii. If $i = 1$, $\mathsf{Post.CT}_1 = \mathsf{ct}_1$. Else $\mathsf{Post.CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.

(d) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(e) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.87. *If SKE' is a secure encryption scheme, then for all $\lambda \in \mathbb{N}$ and all PPT adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}_9^{\mathcal{A}}(1^\lambda), \mathbf{Hybrid}_{10}^{\mathcal{A}}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. The lemma follows by the security of SKE' since we only use the keys k'_i to encrypt c'_i.

For $t \in [t^*, n]$, define $\mathbf{Hybrid}_{9,t}^{\mathcal{A}}$ be the same as $\mathbf{Hybrid}_{10}^{\mathcal{A}}$ except that in step 6c of KeyGen, if $i \leq t$, we compute $c'_i \leftarrow \mathsf{SKE}'.\mathsf{Enc}(k'_i, \mathsf{sk}_{g_i})$ as in $\mathbf{Hybrid}_{10}^{\mathcal{A}}$ and if $i > t$, we compute $c'_i \leftarrow \mathsf{SKE}'.\mathsf{Enc}(k'_i, \perp)$

as in $\mathbf{Hybrid}_9^{\mathcal{A}}$. Observe that $\mathbf{Hybrid}_9^{\mathcal{A}} = \mathbf{Hybrid}_{9,t^*}^{\mathcal{A}}$ since moving the computation of $\mathsf{sk}_{g,i}$ up in the hybrid does not affect its output. Additionally, $\mathbf{Hybrid}_{10}^{\mathcal{A}} = \mathbf{Hybrid}_{9,n}^{\mathcal{A}}$.

By a reduction to the security of SKE, we show that for all $t \in [t^*+1, n]$, $\mathbf{Hybrid}_{9,t-1}^{\mathcal{A}}$ and $\mathbf{Hybrid}_{9,t}^{\mathcal{A}}$ are indistinguishable. This implies our lemma. In the reduction, without sampling r'_{k_t} or computing k'_t, we run $\mathbf{Hybrid}_{9,t-1}^{\mathcal{A}}$ up to just before step 6c of KeyGen. We sample $\theta_t \leftarrow \{0,1\}^{\ell_{\mathcal{Y}}}$, compute $\mathsf{ct}'_{t+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t+1}, (f, \widetilde{\mathsf{st}}_{t+1}, r_{\mathsf{msk}_{t+1}}, r_{\mathsf{KeyGen}_{t+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{t+1}})$, compute $c_t \leftarrow \mathsf{SKE.Enc}(k_t, (\theta_t, \mathsf{ct}'_{t+1}))$, and compute $\mathsf{sk}_{g_t} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_t, g_t; r_{\mathsf{KeyGen}_t})$ for $g_t = g_{f,\widetilde{\mathsf{st}}_t,c_t}$. We then send $(\bot, \mathsf{sk}_{g_t})$ to the SKE challenger and receive an encryption c^* of one of the two messages. We set $c'_t = c^*$ and compute $\mathsf{sk}'_{h_t} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_t, h_t)$ for h_{c_t,c'_t}. Next, we run step 6c of KeyGen of $\mathbf{Hybrid}_{9,t-1}^{\mathcal{A}}$ for $i \in [t^*+1, n] \backslash \{t\}$. We then run the rest of $\mathbf{Hybrid}_{9,t-1}^{\mathcal{A}}$ starting from step 6d of KeyGen. Observe that if c^* was an encryption of $\bot$ then we exactly emulate $\mathbf{Hybrid}_{9,t-1}^{\mathcal{A}}$, and if c^* was an encryption of sk_{g_t} then we exactly emulate $\mathbf{Hybrid}_{9,t}^{\mathcal{A}}$. Thus, by SKE security, the outputs of these hybrids must be indistinguishable. □

Hybrid$^{\mathcal{A}}_{11,t,0}$: We change the message encrypted in ct'_t so that we use the $\alpha'_t = 1$ branch of h_{c_t,c'_t}.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.

   (b) For $i \in [n+1]$,

      i. $r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i} \leftarrow \{0,1\}^\lambda$.

      ii. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$, $\mathsf{msk}'_i = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i})$, $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$, $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.

      iii. If $i < t^* + 1$,

         A. $p_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

         B. $K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i})$.

      iv. If $i \geq t^* + 1$,

         A. $\widetilde{\mathsf{st}}_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) **Compute phase 1 ciphertexts:**

      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.

      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.

   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) **Compute phase 1 state and output values:**

      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.

      ii. For $i \in [t^*]$,

         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

      iii. $p_{t^*+1} = \mathsf{st}_{t^*+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{t^*+1}$.

   (c) **Compute** sk'_{h_i}: For $i \in [t^*+1, n]$,

      i. $\theta_i \leftarrow \{0,1\}^{\ell_\mathcal{Y}}$.

      ii. If $i+1 \leq t$, $\mathsf{ct}'_{i+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (0^{\ell_\mathcal{F}}, 0^{\ell_\mathcal{S}}, 0^\lambda, 0^\lambda, 1, r'_{k_{i+1}}); r'_{\mathsf{Enc}_{i+1}})$.

      iii. If $i+1 > t$, $\mathsf{ct}'_{i+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (f, \widetilde{\mathsf{st}}_{i+1}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{i+1}})$.

iv. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, (\theta_i, \mathsf{ct}'_{i+1}))$.

v. $\mathsf{sk}_{g_i} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_i, g_i; r_{\mathsf{KeyGen}_i})$ for $g_i = g_{f,\widetilde{\mathsf{st}}_i,c_i}$.

vi. If $i > 1$,

A. $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \mathsf{sk}_{g_i})$.

B. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$ for $h_i = h_{c_i,c'_i}$.

(d) **Compute hardcoded values:**

i. If $t^*+1 = 1$, $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$.

ii. If $t^*+1 > 1$,

A. $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.

B. $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (0^{\ell_{\mathcal{F}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 1, r'_{k_{t^*+1}}); r'_{\mathsf{Enc}_{t^*+1}})$.

C. $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$.

(e) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

(f) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute p_i and ψ_i:**

i. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

ii. $\psi_i = y_i \oplus \theta_i$.

iii. $p_{i+1} = \mathsf{st}_{i+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{i+1}$.

(c) **Compute phase 2 ciphertexts:**

i. If $i < t$, $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (0^{\ell_{\mathcal{X}}}, 0^{\ell_{\mathcal{S}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 0^\lambda, 0^\lambda, 1, r_{k_i}, \psi_i))$.

ii. If $i \geq t$, $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (x_i^{(b)}, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_{\mathcal{Y}}}))$.

iii. If $i = 1$, $\mathsf{Post.CT}_1 = \mathsf{ct}_1$. Else $\mathsf{Post.CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.

(d) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(e) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.88. *If* OneCompFE *is a single-key, single-ciphertext, selective secure functional encryption scheme, then for all $\lambda \in \mathbb{N}$ and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}_{10}^{\mathcal{A}}(1^\lambda), \mathbf{Hybrid}_{11,t^*+1,0}^{\mathcal{A}}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. The lemma follows by a reduction to the security of OneCompFE since we only use $(r'_{\mathsf{msk}_{t^*+1}}, \mathsf{msk}'_{t^*+1}, r'_{\mathsf{Enc}_{t^*+1}})$ to compute ct'_{t^*+1} and $\mathsf{sk}'_{h_{t^*+1}}$.

Observe that since $t = t^*+1$, then step 6c of KeyGen and step 7c of **Encryption Phase 2** are identical in both hybrids. Thus, the only change is to ct'_{t^*+1}.

In the reduction, without sampling or computing $(r'_{\mathsf{msk}_{t^*+1}}, \mathsf{msk}'_{t^*+1}, r'_{\mathsf{Enc}_{t^*+1}})$, we run $\mathbf{Hybrid}_{11,t^*+1,0}^{\mathcal{A}}$ up to just before step 6c of KeyGen. We then run step 6c of KeyGen except that we skip over computing $\mathsf{sk}'_{h_{t^*+1}}$. This is possible without needing to know $(r'_{\mathsf{msk}_{t^*+1}}, \mathsf{msk}'_{t^*+1}, r'_{\mathsf{Enc}_{t^*+1}})$ since

- We only need to know K_i for $i < t^* + 1$.
- In step 6c of KeyGen, to compute all the ct'_{i+1}, we only need to know $(r'_{\mathsf{msk}_{i+1}}, \mathsf{msk}'_{i+1}, r'_{\mathsf{Enc}_{i+1}})$ for $i \geq t^* + 1$.
- In step 6c of KeyGen, to compute sk'_{h_i} for $i \neq t^* + 1$, we only need to know $(r'_{\mathsf{msk}_i}, \mathsf{msk}'_i, r'_{\mathsf{Enc}_i})$ for $i \neq t^* + 1$.

We then send challenge message pair (m_0, m_1) to the $\mathsf{OneFSFE}$ challenger where

$$m_0 = (f, \widetilde{\mathsf{st}}_{t^*+1}, r_{\mathsf{msk}_{t^*+1}}, r_{\mathsf{KeyGen}_{t^*+1}}, 0, 0^\lambda)$$

$$m_1 = (0^{\ell_{\mathcal{F}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 1, r'_{k_{t^*+1}})$$

and receive back a $\mathsf{OneFSFE}$ encryption ct^* of one of them. We send function $h_{c_{t^*+1}, c'_{t^*+1}}$ to the $\mathsf{OneFSFE}$ challenger and receive back a function key sk^*. We set $\mathsf{ct}'_{t^*+1} = \mathsf{ct}^*$ and $\mathsf{sk}'_{h_{t^*+1}} = \mathsf{sk}^*$. As needed for the security game, we can observe that

$$\begin{aligned}
&h_{c_{t^*+1}, c'_{t^*+1}}(0^{\ell_{\mathcal{F}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 1, r'_{k_{t^*+1}}) \\
&= \mathsf{SKE}'.\mathsf{Dec}(k'_{t^*+1}, c'_{t^*+1}) \\
&= \mathsf{sk}_{g_{t^*+1}} \\
&= h_{c_{t^*+1}, c'_{t^*+1}}(f, \widetilde{\mathsf{st}}_{t^*+1}, r_{\mathsf{msk}_{t^*+1}}, r_{\mathsf{KeyGen}_{t^*+1}}, 0, 0^\lambda)
\end{aligned}$$

This is because c'_{t^*+1} encrypts $\mathsf{sk}_{g_{t^*+1}}$ where $\mathsf{sk}_{g_{t^*+1}}$ is generated in the same way as in the $\alpha'_{t^*+1} = 0$ branch of $h_{c_{t^*+1}, c'_{t^*+1}}$.

Then, if $t^* + 1 = 1$, we set $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$. Else, we sample $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \widetilde{\mathsf{st}}_1)$ and set $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$. We then compute the remainder of $\mathbf{Hybrid}^{\mathcal{A}}_{11, t^*+1, 0}$ starting from step 6e of KeyGen. Note that we do not need to know $(r'_{\mathsf{msk}_{t^*+1}}, \mathsf{msk}'_{t^*+1}, r'_{\mathsf{Enc}_{t^*+1}})$ to compute the remainder of the hybrid since **Encryption Phase 2** only needs to know $(r'_{\mathsf{msk}_{t^*+1}}, r'_{\mathsf{Enc}_{t^*+1}})$ for $i > t^* + 1$.

Observe that if c^* was an encryption of m_0 then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{10}$, and if c^* was an encryption of m_1 then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{11, t^*+1, 0}$. Thus, by the security of $\mathsf{OneCompFE}$, the outputs of these hybrids must be indistinguishable. □

Hybrid$^{\mathcal{A}}_{11,t,1}$:We change the message encrypted in ct_t so that we use the $\alpha_t = 1$ branch of $g_{f,\widetilde{\mathsf{st}}_t,c_t}$.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.

   (b) For $i \in [n+1]$,

      i. $r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i} \leftarrow \{0,1\}^\lambda$.

      ii. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$, $\mathsf{msk}'_i = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i})$, $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$, $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.

      iii. If $i < t^* + 1$,

         A. $p_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

         B. $K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i})$.

      iv. If $i \geq t^* + 1$,

         A. $\widetilde{\mathsf{st}}_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) **Compute phase 1 ciphertexts:**

      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.

      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.

   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) **Compute phase 1 state and output values:**

      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.

      ii. For $i \in [t^*]$,

         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

      iii. $p_{t^*+1} = \mathsf{st}_{t^*+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{t^*+1}$.

   (c) **Compute** sk'_{h_i}: For $i \in [t^*+1, n]$,

      i. $\theta_i \leftarrow \{0,1\}^{\ell_\mathcal{Y}}$.

      ii. If $i+1 \leq t$, $\mathsf{ct}'_{i+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (0^{\ell_\mathcal{F}}, 0^{\ell_\mathcal{S}}, 0^\lambda, 0^\lambda, 1, r'_{k_{i+1}}); r'_{\mathsf{Enc}_{i+1}})$.

      iii. If $i+1 > t$, $\mathsf{ct}'_{i+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (f, \widetilde{\mathsf{st}}_{i+1}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda); r'_{\mathsf{Enc}_{i+1}})$.

iv. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, (\theta_i, \mathsf{ct}'_{i+1}))$.

v. $\mathsf{sk}_{g_i} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_i, g_i; r_{\mathsf{KeyGen}_i})$ for $g_i = g_{f,\widetilde{\mathsf{st}}_i,c_i}$.

vi. If $i > 1$,

A. $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \mathsf{sk}_{g_i})$.

B. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$ for $h_i = h_{c_i,c'_i}$.

(d) **Compute hardcoded values:**

i. If $t^*+1 = 1$, $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$.

ii. If $t^*+1 > 1$,

A. $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.

B. $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (0^{\ell_{\mathcal{F}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 1, r'_{k_{t^*+1}}); r'_{\mathsf{Enc}_{t^*+1}})$.

C. $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$.

(e) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

(f) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute p_i and ψ_i:**

i. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

ii. $\psi_i = y_i \oplus \theta_i$.

iii. $p_{i+1} = \mathsf{st}_{i+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{i+1}$.

(c) **Compute phase 2 ciphertexts:**

i. If $i \le t$, $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (0^{\ell_{\mathcal{X}}}, 0^{\ell_{\mathcal{S}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 0^\lambda, 0^\lambda, 1, r_{k_i}, \psi_i))$.

ii. If $i > t$, $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (x_i^{(b)}, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_{\mathcal{Y}}}))$.

iii. If $i = 1$, $\mathsf{Post.CT}_1 = \mathsf{ct}_1$. Else $\mathsf{Post.CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.

(d) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(e) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.89. *If* OneFSFE *is a single-key, single-ciphertext, function-selective secure functional encryption scheme, then for all $\lambda \in \mathbb{N}$, all $t \in [t^*+1, n]$, and all* PPT *adversaries $\mathcal{A}$,*

$$\Delta(\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{11,t,0}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{11,t,1}(1^\lambda)) \le \mathsf{negl}(\lambda)$$

Proof. This lemma follows by a reduction to the security of OneFSFE since we only use $(r_{\mathsf{msk}_t}, \mathsf{msk}_t, r_{\mathsf{KeyGen}_t})$ to compute ct_t and sk_{g_t}.

In the reduction, without sampling or computing $(r_{\mathsf{msk}_t}, \mathsf{msk}_t, r_{\mathsf{KeyGen}_t})$, we run $\mathbf{Hybrid}^{\mathcal{A}}_{11,t^*+1,1}$ up to just before step 6c of KeyGen. We then run step 6c of KeyGen except that we skip over computing $(\mathsf{sk}_{g_t}, c'_t, \mathsf{sk}'_{h_t})$. This is possible without needing to know $(r_{\mathsf{msk}_t}, \mathsf{msk}_t, r_{\mathsf{KeyGen}_t})$ since

- We only need to know K_i for $i < t^*+1 \le t$.

- In step 6c of KeyGen, to compute all the ct'_{i+1}, we only need to know $(r_{\mathsf{msk}_{i+1}}, \mathsf{msk}_{i+1}, r_{\mathsf{KeyGen}_{i+1}})$ for $i+1 > t$.
- In step 6c of KeyGen, to compute $(\mathsf{sk}_{g_i}, c'_i, \mathsf{sk}'_{h_i})$ for $i \neq t$, we only need to know $(r_{\mathsf{msk}_i}, \mathsf{msk}_i, r_{\mathsf{KeyGen}_i})$ for $i \neq t$.

We then send challenge function $g_{f,\widetilde{\mathsf{st}}_t,c_t}$ to the $\mathsf{OneFSFE}$ challenger and receive back a function key sk^*. We set $\mathsf{sk}_{g_t} = \mathsf{sk}^*$. We then compute $c'_t \leftarrow \mathsf{SKE}'.\mathsf{Enc}(k'_t, \mathsf{sk}_{g,t})$ and $\mathsf{sk}'_{h_t} \leftarrow \mathsf{OneCompFE}.\mathsf{KeyGen}(\mathsf{msk}'_t, h_t)$ for $h_t = h_{c_t,c'_t}$. We then compute $\mathbf{Hybrid}^{\mathcal{A}}_{11,t,1}$ from step 6d of KeyGen to just before step 7c of iteration $i = t$ of **Encryption Phase 2**. Note that this computation only requires $(r_{\mathsf{msk}_i}, \mathsf{msk}_i, r_{\mathsf{KeyGen}_i})$ for $i < t$.

We send challenge message pair (m_0, m_1) to the $\mathsf{OneFSFE}$ challenger where

$$m_0 = (x_i^{(b)}, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_{\mathcal{Y}}})$$

$$m_1 = (0^{\ell_{\mathcal{X}}}, 0^{\ell_{\mathcal{S}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 0^\lambda, 1, r_{k_i}, \psi_i)$$

and receive back a $\mathsf{OneFSFE}$ encryption ct^* of one of them. We set $\mathsf{ct}_t = \mathsf{ct}^*$. If $i = 1$, $\mathsf{Post}.\mathsf{CT}_1 = \mathsf{ct}_1$. Else $\mathsf{Post}.\mathsf{CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$. As needed for the security game, we can observe that

$$\begin{aligned}
&g_{c_t}(0^{\ell_{\mathcal{X}}}, 0^{\ell_{\mathcal{S}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 0^\lambda, 1, r_{k_t}, \psi_t) \\
&= \mathsf{SKE}.\mathsf{Dec}(k_t, c_t) \oplus (\psi_t, 0^{|\mathsf{ct}'_{t+1}|}) \\
&= (\theta_t \oplus \psi_t, \mathsf{ct}'_{t+1}) \\
&= (y_t, \mathsf{ct}'_{t+1}) \\
&= g_{c_t}(x_i^{(b)}, p_i, p_{i+1}, r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}}, r_{\mathsf{msk}_{i+1}}, r_{\mathsf{KeyGen}_{i+1}}, 0, 0^\lambda, 0^{\ell_{\mathcal{Y}}})
\end{aligned}$$

This is because c_t encrypts $(\theta_t, \mathsf{ct}'_{t+1})$ where $\theta_t \oplus \psi_t = y_t$ and where ct'_{t+1} is generated in the same way as in the $\alpha_t = 0$ branch of g_t.

We then compute the rest of $\mathbf{Hybrid}^{\mathcal{A}}_{11,t,1}$ starting from step 7d of iteration $i = t$ of **Encryption Phase 2**. Note that this computation only requires $(r_{\mathsf{msk}_i}, \mathsf{msk}_i, r_{\mathsf{KeyGen}_i})$ for $i > t$.

Observe that if c^* was an encryption of m_0 then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{11,t,0}$, and if c^* was an encryption of m_1 then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{11,t,1}$. Thus, by the security of $\mathsf{OneFSFE}$, the outputs of these hybrids must be indistinguishable. □

Lemma C.90. *If* $\mathsf{OneCompFE}$ *is a single-key, single-ciphertext, selective secure functional encryption scheme, then for all* $\lambda \in \mathbb{N}$*, all* $t \in [t^* + 2, n + 1]$*, and all* PPT *adversaries* $\mathcal{A}$*,*

$$\Delta(\mathbf{Hybrid}^{\mathcal{A}}_{11,t-1,1}(1^\lambda), \mathbf{Hybrid}^{\mathcal{A}}_{11,t,0}(1^\lambda)) \leq \mathsf{negl}(\lambda)$$

Proof. Observe that the only difference between $\mathbf{Hybrid}^{\mathcal{A}}_{11,t-1,1}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{11,t,0}$ is in how we compute ct'_t in the latter hybrid. Thus, the lemma follows by a reduction to the security of $\mathsf{OneCompFE}$ since we only use $(r'_{\mathsf{msk}_t}, \mathsf{msk}'_t, r'_{\mathsf{Enc}_t})$ to compute ct'_t and sk'_{h_t}.

In the reduction, without sampling or computing $(r'_{\mathsf{msk}_t}, \mathsf{msk}'_t, r'_{\mathsf{Enc}_t})$, we run $\mathbf{Hybrid}^{\mathcal{A}}_{11,t,0}$ up to just before iteration $i = t - 1$ of step 6c of KeyGen. This is possible without needing to know $(r'_{\mathsf{msk}_t}, \mathsf{msk}'_t, r'_{\mathsf{Enc}_t})$ since

- We only need to know K_i for $i < t^* + 1 < t$.
- For iterations $i < t-1$ of step 6c, to compute ct'_{i+1}, we only need to know $(r'_{\mathsf{msk}_{i+1}}, \mathsf{msk}'_{i+1}, r'_{\mathsf{Enc}_{i+1}})$ for $i + 1 < t$.
- For iterations $i < t - 1$ of step 6c, to compute sk'_{h_i}, we only need to know $(r'_{\mathsf{msk}_i}, \mathsf{msk}'_i, r'_{\mathsf{Enc}_i})$ for $i < t - 1$.

We then send challenge message pair (m_0, m_1) to the $\mathsf{OneFSFE}$ challenger where

$$m_0 = (f, \widetilde{\mathsf{st}}_t, r_{\mathsf{msk}_t}, r_{\mathsf{KeyGen}_t}, 0, 0^\lambda)$$

$$m_1 = (0^{\ell_{\mathcal{F}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 1, r'_{k_t})$$

and receive back a $\mathsf{OneFSFE}$ encryption ct^* of one of them. We set $\mathsf{ct}'_t = \mathsf{ct}^*$.

We then run the rest of step 6c of KeyGen of $\mathbf{Hybrid}^{\mathcal{A}}_{11,t,0}$ starting from step 6c.iv in iteration $i = t-1$, but skip over computing sk'_{h_t}. This is possible without needing to know $(r'_{\mathsf{msk}_t}, \mathsf{msk}'_t, r'_{\mathsf{Enc}_t})$ since

- For iterations $i > t-1$ of step 6c, to compute ct'_{i+1}, we only need to know $(r'_{\mathsf{msk}_{i+1}}, \mathsf{msk}'_{i+1}, r'_{\mathsf{Enc}_{i+1}})$ for $i + 1 > t$.
- For iterations $i \geq t - 1$ of step 6c, to compute sk'_{h_i} for $i \neq t$, we only need to know $(r'_{\mathsf{msk}_i}, \mathsf{msk}'_i, r'_{\mathsf{Enc}_i})$ for $i \neq t$.

We then send function h_{c_t, c'_t} to the $\mathsf{OneFSFE}$ challenger and receive back a function key sk^*. We set $\mathsf{sk}'_{h_t} = \mathsf{sk}^*$. As needed for the security game, we can observe that

$$\begin{aligned}
&h_{c_t, c'_t}(0^{\ell_{\mathcal{F}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 1, r'_{k_t}) \\
&= \mathsf{SKE}'.\mathsf{Dec}(k'_t, c'_t) \\
&= \mathsf{sk}_{g_t} \\
&= h_{c_t, c'_t}(f, \widetilde{\mathsf{st}}_t, r_{\mathsf{msk}_t}, r_{\mathsf{KeyGen}_t}, 0, 0^\lambda)
\end{aligned}$$

This is because c'_t encrypts sk_{g_t} where sk_{g_t} is generated in the same way as in the $\alpha'_t = 0$ branch of h_{c_t, c'_t}.

We then compute the remainder of $\mathbf{Hybrid}^{\mathcal{A}}_{11,t,0}$ starting from step 6d of KeyGen. Note that we do not need to know $(r'_{\mathsf{msk}_t}, \mathsf{msk}'_t, r'_{\mathsf{Enc}_t})$ to compute the remainder of the hybrid since

- Computing ct'_{t^*+1} only requires $(r'_{\mathsf{msk}_{t^*+1}}, r'_{\mathsf{Enc}_{t^*+1}})$ and $t^* + 1 < t$.

- **Encryption Phase 2** only needs to know $(r'_{\mathsf{msk}_{i+1}}, r'_{\mathsf{Enc}_{i+1}})$ for $i \geq t$.

Observe that if c^* was an encryption of m_0 then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{10,t-1,1}$, and if c^* was an encryption of m_1 then we exactly emulate $\mathbf{Hybrid}^{\mathcal{A}}_{11,t,0}$. Thus, by the security of $\mathsf{OneCompFE}$, the outputs of these hybrids must be indistinguishable. □

Hybrid$_{12}^{\mathcal{A}}$: This is the same as **Hybrid**$_{11,n+1,0}^{\mathcal{A}}$. Observe that we no longer need to compute p_i for $i \geq t^* + 1$.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.

   (b) For $i \in [n+1]$,

      i. $r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i} \leftarrow \{0,1\}^\lambda$.

      ii. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$, $\mathsf{msk}'_i = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i})$, $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$, $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.

      iii. If $i < t^* + 1$,

         A. $p_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

         B. $K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i})$.

      iv. If $i \geq t^* + 1$,

         A. $\widetilde{\mathsf{st}}_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) **Compute phase 1 ciphertexts:**

      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.

      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.

   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) **Compute phase 1 state and output values:**

      i. $\mathsf{st}_1^{(b)} = \mathsf{st}_1$.

      ii. For $i \in [t^*]$,

         A. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

      iii. ~~$p_{t^*+1} = \mathsf{st}_{t^*+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{t^*+1}$.~~

   (c) **Compute** sk'_{h_i}: For $i \in [t^* + 1, n]$,

      i. $\theta_i \leftarrow \{0,1\}^{\ell_\mathcal{Y}}$.

      ii. $\mathsf{ct}'_{i+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (0^{\ell_\mathcal{F}}, 0^{\ell_\mathcal{S}}, 0^\lambda, 0^\lambda, 1, r'_{k_{i+1}}); r'_{\mathsf{Enc}_{i+1}})$.

iii. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, (\theta_i, \mathsf{ct}'_{i+1}))$.

iv. $\mathsf{sk}_{g_i} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_i, g_i; r_{\mathsf{KeyGen}_i})$ for $g_i = g_{f,\widetilde{\mathsf{st}}_i,c_i}$.

v. If $i > 1$,

A. $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \mathsf{sk}_{g_i})$.

B. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$ for $h_i = h_{c_i,c'_i}$.

(d) **Compute hardcoded values:**

i. If $t^* + 1 = 1$, $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$.

ii. If $t^* + 1 > 1$,

A. $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.

B. $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (0^{\ell_{\mathcal{F}}}, 0^{\ell_{\mathcal{S}}}, 0^{\lambda}, 0^{\lambda}, 1, r'_{k_{t^*+1}}); r'_{\mathsf{Enc}_{t^*+1}})$.

C. $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$.

(e) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

(f) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^* + 1, t^* + 2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute p_i and ψ_i:**

i. $(y_i, \mathsf{st}_{i+1}^{(b)}) = f(x_i^{(b)}, \mathsf{st}_i^{(b)})$.

ii. $\psi_i = y_i \oplus \theta_i$.

iii. ~~$p_{i+1} = \mathsf{st}_{i+1}^{(b)} \oplus \widetilde{\mathsf{st}}_{i+1}$.~~

(c) **Compute phase 2 ciphertexts:**

i. $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (0^{\ell_{\mathcal{X}}}, 0^{\ell_{\mathcal{S}}}, 0^{\ell_{\mathcal{S}}}, 0^{\lambda}, 0^{\lambda}, 0^{\lambda}, 0^{\lambda}, 1, r_{k_i}, \psi_i))$.

ii. If $i = 1$, $\mathsf{Post.CT}_1 = \mathsf{ct}_1$. Else $\mathsf{Post.CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.

(d) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(e) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.91. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}_{11,n+1,0}^{\mathcal{A}}(1^{\lambda}), \mathbf{Hybrid}_{12}^{\mathcal{A}}(1^{\lambda})) = 0$$

Proof. The hybrids are identical. □

Hybrid$_{13}^{\mathcal{A}}$: Finally, we observe that for $i \geq t^* + 1$ the hybrid does not depend on the challenge stream except for the values of st_1 and $\{y_i\}_{i\in[n]}$. Thus, since $x^{(b)}$ and $x^{(0)}$ have the same starting state and output values, we can exchange stream $x^{(b)}$ for $x^{(0)}$. The view of the adversary in the final hybrid is independent of the bit b.

1. **Guess Stream Length:** $(t^*, n) \leftarrow [T_{\mathcal{A},\lambda}] \times [T_{\mathcal{A},\lambda}]$ where $T_{\mathcal{A},\lambda}$ is the maximum runtime of $\mathcal{A}$ on security parameter λ.

2. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.

3. **Setup:**

   (a) $\mathsf{Pre.MSK} \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Setup}(1^\lambda, 1^{L_\mathcal{F}}, 1^{L_\mathcal{S}}, 1^{L_\mathcal{X}}, 1^{L_\mathcal{Y}})$.

   (b) For $i \in [n+1]$,

      i. $r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i} \leftarrow \{0,1\}^\lambda$.

      ii. $\mathsf{msk}_i = \mathsf{OneFSFE.Setup}(1^\lambda; r_{\mathsf{msk}_i})$, $\mathsf{msk}'_i = \mathsf{OneCompFE.Setup}(1^\lambda; r'_{\mathsf{msk}_i})$, $k_i = \mathsf{SKE.Setup}(1^\lambda; r_{k_i})$, $k'_i = \mathsf{SKE'.Setup}(1^\lambda; r'_{k_i})$.

      iii. If $i < t^* + 1$,

         A. $p_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

         B. $K_i = (p_i, r_{\mathsf{msk}_i}, r'_{\mathsf{msk}_i}, r_{k_i}, r'_{k_i}, r_{\mathsf{KeyGen}_i}, r'_{\mathsf{Enc}_i})$.

      iv. If $i \geq t^* + 1$,

         A. $\widetilde{\mathsf{st}}_i \leftarrow \{0,1\}^{\ell_\mathcal{S}}$.

4. **Challenge Bit:** $b \leftarrow \{0,1\}$.

5. **Encryption Phase 1:** For $i = 1, 2, \ldots, t^*$:
   If the adversary does not make exactly t^* queries during this phase, output $\perp$ and halt.

   (a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

   (b) **Compute phase 1 ciphertexts:**

      i. If $i < t^*$, $\mathsf{Post.CT}_i = \mathsf{Post\text{-}One\text{-}sFE.EncLocal}((K_i, K_{i+1}), i, x_i^{(0)})$.

      ii. If $i = t^*$, $\mathsf{Post.CT}_i = \perp$.

   (c) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

   (d) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

6. **KeyGen:**

   (a) $\mathcal{A}$ sends f to the challenger. Define $\mathsf{st}_1 = \perp$.

   (b) **Compute phase 1 state and output values:**

      i. $\mathsf{st}_1^{(0)} = \mathsf{st}_1$.

      ii. For $i \in [t^*]$,

         A. $(y_i, \mathsf{st}_{i+1}^{(0)}) = f(x_i^{(0)}, \mathsf{st}_i^{(0)})$.

   (c) **Compute** sk'_{h_i}: For $i \in [t^* + 1, n]$,

      i. $\theta_i \leftarrow \{0,1\}^{\ell_\mathcal{Y}}$.

      ii. $\mathsf{ct}'_{i+1} \leftarrow \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{i+1}, (0^{\ell_\mathcal{F}}, 0^{\ell_\mathcal{S}}, 0^\lambda, 0^\lambda, 1, r'_{k_i}); r'_{\mathsf{Enc}_{i+1}})$.

iii. $c_i \leftarrow \mathsf{SKE.Enc}(k_i, (\theta_i, \mathsf{ct}'_{i+1}))$.

iv. $\mathsf{sk}_{g_i} \leftarrow \mathsf{OneFSFE.KeyGen}(\mathsf{msk}_i, g_i; r_{\mathsf{KeyGen}_i})$ for $g_i = g_{f,\widetilde{\mathsf{st}}_i,c_i}$.

v. If $i > 1$,

A. $c'_i \leftarrow \mathsf{SKE'.Enc}(k'_i, \mathsf{sk}_{g_i})$.

B. $\mathsf{sk}'_{h_i} \leftarrow \mathsf{OneCompFE.KeyGen}(\mathsf{msk}'_i, h_i)$ for $h_i = h_{c_i,c'_i}$.

(d) **Compute hardcoded values:**

i. If $t^*+1 = 1$, $\mathsf{Post.Dec.st}_1 = \mathsf{sk}_{g_1}$.

ii. If $t^*+1 > 1$,

A. $\mathsf{Post.Dec.st}_1 \leftarrow \mathsf{Post\text{-}One\text{-}sFE.KeyGenLocal}(K_1, f, \mathsf{st}_1)$.

B. $\mathsf{ct}'_{t^*+1} = \mathsf{OneCompFE.Enc}(\mathsf{msk}'_{t^*+1}, (0^{\ell_{\mathcal{F}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 1, r'_{k_{t^*+1}}); r'_{\mathsf{Enc}_{t^*+1}})$.

C. $\mathsf{Post.Dec.st}_{t^*+1} = \mathsf{ct}'_{t^*+1}$.

(e) $\mathsf{Pre.SK}_f \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.KeyGenHardwire}(\mathsf{Pre.MSK}, \mathsf{Post\text{-}One\text{-}sFE.Dec}, \mathsf{Post.Dec.st}_1, t^*, y_{t^*-1}, y_{t^*}, \mathsf{Post.Dec.st}_{t^*+1})$

(f) Send $\mathsf{SK}_f = \mathsf{Pre.SK}_f$ to $\mathcal{A}$.

7. **Encryption Phase 2:** For $i = t^*+1, t^*+2, \ldots, n$:
If the adversary does not make exactly $n - t^*$ queries during this phase, output $\perp$ and halt.

(a) $\mathcal{A}$ sends $(x_i^{(0)}, x_i^{(1)})$ to the challenger.

(b) **Compute ψ_i:**

i. $(y_i, \mathsf{st}_{i+1}^{(0)}) = f(x_i^{(0)}, \mathsf{st}_i^{(0)})$.

ii. $\psi_i = y_i \oplus \theta_i$.

(c) **Compute phase 2 ciphertexts:**

i. $\mathsf{ct}_i \leftarrow \mathsf{OneFSFE.Enc}(\mathsf{msk}_i, (0^{\ell_{\mathcal{X}}}, 0^{\ell_{\mathcal{S}}}, 0^{\ell_{\mathcal{S}}}, 0^\lambda, 0^\lambda, 0^\lambda, 1, r_{k_i}, \psi_i))$.

ii. If $i = 1$, $\mathsf{Post.CT}_1 = \mathsf{ct}_1$. Else $\mathsf{Post.CT}_i = (\mathsf{ct}_i, \mathsf{sk}'_{h_i})$.

(d) $\mathsf{Pre.CT}_i \leftarrow \mathsf{Pre\text{-}One\text{-}sFE.Enc}(\mathsf{Pre.MSK}, i, \mathsf{Post.CT}_i)$.

(e) Send $\mathsf{CT}_i = \mathsf{Pre.CT}_i$ to $\mathcal{A}$.

8. **Output**: $\mathcal{A}$ sends b' to the challenger. Output 1 if $b = b'$, and output 0 else.

Lemma C.92. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Delta(\mathbf{Hybrid}_{12}^{\mathcal{A}}(1^\lambda), \mathbf{Hybrid}_{13}^{\mathcal{A}}(1^\lambda)) = 0$$

Proof. The hybrids are identical. In step 6b of KeyGen and step 7b of **Encryption Phase 2**, the two hybrids compute the same values of y_i and ψ_i since $x^{(b)}$ and $x^{(0)}$ have the same output values. Since the previous hybrid did not depend on any other values computed during these two steps, then exchanging stream $x^{(b)}$ to $x^{(0)}$ as in the current hybrid does not affect the output of the hybrid. □

Lemma C.93. *For all $\lambda \in \mathbb{N}$ and all adversaries $\mathcal{A}$,*

$$\Pr[\mathsf{Wrap}(\mathbf{Hybrid}_{13}^{\mathcal{A}})(1^\lambda) = 1] \leq \frac{1}{2}$$

Proof. The proof relies on the fact that the adversary's view in $\mathbf{Hybrid}_{13}^{\mathcal{A}}$ is independent of the challenge bit b.

In each iteration of Wrap, we run an instance of $\mathbf{Hybrid}_{13}^{\mathcal{A}}$. The output of Wrap is defined to be the output of the first instance of $\mathbf{Hybrid}_{13}^{\mathcal{A}}$ where we did not output $\bot$, or 0 if all instances output $\bot$. Consider any instance of $\mathbf{Hybrid}_{13}^{\mathcal{A}}$. Conditioned on not outputting $\bot$, we will only output 1 if the adversary correctly guesses $b = b'$. However, since the adversary's view in $\mathbf{Hybrid}_{13}^{\mathcal{A}}$ is independent of the bit b, then conditioned on not outputting $\bot$, the probability that the adversary makes it to the end of the hybrid and guesses correctly is at most $\frac{1}{2}$. Thus, the probability that $\mathsf{Wrap}(\mathbf{Hybrid}_{13}^{\mathcal{A}})(1^\lambda)$ outputs 1 is at most $\frac{1}{2}$. □

Lemma C.94. *Assuming iO for* P/Poly *and injective* PRG*s,* One-sFE *is a single-key, single-ciphertext, adapively secure* sFE *schme.*

Proof. We start by combining all of our intermediate lemmas to prove indistinguishability between $\mathbf{Hybrid}_0^{\mathcal{A}}$ and $\mathsf{Wrap}(\mathbf{Hybrid}_{13}^{\mathcal{A}})$. By Lemma C.6, $\mathbf{Hybrid}_0^{\mathcal{A}}$ and $\mathsf{Wrap}(\mathbf{Hybrid}_1^{\mathcal{A}})$ are indistinguishable. We then show indistinguishability between $\mathsf{Wrap}(\mathbf{Hybrid}_1^{\mathcal{A}})$ and $\mathsf{Wrap}(\mathbf{Hybrid}_{13}^{\mathcal{A}})$ by iterating through the wrapped versions of each of the intermediate hybrids of this security proof. To prove indistinguishability between each pair of intermediate wrapped hybrids, we rely on Lemma C.5 which says that it is sufficient to prove indistinguishability between the unwrapped hybrids, which we have already done.

Since $\mathbf{Hybrid}_0^{\mathcal{A}}$ and $\mathsf{Wrap}(\mathbf{Hybrid}_{13}^{\mathcal{A}})$ are indistinguishable, then by Lemma C.93, for all PPT adversaries $\mathcal{A}$,

$$\Pr[\mathbf{Hybrid}_0^{\mathcal{A}}(1^\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

which implies that One-sFE is a single-key, single-ciphertext, adapively secure sFE scheme. □

D Security Proof from Section 7

In this section, we prove that sFE is adaptively secure (see Definition 3.30). In this proof, we will use an alternate, but equivalent definition of adaptive security.

Definition D.1 (Adaptive Security for Public-Key sFE, Equivalent Definition)**.** *A public-key streaming FE scheme* sFE *for* $\mathsf{P/Poly}$ *is adaptively secure if there exists a negligible function* μ *such that for all* $\lambda \in \mathbb{N}$ *and all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{ExptGuess}_{\mathcal{A}}^{\mathsf{sFE\text{-}Adaptive}}(1^\lambda) = 1]\right| \leq \frac{1}{2} + \mu(\lambda)$$

where for each $\lambda \in \mathbb{N}$*, we define*

$\mathsf{ExptGuess}_{\mathcal{A}}^{\mathsf{sFE\text{-}Adaptive}}(1^\lambda)$

1. ***Parameters****:* $\mathcal{A}$ *takes as input* 1^λ*, and outputs a function size* $1^{\ell_{\mathcal{F}}}$*, a state size* $1^{\ell_{\mathcal{S}}}$*, an input size* $1^{\ell_{\mathcal{X}}}$*, and an output size* $1^{\ell_{\mathcal{Y}}}$*.*
2. ***Setup****: Compute* $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{sFE.Setup}(1^\lambda, 1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}})$*.*
3. ***Public Key****: Send* mpk *to* $\mathcal{A}$*.*
4. ***Challenge Bit****: Sample* $b \leftarrow \{0,1\}$*.*
5. *For a polynomial number of rounds, the adversary can do either one of the following in each round:*
   (a) ***Function Query****:*
      i. $\mathcal{A}$ *outputs a streaming function query* $f \in \mathcal{F}[\ell_{\mathcal{F}}, \ell_{\mathcal{S}}, \ell_{\mathcal{X}}, \ell_{\mathcal{Y}}]$*.*
      ii. $\mathsf{sk}_f \leftarrow \mathsf{sFE.KeyGen}(\mathsf{msk}, f)$*.*
      iii. *Send* sk_f *to* $\mathcal{A}$*.*
   (b) ***Challenge Message Query****:*
      i. *If this is the first challenge message query, sample* $\mathsf{Enc.st} \leftarrow \mathsf{sFE.EncSetup}(\mathsf{mpk})$ *and initialize the index* $i = 0$*. Else, increment the index* i *by* 1*.*
      ii. $\mathcal{A}$ *outputs a challenge message pair* $(x_i^{(0)}, x_i^{(1)}) \in \{0,1\}^{\ell_{\mathcal{X}}} \times \{0,1\}^{\ell_{\mathcal{X}}}$*.*
      iii. $\mathsf{ct}_i \leftarrow \mathsf{sFE.Enc}(\mathsf{mpk}, \mathsf{Enc.st}, i, x_i^{(b)})$*.*
      iv. *Send* ct_i *to* $\mathcal{A}$*.*
6. ***Experiment Outcome****:* $\mathcal{A}$ *outputs a bit* b'*. The output of the experiment is set to* 1 *if* $b = b'$*, and* 0 *otherwise.*

Additionally, when running the experiment, we immediately halt and output 0 *if the adversary ever aborts or if it at any point some function query* f *submitted by the adversary yields different outputs on the challenge message streams submitted so far (i.e. if* $f(x^{(0)}) \neq f(x^{(1)})$ *for some function query* f *submitted by the adversary where* $\{(x_i^{(0)}, x_i^{(1)})\}_{i \in [t]}$ *are the message queries submitted so far,* $x^{(0)} = x_1^{(0)} \ldots x_t^{(0)}$*, and* $x^{(1)} = x_1^{(1)} \ldots x_t^{(1)}$*).*

Using standard techniques, it is easy to show that this is equivalent to the regular definition of adaptive security.

D.1 Proof Overview

To build intuition, we provide a brief overview of each hybrid below.

- $\mathbf{Hybrid}_0^{\mathcal{A}}$: This is the real world experiment. The adversary first receives the security parameter and chooses the function size, state size, input size, and output size. Then, the adversary receives the master public key MPK. After that, the adversary can repeatedly and adaptively submit either a streaming function f_j and receive a function key sk_{f_j} for f_j, or submit a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ and receives a ciphertext of $x_i^{(b)}$ for a fixed random bit $b \in \{0,1\}$. The adversary guesses b and wins if it guesses b correctly. Throughout the process, we require that for all f_j queried by the adversary, $f_j(x^{(0)}) = f_j(x^{(1)})$.

- $\mathbf{Hybrid}_1^{\mathcal{A}}$: We hardcode in values for the $\alpha = 1$ branch of G_{f_j,s_j,c_j} for each function key. For each function query f_j, we hardcode into c_j the values $(\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j)$ that are output by G_{f_j,s_j,c_j} on the $\alpha = 0$ branch if we run it on the input $(\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}})$ generated by the challenge message. Note that this input is independent of the choice of challenge messages $(x^{(0)}, x^{(1)})$. (By hardcode, we mean that we generate $c_i \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j)))$. The objective is to use the security of FE in the next hybrid to switch to the $\alpha = 1$ branch of each G_{f_j,c_j,s_j}, which does not require knowledge of $\mathsf{PRF}.K$ or $\mathsf{FPFE.msk}$ in the input. As $\mathsf{PRF}.K$ is used to generate all of the $\mathsf{One\text{-}sFE}$ master secret keys, being able to remove this value will allow us to hide these $\mathsf{One\text{-}sFE}$ master secret keys in later hybrids. The indistinguishability of $\mathbf{Hybrid}_0^{\mathcal{A}}$ and $\mathbf{Hybrid}_1^{\mathcal{A}}$ holds by the pseudorandom ciphertext property of SKE.

- $\mathbf{Hybrid}_2^{\mathcal{A}}$: In the challenge ciphertext, instead of encrypting

$$\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}}))$$

  we encrypt

$$\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.K_\lambda}}, 1, \mathsf{SKE}.k))$$

  Observe that the only functions keys generated using the corresponding $\mathsf{FE.msk}$ are for functions G_{f_j,s_j,c_j}. However, because we have hardcoded the correct output values into each c_j in our previous hybrid, then for all j,

$$G_{f_j,s_j,c_j}(\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}}) = G_{f_j,s_j,c_j}(0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.K_\lambda}}, 1, \mathsf{SKE}.k)$$

  Thus, the indistinguishability of $\mathbf{Hybrid}_1^{\mathcal{A}}$ and $\mathbf{Hybrid}_2^{\mathcal{A}}$ holds by the selective-security of FE. Selective security is sufficient as the messages $(\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}})$ and $(0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.K_\lambda}}, 1, \mathsf{SKE}.k)$ can be computed at the beginning of the experiment, even before learning $\mathsf{FE.mpk}$.

- $\mathbf{Hybrid}_3^{\mathcal{A}}$: For each j, to determine the values we need to hardcode into c_j, we use randomness $r_{\mathsf{Setup},j}, r_{\mathsf{KeyGen},j}, r_{\mathsf{EncSetup},j}, r_{\mathsf{PRF2},j}, r_{\mathsf{Enc},j}$ to generate $\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j$, $\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{PRF2}.k_j$, and $\mathsf{FPFE.ct}_j$. Instead of generating these random values using $\mathsf{PRF}.K$, we now generate these values using true randomness. Because of the change made in our previous hybrid, the key $\mathsf{PRF}.K$ is not used anywhere else in our experiment, so the indistinguishability of $\mathbf{Hybrid}_2^{\mathcal{A}}$ and $\mathbf{Hybrid}_3^{\mathcal{A}}$ holds by the security of PRF.

- $\mathbf{Hybrid}_4^{\mathcal{A}}$: In the ciphertext, we replace the FPFE function keys for $H_{i,x_i^{(b)},t_i}$ with function keys for new functions $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$ (defined in Figure 8) that have additional branches of computation.

– When $\beta = 0$, $H^*_{i,x_i^{(b)},x_i^{(0)},v_i}$ will act the same as $H_{i,x_i^{(b)},t_i}$ and will generate a $\mathsf{One\text{-}sFE}$ ciphertext for $x_i^{(b)}$.

– When $\beta = 1$, $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$ will instead generate a $\mathsf{One\text{-}sFE}$ ciphertext for $x_i^{(0)}$.

– When $\beta = 2$, $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$ will simply output v_i (which is set to 0 in this hybrid).

As $H_{i,x_i^{(b)},t_i}$ and $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$ act the same when $\beta = 0$, and we only encrypt FPFE messages where $\beta = 0$, then the indistinguishability of $\mathbf{Hybrid}_3^{\mathcal{A}}$ and $\mathbf{Hybrid}_4^{\mathcal{A}}$ holds by the function privacy of FPFE.

- We will now go through a series of hybrids for $k = 1$ to q where $q = q(\lambda)$ is the runtime of $\mathcal{A}$ and an implicit bound on the number of function queries made by $\mathcal{A}$. At a high level, the goal is to one by one switch to the $\beta = 1$ branch in every FPFE ciphertext. This will allow us to use the function privacy of FPFE to remove the dependence on b present in the $\beta = 0$ branch of each $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$.

  – $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}$: We prepare to switch to the $\beta = 2$ branch in the k^{th} FPFE ciphertext. For each i, we replace the value v_i in the FPFE function key of $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$ (or for $k > 1$, the value $v_{i,k-1}$ in $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k-1}}$) with a new value $v_{i,k}$ which corresponds to the output of $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$ on the message $(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, \mathsf{PRF2}.k_k, 0)$ encrypted in the k^{th} FPFE ciphertext. This value $v_{i,k}$ is an encryption of $x_i^{(b)}$ under $\mathsf{One\text{-}sFE.msk}_k$ using randomness generated by $\mathsf{PRF2}.k_k$. Since the value of v_i (or $v_{i,k-1}$) only affects the $\beta = 2$ branch of $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$ (or $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k-1}}$), and we only encrypt FPFE ciphertexts where $\beta = 0$ or $\beta = 1$, then we can perform this change due to the function privacy of FPFE.

  – $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}$: We now switch to the $\beta = 2$ branch of the k^{th} FPFE ciphertext. When we hardcode values into c_k in our function key, instead of encrypting

$$\mathsf{FPFE.ct}_k \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, \mathsf{PRF2}.k_k, 0))$$

  we encrypt

$$\mathsf{FPFE.ct}_k \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$$

  Observe that the only FPFE function keys generated using $\mathsf{FPFE.msk}$ are for functions $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$. However, because we hardcoded the correct output values into each $v_{i,k}$, then

$$\begin{aligned} &H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, \mathsf{PRF2}.k_k, 0) \\ &= H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}} 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2) \end{aligned}$$

  Thus, the indistinguishability of $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}$ holds by the message privacy of FPFE.

– $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}$: We would now like to change $v_{i,k}$ from a $\mathsf{One\text{-}sFE}$ encryption of $x_i^{(b)}$ to a $\mathsf{One\text{-}sFE}$ encryption of $x_i^{(0)}$. However, in order to perform that step, we first need to use true randomness for the encryption. Thus, in this hybrid, instead of generating $r_{i,k}$ (which is the randomness used to generate $v_{i,k}$: the i^{th} ciphertext of $x_i^{(b)}$ under $\mathsf{One\text{-}sFE.msk}_k$ and $\mathsf{One\text{-}sFE.Enc.st}_k$) using $\mathsf{PRF2}.k_k$, we generate $r_{i,k}$ using true randomness. Observe that $\mathsf{PRF2}.k_k$ was removed from our experiment in the previous hybrid when we switched to the $\beta = 2$ branch in $\mathsf{FPFE.ct}_k$. Thus, the indistinguishability of $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}$ holds by the security of $\mathsf{PRF2}$.

– $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}$: We now invoke the security of $\mathsf{One\text{-}sFE}$ to change the value of $v_{i,k}$. For each i, instead of computing

$$v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(b)})$$

we compute

$$v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(0)})$$

Observe that this is equivalent to switching from an encryption of $x^{(b)}$ under $\mathsf{One\text{-}sFE.msk}_k$ to an encryption of $x^{(0)}$ under $\mathsf{One\text{-}sFE.msk}_k$. (If for $d \in \{0,1\}$, $\mathsf{CT}^{(d)} = \{\mathsf{CT}_i^{(d)}\}_{i\in[n]}$ is an encryption of $x^{(d)}$ under $\mathsf{One\text{-}sFE.msk}_k$, then $v_{i,k} = \mathsf{CT}_i^{(b)}$ in the former case and $v_{i,k} = \mathsf{CT}_i^{(0)}$ in the latter.) To allow this change under the single-key, single-ciphertext, adaptive-security of $\mathsf{One\text{-}sFE}$, we need to ensure the following:

1. We only use $\mathsf{One\text{-}sFE.msk}_k$ and $\mathsf{One\text{-}sFE.Enc.st}_k$ for one ciphertext and one function key. For our challenge message, every function query generates a different $\mathsf{One\text{-}sFE}$ master secret key. Thus, we only use these values for one ciphertext (namely the challenge ciphertext) and one key (corresponding to the k^{th} function query f_k).
2. The $\mathsf{One\text{-}sFE}$ challenge function f_k has the same output value on the challenge messages $x^{(b)}$ and $x^{(0)}$. This holds since the sFE security game requires $f_j(x^{(0)}) = f_j(x^{(1)})$ for all functions f_j queried, so indeed $f_k(x^{(b)}) = f_k(x^{(0)})$.
3. We do not leak additional information about $\mathsf{One\text{-}sFE.msk}_k$, $\mathsf{One\text{-}sFE.Enc.st}_k$, or the randomness used to generate the ciphertext or function key. Except for their appearances in the k^{th} $\mathsf{One\text{-}sFE}$ ciphertext and function key, the only place that $\mathsf{One\text{-}sFE.msk}_k$ and $\mathsf{One\text{-}sFE.Enc.st}_k$ appeared was in $\mathsf{FPFE.ct}_k$. However, we removed these values from $\mathsf{FPFE.ct}_k$ in a previous hybrid when we switched to the $\beta = 2$ branch. Observe also that the randomness used is independent and uniform as we have already removed $\mathsf{PRF}.K$ and $\mathsf{PRF2}.k_k$ from the experiment.

Thus, the indistinguishability of $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}$ holds by the security of $\mathsf{One\text{-}sFE}$.

– $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}$: We undo the change made in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}$. Instead of computing $v_{i,k}$ using true randomness, we compute $v_{i,k}$ using randomness $r_{i,k}$ generated by the k^{th} $\mathsf{PRF2}$ key $\mathsf{PRF2}.k_k$. The indistinguishability of $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}$ holds by the security of $\mathsf{PRF2}$.

– $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,5}$: We now switch to the $\beta = 1$ branch in the k^{th} ciphertext. When we hardcode values into c_k in our function key, instead of encrypting

$$\mathsf{FPFE.ct}_k \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$$

we encrypt

$$\mathsf{FPFE.ct}_k \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, \mathsf{PRF2}.k_k, 1))$$

Observe that the only FPFE function keys we generated using FPFE.msk are for functions $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$. However, we observe that the value of $v_{i,k}$ is now in fact equal to $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k,$ $\mathsf{PRF2}.k_k, 1)$ as it is an encryption of $x_i^{(0)}$ under $\mathsf{One\text{-}sFE.msk}_k$ using randomness generated by $\mathsf{PRF2}.k_k$. Therefore,

$$\begin{aligned}&H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, \mathsf{PRF2}.k_k, 1)\\ &= H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}} 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2)\end{aligned}$$

and the indistinguishability of $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}$ and $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,5}$ holds by the message privacy of FPFE.

- $\mathbf{Hybrid}^{\mathcal{A}}_6$: In the ciphertext, we replace the FPFE function keys for $H^*_{i,x_i^{(b)},x_i^{(0)},t_i v_{i,q}}$ (where q is the runtime of $\mathcal{A}$) with FPFE function keys for functions $H^*_{i,x_i^{(0)},x_i^{(0)},t_i,v_i}$ where v_i is set to 0. Observe that q is an implicit bound on the number of function queries made by $\mathcal{A}$ and thus on the number of FPFE ciphertexts that we generate. Therefore, by the time we reach $\mathbf{Hybrid}^{\mathcal{A}}_{5,q,5}$, we will have switched all FPFE ciphertexts to the $\beta = 1$ branch. But since $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,q}}$ and $H^*_{i,x_i^{(0)},x_i^{(0)},t_i,v_i}$ act the same when $\beta = 1$, then the indistinguishability of $\mathbf{Hybrid}^{\mathcal{A}}_{5,q,5}$ and $\mathbf{Hybrid}_6$ holds by the function privacy of FPFE.

Our final hybrid $\mathbf{Hybrid}^{\mathcal{A}}_6$ is independent of the bit b. Thus, any adversary's advantage in guessing b in $\mathbf{Hybrid}^{\mathcal{A}}_6$ is zero. But our proof shows that for any PPT adversary $\mathcal{A}$, $\mathcal{A}$'s advantage in guessing b in $\mathbf{Hybrid}^{\mathcal{A}}_1$ is negligibly close to $\mathcal{A}$'s advantage in guessing b in $\mathbf{Hybrid}^{\mathcal{A}}_6$. Thus, for any PPT adversary $\mathcal{A}$, the advantage in guessing b in the real world must be negligible, so security holds.

D.2 Formal Proof

We now formally prove security via a hybrid argument.

Remark D.2. We require all of our hybrids to immediately halt and output 0 if the adversary ever aborts or if it at any point some function query f submitted by the adversary yields different outputs on the challenge message streams submitted so far (i.e. if $f(x^{(0)}) \neq f(x^{(1)})$ for some function query f submitted by the adversary where $\{(x_i^{(0)}, x_i^{(1)})\}_{i\in[t]}$ are the message queries submitted so far, $x^{(0)} = x_1^{(0)} \dots x_t^{(0)}$, and $x^{(1)} = x_1^{(1)} \dots x_t^{(1)}$). We will consider the latter behavior to also be an abort condition. For notational simplicity, we will omit this requirement from the description of our hybrids.

$\mathbf{Hybrid}_0^{\mathcal{A}}(1^\lambda)$: This is the real world experiment. Though we have reordered some steps for the sake of the proof, this does not affect the outcome of the experiment.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{PRF}.K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$
   (c) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (d) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}}))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. $c_j \leftarrow \{0,1\}^{\ell_{\mathsf{SKE}.\mathsf{ct}_\lambda}}$
      iii. Let $G_j = G_{f_j, s_j, c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. Let $H_i = H_{i, x_i^{(b)}, t_i}$ as defined in Figure 5 (page 85).
      iii. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$
      iv. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.
      v. Send CT_i to the adversary.
6. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Hybrid$_1^{\mathcal{A}}(1^\lambda)$: For each j, we hardcode into c_j the values

$$(\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j) = G_{f_j,s_j,c_j}(\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}})$$

which would be generated in the real world experiment. This will allow us to later switch to the $\alpha = 1$ branch in G_{f_j,s_j,c_j} using the security of FE. Observe that the values being hardcoded into c_j can be computed before knowing $x^{(0)}$ or $x^{(1)}$.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{PRF}.K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$
   (c) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (d) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (e) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}}))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. $(r_{\mathsf{Setup},j}, r_{\mathsf{KeyGen},j}, r_{\mathsf{EncSetup},j}, r_{\mathsf{PRF2},j}, r_{\mathsf{Enc},j}) \leftarrow \mathsf{PRF.Eval}(\mathsf{PRF}.K, s_j)$
         B. $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda; r_{\mathsf{Setup},j})$
         C. $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j; r_{\mathsf{EncSetup}_j})$
         D. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j; r_{\mathsf{KeyGen},j})$
         E. $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda; r_{\mathsf{PRF2},j})$
         F. $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0); r_{\mathsf{Enc},j})$
         G. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j,s_j,c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. Let $H_i = H_{i,x_i^{(b)},t_i}$ as defined in Figure 5 (page 85).
      iii. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$

iv. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.

v. Send CT_i to the adversary.

6. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.3. *If* SKE *has pseudorandom ciphertexts, then for all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathbf{Hybrid}_0^{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}_1^{\mathcal{A}}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ such that

$$\left|\Pr[\mathbf{Hybrid}_0^{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}_1^{\mathcal{A}}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{1}$$

We build a PPT adversary $\mathcal{B}$ that breaks the pseudorandom ciphertext property of SKE. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}}$. $\mathcal{B}$ then sends message length $1^{\ell_{\mathsf{SKE}.m_\lambda}}$ to its SKE challenger where where $\ell_{\mathsf{SKE}.m_\lambda}$ is computed as described in the parameter section. $\mathcal{B}$ then computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{PRF}.K, \mathsf{FPFE.msk}, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}_0^{\mathcal{A}}$ and sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ samples $b \leftarrow \{0,1\}$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes $s_j \leftarrow \{0,1\}^\lambda$ and $(\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j)$ as in $\mathbf{Hybrid}_1^{\mathcal{A}}$. $\mathcal{B}$ sends $(\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j)$ as its challenge message to its SKE challenger and receives c_j which is either a uniform random value or an encryption of $(\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j)$ under SKE. $\mathcal{B}$ then computes $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_{f_j, s_j, c_j})$ and sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$.

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ computes CT_i as in $\mathbf{Hybrid}_0^{\mathcal{A}}$, and sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if every c_j is an independent uniform random value, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}_0^{\mathcal{A}}$, and if each c_j is an encryption of $\mathcal{B}$'s j^{th} challenge message $(\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j)$ under SKE, then $\mathcal{B}$ emulates $\mathbf{Hybrid}_1^{\mathcal{A}}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{SKE}.k$ to carry out this experiment. Thus, by Equation 1, this means that $\mathcal{B}$ breaks the pseudorandom ciphertext property of SKE as $\mathcal{B}$ can distinguish between receiving random values and valid ciphertexts with non-negligible probability. □

Hybrid$_2^{\mathcal{A}}$: We change the message encrypted in $\mathsf{FE.ct}$ so that we use the $\alpha = 1$ branch of every G_{f_j,s_j,c_j}. This allows us to remove $\mathsf{FPFE.msk}$ and $\mathsf{PRF}.K$ from $\mathsf{FE.ct}$.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{PRF}.K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$
   (c) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (d) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (e) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.k_\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. $(r_{\mathsf{Setup},j}, r_{\mathsf{KeyGen},j}, r_{\mathsf{EncSetup},j}, r_{\mathsf{PRF2},j}, r_{\mathsf{Enc},j}) \leftarrow \mathsf{PRF.Eval}(\mathsf{PRF}.K, s_j)$
         B. $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda; r_{\mathsf{Setup},j})$
         C. $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j; r_{\mathsf{EncSetup}_j})$
         D. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j; r_{\mathsf{KeyGen},j})$
         E. $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda; r_{\mathsf{PRF2},j})$
         F. $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0); r_{\mathsf{Enc},j})$
         G. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j,s_j,c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. Let $H_i = H_{i,x_i^{(b)},t_i}$ as defined in Figure 5 (page 85).
      iii. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$
      iv. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.
      v. Send CT_i to the adversary.
6. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.4. *If* FE *is selectively secure, then for all PPT adversaries,*

$$\left|\Pr[\mathbf{Hybrid}_1^{\mathcal{A}}(1^\lambda)=1]-\Pr[\mathbf{Hybrid}_2^{\mathcal{A}}(1^\lambda)=1]\right|\leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ such that

$$\left|\Pr[\mathbf{Hybrid}_1^{\mathcal{A}}(1^\lambda)=1]-\Pr[\mathbf{Hybrid}_2^{\mathcal{A}}(1^\lambda)=1]\right|> \mathsf{negl}(\lambda) \tag{2}$$

We build a PPT adversary $\mathcal{B}$ that breaks the selective-security of FE. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}}$. $\mathcal{B}$ then sends function size $1^{\ell_{G_\lambda}}$, input size $1^{\ell_{\mathsf{FE}.m_\lambda}}$, and output size $1^{\ell_{\mathsf{FE}.out_\lambda}}$ to its FE challenger where $\ell_{G_\lambda}, \ell_{\mathsf{FE}.m_\lambda}, \ell_{\mathsf{FE}.out_\lambda}$ are computed as described in the parameter section. $\mathcal{B}$ computes $(\mathsf{PRF}.K, \mathsf{FPFE.msk}, \mathsf{SKE}.k)$ as in $\mathbf{Hybrid}_1^{\mathcal{A}}$. $\mathcal{B}$ sends challenge message pair $((\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}}), (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.K_\lambda}}, 1, \mathsf{SKE}.k))$ to its FE challenger and receives $(\mathsf{FE.mpk}, \mathsf{FE.ct})$ where $\mathsf{FE.ct}$ is either an encryption of $(\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}})$ or an encryption of $(0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.K_\lambda}}, 1, \mathsf{SKE}.k)$. $\mathcal{B}$ then sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$ and samples $b \leftarrow \{0,1\}$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes (s_j, c_j) as in $\mathbf{Hybrid}_1^{\mathcal{A}}$. $\mathcal{B}$ then sends function query $G_j = G_{f_j, s_j, c_j}$ to its FE challenger and receives a function key $\mathsf{FE.sk}_{G_j}$ in return. This is a valid function query since for all j,

$$G_{f_j,s_j,c_j}(\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}}) = G_{f_j,s_j,c_j}(0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.K_\lambda}}, 1, \mathsf{SKE}.k)$$

because c_j encrypts $(\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j)$ which are generated in the same way as in the $\alpha = 0$ branch of G_{f_j,s_j,c_j}. $\mathcal{B}$ then sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$.

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ computes CT_i as in $\mathbf{Hybrid}_1^{\mathcal{A}}$, and sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathsf{FE.ct}$ is an encryption of $(\mathsf{FPFE.msk}, \mathsf{PRF}.K, 0, 0^{\ell_{\mathsf{SKE}.k_\lambda}})$, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}_1^{\mathcal{A}}$, and if $\mathsf{FE.ct}$ is an encryption of $(0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.K_\lambda}}, 1, \mathsf{SKE}.k)$ then $\mathcal{B}$ emulates $\mathbf{Hybrid}_2^{\mathcal{A}}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{FE.msk}$ to carry out this experiment. Thus, by Equation 2, this means that $\mathcal{B}$ breaks the selective-security of FE as $\mathcal{B}$ can distinguish between the two ciphertexts with non-negligible probability. □

Hybrid$_3^{\mathcal{A}}$: We exchange the randomness generated by $\mathsf{PRF}.K$ with true randomness.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) ~~$\mathsf{PRF}.K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$~~
   (c) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (d) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (e) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.k_\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. ~~$(r_{\mathsf{Setup},j}, r_{\mathsf{KeyGen},j}, r_{\mathsf{EncSetup}_j} r_{\mathsf{PRF2},j}, r_{\mathsf{Enc},j}) \leftarrow \mathsf{PRF.Eval}(\mathsf{PRF}.K, s_j)$~~
         B. $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$
         C. $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$
         D. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$
         E. $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$
         F. $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$
         G. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j, s_j, c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. Let $H_i = H_{i, x_i^{(b)}, t_i}$ as defined in Figure 5 (page 85).
      iii. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$
      iv. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.
      v. Send CT_i to the adversary.
6. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.5. *If* PRF *is a secure PRF, then for all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathbf{Hybrid}_2^{\mathcal{A}}(1^\lambda)=1]-\Pr[\mathbf{Hybrid}_3^{\mathcal{A}}(1^\lambda)=1]\right|\leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ such that

$$\left|\Pr[\mathbf{Hybrid}_2^{\mathcal{A}}(1^\lambda)=1]-\Pr[\mathbf{Hybrid}_3^{\mathcal{A}}(1^\lambda)=1]\right|> \mathsf{negl}(\lambda) \tag{3}$$

We build a PPT adversary $\mathcal{B}$ that breaks the security of PRF. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}}$. $\mathcal{B}$ then sends input size 1^λ, and output size $1^{5\lambda}$ to its PRF challenger. $\mathcal{B}$ is then given oracle access to either $\mathsf{PRF.Eval}(\mathsf{PRF}.K,\cdot)$ for some $\mathsf{PRF}.K\leftarrow\mathsf{PRF.Setup}(1^\lambda,1^\lambda,1^{5\lambda})$ or to a uniformly random function $R\leftarrow\mathcal{R}_{\lambda,5\lambda}$ where $\mathcal{R}_{\lambda,5\lambda}$ is the set of all functions from $\{0,1\}^\lambda$ to $\{0,1\}^{5\lambda}$. $\mathcal{B}$ computes $(\mathsf{FE.mpk},\mathsf{FE.msk},\mathsf{FPFE.msk},\mathsf{SKE}.k)$ as in $\mathbf{Hybrid}_2^{\mathcal{A}}$ and computes $\mathsf{FE.ct}\leftarrow\mathsf{FE.Enc}(\mathsf{FE.mpk},0^{\ell_{\mathsf{FPFE.msk}_\lambda}},0^{\ell_{\mathsf{PRF}.k_\lambda}},1,\mathsf{SKE}.k)$. (This does not require knowledge of $\mathsf{PRF}.K$). $\mathcal{B}$ then sends $\mathsf{MPK}=\mathsf{FE.mpk}$ to $\mathcal{A}$ and samples $b\leftarrow\{0,1\}$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ does the following: $\mathcal{B}$ samples $s_j\leftarrow\{0,1\}^\lambda$ and sets $(r_{\mathsf{Setup},j},r_{\mathsf{KeyGen},j},r_{\mathsf{EncSetup},j},r_{\mathsf{PRF2},j},r_{\mathsf{Enc},j})$ equal to the output of $\mathcal{B}$'s oracle on s_j.
$\mathcal{B}$ then computes $\mathsf{One\text{-}sFE.msk}_j\leftarrow\mathsf{One\text{-}sFE.Setup}(1^\lambda;r_{\mathsf{Setup},j})$,
$\mathsf{One\text{-}sFE.Enc.st}_j\leftarrow\mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j;r_{\mathsf{EncSetup},j})$,
$\mathsf{One\text{-}sFE.sk}_{f_j}\leftarrow\mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j,f_j;r_{\mathsf{KeyGen},j})$, $\mathsf{PRF2}.k_j\leftarrow\mathsf{PRF2.Setup}(1^\lambda;r_{\mathsf{PRF2},j})$, and $\mathsf{FPFE.ct}_j\leftarrow\mathsf{FPFE.Enc}(\mathsf{FPFE.msk},(\mathsf{One\text{-}sFE.msk}_j,\mathsf{One\text{-}sFE.Enc.st}_j,\mathsf{PRF2}.k_j,0);r_{\mathsf{Enc},j})$ using these values as randomness. $\mathcal{B}$ computes c_j and SK_{f_j} from these values as in $\mathbf{Hybrid}_2^{\mathcal{A}}$ and sends SK_{f_j} to $\mathcal{A}$.

For each challenge message query $(x_i^{(0)},x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ computes CT_i as in $\mathbf{Hybrid}_2^{\mathcal{A}}$, and sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b=b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathcal{B}$'s oracle was a uniform random function R, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}_3^{\mathcal{A}}$, and if $\mathcal{B}$'s oracle was $\mathsf{PRF.Eval}(\mathsf{PRF}.K,\cdot)$, then $\mathcal{B}$ emulates $\mathbf{Hybrid}_2^{\mathcal{A}}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{PRF}.K$ to carry out this experiment. Thus, by Equation 3, this means that $\mathcal{B}$ breaks the security of PRF as $\mathcal{B}$ can distinguish between a random function and the PRF. □

$H^*_{i,x_i,x'_i,t_i,v_i}(\mathsf{One\text{-}sFE.msk}, \mathsf{One\text{-}sFE.Enc.st}, \mathsf{PRF2}.k, \beta)$:

- If $\beta = 0$
  1. $r_i \leftarrow \mathsf{PRF2.Eval}(\mathsf{PRF2}.k, t_i)$
  2. Output $\mathsf{One\text{-}sFE.ct}_i \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}, \mathsf{One\text{-}sFE.Enc.st}, i, x_i; r_i)$
- If $\beta = 1$
  1. $r_i \leftarrow \mathsf{PRF2.Eval}(\mathsf{PRF2}.k, t_i)$
  2. Output $\mathsf{One\text{-}sFE.ct}_i \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}, \mathsf{One\text{-}sFE.Enc.st}, i, x'_i; r_i)$
- Else, output v_i

Figure 8:

Hybrid$_4^{\mathcal{A}}(1^\lambda)$: We replace each $H_{i,x_i^{(b)},t_i}$ with a new function $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$ that has additional branches of computation. We also move some steps further up in the proof, which we can do since these steps do not depend on the function queries.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (c) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (d) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.k_\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. **Pre-Compute FPFE Ciphertexts**: For $j \in [q]$ where $q = q(\lambda)$ is a bound on the runtime of $\mathcal{A}$,
   (a) $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$.
   (b) $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$.
   (c) $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$
   (d) $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$
6. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$

ii. **Compute c_j:**
   A. ~~$\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$~~
   B. ~~$\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$~~
   C. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$
   D. ~~$\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$~~
   E. ~~$\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$~~
   F. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$

iii. Let $G_j = G_{f_j, s_j, c_j}$ as defined in Figure 6 (page 86).

iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$

v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.

(b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.

i. $t_i \leftarrow \{0,1\}^\lambda$

ii. $v_i = 0^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}}$

iii. Let $H_i = H^*_{i, x_i^{(b)}, x_i^{(0)}, t_i, v_i}$ as defined in Figure 8.

iv. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$

v. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.

vi. Send CT_i to the adversary.

7. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.6. *If* FPFE *is a function-private-selective-secure* FE *scheme, then for all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathbf{Hybrid}_3^\mathcal{A}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}_4^\mathcal{A}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ such that

$$\left|\Pr[\mathbf{Hybrid}_3^\mathcal{A}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}_4^\mathcal{A}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{4}$$

We build a PPT adversary $\mathcal{B}$ that breaks the function-private-selective-security of FPFE. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}$. $\mathcal{B}$ then sends function size $1^{\ell_{H_\lambda}}$, input size $1^{\ell_{\mathsf{FPFE}.m_\lambda}}$, and output size $1^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}}$ to its FPFE challenger where $\ell_{H_\lambda}, \ell_{\mathsf{FPFE}.m_\lambda}, \ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ are computed as described in the parameter section. $\mathcal{B}$ computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{SKE}.k, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}_3^\mathcal{A}$ and sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ then samples $b \leftarrow \{0,1\}$.

Let $q = q(\lambda)$ be the running time of $\mathcal{A}$. Observe that $q = \mathsf{poly}(\lambda)$ as $\mathcal{A}$ is polytime and that $\mathcal{A}$ outputs at most $q(\lambda)$ function queries on security parameter λ. For $j \in [q]$, $\mathcal{B}$ computes $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j)$ as in $\mathbf{Hybrid}_3^\mathcal{A}$. (This does not require knowledge of $\mathsf{FPFE.msk}$ or f_j). $\mathcal{B}$ then sends challenge message pairs $\{((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0), (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))\}_{j\in[q]}$ to its FPFE challenger and receives $\{\mathsf{FPFE.ct}_j\}_{j\in[q]}$ where each $\mathsf{FPFE.ct}_j$ is an encryption of $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0)$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ computes $s_j \leftarrow \{0,1\}^\lambda$, $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$, $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$, and $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_{f_j, s_j, c_j})$, and sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$. (This is possible to compute as q is at least as large as the number of function queries that $\mathcal{A}$ makes).

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ does the following: $\mathcal{B}$ samples $t_i \leftarrow \{0,1\}^\lambda$ and sets $v_i = 0^{\ell_{\mathsf{One\text{-}sFE}.ct\lambda}}$. $\mathcal{B}$ sends a challenge function pair $(H_{i,x_i^{(b)},t_i}, H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i})$ to its FPFE challenger and receives an FPFE function key $\mathsf{FPFE.sk}_{H_i}$ which is either a function key for $H_{i,x_i^{(b)},t_i}$ or a function key for $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$. This is a valid function query pair since for all $j \in [q]$,

$$\begin{aligned}
&H_{i,x_i^{(b)},t_i}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0) \\
&= H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0)
\end{aligned}$$

as the two function act the same when $\beta = 0$. If $i = 1$, $\mathcal{B}$ sets $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, $\mathcal{B}$ sets $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$. $\mathcal{B}$ sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathcal{B}$ received only ciphertexts and function keys for the first message or function of each of its challenge pairs, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}_3^{\mathcal{A}}$, and if $\mathcal{B}$ received only ciphertexts and function keys for the second message or function of each of its challenge pairs, then $\mathcal{B}$ emulates $\mathbf{Hybrid}_4^{\mathcal{A}}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{FPFE.msk}$ to carry out this experiment. Thus, by Equation 4, this means that $\mathcal{B}$ breaks the function-private-selective-security of FPFE as $\mathcal{B}$ can distinguish between the two security games with non-negligible probability. □

Hybrid$^{\mathcal{A}}_{5,k,0}(1^\lambda)$: We replace v_i with $v_{i,k} = H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{PRF2}.k_k, 0)$. This will allow us to later use the security of FPFE to change the input message in the k^{th} ciphertext $\mathsf{FPFE.ct}_k$ so that it uses the $\beta = 2$ branch of H_i.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (c) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (d) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.k_\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. **Pre-Compute FPFE Ciphertexts**: For $j \in [q]$ where $q = q(\lambda)$ is a bound on the runtime of $\mathcal{A}$,
   (a) $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$.
   (b) $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$.
   (c) $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$.
   (d) If $j < k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$
   (e) If $j = k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$
   (f) If $j > k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$
6. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$
         B. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j,s_j,c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. $r_{i,k} = \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, t_i)$
      iii. $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(b)}; r_{i,k})$
      iv. Let $H_i = H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$ as defined in Figure 8.

v. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$

vi. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.

vii. Send CT_i to the adversary.

7. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.7. *If* FPFE *is a function-private-selective-secure* FE *scheme, then for all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathbf{Hybrid}_4^{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}_{5,1,0}^{\mathcal{A}}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ such that

$$\left|\Pr[\mathbf{Hybrid}_4^{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}_{5,1,0}^{\mathcal{A}}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{5}$$

We build a PPT adversary $\mathcal{B}$ that breaks the function-private-selective-security of FPFE. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}}$. $\mathcal{B}$ then sends function size $1^{\ell_{H_\lambda}}$, input size $1^{\ell_{\mathsf{FPFE}.m_\lambda}}$, and output size $1^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}}$ to its FPFE challenger where $\ell_{H_\lambda}, \ell_{\mathsf{FPFE}.m_\lambda}, \ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ are computed as described in the parameter section. $\mathcal{B}$ computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{SKE}.k, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}_4^{\mathcal{A}}$ and sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ samples $b \leftarrow \{0,1\}$.

Let $q = q(\lambda)$ be the running time of $\mathcal{A}$. Observe that $q = \mathsf{poly}(\lambda)$ as $\mathcal{A}$ is polytime and that $\mathcal{A}$ outputs at most $q(\lambda)$ function queries on security parameter λ. For $j \in [q]$, $\mathcal{B}$ computes $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j)$ as in $\mathbf{Hybrid}_4^{\mathcal{A}}$. (This does not require knowledge of $\mathsf{FPFE.msk}$ or f_j). $\mathcal{B}$ then sends challenge message pairs $\{((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0), (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))\}_{j\in[q]}$ to its FPFE challenger and receives $\{\mathsf{FPFE.ct}_j\}_{j\in[q]}$ where each $\mathsf{FPFE.ct}_j$ is an encryption of $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0)$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ computes $s_j \leftarrow \{0,1\}^\lambda$, $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j), c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$, and $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_{f_j,s_j,c_j})$, and sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$. (This is possible to compute as q is at least as large as the number of function queries that $\mathcal{A}$ makes).

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ does the following: $\mathcal{B}$ samples $t_i \leftarrow \{0,1\}^\lambda$, sets $v_i = 0^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}}$, sets $r_{i,1} = \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_1, t_i)$, and computes $v_{i,1} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_1, \mathsf{One\text{-}sFE.Enc.st}_1, i, x_i^{(b)}; r_{i,1})$. $\mathcal{B}$ sends challenge function pair $(H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}, H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,1}})$ to its FPFE challenger and receives an FPFE function key $\mathsf{FPFE.sk}_{H_i}$ which is either a function key for $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}$ or a function key for $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,1}}$. This is a valid function query pair since for all $j \in [q]$,

$$\begin{aligned}&H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_i}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0)\\ &= H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,1}}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0)\end{aligned}$$

as the two function act the same when $\beta = 0$. If $i = 1$, $\mathcal{B}$ sets $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, $\mathcal{B}$ sets $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$. $\mathcal{B}$ sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathcal{B}$ received only ciphertexts and function keys for the first message or function of each of its challenge pairs, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}_4^{\mathcal{A}}$, and if $\mathcal{B}$ received only ciphertexts and function keys for the second message or function of each of its challenge pairs, then $\mathcal{B}$ emulates $\mathbf{Hybrid}_{5,1,0}^{\mathcal{A}}$. Additionally, $\mathcal{B}$

does not need to know $\mathsf{FPFE.msk}$ to carry out this experiment. Thus, by Equation 5, this means that $\mathcal{B}$ breaks the function-private selective-security of FPFE as $\mathcal{B}$ can distinguish between the two security games with non-negligible probability. □

Hybrid$^{\mathcal{A}}_{5,k,1}(1^\lambda)$: We change the message encrypted in $\mathsf{FPFE.ct}_k$ so that we use the $\beta = 2$ branch of every $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$. This allows us to remove $\mathsf{One\text{-}sFE.msk}_k$ and $\mathsf{PRF2}.k_k$ from $\mathsf{FPFE.ct}_k$.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (c) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (d) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}\lambda}}, 0^{\ell_{\mathsf{PRF}.k\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Pre-Compute FPFE Ciphertexts**: For $j \in [q]$ where $q = q(\lambda)$ is a bound on the runtime of $\mathcal{A}$,
   (a) $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$.
   (b) $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$.
   (c) $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$.
   (d) If $j < k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$
   (e) If $j = k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (0^{\ell_{\mathsf{One\text{-}sFE.msk}\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}\lambda}}, 0^{\ell_{\mathsf{PRF2}.k\lambda}}, 2))$
   (f) If $j > k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$
5. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
6. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$
         B. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j,s_j,c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. $r_{i,k} = \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, t_i)$
      iii. $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(b)}; r_{i,k})$
      iv. Let $H_i = H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$ as defined in Figure 8.

v. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$

vi. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.

vii. Send CT_i to the adversary.

7. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.8. *If* FPFE *is a function-private-selective-secure* FE *scheme, then for all PPT adversaries* $\mathcal{A}$ *and for all* $k \in \mathbb{N}$*,*

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ and $k \in \mathbb{N}$ such that

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{6}$$

We build a PPT adversary $\mathcal{B}$ that breaks the function-private-selective-security of FPFE. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}$. $\mathcal{B}$ then sends function size $1^{\ell_{H_\lambda}}$, input size $1^{\ell_{\mathsf{FPFE}.m_\lambda}}$, and output size $1^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}}$ to its FPFE challenger where $\ell_{H_\lambda}, \ell_{\mathsf{FPFE}.m_\lambda}, \ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ are computed as described in the parameter section. $\mathcal{B}$ computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{SKE}.k, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}$ and sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ samples $b \leftarrow \{0,1\}$.

Let $q = q(\lambda)$ be the running time of $\mathcal{A}$. Observe that $q = \mathsf{poly}(\lambda)$ as $\mathcal{A}$ is polytime and that $\mathcal{A}$ outputs at most $q(\lambda)$ function queries on security parameter λ. For $j \in [q]$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j)$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}$. (This does not require knowledge of $\mathsf{FPFE.msk}$ or f_j).

- If $j < k$, $\mathcal{B}$ sets its j^{th} challenge message pair to be $((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1), (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$.
- If $j = k$, $\mathcal{B}$ sets its j^{th} challenge message pair to be $((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0), (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$
- If $j > k$, $\mathcal{B}$ sets its j^{th} challenge message pair to be $((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0), (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$

$\mathcal{B}$ then sends all q challenge message pairs to its FPFE challenger and receives $\{\mathsf{FPFE.ct}_j\}_{j\in[q]}$ where either each $\mathsf{FPFE.ct}_j$ is an encryption of the first message of the j^{th} challenge message pair, or each $\mathsf{FPFE.ct}_j$ is an encryption of the second message of the j^{th} challenge message pair.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ computes $s_j \leftarrow \{0,1\}^\lambda$, $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j), c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$, and $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_{f_j,s_j,c_j})$, and sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$. (This is possible to compute as q is at least as large as the number of function queries that $\mathcal{A}$ makes.)

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ does the following: $\mathcal{B}$ samples $t_i \leftarrow \{0,1\}^\lambda$, sets $r_{i,k} = \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, t_i)$, and computes $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(b)}; r_{i,k})$. $\mathcal{B}$ sends challenge function pair $(H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}, H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}})$ to its FPFE challenger and receives a FPFE function key $\mathsf{FPFE.sk}_{H_i}$ which is a function key for

$H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$. This is a valid function query pair since for all $j \in [q]$ and $\beta \in \{0,1\}$, we clearly have,

$$H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, \beta)$$
$$= H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, \beta)$$

and additionally,

$$H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, \mathsf{PRF2}.k_k, 0)$$
$$= H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2)$$

as when $\beta = 2$, the output is $v_{i,k}$ which has been programmed to be equal to $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, \mathsf{PRF2}.k_k, 0)$. If $i = 1$, $\mathcal{B}$ sets $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, $\mathcal{B}$ sets $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$. $\mathcal{B}$ sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathcal{B}$ received only ciphertexts and function keys for the first message or function of each of its challenge pairs, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}$, and if $\mathcal{B}$ received only ciphertexts and function keys for the second message or function of each of its challenge pairs, then $\mathcal{B}$ emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{FPFE.msk}$ to carry out this experiment. Thus, by Equation 6, this means that $\mathcal{B}$ breaks the function-private-selective-security of FPFE as $\mathcal{B}$ can distinguish between the two security games with non-negligible probability. □

Hybrid$^{\mathcal{A}}_{5,k,2}(1^\lambda)$: For each i, instead of sampling $r_{i,k}$ using $\mathsf{PRF2}.k_k$, we sample $r_{i,k}$ uniformly at random.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (c) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (d) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.k_\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. **Pre-Compute FPFE Ciphertexts**: For $j \in [q]$ where $q = q(\lambda)$ is a bound on the runtime of $\mathcal{A}$,
   (a) $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$.
   (b) $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$.
   (c) If $j \neq k$, $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$
   (d) If $j < k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$
   (e) If $j = k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$
   (f) If $j > k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$
6. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$
         B. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j, s_j, c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. ~~$r_{i,k} \leftarrow \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, t_i)$~~
      iii. $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(b)})$
      iv. Let $H_i = H^*_{i, x_i^{(b)}, x_i^{(0)}, t_i, v_{i,k}}$ as defined in Figure 8.
      v. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$

vi. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.

vii. Send CT_i to the adversary.

7. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.9. *If* PRF2 *is a secure PRF, then for all PPT adversaries* $\mathcal{A}$ *and for all* $k \in \mathbb{N}$*,*

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ and $k \in \mathbb{N}$ such that

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{7}$$

We build a PPT adversary $\mathcal{B}$ that breaks the security of PRF2. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}$. $\mathcal{B}$ then sends input size 1^λ, and output size 1^λ to its PRF2 challenger. $\mathcal{B}$ is then given oracle access to either $\mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, \cdot)$ for some $\mathsf{PRF2}.k_k \leftarrow \mathsf{PRF2.Setup}(1^\lambda, 1^\lambda, 1^\lambda)$ or to a uniformly random function $R2 \leftarrow \mathcal{R}2_{\lambda,\lambda}$ where $\mathcal{R}2_{\lambda,\lambda}$ is the set of all functions from $\{0,1\}^\lambda$ to $\{0,1\}^\lambda$. $\mathcal{B}$ computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{FPFE.msk}, \mathsf{SKE}.k, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}$. $\mathcal{B}$ then sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ samples $b \leftarrow \{0,1\}$.

Let $q = q(\lambda)$ be the running time of $\mathcal{A}$. Observe that $q = \mathsf{poly}(\lambda)$ as $\mathcal{A}$ is polytime and that $\mathcal{A}$ outputs at most $q(\lambda)$ function queries on security parameter λ. For $j \in [q]$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j)$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}$. If $j \neq k$, $\mathcal{B}$ also computes $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda, 1^\lambda, 1^\lambda)$.

- If $j < k$, $\mathcal{B}$ computes $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$.
- If $j = k$, $\mathcal{B}$ computes $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$.
- If $j > k$, $\mathcal{B}$ computes $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ computes $s_j \leftarrow \{0,1\}^\lambda$, $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j), c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$, and $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_{f_j,s_j,c_j})$, and sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$. (This is possible to compute as q is at least as large as the number of function queries that $\mathcal{A}$ makes.)

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ does the following: $\mathcal{B}$ samples $t_i \leftarrow \{0,1\}^\lambda$ and sets $r_{i,k}$ equal to the output of its oracle on input t_i. $\mathcal{B}$ computes $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(b)}; r_{i,k})$ and $\mathsf{FPFE.sk}_{H_i} \leftarrow \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}})$. If $i = 1$, $\mathcal{B}$ sets $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, $\mathcal{B}$ sets $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$. $\mathcal{B}$ sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathcal{B}$'s oracle was a uniform random function $R2$, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}$, and if $\mathcal{B}$'s oracle was $\mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, \cdot)$, then $\mathcal{B}$ emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,1}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{PRF2}.k_k$ to carry out this experiment. Thus, by Equation 7, this means that $\mathcal{B}$ breaks the security of PRF2 as $\mathcal{B}$ can distinguish between a random function and PRF2. □

Hybrid$^{\mathcal{A}}_{5,k,3}(1^\lambda)$: We now invoke the security of $\mathsf{One\text{-}sFE}$ to change $v_{i,k}$ from an encryption of $x^{(b)}$ under $\mathsf{One\text{-}sFE.msk}_k$ to an encryption of $x^{(0)}$ under $\mathsf{One\text{-}sFE.msk}_k$.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (c) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (d) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.k_\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. **Pre-Compute FPFE Ciphertexts**: For $j \in [q]$ where $q = q(\lambda)$ is a bound on the runtime of $\mathcal{A}$,
   (a) $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$.
   (b) $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$.
   (c) If $j \neq k$, $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$
   (d) If $j < k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$
   (e) If $j = k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$
   (f) If $j > k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$
6. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$
         B. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j, s_j, c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(0)})$
      iii. Let $H_i = H^*_{i, x_i^{(b)}, x_i^{(0)}, t_i, v_{i,k}}$ as defined in Figure 8.
      iv. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$

v. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.

vi. Send CT_i to the adversary.

7. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.10. *If* One-sFE *is single-key, single-ciphertext, adaptively secure, then for all PPT adversaries* $\mathcal{A}$ *and for all* $k \in \mathbb{N}$*,*

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ and $k \in \mathbb{N}$ such that

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{8}$$

We build a PPT adversary $\mathcal{B}$ that breaks the single-key, single-ciphertext, adaptive-security of One-sFE. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}}$. $\mathcal{B}$ then sends function size $1^{\ell_{\mathcal{F}}}$, state size $1^{\ell_{\mathcal{S}}}$, input size $1^{\ell_{\mathcal{X}}}$, and output size $1^{\ell_{\mathcal{Y}}}$ to its One-sFE challenger. $\mathcal{B}$ computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{FPFE.msk}, \mathsf{SKE}.k, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}$. $\mathcal{B}$ then sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ samples $b \leftarrow \{0,1\}$.

Let $q = q(\lambda)$ be the running time of $\mathcal{A}$. Observe that $q = \mathsf{poly}(\lambda)$ as $\mathcal{A}$ is polytime and that $\mathcal{A}$ outputs at most $q(\lambda)$ function queries on security parameter λ. For $j \in [q]$, $\mathcal{B}$ does the following: If $j \neq k$, $\mathcal{B}$ computes $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$, $\mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$, and $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$.

- If $j < k$, $\mathcal{B}$ computes $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$.
- If $j = k$, $\mathcal{B}$ computes $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$.
- If $j > k$, $\mathcal{B}$ computes $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes $s_j \leftarrow \{0,1\}^\lambda$. If $j \neq k$, $\mathcal{B}$ computes $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$. If $j = k$, $\mathcal{B}$ sends f_k to its One-sFE challenger and receives a function key $\mathsf{One\text{-}sFE.sk}_{f_k}$ in return. $\mathcal{B}$ computes c_j and SK_{f_j} from these values as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}$ and sends SK_{f_j} to $\mathcal{A}$.

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ submitted by $\mathcal{A}$, $\mathcal{B}$ sends challenge message pair $(x_i^{(b)}, x_i^{(0)})$ to its One-sFE challenger, and receives a ciphertext $\mathsf{One\text{-}sFE.ct}_i$ of either $x_i^{(b)}$ or $x_i^{(0)}$. Then, $\mathcal{B}$ samples $t_i \leftarrow \{0,1\}^\lambda$, sets $v_{i,k} = \mathsf{One\text{-}sFE.ct}_i$, and computes $\mathsf{FPFE.sk}_{H_i} \leftarrow \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}})$. If $i = 1$, $\mathcal{B}$ sets $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, $\mathcal{B}$ sets $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$. $\mathcal{B}$ sends CT_i to $\mathcal{A}$.

Observe that these are valid message and function queries to the One-sFE challenger since the adversary is required to have $f_j(x^{(0)}) = f_j(x^{(1)}) = f_j(x^{(b)})$ for all f_j queried by $\mathcal{A}$ and for streams $x^{(0)} = x_1^{(0)}, \ldots, x_t^{(0)}$ and $x^{(1)} = x_1^{(1)}, \ldots, x_t^{(1)}$ where $\{(x_i^{(0)}, x_i^{(1)})\}_{i \in [t]}$ are the messages queried by $\mathcal{A}$ so far. (If not, the adversary is considered aborting, so the hybrid would immediately halt and output 0.)

After $\mathcal{A}$ is done making queries, $\mathcal{B}$ receives b' from $\mathcal{A}$. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathsf{One\text{-}sFE.ct}_i$ is an encryption of $x_i^{(b)}$ for all $i \in [n]$, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}$, and if $\mathsf{One\text{-}sFE.ct}$ is an encryption of $x^{(0)}$, then $\mathcal{B}$ emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{One\text{-}sFE.msk}_k$ to carry out this experiment. Thus, by Equation 8, this means that $\mathcal{B}$ breaks the single-key, single-ciphertext, adaptive-security of $\mathsf{One\text{-}sFE}$, as $\mathcal{B}$ can distinguish between the two ciphertexts with non-negligible probability. □

$\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}(1^\lambda)$: We now reverse the change we made in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,2}$. For each i, instead of sampling $r_{i,k}$ uniformly at random, we sample $r_{i,k}$ using $\mathsf{PRF2}.k_k$.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (c) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (d) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.k_\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. **Pre-Compute FPFE Ciphertexts**: For $j \in [q]$ where $q = q(\lambda)$ is a bound on the runtime of $\mathcal{A}$,
   (a) $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$.
   (b) $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$.
   (c) ~~If $j \neq k$,~~ $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$
   (d) If $j < k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$
   (e) If $j = k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$
   (f) If $j > k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$
6. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$
         B. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j, s_j, c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. $r_{i,k} \leftarrow \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, t_i)$
      iii. $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(0)}; r_{i,k})$
      iv. Let $H_i = H^*_{i, x_i^{(b)}, x_i^{(0)}, t_i, v_{i,k}}$ as defined in Figure 8.
      v. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$

vi. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.

vii. Send CT_i to the adversary.

7. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.11. *If* PRF2 *is a secure PRF, then for all PPT adversaries* $\mathcal{A}$ *and for all* $k \in \mathbb{N}$*,*

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ and $k \in \mathbb{N}$ such that

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{9}$$

We build a PPT adversary $\mathcal{B}$ that breaks the security of PRF2. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}$. $\mathcal{B}$ then sends input size 1^λ, and output size 1^λ to its PRF2 challenger. $\mathcal{B}$ is then given oracle access to either $\mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, \cdot)$ for some $\mathsf{PRF2}.k_k \leftarrow \mathsf{PRF2.Setup}(1^\lambda, 1^\lambda, 1^\lambda)$ or to a uniformly random function $R2 \leftarrow \mathcal{R}2_{\lambda,\lambda}$ where $\mathcal{R}2_{\lambda,\lambda}$ is the set of all functions from $\{0,1\}^\lambda$ to $\{0,1\}^\lambda$. $\mathcal{B}$ computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{FPFE.msk}, \mathsf{SKE}.k, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}$. $\mathcal{B}$ then sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ samples $b \leftarrow \{0,1\}$.

Let $q = q(\lambda)$ be the running time of $\mathcal{A}$. Observe that $q = \mathsf{poly}(\lambda)$ as $\mathcal{A}$ is polytime and that $\mathcal{A}$ outputs at most $q(\lambda)$ function queries on security parameter λ. For $j \in [q]$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{One\text{-}sFE.sk}_{f_j})$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}$. If $j \neq k$, $\mathcal{B}$ computes $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$

- If $j < k$, $\mathcal{B}$ computes
  $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$.
- If $j = k$, $\mathcal{B}$ computes
  $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$.
- If $j > k$, $\mathcal{B}$ computes
  $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ computes $s_j \leftarrow \{0,1\}^\lambda$, $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j), c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$, and $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_{f_j,s_j,c_j})$, and sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$. (This is possible to compute as q is at least as large as the number of function queries that $\mathcal{A}$ makes.)

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ does the following: $\mathcal{B}$ samples $t_i \leftarrow \{0,1\}^\lambda$ and sets $r_{i,k}$ equal to the output of its oracle on input t_i. $\mathcal{B}$ computes $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(0)}; r_{i,k})$ and computes $\mathsf{FPFE.sk}_{H_i} \leftarrow \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}})$. If $i = 1$, $\mathcal{B}$ sets $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, $\mathcal{B}$ sets $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$. $\mathcal{B}$ sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathcal{B}$'s oracle was a uniform random function $R2$, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,3}$, and if $\mathcal{B}$'s oracle was $\mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, \cdot)$, then $\mathcal{B}$ emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{PRF2}.k_k$ to carry out this experiment. Thus, by Equation 9, this means that $\mathcal{B}$ breaks the security of PRF2 as $\mathcal{B}$ can distinguish betwewen a random function and PRF2. □

Hybrid$^{\mathcal{A}}_{5,k,5}(1^\lambda)$: We change the message encrypted in $\mathsf{FPFE.ct}_k$ so that it uses the $\beta = 1$ branch of every $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (c) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (d) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.k_\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. **Pre-Compute FPFE Ciphertexts**: For $j \in [q]$ where $q = q(\lambda)$ is a bound on the runtime of $\mathcal{A}$,
   (a) $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$.
   (b) $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$.
   (c) $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$
   (d) If $j < k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$
   (e) If $j = k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$
   (f) If $j > k$, $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$
6. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$
         B. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j, s_j, c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. $r_{i,k} \leftarrow \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, t_i)$
      iii. $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(0)}; r_{i,k})$
      iv. Let $H_i = H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$ as defined in Figure 8.

v. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$

vi. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.

vii. Send CT_i to the adversary.

7. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.12. *If* FPFE *is function-private-selective-secure, then for all PPT adversaries* $\mathcal{A}$ *and for all* $k \in \mathbb{N}$*,*

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,5}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ and $k \in \mathbb{N}$ such that

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,5}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{10}$$

We build a PPT adversary $\mathcal{B}$ that breaks the function-private-selective-security of FPFE. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_\mathcal{F}}, 1^{\ell_\mathcal{S}}, 1^{\ell_\mathcal{X}}, 1^{\ell_\mathcal{Y}}$. $\mathcal{B}$ then sends function size $1^{\ell_{H_\lambda}}$, input size $1^{\ell_{\mathsf{FPFE}.m_\lambda}}$, and output size $1^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}}$ to its FPFE challenger where $\ell_{H_\lambda}, \ell_{\mathsf{FPFE}.m_\lambda}, \ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ are computed as described in the parameter section. $\mathcal{B}$ computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{SKE}.k, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}$ and sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ samples $b \leftarrow \{0,1\}$.

Let $q = q(\lambda)$ be the running time of $\mathcal{A}$. Observe that $q = \mathsf{poly}(\lambda)$ as $\mathcal{A}$ is polytime and that $\mathcal{A}$ outputs at most $q(\lambda)$ function queries on security parameter λ. For $j \in [q]$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j)$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}$. (This does not require knowledge of $\mathsf{FPFE.msk}$ or f_j).

- If $j < k$, $\mathcal{B}$ sets its j^{th} challenge message pair to be $((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1), (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$.
- If $j = k$, $\mathcal{B}$ sets its j^{th} challenge message pair to be $((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1), (0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2))$
- If $j > k$, $\mathcal{B}$ sets its j^{th} challenge message pair to be $((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0), (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$

$\mathcal{B}$ then sends all q challenge message pairs to its FPFE challenger and receives $\{\mathsf{FPFE.ct}_j\}_{j\in[q]}$ where either each $\mathsf{FPFE.ct}_j$ is an encryption of the first message of the j^{th} challenge message pair, or each $\mathsf{FPFE.ct}_j$ is an encryption of the second message of the j^{th} challenge message pair.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ computes $s_j \leftarrow \{0,1\}^\lambda$, $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j), c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$, and $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_{f_j,s_j,c_j})$, and sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$. (This is possible to compute as q is at least as large as the number of function queries that $\mathcal{A}$ makes.)

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ does the following: $\mathcal{B}$ samples $t_i \leftarrow \{0,1\}^\lambda$, sets $r_{i,k} = \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, t_i)$, and computes $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(0)}; r_{i,k})$. $\mathcal{B}$ sends a challenge function pair $(H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}, H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}})$ to its FPFE challenger and receives an FPFE function key $\mathsf{FPFE.sk}_{H_i}$ which is a function key for

$H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$. This is a valid function query pair since for all $j \in [q]$ and $\beta \in \{0,1\}$, we clearly have

$$H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, \beta)$$
$$= H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, \beta)$$

and additionally

$$H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, \mathsf{PRF2}.k_k, 1)$$
$$= H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(0^{\ell_{\mathsf{One\text{-}sFE.msk}_\lambda}}, 0^{\ell_{\mathsf{One\text{-}sFE.Enc.st}_\lambda}}, 0^{\ell_{\mathsf{PRF2}.k_\lambda}}, 2)$$

as when $\beta = 2$, the output is $v_{i,k}$ which has been programmed to be equal to $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, \mathsf{PRF2}.k_k, 1)$. If $i = 1$, $\mathcal{B}$ sets $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, $\mathcal{B}$ sets $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$. $\mathcal{B}$ sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathcal{B}$ received only ciphertexts and function keys for the first message or function of each of its challenge pairs, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,5}$, and if $\mathcal{B}$ received only ciphertexts and function keys for the second message or function of each of its challenge pairs, then $\mathcal{B}$ emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,4}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{FPFE.msk}$ to carry out this experiment. Thus, by Equation 10, this means that $\mathcal{B}$ breaks the function-private-selective-security of FPFE as $\mathcal{B}$ can distinguish between the two security games with non-negligible probability. □

Lemma D.13. *If* FPFE *is a function-private-selective-secure* FE *scheme, then for all PPT adversaries* $\mathcal{A}$ *and for all* $k \in \mathbb{N}\backslash\{1\}$*,*

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k-1,5}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

Proof. Suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ and a $k \in \mathbb{N}\backslash\{1\}$ such that

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k-1,5}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{11}$$

We build a PPT adversary $\mathcal{B}$ that breaks the function-private-selective-security of FPFE. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}}$. $\mathcal{B}$ then sends function size $1^{\ell_{H_\lambda}}$, input size $1^{\ell_{\mathsf{FPFE}.m_\lambda}}$, and output size $1^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}}$ to its FPFE challenger where $\ell_{H_\lambda}, \ell_{\mathsf{FPFE}.m_\lambda}, \ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ are computed as described in the parameter section. $\mathcal{B}$ computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{SKE}.k, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k-1,5}$ and sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ samples $b \leftarrow \{0,1\}$.

Let $q = q(\lambda)$ be the running time of $\mathcal{A}$. Observe that $q = \mathsf{poly}(\lambda)$ as $\mathcal{A}$ is polytime and that $\mathcal{A}$ outputs at most $q(\lambda)$ function queries on security parameter λ. For $j \in [q]$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j)$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,k-1,5}$. (This does not require knowledge of $\mathsf{FPFE.msk}$ or f_j).

- If $j < k$, $\mathcal{B}$ sets its j^{th} challenge message pair to be $((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1), (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$.
- If $j \geq k$, $\mathcal{B}$ sets its j^{th} challenge message pair to be $((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0), (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0))$.

$\mathcal{B}$ then sends all q challenge message pairs to its FPFE challenger and receives $\{\mathsf{FPFE.ct}_j\}_{j\in[q]}$ where for $j < k$, $\mathsf{FPFE.ct}_j$ is an encryption of $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1)$, and for $j \geq k$, $\mathsf{FPFE.ct}_j$ is an encryption of $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 0)$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ samples $s_j \leftarrow \{0,1\}^\lambda$, computes $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j), c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$, and $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_{f_j,s_j,c_j})$, and sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$. (This is possible to compute as q is at least as large as the number of function queries that $\mathcal{A}$ makes.)

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes $t_i \leftarrow \{0,1\}^\lambda$, $r_{i,k-1} = \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_{k-1}, t_i)$, $v_{i,k-1} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_{k-1}, \mathsf{One\text{-}sFE.Enc.st}_{k-1}, i, x_i^{(0)}; r_{i,k-1})$, $r_{i,k} = \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_k, t_i)$, and $v_{i,k} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_k, \mathsf{One\text{-}sFE.Enc.st}_k, i, x_i^{(b)}; r_{i,k})$. $\mathcal{B}$ sends a challenge function pair $(H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k-1}}, H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}})$ to its FPFE challenger and receives an FPFE function key $\mathsf{FPFE.sk}_{H_i}$ which is either a function key for $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k-1}}$ or a function key for $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$. This is a valid function query pair since for all $j \in [q]$ and $\beta \in \{0,1\}$,

$$\begin{aligned}&H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k-1}}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, \beta)\\ &= H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, \beta)\end{aligned}$$

as the two functions act the same when $\beta = 0$ or $\beta = 1$. If $i = 1$, $\mathcal{B}$ sets $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, $\mathcal{B}$ sets $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$. $\mathcal{B}$ sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathcal{B}$ received only ciphertexts and function keys for the first message or function of each of its challenge pairs, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k-1,5}$, and if $\mathcal{B}$ received only ciphertexts and function keys for the second message or function of each of its challenge pairs, then $\mathcal{B}$ emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,k,0}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{FPFE.msk}$ to carry out this experiment. Thus, by Equation 11, this means that $\mathcal{B}$ breaks the function-private selective-security of FPFE as $\mathcal{B}$ can distinguish between the two security games with non-negligible probability. □

Hybrid$_6^{\mathcal{A}}(1^\lambda)$: We replace each $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,k}}$ with a function $H^*_{i,x_i^{(0)},x_i^{(0)},t_i,v_i}$ which is independent of b.

1. **Parameters**: The adversary $\mathcal{A}$ receives security parameter 1^λ, and outputs a function size $1^{\ell_\mathcal{F}}$, a state size $1^{\ell_\mathcal{S}}$, an input size $1^{\ell_\mathcal{X}}$, and an output size $1^{\ell_\mathcal{Y}}$.
2. **Setup:**
   (a) $(\mathsf{FE.mpk}, \mathsf{FE.msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$
   (b) $\mathsf{FPFE.msk} \leftarrow \mathsf{FPFE.Setup}(1^\lambda)$
   (c) $\mathsf{SKE}.k \leftarrow \mathsf{SKE.Setup}(1^\lambda)$
   (d) $\mathsf{FE.ct} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.mpk}, (0^{\ell_{\mathsf{FPFE.msk}_\lambda}}, 0^{\ell_{\mathsf{PRF}.k_\lambda}}, 1, \mathsf{SKE}.k))$
3. **Public Key:** Send $\mathsf{MPK} = \mathsf{FE.mpk}$ to the adversary.
4. **Challenge Bit**: Sample $b \leftarrow \{0,1\}$.
5. **Pre-Compute FPFE Ciphertexts**: For $j \in [q]$ where $q = q(\lambda)$ is a bound on the runtime of $\mathcal{A}$,
   (a) $\mathsf{One\text{-}sFE.msk}_j \leftarrow \mathsf{One\text{-}sFE.Setup}(1^\lambda)$.
   (b) $\mathsf{One\text{-}sFE.Enc.st}_j \leftarrow \mathsf{One\text{-}sFE.EncSetup}(\mathsf{One\text{-}sFE.msk}_j)$.
   (c) $\mathsf{PRF2}.k_j \leftarrow \mathsf{PRF2.Setup}(1^\lambda)$
   (d) $\mathsf{FPFE.ct}_j \leftarrow \mathsf{FPFE.Enc}(\mathsf{FPFE.msk}, (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))$
6. For a polynomial number of rounds, the adversary can do either one of the following in each round:
   (a) **Function Query:** For the j^{th} function query $f_j \in \mathcal{F}[\ell_\mathcal{F}, \ell_\mathcal{S}, \ell_\mathcal{X}, \ell_\mathcal{Y}]$ made by the adversary:
      i. $s_j \leftarrow \{0,1\}^\lambda$
      ii. **Compute c_j:**
         A. $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j)$
         B. $c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$
      iii. Let $G_j = G_{f_j,s_j,c_j}$ as defined in Figure 6 (page 86).
      iv. $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_j)$
      v. Send $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to the adversary.
   (b) **Challenge Message Query:** For the i^{th} challenge message query, $\mathcal{A}$ outputs a challenge message pair $(x_i^{(0)}, x_i^{(1)})$ where $x_i^{(0)}, x_i^{(1)} \in \{0,1\}^{\ell_\mathcal{X}}$.
      i. $t_i \leftarrow \{0,1\}^\lambda$
      ii. $v_i = 0^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}}$
      iii. Let $H_i = H^*_{i,x_i^{(0)},x_i^{(0)},t_i,v_i}$ as defined in Figure 8.
      iv. $\mathsf{FPFE.sk}_{H_i} = \mathsf{FPFE.KeyGen}(\mathsf{FPFE.msk}, H_i)$
      v. If $i = 1$, let $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, let $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$.
      vi. Send CT_i to the adversary.

7. **Experiment Outcome**: $\mathcal{A}$ outputs a bit b'. The output of the experiment is set to 1 if $b = b'$, and 0 otherwise.

Lemma D.14. *If* FPFE *is a function-private-selective-secure* FE *scheme, then for all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,q,5}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{6}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

where $q = q(\lambda)$ *is the runtime of* $\mathcal{A}$ *on security parameter* λ*.*

Proof. First, observe that if $q(\lambda)$ is the runtime of $\mathcal{A}$, then $\mathcal{A}$ outputs at most $q(\lambda)$ function queries on security parameter λ. Thus, $\mathbf{Hybrid}^{\mathcal{A}}_{5,q,5}$ always uses the $\beta = 1$ branch when encrypting $\mathsf{FPFE.ct}_j$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{6}$. Now, suppose for sake of contradiction that there exists a PPT adversary $\mathcal{A}$ such that

$$\left|\Pr[\mathbf{Hybrid}^{\mathcal{A}}_{5,q,5}(1^\lambda) = 1] - \Pr[\mathbf{Hybrid}^{\mathcal{A}}_{6}(1^\lambda) = 1]\right| > \mathsf{negl}(\lambda) \tag{12}$$

We build a PPT adversary $\mathcal{B}$ that breaks the function-private-selective-security of FPFE. $\mathcal{B}$ first runs $\mathcal{A}$ on input 1^λ and receives parameters $1^{\ell_{\mathcal{F}}}, 1^{\ell_{\mathcal{S}}}, 1^{\ell_{\mathcal{X}}}, 1^{\ell_{\mathcal{Y}}}$. $\mathcal{B}$ then sends function size $1^{\ell_{H_\lambda}}$, input size $1^{\ell_{\mathsf{FPFE}.m_\lambda}}$, and output size $1^{\ell_{\mathsf{One\text{-}sFE.ct}_\lambda}}$ to its FPFE challenger where $\ell_{H_\lambda}, \ell_{\mathsf{FPFE}.m_\lambda}, \ell_{\mathsf{One\text{-}sFE.ct}_\lambda}$ are computed as described in the parameter section. $\mathcal{B}$ computes $(\mathsf{FE.mpk}, \mathsf{FE.msk}, \mathsf{SKE}.k, \mathsf{FE.ct})$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,q,5}$ and sends $\mathsf{MPK} = \mathsf{FE.mpk}$ to $\mathcal{A}$. $\mathcal{B}$ samples $b \leftarrow \{0,1\}$.

For $j \in [q]$, $\mathcal{B}$ computes $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j)$ as in $\mathbf{Hybrid}^{\mathcal{A}}_{5,q,5}$. (This does not require knowledge of $\mathsf{FPFE.msk}$ or f_j). $\mathcal{B}$ then sends challenge message pairs $\{((\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1), (\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1))\}_{j\in[q]}$ to its FPFE challenger and receives $\{\mathsf{FPFE.ct}_j\}_{j\in[q]}$ where each $\mathsf{FPFE.ct}_j$ is an encryption of $(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1)$.

For each function query f_j that $\mathcal{A}$ sends to $\mathcal{B}$, $\mathcal{B}$ samples $s_j \leftarrow \{0,1\}^\lambda$, computes $\mathsf{One\text{-}sFE.sk}_{f_j} \leftarrow \mathsf{One\text{-}sFE.KeyGen}(\mathsf{One\text{-}sFE.msk}_j, f_j), c_j \leftarrow \mathsf{SKE.Enc}(\mathsf{SKE}.k, (\mathsf{One\text{-}sFE.sk}_{f_j}, \mathsf{FPFE.ct}_j))$, and $\mathsf{FE.sk}_{G_j} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.msk}, G_{f_j,s_j,c_j})$, and sends $\mathsf{SK}_{f_j} = \mathsf{FE.sk}_{G_j}$ to $\mathcal{A}$. (This is possible to compute as q is at least as large as the number of function queries that $\mathcal{A}$ makes.)

For each challenge message query $(x_i^{(0)}, x_i^{(1)})$ output by $\mathcal{A}$, $\mathcal{B}$ does the following: $\mathcal{B}$ computes $t_i \leftarrow \{0,1\}^\lambda$, $v_i = 0^{\ell_{\mathsf{One\text{-}sFE.ct}\lambda}}$, $r_{i,q} \leftarrow \mathsf{PRF2.Eval}(\mathsf{PRF2}.k_q, t_i)$, and $v_{i,q} \leftarrow \mathsf{One\text{-}sFE.Enc}(\mathsf{One\text{-}sFE.msk}_q, \mathsf{One\text{-}sFE.Enc.st}_q, i, x_i^{(0)}; r_{i,q})$. $\mathcal{B}$ sends challenge function pair $(H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,q}}, H^*_{i,x_i^{(0)},x_i^{(0)},t_i,v_i})$ to its FPFE challenger and receives an FPFE function key $\mathsf{FPFE.sk}_{H_i}$ which is either a function key for $H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,q}}$ or a function key for $H^*_{i,x_i^{(0)},x_i^{(0)},t_i,v_i}$. This is a valid function query pair since for all $j \in [q]$,

$$\begin{aligned}&H^*_{i,x_i^{(b)},x_i^{(0)},t_i,v_{i,q}}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1)\\ &= H^*_{i,x_i^{(0)},x_i^{(0)},t_i,v_i}(\mathsf{One\text{-}sFE.msk}_j, \mathsf{One\text{-}sFE.Enc.st}_j, \mathsf{PRF2}.k_j, 1)\end{aligned}$$

as the two function act the same when $\beta = 1$. If $i = 1$, $\mathcal{B}$ sets $\mathsf{CT}_1 = (\mathsf{FE.ct}, \mathsf{FPFE.sk}_{H_1})$. Else, $\mathcal{B}$ sets $\mathsf{CT}_i = \mathsf{FPFE.sk}_{H_i}$. $\mathcal{B}$ sends CT_i to $\mathcal{A}$.

After $\mathcal{A}$ is done making queries, $\mathcal{A}$ outputs b'. $\mathcal{B}$ outputs 1 if $b = b'$, and outputs 0 otherwise. If the experiment for $\mathcal{A}$ aborts for any reason, $\mathcal{B}$ always outputs 0. Observe that if $\mathcal{B}$ received only ciphertexts and function keys for the first message or function of each of its challenge pairs, then $\mathcal{B}$ exactly emulates $\mathbf{Hybrid}^{\mathcal{A}}_{5,q,5}$, and if $\mathcal{B}$ received only ciphertexts and function keys for the second message or function of each of its challenge pairs, then $\mathcal{B}$ emulates $\mathbf{Hybrid}^{\mathcal{A}}_{6}$. Additionally, $\mathcal{B}$ does not need to know $\mathsf{FPFE.msk}$ to carry out this experiment. Thus, by Equation 12, this means

that $\mathcal{B}$ breaks the function-private-selective-security of FPFE as $\mathcal{B}$ can distinguish between the two security games with non-negligible probability. □

Lemma D.15. *For all adversaries $\mathcal{A}$,*

$$\Pr[\mathbf{Hybrid}_6^{\mathcal{A}}(1^\lambda) = 1] \leq \frac{1}{2}$$

Proof. The messages sent to $\mathcal{A}$ in $\mathbf{Hybrid}_6^{\mathcal{A}}$ are independent of b. Thus, the probability that $\mathcal{A}$ correctly guesses b in $\mathbf{Hybrid}_6^{\mathcal{A}}$ is $\frac{1}{2}$. The lemma then follows since the probability that $\mathbf{Hybrid}_6^{\mathcal{A}}$ outputs 1 is at most the probability that $\mathcal{A}$ correctly guesses b. □

Thus, our lemmas give us the following corollary:

Corollary D.16. *If*

- PRF *and* $\mathsf{PRF2}$ *are secure PRFs,*
- SKE *is a secure symmetric key encryption scheme with pseudorandom ciphertexts,*
- $\mathsf{One\text{-}sFE}$ *is single-key, single-ciphertext, adaptively secure,*
- FPFE *is function-private-selective-secure,*
- *and* FE *is selective-secure,*

then sFE *is adaptively secure.*

Proof. By combining the hybrid indistinguishability lemmas above, we get that for all PPT adversaries $\mathcal{A}$,

$$\left|\Pr[\mathsf{ExptGuess}_{\mathcal{A}}^{\mathsf{sFE\text{-}Adaptive}}(1^\lambda) = 1]\right| = \left|\Pr[\mathbf{Hybrid}_1^{\mathcal{A}}(1^\lambda) = 1]\right| \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

The corollary then follows immediately. □

Corollary D.16 then implies Theorem 7.1 (page 82) , since as shown earlier, we can instantiate the required primitives from a selectively secure, public-key FE scheme for $\mathsf{P/Poly}$ and a single-key, single-ciphertext, adaptively secure, secret-key, sFE scheme for $\mathsf{P/Poly}$.