CASE: A New Frontier in Public-Key Authenticated Encryption

Shashank Agrawal[1], Shweta Agrawal[2], Manoj Prabhakaran[3], Rajeev Raghunath[3], and Jayesh Singla[3]

[1] Coinbase
sagrawal@protonmail.ch
[2] IIT Madras, Chennai, India
shweta.a@gmail.com
[3] IIT Bombay, Mumbai, India
{mp,mrrajeev,jayeshs}@cse.iitb.ac.in

Abstract. We introduce a new cryptographic primitive, called Completely Anonymous Signed Encryption (CASE). CASE is a public-key authenticated encryption primitive, that offers anonymity for senders as well as receivers. A "case-packet" should appear, without a (decryption) key for opening it, to be a blackbox that reveals no information at all about its contents. To *decase* a case-packet fully – so that the message is retrieved and authenticated – a verification key is also required.

Defining security for this primitive is subtle. We present a relatively simple *Chosen Objects Attack* (COA) security definition. Validating this definition, we show that it implies a comprehensive indistinguishability-preservation definition in the real-ideal paradigm. To obtain the latter definition, we extend the Cryptographic Agents framework of [2, 3] to allow maliciously created objects.

We also provide a novel and practical construction for COA-secure CASE under standard assumptions in public-key cryptography, and in the standard model.

We believe CASE can be a staple in future cryptographic libraries, thanks to its robust security guarantees and efficient instantiations based on standard assumptions.

1 Introduction

In this work, we introduce a new cryptographic primitive, called Completely Anonymous Signed Encryption (CASE). CASE is a public-key authenticated encryption primitive, that offers anonymity for senders as well as receivers. CASE captures the intuition that once a message is "encased" – resulting in a case-packet – it should appear, to someone without a (decryption) key for opening the case-packet, to be a blackbox that reveals no information at all about its contents.[4] To *decase* a case-packet fully – so that the message is retrieved and authenticated – a verification key is also required.

The significance of such a primitive stems from its *fundamental nature* as well as its potential as a *practical tool*. For instance, in blockchain-like systems where data packets can be publicly posted, for privacy, not only the contents of the packet should be hidden, but also the originator and the intended recipient of the data should remain anonymous. Further, we may require that even the recipient of a packet should not learn about its sender unless they have acquired a verification key that allows them to authenticate packets from the sender (this is what we call *complete* anonymity).

CASE, while fundamental in nature, is still a fairly complex primitive, and formally defining security for it is a non-trivial task. It involves two pairs of keys (public and secret keys, for encryption and signature), used in different combinations (e.g., a decryption key is enough to open the case-packet for reading a message, but a verification key is also needed for authentication), and multiple security requirements based on which keys are available to the adversary and which are not.

Public-key authenticated encryption has been well-explored in the literature (see Section 1.1) and has also been making its way into standards (e.g., [4, 11]). However, these notions do not incorporate anonymity as we do here. Further, we seek and achieve *significantly more comprehensive security guarantees and strong key-hiding properties*. In particular, we seek security against active adversaries who can access oracles that

[4] For simplicity, we consider a finite message space. If messages of arbitrary length are to be allowed, we will let a case-packet reveal the length of the message (possibly after padding). All our definitions and results can be readily generalized to this setting.

combine honest objects with adversarial objects, where "objects" refer to both keys as well as case-packets. For instance, the adversary can query a decasing oracle with its own decryption key and case-packet, but requesting to use one of two verification keys picked by the experiment. We term such attacks **Chosen Objects Attack** (COA), as a generalization of Chosen Ciphertext Attack. We present a relatively simple definition of COA-secure CASE consisting of three elegant experiments (Total-Hiding, Sender-Anonymity, Unforgeability),[5] correctness conditions, an unpredictability condition, and a set of natural – but new – *existential consistency* requirements.

Is COA Security Comprehensive? (Yes!) At first glance, our COA security definition for CASE may appear as an incomplete list of desirable properties. Indeed, given the subtleties of defining security for a complex primitive, it is not possible to appeal to intuition to argue that all vulnerabilities have been covered by this definition. Instead, one should use a comprehensive definition in the *real-ideal* paradigm, where the ideal model is intuitively convincing. This approach has formed the foundation for general frameworks like Universally Composable security [15] and Constructive Cryptography [31]. However, using a simulation based security definition for modeling *objects that can be passed around* (rather than functionalities implemented using protocols wherein parties never transfer their secret keys) quickly leads to impossibility results in the standard model without random oracles (see Section 6.4). To avoid such outright impossibility results, we consider a definition in the real-ideal paradigm that uses **indistinguishability-preservation** [2, 3] as the security notion, rather than simulation. In the process, we extend the Cryptographic Agents framework of [2, 3] to allow maliciously created objects, which is an *important additional contribution of this work*.

Once the definitions are in place, our main results are a novel construction of a COA-secure CASE from standard assumptions in public-key cryptography, and also showing that COA-secure CASE meets the real-ideal security definition for CASE.

Our Contributions. We summarize our contributions here.

- We introduce CASE as a practical and powerful cryptographic primitive.
- We present a strong security definition for CASE, called COA security (Section 4).
- We give a construction for COA-secure CASE under standard assumptions in the standard model (Section 5). We also show how to leverage the efficiency of any symmetric-key encryption scheme to get a correspondingly efficient COA-secure CASE (Section 5.4).
- We present the Active Agents Framework as an extension of the Cryptographic Agents model, to capture comprehensive security guarantees for complex primitives like CASE under the real-ideal paradigm (Section 6).
- We show that COA secure CASE yields a secure implementation of CASE in the active agents framework (Section 7).

While we present the COA security definition upfront, it is important to point out that this definition was arrived at starting from the security definition in the active agents framework, and working through the demands of satisfying that definition.

1.1 Related Work

Public-key authenticated encryption has been extensively studied since signcryption was introduced by Zheng [43]. Despite being a fundamental primitive studied for over two decades, it has proved challenging to find the right definitions of security for this notion. Indeed, the original scheme by Zheng was proven secure several years after its introduction [8]. A sequence of works [5, 6, 8, 36, 42] formalized security in the so called "outsider security model" and "insider security model" where the former is used to model network attacks while the latter is used to model (a priori) legitimate users whose keys have been compromised. Even as these basic security definitions remained ad hoc, a significant number of works have constructed concrete schemes based on different assumptions [27, 28, 40, 43, 44], and gone on to realize advanced properties [9, 13, 14, 18, 19, 20, 23, 27, 29, 30, 38, 39, 41].

[5] These distinct experiments can be combined to give an equivalent unified experiment in which the adversary is allowed to adaptively attack any of the above security properties over a collection of keys and case-packets. Such a definition is presented as an intermediate step to showing the comprehensiveness of this definition (see below).

An early attempt by Gjøsteen and Kråkmo [24] modelled unforgeability and confidentiality in the outsider security model by using an ideal functionality. More recently, [7] provided a constructive cryptography perspective of the basic security notions of signcryption. This work modelled the goal of authenticated public-key encryption as a secure communication network, with static corruption of nodes. As it used a simulation-based definition for the communication functionality, it does not account (and could not have accounted) for secret key transfers, or more generally, the use of the scheme's objects in non-standard ways outside of the prescribed communication protocols (e.g., posting ciphertexts on a bulletin board or forwarding them, using signatures to prove the possession of a signing key, etc.).

Recently, Bellare and Stepanovs studied signcryption from a quantitative perspective due to its use in various practical systems and standards [11]. More recently, Alwen et al. [4] conducted a thorough study of the "authenticated mode" of the Hybrid Public Key Encryption (HPKE) standard, which combines a Key Encapsulation Mechanism and an Authenticated Encryption. They abstract this notion using a new primitive which they call Authenticated Public Key Encryption. However, their study is tailored to the HPKE standard, and primarily studies weaker variants of security. Another recent work by Maurer et al. [32] studied the related notion of "Multi-Designated Receiver Signed Public Key Encryption" which allows a sender to select a set of designated receivers and both encrypt and sign a message that only these receivers will be able to read and authenticate.

While the aforementioned works make important progress towards the goal of finding the right formalization for public-key authenticated encryption, *none of them consider anonymity* of the sender and intended receiver. They also work with *relatively weak or ad hoc security definitions* and do not comprehensively model an adversary that can combine honest and adversarial objects via oracles.

2 Technical Overview

We proceed to provide a technical overview of our definitions, constructions and proofs of security.

2.1 Defining COA-Secure CASE

CASE is a fairly complex primitive. For instance, in contrast to symmetric-key authenticated encryption, encasing and decasing a message involves four keys. Further, in comparison to signcryption, which itself has been the subject of an extensive body of work, CASE requires strong *key-hiding* properties. We also require that even if one of the two keys used to create a case-packet, or used to decase a possibly malicious case-packet, is maliciously crafted, the residual hiding assurances for the honestly created key should hold.

We start off by presenting a fairly intuitive set of security games and correctness properties. We term our definition security against *Chosen Objects Attack*, or **COA-security** (Section 4), since the adversary needs to be provided with oracles which take not only malicious "ciphertexts" (or case-packets), but also malicious keys; both encasing and decasing oracles need to be provided to the adversary. There are standard correctness requirements and three security games – **total hiding** and **sender anonymity** games with a flavor of CCA security, and an **unforgeability** game paralleling a standard signature unforgeability requirement. In addition, there is an **unpredictability** requirement and a set of **existential consistency** requirements, which are crucial for security against malicious keys. The former requires that encasing a message with any encryption key and signing key results in a case-packet with high min-entropy (or results in an error); while this is implied by the above security experiments for honestly generated keys, the additional requirement is that it holds for *all* keys in the key-space. The existential consistency conditions require that a case-packet should have at most one set of keys and message that can be associated with it, and similarly a verification key should have at most one signing key, and an encryption key should have at most one decryption key. Like the unpredictability requirement, the consistency requirements are also remarkably unremarkable in nature – indeed, one may feel that they are to be expected in any reasonable scheme – but, they are non-trivial to enforce.

2.2 Constructing a COA-Secure CASE

We start with a sign-then-encrypt strategy. Indeed, in the setting of (non-anonymous) signcryption, sign-then-encrypt is a generic composition that is known to yield a secure signcryption [6], but only with the weakened form of "replayable CCA" security (introduced in [6] as *generalized CCA* or gCCA). The main drawback of this construction is replayability: suppose Eve receives a case-packet CP signed by Alice and encrypted using Eve's encryption key; then, Eve can decrypt it and reencrypt using any encryption key of its choice (without needing to modify the underlying signature of Alice). This is clearly problematic because, if Bob receives a case-packet that he can decase and authenticate to be from Alice, he still cannot be sure if Alice had actually sent it to him, or to someone like Eve (who then carried out the above attack). An immediate solution to this is to include in the signed message the encryption key to be used as well; this would prevent Eve from passing off the signed message with her encryption key in it as a message intended for Bob. However, this still leaves some non-ideal behavior: On receiving one case-packet from Alice, Eve can construct many *distinct* case-packets by decrypting and reencrypting it with its encryption key many times. Each of these case-packets would verify as coming from Alice by someone with Eve's decryption key. Whether this translates to concrete harm or not is application dependent – but this a behavior that is not possible in the ideal setting.

We thus want to authenticate the entire case-packet (rather than just the message and the encryption key) in the signature. However, this leads to a circularity as the case-packet is determined only after the signature is computed. It turns out that one can circumvent this circularity by exposing a little more structure from the underlying PKE scheme. The idea is as follows, instead of signing the case-packet itself, it is enough to sign everything that goes into the case-packet other than the signature itself – i.e., the message, the encryption key, and the *randomness that will be used to create the encryption.* This idea should be implemented with some care, so that the security of the encryption scheme (which is not designed to support message-dependent-randomness) remains un-affected.

We call an encryption scheme **quasi-deterministic** if any ciphertext generated by it includes a part τ that is independent of the message, but is a perfectly binding encoding of all the randomness r used in the encryption. As a simple example, El Gamal encryption is quasi-deterministic, since $\mathrm{Enc}_{\mathrm{ElGamal}}((g,h),m;r) = (g^r, m \cdot h^r)$ where (g,h) is the public-key, m the message and r the randomness, and g^r is a binding encoding of r. The same is true for Cramer-Shoup encryption [17].

This gives us the structure of our final scheme: we need a signature scheme (with sufficiently short signatures) and a quasi-deterministic PKE scheme (with sufficiently long messages). To encase m, we first pick the randomness r for the PKE scheme and compute the first component τ of the ciphertext (without needing the message). Then, we set the case-packet to be $\mathsf{pkeEnc}(EK, m||\sigma; r)$ where $\sigma = \mathsf{sigSign}(SK, m||EK||\tau)$. Note that, the ciphertext produced by pkeEnc using randomness r will contain τ as a part, and during decasing, the signature σ can be verified.

To make this construction work, we need the right kind of PKE and signature schemes, with their own anonymity and existential consistency in addition to the standard security guarantees (CCA and strong unforgeability, resp.). We capture these security requirements as *COA-secure Quasi-Deterministic PKE* (COA-QD-PKE) and *Existentially Consistent Anonymous Signatures* (ECAS).

COA Secure Quasi-Deterministic PKE The definition of COA security of PKE consists of a single indistinguishability requirement – Anonymous-CCA-QD security (adapted from Anonymous-CCA security [1, 12]) – plus a set of existential consistency requirements.

To be able to exploit the quasi-determinism (described above), we need to modify the CCA security game slightly into a CCA-QD game as follows. The adversary receives the first part τ of the challenge ciphertext (which does not depend on the message) upfront along with the public-key; it receives the rest of the ciphertext after it submits a pair of challenge messages.

To construct a COA-QD-PKE scheme, we start from an Anonymous-CCA-QD secure scheme. As it turns out, we already have a construction in the literature that is Anonymous-CCA-QD secure: [1] showed that with a slight modification, the Cramer-Shoup encryption scheme [17] becomes Anonymous-CCA secure; we reanalyze this scheme to show that it is Anonymous-CCA-QD secure as well.[6]

[6] We note that, CCA-QD security is not implied by CCA security and the QD structure alone. E.g., one can modify a CCA-QD secure PKE scheme such that, if the encoding of the randomness (the pre-computed component of the ciphertext) happens to equal the message, it simply sets the second component to $\perp$, thereby revealing the message; while this remains CCA secure, an adversary in the CCA-QD game can set one of the challenge messages to be equal to the encoding of the randomness and break CCA-QD security.

We also require existential consistency s.t. if a ciphertext decrypts successfully, it can only decrypt to at most a single message with at most a single decryption key. We now show how a given Anonymous-CCA-QD-PKE with perfect correctness (such as the modified Cramer-Shoup scheme [1]) can be modified to be existentially consistent while retaining its original security. Note that, perfect correctness only refers to honestly generated keys and ciphertexts, and does not entail existential consistency.

A helpful first step in preventing invalid secret-keys is to redefine it to be the randomness used to generate the original secret-key. Further towards enforcing existential consistency, we augment the public-key to include a perfectly binding commitment to the secret-key, and the ciphertext is augmented to include one to the public-key. That is, the ciphertext has the form (α, β), where α is a commitment to the public-key and β is a ciphertext in the original scheme. To preserve anonymous-CCA security, we need to tie α and β together: it turns out to be enough to let β be the encryption of $m||d$ where d is the canonical decommitment information for α (from which α also can be computed).

Here we point out one subtlety in the above construction. Note that the public-key is required to include a binding commitment of the secret-key. But we in fact require that the public-key can be *deterministically* computed from the secret-key (since this property will be required of our CASE scheme). Hence the randomness needed to compute this commitment must already be part of the secret-key, leading to a circularity. This circularity can be avoided by using a commitment scheme that is "fully binding" – i.e., the output of the commitment is perfectly binding not only for the message, but also for the randomness used. An example of such a scheme, under the DDH assumption, is obtained from the El Gamal encryption scheme mentioned above: $\mathrm{Com}(m; g, h, r) = (g, h, g^r, mh^r)$.

Existentially Consistent Anonymous Signature . We require ECAS to be a (strongly unforgeable) signature scheme with an anonymity guarantee: without knowing a verification key, one cannot tell if two signatures are signed using the same key or not. We shall also require existential consistency guarantees of ECAS.

To construct an ECAS scheme, we start with a plain (strongly unforgeable) signature scheme, which w.l.o.g., has uniformly random signing keys from which verification keys are deterministically derived (by considering the randomness of the key-generation process as the signing key). We first augment this scheme to support anonymity by adding a layer of encryption, and include the decryption key in the signing and verification keys of the ECAS scheme. To obtain existential consistency, we make the following modifications:

- The signing key SK includes the underlying scheme's signing key, the decryption key for the encryption layer, and additional randomness for making the commitment below.
- The verification key VK includes the underlying verification key, the decryption key for the encryption layer and a commitment to the underlying signing key (using a fully binding commitment scheme as above).
- The signature includes a commitment to VK (but to the encryption key in it) using fresh randomness $\hat{r}$, and a *quasi-deterministic* encryption of $(\hat{r}||\sigma)$ where σ is a signature on $m||\hat{r}||\tau$ using the underlying signature scheme, where τ is the first component of the quasi-deterministic ciphertext.
- Verification corresponds to decrypting the ciphertext, verifying the signature according to the underlying signature scheme and then verifying the consistency of the commitment.

For existential consistency, as well as (strong) unforgeability, we will rely on the encryption scheme to be a COA-QD-PKE. Note that we have rely on the quasi-deterministic nature of the encryption scheme to prevent forgeries which simply refresh the encryption layer (decrypt and re-encrypt).

We point out one subtlety in the above construction. We have defined the signature above to include a commitment to (SK^*, c, EK^*) rather than the actual verification key $VK = (SK^*, c, DK^*)$. This is to avoid the following circularity: the commitment would have the decryption key in it while the encryption would have the randomness used for this commitment. This would prevent us from arguing the properties of ECAS.

Please refer to Appendix B.2 for the full details. Note that this construction shares several similarities with our CASE construction. If one unrolls our CASE construction, there are two layers of COA-QD-PKE, but using two different keys.

Improving the Efficiency. As described in Section 5.4, CASE admits an analogue of "hybrid encryption," whereby long messages can be encased at the cost of applying symmetric-key encryption (SKE) and collision-resistant hashing to the original message, plus the cost of encasing a fixed size message (consisting of the keys for SKE and hashing, and the hash of the message). This makes our CASE construction quite practical.

2.3 A Real-Ideal Definition

A major concern with game-based security definitions is that they may leave out several subtler aspects of security. For instance, even for the simpler (and heavily studied) setting of public-key encryption, the security definition has been strengthened incrementally through a sequence of notions that emerged over the decades: Semantic security or IND-CPA [26], IND-CCA (1 and 2) [21, 34, 37], anonymity [12] and robustness [1, 22, 33]. With CASE, this is clearly an even more pressing concern, given its complexity. In particular, our definition of COA-secure CASE has several games and conditions as part of it, and one may suspect that more such components could be added in the future.

To address this concern, we seek a definition following the *real-ideal paradigm*, where by inspecting the ideal world, one can be easily convinced about the meaningfulness of the definition. However, a *simulation-based definition* quickly leads us to impossibility results. Even for PKE with adaptive security (when decryption keys may be revealed adaptively – a situation we do intend to cover), as observed by Nielsen [35], a simulation based definition is impossible to achieve in the standard model.

In this work, we develop a new definition in the real-ideal paradigm that avoids simulation, but is nevertheless powerful enough to subsume game-based definitions like IND-CCA security. Our definition is based on the *indistinguishability-preserving* security notion of the Cryptographic Agents framework [2, 3]. The original framework of [2, 3] did not allow an adversary to send (possibly maliciously created) objects to an honest party, and as such was not powerful to capture even IND-CCA security. We remove this restriction from the framework and extend it with other useful features. Then, we model CASE in this framework using a natural idealized version, and seek an indistinguishability-preserving implementation for it.

Our main result in this model, informally, is that a COA-secure CASE scheme is in fact, an indistinguishability-preserving implementation of ideal CASE. This validates our COA security definition for CASE.

Active Agents Framework. We briefly discuss the active agents framework (with more technical details in Section 6). The framework is minimalistic and conceptually simple, and consists of the following:

– *Two arbitrary entities.* Test models the honest party, and User models the adversary.
– *The ideal model* has a trusted party $\mathcal{B}$ which hands out *handles* to Test and User for manipulating data stored with it via an idealized interface called "schema"(akin to a functionality in the UC security model).
– *The real model* has Test and User interact with each other using cryptographic objects, in place of ideal handles.
⋆ *Indistinguishability Preservation:* The security requirement in this model is as follows. For any predicate on Test's inputs that is hidden from User in the ideal world, it should be hidden in the real world as well.

An ideal world schema will have an interface corresponding to each algorithm of an application (such as key generation, encasing and decasing for CASE) and an agent corresponding to each cryptographic object (such as keys and ciphertexts). Both Test and User only get handle numbers to agents. Constructing objects via algorithms is modelled as invoking the corresponding schema command and getting a handle for a new agent. Sending cryptographic objects is modelled via a special command called **Transfer**. Test (respectively User) can transfer its agents (via handles) to User (respectively Test), which gets a new handle number to the transferred agent.

Δ-s-IND-PRE Security. To obtain our full definition, we need to further qualify indistinguishability-preservation by specifying the class of Tests and Users in the ideal model. We denote s-IND-PRE as the class of all PPT Test that are hiding against *even unbounded* Users in the *ideal world* (as in [3]). [7]

[7] So that, it is statistical indistinguishability in the ideal model that is required to be preserved as computational indistinguishability in the real model.

The strongest possible s-IND-PRE definition one can ask for in the active agents framework is for the test-family of all PPT programs, which results in a definition that is impossible to realize (even for symmetric key encryption and even in the original framework of [2] – see Section 6.4). However, a more restricted test-family called $\varDelta$ suffices to subsume all possible IND-style (a.k.a. "real-or-random") definitions. Informally, a $\mathsf{Test} \in \varDelta$ reveals everything about the handles for agents it uses in its interaction with User except for a test-bit b corresponding to some arbitrary predicate. When transferring an agent to User, Test chooses two handles h_0, h_1 and communicates these to the user but *transfers only agent for* h_b. Thus, User knows that Test has transferred one of two known agents to her, but does not know which. User may proceed to perform any idealized operation with this newly transferred agent.

In intuitive terms, $\varDelta$-s-IND-PRE formalizes the following guarantee: *as long as* Test does not reveal a secret in the ideal world, the real world will also keep it hidden. It *subsumes essentially all meaningful IND security definitions* for a given interface of the primitive: for any such IND security game, there is $\mathsf{Test} \in \varDelta$ which carries out this game, such that it statistically hides the test-bit when an ideal encryption scheme is used (e.g., in the case of IND-CCA security this formulation corresponds to a game that never decrypts a ciphertext that is identical to the ciphertext that was earlier given as the challenge, called IND-CCA-SE in [10]), and $\varDelta$-s-IND-PRE security applied to this Test translates to the security guarantee in the IND security game.

In particular, $\varDelta$-s-IND-PRE security directly addresses the chosen object attacks of interest, as they can all be captured using specific tests.

Beyond CASE. We point out that the active agents framework developed here is quite general and can be used to model security for other schemas in the presence of adversarially created objects. The original frameworks of [2, 3] modeled security notions for more advanced primitives like indistinguishability obfuscation, differing-inputs obfuscation and VGB obfuscation by using different test families. Transferring these definitions to our new model would yield stronger notions with additional non-malleability guarantees; the resulting primitives remain to be explored. Indeed, as the basic security definitions for obfuscation and functional encryption are increasingly considered to be realizable, the achievability of stronger definitions emerges as an important question.

Limits of $\varDelta$-s-IND-PRE. Even though $\varDelta$-s-IND-PRE security is based on an ideal world model, and subsumes *all possible* IND definitions, we advise caution against interpreting $\varDelta$-s-IND-PRE security on par with a simulation-based security definition (which is indeed unrealizable). For instance, $\varDelta$-s-IND-PRE does not require preserving non-negligible advantages: e.g., a distinguishing advantage of 0.1 in the ideal world could translate to an advantage of 0.2 in the real world. Note that this is usually not a concern, since it corresponds to an ideal world that is already "insecure".

Another issue is that, while an ideal encryption scheme could be used as a non-malleable commitment scheme, $\varDelta$-s-IND-PRE security makes no such assurances. This is because, in the ideal world, if a commitment is to be opened such that indistinguishability ceases, then IND-PRE security makes no more guarantees. We leave it as an intriguing question whether $\varDelta$-s-IND-PRE secure encryption could be leveraged in an indirect way to obtain a non-malleable commitment scheme.

$\varDelta$-s-IND-PRE definition also does not cover side-channel attacks. One can extend the definition to allow the interface of an implementation to have more commands (corresponding to leakage) than in the ideal interface of the schema. We defer this to future work.

Finally, the idealized model in the Agents framework excludes certain kinds of usages that a simulation-based idealization would permit. Specifically, since the ideal interface provides honest users only with handles (serial numbers) for the cryptographic objects they create or receive, they cannot use a cryptographic object as input to another algorithm, or even to an algorithm in the same scheme (e.g., a key cannot be used as a message that is encased). We remark that this restriction is, in fact, a *desirable feature* in a programming interface for a cryptographic library; violating this interface should not be up to the programmer, but should be carefully designed, analyzed and exposed as a new schema by the creators of the cryptographic library.

2.4 Proving COA Security $\Rightarrow$ Δ-s-IND-PRE Secure CASE

Implementing the schema Σ_{case} is a challenging task because it is highly idealized and implies numerous security guarantees that may not be immediately apparent. (For instance, in the ideal world, to produce a case-packet, not only is the signing key needed, but so is the encryption key; hence an adversary with the signing key who gets oracle access to encasing and decasing, should not be able to create a new valid case-packet.) These guarantees are not explicit in the definition of COA security. Nevertheless, we show the following:

Theorem 1. *A Δ-s-IND-PRE secure implementation of Σ_{case} exists if a COA secure CASE scheme exists.*

The construction itself is direct, syntactically translating the elements of a CASE scheme into those of an implementation of Σ_{case}. However, the proof of security is quite non-trivial. This should not be surprising given the simplicity of the COA security definition vis-à-vis the generality of Δ-s-IND-PRE security. We use a careful sequence of hybrids to argue indistinguishability preservation, where some of the hybrids involve the use of an "extended schema" (which is partly ideal and partly real). To switch between these hybrids, we use both PPT simulators (which rely on the indistinguishability and unforgeability guarantees in the COA security) and computationally unbounded simulators (which rely on existential consistency). As we shall see, the simulators heavily rely on the fact that $\mathsf{Test} \in \Delta$, and hence the only uncertainty regarding agents transferred by Test is the choice between one of two known agents, determined by the test-bit b given as input to Test. The essential ingredients of these simulators are summarized below.[8]

- First, we move from the real execution to a hybrid execution in which objects originating from Test are replaced with ideal agents, while the objects originating from the adversary are replaced – by an efficient simulator $\mathcal{S}_b^\dagger$ (which knows the test bit b) – with ideal agents only when their structure can be deduced efficiently based on the objects already in the transcript; otherwise $\mathcal{S}_b^\dagger$ prepares non-ideal agents which internally contain cryptographic objects and transfers them.
In this hybrid, an "extended" schema which allows both ideal and non-ideal agents is used. The extended schema is carefully designed to allow sessions to run correctly, even when non-ideal agents (prepared by $\mathcal{S}_b^\dagger$) and ideal agents interact with each other.
A detailed analysis, using a graph $\mathbb{G}_b^\dagger$ which encodes the combined view of Test and $\mathcal{A}$, is used to argue that the modifications in this hybrid will cause the execution to deviate only if certain "bad events" occur (see Figure 24). The bad events mainly correspond to the violation of conditions explicitly included in the COA security definition (like correctness, unforgeability and unpredictability) or other consequences of the definition (like encasing resistance, in Section 4.1). Since these bad events can all shown to have negligible probability, making this modification keeps the experiment's outcome indistinguishable.[9]
- The next step is to show that there is a simulator $\mathcal{S}^\ddagger$ which does not need to know the bit b to carry out the above simulation. This is perhaps the most delicate part of the proof. The high-level idea is to argue that the executions for $b = 0$ and $b = 1$ should proceed identically from the point of view of the adversary (as Test hides the bit b in the ideal world), and hence a joint simulation should be possible. $\mathcal{S}^\ddagger$ will abort when it cannot assign a single simulated object for the two possible choices of a transferred agent, corresponding to $b = 0$ and $b = 1$. Intuitively, this event corresponds to revealing the test-bit b in the ideal execution. This argument crucially relies on the hiding properties that are part of COA security. These hiding properties are used to first show indistinguishability in an augmented security game (Section 4.1) which resembles the over all system conditioned on Test keeping the bit b hidden statistically in the ideeal execution. Then it is argued that if Test hides the test bit in this execution, then the simulation is good, unless the augmented security guarantee can be broken.
The execution of $\mathcal{S}^\ddagger$ involves assigning "tentative" objects to handles when they are needed to compute objects that are being transferred to the adversary, but they are finalized only they themselves are transferred. The conditions corresponding to the simulator $\mathcal{S}^\ddagger$ failing are carefully restricted to only those cases which reveal

[8] To facilitate keeping track of the arguments being made, we describe the corresponding hybrids from Section 7. The goal is to show $\mathsf{H}_0 \approx \mathsf{H}_7$, for hybrids corresponding to real executions with $b = 0$ and $b = 1$ respectively.

[9] This corresponds to $\mathsf{H}_0 \approx \mathsf{H}_1$ (with $b = 0$) and $\mathsf{H}_6 \approx \mathsf{H}_7$ (with $b = 1$).

the test-bit. For example, suppose Test transfers a case-packet agent such that it has different messages in the two executions corresponding to $b = 0$ and $b = 1$. Then there is no consistent assignment of that agent to an object that works for both $b = 0$ and $b = 1$. Nevertheless this may still keep b hidden, as long as the corresponding decryption keys are not transferred. So $\mathcal{S}^{\ddagger}$ can assign a random case-packet to this agent, provided that a decryption key which can decase the case-packet will be never transferred.
Here, b not being hidden does not yield a contradiction yet.[10]

- The next simulator $\mathcal{S}^*$ is computationally unbounded, and helps us move from the ideal world with the extended schema to the ideal world involving only the schema Σ_{case}. The key to this step is existential consistency: $\mathcal{S}^*$ will use unbounded computational power to break open objects sent by the adversary and map them to ideal agents. It replaces the non-ideal agents from before with ideal agents. $\mathcal{S}^*$ can be thought of as simulating the interface of the extended schema to $\mathcal{S}^{\ddagger}$, while itself interacting with the ideal schema. Existential consistency guarantees help ensure that the view of Test and $\mathcal{A}$ remains the same.[11]
- To prove Δ-s-IND-PRE security we need only consider $\mathsf{Test} \in \Delta$ such that the bit b remains hidden against a *computationally unbounded* adversary. For such a Test, the above two hybrids are indistinguishable from each other.[12]

Together these steps establish that if b is statistically hidden in the ideal execution, then that it is (computationally) hidden in the real execution. Section 7 and Appendix C together present the complete argument.

3 Preliminaries and Definitions

3.1 Formalism of Agents

For the sake of completeness, we include a formalism for modeling agents and sessions, borrowed from [2] (with minor changes).

Definition 1 (Agents). An *agent* is an *interactive Turing Machine*, with the following modifications:

- There is an *a priori* restriction on the size of all the tapes other than the randomness tape (including input, communication and work tapes), as a function of the security parameter.
- There is a special *blocking state* such that if the machine enters such a state, it remains there if the input tape is empty. Similarly, there are blocking states which let the machine block if any combination of the communication tape and the input tape is empty.

◁

We can allow *non-uniform agents* by allowing an additional advice tape. Our framework and basic results work in the uniform and non-uniform model equally well.

Note that an agent who enters a blocking state can move out of it if its configuration is changed by adding a message to its input tape and/or communication tape. However, if the agent enters a halting state, it will not move out of that state. An agent who never enters a blocking state is called a *non-reactive agent*. An agent who never reads or writes from a communication tape is called a *non-interactive agent*.

Definition 2 (Session). A session maps a finite ordered set of agents, their configurations and inputs, to outputs and (updated) configurations of the same agents, as follows. The agents are initialized with the given inputs on their input tapes, and then executed together until they are deadlocked.[13] The result of applying the session is defined as the collection of outputs and configurations of the agents when the session terminates (if it terminates; if not, the result is left undefined). ◁

[10] This corresponds to showing that *if* $\mathsf{H}_2 \approx \mathsf{H}_5$, *then* $\mathsf{H}_1 \approx \mathsf{H}_2$ and $\mathsf{H}_5 \approx \mathsf{H}_6$.

[11] This shows $\mathsf{H}_2 \approx \mathsf{H}_3$ and $\mathsf{H}_4 \approx \mathsf{H}_5$.

[12] That is, $\mathsf{H}_3 \approx \mathsf{H}_4$.

[13] More precisely, the first agent is executed till it enters a blocking or halting state, and then the second and so forth, in a round-robin fashion, until all the agents remain in blocking or halting states for a full round. After each execution of an agent, the contents of its outgoing communication tape are interpreted as an ordered sequence of messages to each of the other agents in the session (some or all of them possibly being empty messages), and copied over to the respective agents' incoming communication tapes.

We shall be restricting ourselves to collections of agents such that sessions involving them are guaranteed to terminate. Note that we have defined a session to have only an initial set of inputs, so that the outcome of a session is well-defined (without the need to specify how further inputs would be chosen).

3.2 Hash Schemes

Definition 3 (Collision-Resistant Hash Function). A CRHF scheme hash, parametrized by a key-length K and digest-length n (polynomial in the security parameter κ) is a deterministic polynomial time algorithm which takes an index $k \in \{0,1\}^K$ and a message $m \in \{0,1\}^*$ and outputs an n-bit hash digest such that the following property holds: For any PPT adversary $\mathcal{A}$, there exists a negligible function negl in κ s.t.

$$\Pr_{k \leftarrow K}\Big[\mathcal{A}(k) = (m_0, m_1) \ \wedge \ m_0 \neq m_1 \ \wedge \ \mathsf{hash}(k, m_0) = \mathsf{hash}(h_k, m_1)\Big] \leq \mathsf{negl}(\kappa)$$

◁

3.3 Encryption Schemes

Definition 4 (SKE). A Symmetric-Key Encryption scheme with efficiently recognizable key-spaces ($\mathcal{K}$, $\mathcal{CT}$) and message space $\mathcal{M}$ consists of the following algorithms.

- skeGen: takes security parameter κ and outputs a key $k \in \mathcal{K}$.
- skeEnc: takes a key k, message $m \in \mathcal{M}$ and outputs a ciphertext $CT \in \mathcal{CT}$.
- skeDec: takes a key k, ciphertext CT and outputs a message m or $\perp$.

Of these, skeEnc and skeDec are deterministic algorithms. These algorithms should satisfy the following properties.

1. **Perfect Correctness of encrypt**: $\forall \kappa, \forall x \in \mathcal{M}$, it holds that:

$$\Pr_{k \leftarrow \mathsf{skeGen}(1^\kappa)}\Big[\mathsf{skeDec}\Big(k, \mathsf{skeEnc}(k, x)\Big) = x\Big] = 1$$

2. **IND-CCA security**: For any PPT adversary $\mathcal{A} = (A_0, A_1)$, there exists a negligible function $\mathsf{negl}(.)$ such that for $\mathsf{skeCCAExp}$ as in Figure 1:

$$\Pr\Big[\mathsf{skeCCAExp}(\mathcal{A}) = 1\Big] \leq \frac{1}{2} + \mathsf{negl}(\kappa)$$

Experiment $\mathsf{skeCCAExp}$

Parameter: Let κ be the security parameter.

```
– $k \leftarrow \mathsf{skeGen}(1^\kappa)$
▷ let $\mathcal{E}$ be s.t. $\mathcal{E}(m) = \mathsf{skeEnc}(k, m)$
▷ let $\mathcal{D}$ be s.t. $\mathcal{D}(CT) = \mathsf{skeDec}(k, CT)$
– $(\mathsf{st}_0, m_0, m_1) \leftarrow A_0^{\mathcal{E},\mathcal{D}}(1^\kappa)$
– $b^* \leftarrow \{0,1\}$, $CT^* \leftarrow \mathsf{skeEnc}(k, m_b)$
▷ let $\mathcal{D}'$ be s.t. $\mathcal{D}'(CT) = \mathsf{skeDec}(k, CT)$ if $CT \neq CT^*$, else $\perp$
– $b' \leftarrow A_1^{\mathcal{E},\mathcal{D}'}(\mathsf{st}_0, CT^*)$
– If $b^* = b'$ output 1, else output 0.
```

Fig. 1: IND-CCA security experiment for SKE.

◁

Definition 5 (PKE). A Public-Key Encryption scheme with efficiently recognizable key-spaces ($\mathcal{EK}$, $\mathcal{DK}$, $\mathcal{CT}$) and message space $\mathcal{M}$ consists of the following algorithms.

- $\mathsf{pkeSKGen}$: takes security parameter κ and outputs a secret key $DK \in \mathcal{DK}$.
- $\mathsf{pkePKGen}$: takes $DK \in \mathcal{DK}$ and outputs a public key $EK \in \mathcal{EK}$.
  We define pkeGen as: $\mathsf{pkeGen}(1^\kappa) := \big(DK \leftarrow \mathsf{pkeSKGen}(1^\kappa), EK \leftarrow \mathsf{pkePKGen}(DK)\big)$
- pkeEnc: takes $EK \in \mathcal{EK}$, message $m \in \mathcal{M}$ and outputs a ciphertext $CT \in \mathcal{CT}$.
- pkeDec: takes $DK \in \mathcal{DK}$, $CT \in \mathcal{CT}$ and outputs a message $m \in \mathcal{M}$.

Of these, $\mathsf{pkePKGen}$ and pkeDec are deterministic algorithms. These algorithms should satisfy the following properties.

1. **Perfect Correctness of encrypt**: $\forall \kappa, \forall x \in \mathcal{M}$, it holds that:

$$\Pr_{(DK,EK) \leftarrow \mathsf{pkeGen}(1^\kappa)} \Big[\mathsf{pkeDec}\Big(DK, \mathsf{pkeEnc}(EK, x)\Big) = x\Big] = 1$$

2. **IND-CCA security**: For any PPT adversary $\mathcal{A} = (A_0, A_1)$, there exists a negligible function $\mathsf{negl}(.)$ such that for $\mathsf{pkeCCAExp}$ as in Figure 2:

$$\Pr\Big[\mathsf{pkeCCAExp}(\mathcal{A}) = 1\Big] \leq \frac{1}{2} + \mathsf{negl}(\kappa)$$

Experiment $\mathsf{pkeCCAExp}$

Parameter: Let κ be the security parameter.

```
– $(DK, EK) \leftarrow \mathsf{pkeGen}(1^\kappa)$
▷ let $\mathcal{D}$ be s.t. $\mathcal{D}(CT) = \mathsf{pkeDec}(DK, CT)$
– $(\mathsf{st}_0, m_0, m_1) \leftarrow A_0^{\mathcal{D}}(EK)$
– $b^* \leftarrow \{0,1\}$, $CT^* \leftarrow \mathsf{pkeEnc}(EK, m_b)$
▷ let $\mathcal{D}'$ be s.t. $\mathcal{D}'(CT) = \mathsf{pkeDec}(DK, CT)$ if $CT \neq CT^*$, else $\perp$
– $b' \leftarrow A_1^{\mathcal{D}'}(\mathsf{st}_0, CT^*)$
– If $b^* = b'$ output 1, else output 0.
```

Fig. 2: IND-CCA security experiment for PKE.

◁

3.4 Signature Schemes

Definition 6 (Signature). A Signature scheme with efficiently recognizable key-spaces ($\mathcal{VK}$, $\mathcal{SK}$, Σ) and message space $\mathcal{M}$ consists of the following algorithms.

- $\mathsf{sigSKGen}$: takes security parameter κ and outputs a signing key $SK \in \mathcal{SK}$.
- $\mathsf{sigVKGen}$: takes $SK \in \mathcal{SK}$ and outputs a verification key $vk \in \mathcal{VK}$.
  We define sigGen as: $\mathsf{sigGen}(1^\kappa) := \big(SK \leftarrow \mathsf{sigSKGen}(1^\kappa), VK \leftarrow \mathsf{sigVKGen}(SK)\big)$

- $\mathsf{sigSign}$: takes $SK \in \mathcal{SK}$, message $m \in \mathcal{M}$ and outputs a signature $\sigma \in \Sigma$.
- $\mathsf{sigVerify}$: takes $VK \in \mathcal{VK}$, message $m \in \mathcal{M}$, signature $\sigma \in \Sigma$ and outputs a bit $b \in \{0, 1\}$.

Of these, $\mathsf{sigVKGen}$ and $\mathsf{sigVerify}$ are deterministic algorithms. These algorithms should satisfy the following properties.

1. **Perfect Correctness of verification**: $\forall \kappa, x$, it holds that:

$$\Pr_{(SK,VK)\leftarrow \mathsf{sigGen}(1^\kappa)}\Big[\mathsf{sigVerify}\Big(VK, x, \mathsf{sigSign}(SK, x)\Big) = 1\Big] = 1$$

2. **Strong-Unforgeability**: For any PPT adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(.)$ such that for $\mathsf{SigForgeExp}$ in Figure 3:

$$\Pr\Big[\mathsf{SigForgeExp}(\mathcal{A}) = 1\Big] \leq \mathsf{negl}(\kappa)$$

Experiment $\mathsf{SigForgeExp}$

Parameter: Let κ be the security parameter.

– $(SK, VK) \leftarrow \mathsf{sigGen}(1^\kappa)$.

▷ let $\mathcal{S}$ be s.t. $\mathcal{S}(m) = \mathsf{sigSign}(SK, m)$

– $(m, h) \leftarrow \mathcal{A}^{\mathcal{S}}(VK)$, where σ was not response of $\mathcal{S}$ to any query by $\mathcal{A}$

– Output $\mathsf{sigVerify}(VK, m, \sigma)$.

Fig. 3: Strong-Unforgeability Experiment for Signature.

◁

3.5 Fully-Binding Commitment Schemes

Definition 7 (Commitment). A (non-interactive) fully-binding commitment scheme for a message space $\mathcal{M} = \{\mathcal{M}_\kappa\}_{\kappa\in\mathbb{N}}$ consists of two polynomial time algorithms defined below:

- $\mathsf{com.Commit}$: takes as input a message $m \in \mathcal{M}_\kappa$ and outputs a commitment c and decommitment information d.
- $\mathsf{com.Open}$: takes as input c, d and outputs a message $m \in \mathcal{M}_\kappa$ or $\perp$.

Where, $\mathsf{com.Open}$ is a deterministic algorithm. The algorithms satisfy the following properties:

1. **Correctness.** For every $m \in \mathcal{M}_\kappa$, we require that

$$\Pr\Big[\mathsf{com.Open}(\mathsf{com.Commit}(m)) = m\Big] = 1$$

2. **Fully Binding.** A commitment scheme is fully binding if $\forall m_0, m_1 \in \mathcal{M}_\kappa$, $\forall r_0, r_1 \in \{0, 1\}^\kappa$, it holds that $(m_0, r_0) \neq (m_1, r_1) \Rightarrow \mathsf{com.Commit}(m_0; r_0) \neq \mathsf{com.Commit}(m_1; r_1)$.
3. **Computational Hiding.** A commitment scheme is computationally hiding if for all $m_0, m_1 \in \mathcal{M}_\kappa$ and all PPT adversaries $\mathcal{A}$,

$$\Big|\Pr\Big[\mathcal{A}(\mathsf{com.Commit}(m_0)) = 1\Big] - \Pr\Big[\mathcal{A}(\mathsf{com.Commit}(m_1)) = 1\Big]\Big| \leq \mathsf{negl}(\kappa)$$

$\triangleleft$

Commitment from any One-Way Permutation. For one-bit messages, commitment schemes can be constructed based on any injective one-way function (Construction 4.4.2 in [25]). The scheme relies on the Goldriech-Levin theorem to modify any one-way function into a hardcore-predicate (while retaining its injective property).
Let f be a one-way permutation with hardcore-predicate h.

$\mathsf{com.Commit}(b)$: sample a random input x and output $c = \big(h, f(x), b + h(x)\big)$, $d = x$.
$\mathsf{com.Open}(c, d)$: parse c as $\big(h, y, b'\big)$, parse d as x. If $f(x) = y$, output $b' + h(x)$; else, output $\perp$.

We now informally prove that the above is a valid commitment scheme. Perfect binding holds from the injective property of f: for any y, there exists a single pre-image x s.t. $y = f(x)$. Computational hiding holds from the hardcore-predicate property of h. An adversary that recovers the bit b with advantage α, also recovers $h(x)$ with same advantage.
In order to commit to a string, one can commit to each bit individually (using hard-core functions could give better efficiency).

4 COA Security for CASE

A CASE scheme involves four keys: a signing key (denoted as SK, typically), a verification key (VK), a decryption key (DK) and an encryption key (EK). Two key generation processes sample the signing and decryption keys, and each of them can be deterministically transformed into corresponding verification and encryption keys. Analogous to encryption and decryption, the two operations in CASE are termed ***encasing*** and ***decasing***. We refer to the output of encasing as a ***case-packet*** (denoted as CP). Below we present the syntax and the COA security definition of a CASE scheme.

Definition 8 (COA-secure CASE). A COA-secure CASE scheme with efficiently recognizable key-spaces $(\mathcal{SK}, \mathcal{VK}, \mathcal{DK}, \mathcal{EK})$ and message space $\mathcal{M}$ consists of the following efficient (polynomial in κ) algorithms.

- skGen: takes security parameter as input, outputs a signing key $SK \in \mathcal{SK}$.
- dkGen: takes security parameter as input, outputs a decryption key $DK \in \mathcal{DK}$.
- vkGen: converts $SK \in \mathcal{SK}$ to a verification key $VK \in \mathcal{VK} \cup \{\perp\}$.
- ekGen: converts $DK \in \mathcal{DK}$ to an encryption key $EK \in \mathcal{EK} \cup \{\perp\}$.
- encase: takes $(SK, EK, m) \in \mathcal{SK} \times \mathcal{DK} \times \mathcal{M}$, outputs $CP \in \mathcal{CP} \cup \{\perp\}$.
- decase: takes $(VK, DK, CP) \in \mathcal{VK} \times \mathcal{DK} \times \mathcal{CP}$ and outputs (m, b) where $m \in \mathcal{M} \cup \{\perp\}$ and $b \in \{0, 1\}$.
- acc: takes any string $obj \in \{0,1\}^{poly(\kappa)}$ as input and outputs a token $t \in \{\textsc{sk}, \textsc{vk}, \textsc{dk}, \textsc{ek}, \textsc{cp}, \perp\}$.

Of these, vkGen, ekGen, decase and acc are deterministic algorithms. Below we refer to algorithms $\mathsf{decase\text{-}msg}$ and $\mathsf{decase\text{-}verify}$ derived from decase as follows:

- $\mathsf{decase\text{-}msg}(DK, CP) = m$ where $(m, b) = \mathsf{decase}(\perp, DK, CP)$
- $\mathsf{decase\text{-}verify}(VK, DK, CP) = m$ if $\mathsf{decase}(VK, DK, CP) = (m, 1)$, and $\perp$ otherwise.

We require the algorithms of a CASE scheme to satisfy the following:

1. **Correctness (of Accept and Accepted Objects)**: $\forall SK \in \mathcal{SK}$, $\forall DK \in \mathcal{DK}$, $\mathsf{acc}(SK) = \textsc{sk} \Rightarrow \mathsf{acc}\big(\mathsf{vkGen}(SK)\big) = \textsc{vk}$ and $\mathsf{acc}(DK) = \textsc{dk} \Rightarrow \mathsf{acc}\big(\mathsf{ekGen}(DK)\big) = \textsc{ek}$. Further, there exists a negligible function negl s.t. $\forall \kappa$, $\forall SK \in \mathcal{SK}$, $DK \in \mathcal{DK}$, $EK \in \mathcal{EK}$, $m \in \mathcal{M}$, the following probabilities are at most $\mathsf{negl}(\kappa)$:

$$\Pr\Big[\mathsf{acc}\big(\mathsf{skGen}(1^\kappa)\big) \neq \textsc{sk}\Big] \qquad \Pr\Big[\mathsf{acc}\big(\mathsf{dkGen}(1^\kappa)\big) \neq \textsc{dk}\Big]$$
$$\Pr\Big[\mathsf{acc}(SK) = \textsc{sk} \ \wedge\ \mathsf{acc}(EK) = \textsc{ek} \ \wedge\ \mathsf{acc}\Big(\mathsf{encase}\big(SK, EK, m\big)\Big) \neq \textsc{cp}\Big]$$

$$\Pr\Big[\mathsf{acc}(SK) = \textsc{sk} \ \wedge\ \mathsf{acc}(DK) = \textsc{dk}$$
$$\wedge\ \mathsf{decase\text{-}msg}\Big(DK, \mathsf{encase}\big(SK, \mathsf{ekGen}(DK), m\big)\Big) \neq m\Big]$$
$$\Pr\Big[\mathsf{acc}(SK) = \textsc{sk} \ \wedge\ \mathsf{acc}(DK) = \textsc{dk}$$
$$\wedge\ \mathsf{decase\text{-}verify}\Big(\mathsf{vkGen}(SK), DK, \mathsf{encase}\big(SK, \mathsf{ekGen}(DK), m\big)\Big) \neq m\Big]$$

2. **Total Hiding**: For any PPT adversary $\mathcal{A} = (A_0, A_1)$, there exists a negligible function negl such that, for $\mathsf{distinguish\text{-}sans\text{-}DK}$ as in Figure 4, $\Pr\Big[\mathsf{distinguish\text{-}sans\text{-}DK}(\mathcal{A}, \kappa) = 1\Big] \leq \frac{1}{2} + \mathsf{negl}(\kappa)$.
3. **Sender Anonymity**: For any PPT adversary $\mathcal{A} = (A_0, A_1)$, there exists a negligible function negl such that, for $\mathsf{distinguish\text{-}sans\text{-}VK}$ as in Figure 4:
$$\Pr\Big[\mathsf{distinguish\text{-}sans\text{-}VK}(\mathcal{A}, \kappa) = 1\Big] \leq \frac{1}{2} + \mathsf{negl}(\kappa).$$
4. **Strong-Unforgeability**: For any PPT adversary $\mathcal{A}$, there exists a negligible function negl such that, for forge as in Figure 4, $\Pr\Big[\mathsf{forge}(\mathcal{A}, \kappa) = 1\Big] \leq \mathsf{negl}(\kappa)$.
5. **Unpredictability**: For all $SK \in \mathcal{SK}, EK \in \mathcal{EK}, CP \in \mathcal{CP}$ $(CP \neq \bot)$ and $m \in \mathcal{M}$, there exists a negligible function negl such that $\Pr\Big[\mathsf{encase}\big(SK, EK, m\big) = CP\Big] \leq \mathsf{negl}(\kappa)$.
6. **Existential Consistency**: There exist functions (not required to be computationally efficient) $\mathsf{skId} : \mathcal{VK} \to \mathcal{SK} \cup \{\bot\}$, $\mathsf{vkId} : \mathcal{CP} \to \mathcal{VK} \cup \{\bot\}$, $\mathsf{dkId} : \mathcal{EK} \to \mathcal{DK} \cup \{\bot\}$, $\mathsf{ekId} : \mathcal{CP} \to \mathcal{EK} \cup \{\bot\}$, $\mathsf{msgId} : \mathcal{CP} \to \mathcal{M} \cup \{\bot\}$ such that, for all objects in the appropriate spaces (and not $\bot$):
$$\begin{aligned}
\mathsf{vkGen}(SK) = VK &\Rightarrow \mathsf{skId}(VK) = SK && \forall VK, SK\\
\mathsf{ekGen}(DK) = EK &\Rightarrow \mathsf{dkId}(EK) = DK && \forall EK, DK\\
\mathsf{decase\text{-}msg}(DK, CP) = m \neq \bot &\Rightarrow \mathsf{dkId}(CP) = DK, &&\\
&\quad\ \mathsf{msgId}(CP) = m && \forall DK, CP\\
\mathsf{decase\text{-}verify}(VK, DK, CP) = m \neq \bot &\Rightarrow \mathsf{vkId}(CP) = VK, &&\\
&\quad\ \mathsf{dkId}(\mathsf{ekId}(CP)) = DK, &&\\
&\quad\ \mathsf{msgId}(CP) = m && \forall VK, DK, CP
\end{aligned}$$

◁

Remark 1. *Minor variations of the above definition are also acceptable. For example, one may allow* decase *and* acc *to be randomized and all our results can be extended to this definition too. However, for the sake of convenience, and since our construction allows it, we have required them to be deterministic. Also, one may include an additional* perfect correctness *condition, which our construction meets; but since our results do not rely on this, we leave this out of the definition.*

4.1 Implications of COA-security for CASE

In this section, we remark on some useful consequences of COA-security. Some of these properties will be used later in Section 7 to prove security of the scheme Π_{case} for CASE.

Variants of Total Hiding. The total hiding property of COA-secure CASE in Definition 8 implies the following simpler properties:

- Receiver Anonymity: this property states that an adversary should not learn which of two encryption keys EK_0 and EK_1 were used to construct the case-packet CP. The experiment is essentially $\mathsf{distinguish\text{-}sans\text{-}DK}$ with $m_0 = m_1$. The experiment $\mathsf{distinguish\text{-}sans\text{-}DK}$ must hold for any adversary that outputs arbitrary m_0, m_1. In particular, it must hold for adversaries that output $m_0 = m_1$.

Total Hiding Experiment $\mathsf{distinguish\text{-}sans\text{-}DK}(\mathcal{A},\kappa)$ where $\mathcal{A} = (A_0, A_1)$ is a 2-stage adversary

```
– For each $b \in \{0,1\}$, sample $DK_b \leftarrow \mathsf{dkGen}(1^\kappa)$ and let $EK_b \leftarrow \mathsf{ekGen}(DK_b)$
▷ Let $\mathcal{D}$ be s.t. $\mathcal{D}(b, VK, CP) = \mathsf{decase}(VK, DK_b, CP)$.
– $(\mathsf{st}_{A_0}, SK_0, SK_1, m_0, m_1) \leftarrow A_0^{\mathcal{D}}(EK_0, EK_1)$
– If $\mathsf{acc}(SK_0) = \perp$ or $\mathsf{acc}(SK_1) = \perp$, output $r \leftarrow \{0,1\}$ and terminate □
– $b^* \leftarrow \{0,1\}$, $CP^* \leftarrow \mathsf{encase}(SK_{b^*}, EK_{b^*}, m_{b^*})$
▷ Let $\mathcal{D}'$ be s.t. $\mathcal{D}'(b, VK, CP) = \perp$ if $CP = CP^*$, and $\mathcal{D}(b, VK, CP)$ otherwise.
– $b' \leftarrow A_1^{\mathcal{D}'}(\mathsf{st}_{A_0}, CP^*)$
– Output 1 iff $b^* = b'$ □
```

Sender Anonymity Experiment $\mathsf{distinguish\text{-}sans\text{-}VK}(\mathcal{A},\kappa)$ where $\mathcal{A} = (A_0, A_1)$ is a 2-stage adversary

```
– For each $b \in \{0,1\}$, sample $SK_b \leftarrow \mathsf{skGen}(1^\kappa)$ and let $VK_b \leftarrow \mathsf{vkGen}(SK_b)$
▷ Let $\mathcal{E}$ be s.t. $\mathcal{E}(b, EK, m)$ returns $\mathsf{encase}(SK_b, EK, m)$
▷ Let $\mathcal{D}$ be s.t. $\mathcal{D}(b, DK, CP) = \mathsf{decase}(VK_b, DK, CP)$
– $(\mathsf{st}_{A_0}, EK, m) \leftarrow A_0^{\mathcal{E},\mathcal{D}}$
– If $\mathsf{acc}(EK) = \perp$, output $r \leftarrow \{0,1\}$ and terminate □
– $b^* \leftarrow \{0,1\}$, $CP^* \leftarrow \mathsf{encase}(SK_{b^*}, EK, m)$
▷ Let $\mathcal{D}'$ be s.t. $\mathcal{D}'(b, DK, CP) = \perp$ if $CP = CP^*$, and $\mathcal{D}(b, DK, CP)$ otherwise.
– $b' \leftarrow A_1^{\mathcal{E},\mathcal{D}'}(\mathsf{st}_{A_0}, CP^*)$
– Output 1 iff $b^* = b'$ □
```

Strong-Unforgeability Experiment $\mathsf{forge}(\mathcal{A},\kappa)$

```
– Sample $SK \leftarrow \mathsf{skGen}(1^\kappa)$, $VK \leftarrow \mathsf{vkGen}(SK)$
▷ Let $\mathcal{E}$ be such that $\mathcal{E}(m, EK)$ returns $\mathsf{encase}(SK, EK, m)$
– $(DK, CP) \leftarrow \mathcal{A}^{\mathcal{E}}(VK)$
– Output 1 if $\mathsf{decase\text{-}verify}(VK, DK, CP) \neq \perp$ and $CP$ was not response of any query to $\mathcal{E}$.
  else, output 0 □
```

Fig. 4: Experiments for defining COA security of CASE

- Message Hiding (IND-CCA): this property states that an adversary should not learn which of two messages m_0, m_1 were used to construct the case-packet CP. The experiment is essentially $\mathsf{distinguish\text{-}sans\text{-}DK}$ with $DK_0 = DK_1$. We prove the implication as follows. Suppose that total hiding property holds for a scheme. Consider an adversary $\mathcal{A}$ with advantage α in the IND-CCA experiment. That is, if output of $\mathcal{A}$ is o, then:

  $$2\alpha = \Big| \Pr\big[o = 0 | b = 0\big] - \Pr\big[o = 0 | b = 1\big] \Big|$$

  We build adversaries $\mathcal{A}_0^*$ and $\mathcal{A}_1^*$ for $\mathsf{distinguish\text{-}sans\text{-}DK}$ as follows.

  - $\mathcal{A}_0^*$: it internally runs $\mathcal{A}$ in a straightline black-box way and interacts with the experiment as follows. It receives EK_0, EK_1 from the experiment and sends EK_0 to $\mathcal{A}$. It responds to all decryption queries of $\mathcal{A}$ using the decryption oracle to DK_0 from the experiment. When $\mathcal{A}$ outputs the challenge (SK_0, SK_1, m_0, m_1), it sends them to the experiment as its challenge, gets back challenge ciphertext CP and sends it to $\mathcal{A}$. Finally, it outputs $\mathcal{A}$'s output.
  - $\mathcal{A}_1^*$: it internally runs $\mathcal{A}$ in a straightline black-box way and interacts with the experiment as follows. It receives EK_0, EK_1 from the experiment and sends EK_0 to $\mathcal{A}$. It responds to all decryption queries of

$\mathcal{A}$ using the decryption oracle to DK_0 from the experiment. When $\mathcal{A}$ outputs the challenge messages (SK_0, SK_1, m_0, m_1), it sends (SK_1, SK_1, m_1, m_1) as the challenge to the experiment, gets back challenge ciphertext CP and sends it to $\mathcal{A}$. Finally, it outputs $\mathcal{A}$'s output.

We define the following quantities:

$$x := \Pr\left[o = 0 \text{ given } \mathsf{encase}(SK_0, EK_0, m_0)\right]$$
$$y := \Pr\left[o = 0 \text{ given } \mathsf{encase}(SK_1, EK_0, m_1)\right]$$
$$z := \Pr\left[o = 0 \text{ given } \mathsf{encase}(SK_1, EK_1, m_1)\right]$$

Then, advantage of $\mathcal{A}$ in the IND-CCA experiment is $\frac{|x-y|}{2}$, advantage of $\mathcal{A}_0^*$ in $\mathsf{distinguish\text{-}sans\text{-}DK}$ is $\frac{|x-z|}{2}$ and advantage of $\mathcal{A}_1^*$ in $\mathsf{distinguish\text{-}sans\text{-}DK}$ is $\frac{|y-z|}{2}$. But, trivially, $|x - y| \leq |x - z| + |y - z|$, thus the advantage α of $\mathcal{A}$ in the IND-CCA experiment must be negligible.

Encasing Resistance. We point out an implication of COA security – called "encasing resistance" – that will be useful later. Encasing resistance requires that any PPT adversary who is given access to an honestly generated encryption/decryption key-pair only via oracles for encasing (w.r.t. any signing key) and decasing using those keys, has negligible probability of generating a "new" valid case-packet for these keys (i.e., a case-packet that is different from the ones returned by the encasing oracle queries, and which on feeding to the decasing oracle returns a non-$\perp$ output).

Experiment $\mathsf{encase\text{-}sans\text{-}EK}(\mathcal{A}, \kappa)$

– $DK \leftarrow \mathsf{dkGen}(1^\kappa)$, $EK \leftarrow \mathsf{ekGen}(DK)$
▷ Let $\mathcal{E}, \mathcal{D}$ be oracles, where $\mathcal{E}(SK, m)$ returns $\mathsf{encase}(SK, EK, m)$ and $\mathcal{D}(VK, CP)$ returns $\mathsf{decase}(VK, DK, CP)$
– $CP \leftarrow \mathcal{A}^{\mathcal{E},\mathcal{D}}$
– Output 1 iff $\mathsf{decase\text{-}msg}(DK, CP) \neq \perp$ and CP was not previously returned by $\mathcal{E}$ □

Fig. 5: Encasing-Resistance Experiment for CASE

Definition 9 (Encasing-Resistance). A CASE scheme satisfies encasing-resistance if, for all PPT adversaries $\mathcal{A}$, there exists a negligible function negl s.t. for $\mathsf{encase\text{-}sans\text{-}EK}$ as in Figure 5:

$$\Pr\left[\mathsf{encase\text{-}sans\text{-}EK}(\mathcal{A}, \kappa) = 1\right] \leq \mathsf{negl}(\kappa) \qquad \triangleleft$$

Lemma 1. *Any COA-secure CASE scheme satisfies encasing-resistance.*

Proof sketch: The idea behind the proof is that in the encasing-resistance experiment, the adversary has access to the pair (DK, EK) only through an oracle, and thanks to the total hiding property, it cannot distinguish if the keys used in the oracle are replaced with an independent pair (but the experiment's output is still defined w.r.t. original key pair). Now, in this modified experiment, the adversary's goal is to produce a case-packet that can be decased with a freshly sampled decryption key. This in turn is not feasible, because by existential consistency, a case-packet can be decased by at most one decryption key, and the probability that a freshly sampled decryption key equals the one associated with the the case-packet is negligible. The formal argument is given in Appendix A.1. □

We point out that the proof crucially relies on existential consistency as well as the hiding guarantees. Indeed, a CASE scheme modified to include a "dummy" case-packet for which $\mathsf{decase\text{-}msg}$ yields a non-$\perp$ message for every decryption key continues to satisfy all the other properties; and this dummy case-packet can be used to violate encasing resistance of the modified scheme.

Augmented Security. It would be convenient for us to capture the consequences of the total hiding and sender anonymity conditions in COA security in an "augmented" hiding experiment. This experiment allows an adversary $\mathcal{A}$ to adaptively choose the kind of hiding property it wants to attack. The experiment maintains n decryption/encryption key pairs and n signing/verification key pairs (where n is specified by $\mathcal{A}$), and also allows $\mathcal{A}$ to send more objects to the experiment. Throughout the experiment, the adversary can retrieve the keys, or access the encase or decase oracles using any combination of these objects. In the challenge phase, it can specify two such sets of inputs to an oracle, and one of the two will be randomly used by the experiment. The adversary's goal is to guess which set of inputs was chosen in the challenge phase. The experiment aborts if at any point responding to the adversary will trivially reveal this choice. (E.g., if the two sets of inputs were to encase two different messages, and later on the decryption key for one of the two is requested.)

Definition 10 (Augmented Security). A CASE scheme satisfies augmented security if, for all PPT adversaries $\mathcal{A}$, there exists a negligible function negl s.t. for aug as in Figure 16:

$$\Pr\left[\mathsf{aug}(\mathcal{A}, \kappa) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\kappa) \qquad \triangleleft$$

We prove the following in Appendix A.2.

Lemma 2. *Any COA-secure CASE scheme satisfies augmented security.*

5 Constructing a COA-secure CASE scheme

In this section, we instantiate a COA-secure CASE scheme. Please refer to Section 2.2 for an overview of the construction.

5.1 Building Block: COA-secure QD-PKE

In this section, we define and instantiate a new primitive called COA-secure QD-PKE that will be required in the main construction. Towards this, we first define QD-PKE and show that existing PKE schemes such as the modified Cramer-Shoup construction of [1] satisfy this definition. We then show that any QD-PKE and a fully-binding commitment scheme can be used to get COA-security.

Definition 11 (Quasi-Deterministic Anon-CCA PKE). A PKE scheme is QD anon-CCA PKE if it satisfies the following properties.

1. **Quasi-Deterministic**: There exists an efficient randomized algorithm pkeEnc_1 and an in-efficient deterministic algorithm pkeEnc_2 such that $\forall \kappa$, $\forall x \in \mathcal{M}$, $\forall EK \in \mathcal{EK}$, $\forall r \in \{0,1\}^{poly(\kappa)}$:

$$\mathsf{pkeEnc}(EK, x; r) = \Big(\mathsf{pkeEnc}_1(EK; r),\ \mathsf{pkeEnc}_2\big(EK, \mathsf{pkeEnc}_1(EK; r), x\big)\Big)$$

2. **Quasi-Deterministic Anonymous IND-CCA security**: For any PPT adversary $\mathcal{A} = (A_0, A_1)$, there exists a negligible function $\mathsf{negl}(.)$ such that for $\mathsf{pkeQDAnonCCAExp}$ as in Figure **??**:

$$\Pr\left[\mathsf{pkeQDAnonCCAExp}(\mathcal{A}) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\kappa)$$

$\triangleleft$

We note that, there exist PKE schemes that are quasi-deterministic anon-CCA secure, such as the Cramer-Shoup PKE scheme (with the modification of [1]). Please refer to Appendix B.1 for the proof.

Lemma 3. *Assuming the Decisional Diffie-Hellman assumption (DDH), there exists a Quasi-Deterministic Anon-CCA PKE scheme.*

We now define COA-secure QD-PKE. Similar to CASE, we require the primitive to support malicious objects and have existential consistency.

Definition 12 (COA-secure Quasi-Deterministic PKE). A QD-PKE scheme $(\mathsf{pkeSKGen}, \mathsf{pkePKGen}, \mathsf{pkeEnc}, \mathsf{pkeDec})$ is COA-secure if it has the following additional algorithm

- pkeAcc: takes any string $obj \in \{0,1\}^{poly(\kappa)}$ and outputs a token $t \in \{\textsc{ek}, \textsc{dk}, \textsc{ct}, \bot\}$.

Where, pkeAcc is a deterministic algorithm. We require the algorithms to satisfy the following additional properties:

1. **Correctness**: $\forall m \in \mathcal{M}$, $\forall SK \in \mathcal{DK}$, $\forall EK \in \mathcal{EK}$, the following probabilities are negligible in κ

$$\Pr\left[\mathsf{pkeAcc}\big(\mathsf{pkeSKGen}(1^\kappa)\big) \neq \textsc{dk}\right]$$
$$\Pr\left[\mathsf{pkeAcc}(EK) = \textsc{ek} \;\wedge\; \mathsf{pkeAcc}(\mathsf{pkeEnc}(EK, m)) \neq \textsc{ct}\right]$$
$$\Pr\left[\mathsf{pkeAcc}(DK) = \textsc{dk} \;\wedge\; \mathsf{pkeAcc}(\mathsf{pkePKGen}(DK)) \neq \textsc{ek}\right]$$
$$\Pr\left[\mathsf{pkeAcc}(DK) = \textsc{dk} \;\wedge\; \mathsf{pkeDec}\Big(DK, \mathsf{pkeEnc}\big(\mathsf{pkePKGen}(DK), m\big)\Big) \neq m\right]$$

2. **Existential Consistency**: There exist *computationally inefficient* deterministic extractor algorithms $\mathsf{pkeSKId} : \mathcal{EK} \to \mathcal{DK} \cup \{\bot\}$, $\mathsf{pkePKId} : \mathcal{CT} \to \mathcal{EK} \cup \{\bot\}$, $\mathsf{pkeMsgId} : \mathcal{CT} \to \mathcal{M} \cup \{\bot\}$ such that, $\forall m \in \mathcal{M}$, $\forall EK \in \mathcal{EK}$, $\forall CT \in \mathcal{CT}$, $\forall DK \in \mathcal{DK}$:

$$\begin{aligned}
\mathsf{pkePKGen}(DK) = EK \quad &\Rightarrow \mathsf{pkeSKId}(EK) = DK \\
\mathsf{pkeEnc}(EK, m) = CT \quad &\Rightarrow \mathsf{pkePKId}(CT) = EK \\
\mathsf{pkeDec}(DK, CT) = m \neq \bot \quad &\Rightarrow \mathsf{pkeSKId}(\mathsf{pkePKId}(CT)) = DK \\
\mathsf{pkeDec}(DK, CT) = m \neq \bot \quad &\Rightarrow \mathsf{pkeMsgId}(CT) = m
\end{aligned} \qquad \lhd$$

Instantiating COA-secure QD-PKE. Following the description in Section 2.2, we show how any QD-PKE scheme and fully binding commitment can be transformed into a COA-secure scheme (Figure 6). Please refer to Appendix B.1 for proof of security.

Parameter: Let κ be the security parameter.
Let $P^* = (\mathsf{pkeGen}^*, \mathsf{pkeEnc}^*, \mathsf{pkeDec}^*)$ be a QD anon-CCA PKE scheme.
Let $C = (\mathsf{comGen}, \mathsf{comCommit}, \mathsf{comOpen})$ be a fully binding commitment scheme.

COA-secure QD-PKE scheme P:

- $\mathsf{pkeSKGen}(1^\kappa)$:
  - sample $r_0 \leftarrow \{0,1\}^{poly(\kappa)}$
  - sample $r_1 \leftarrow \{0,1\}^{poly(\kappa)}$
  - output $DK := (r_0, r_1)$

- $\mathsf{pkePKGen}(DK)$:
  - if $\mathsf{pkeAcc}(DK) \neq \textsc{dk}$, output $\bot$
  - parse DK as (r_0, r_1)
  - $(DK^*, EK^*) \leftarrow \mathsf{pkeGen}^*(1^\kappa; r_0)$ using randomness r_0
  - $c \leftarrow \mathsf{comCommit}(r_0; r_1)$ using randomness r_1
  - output $EK := (EK^*, c)$

```
– $\mathsf{pkeEnc}(PK, m)$:
    • if $\mathsf{pkeAcc}(EK) \neq \textsc{ek}$, output $\perp$
    • parse $EK$ as $(EK^*, c)$
    • sample $\hat{r} \leftarrow \{0,1\}^{poly(\kappa)}$
    • $\hat{c} \leftarrow \mathsf{comCommit}^*(EK; \hat{r})$ using randomness $\hat{r}$
    • $CT^* \leftarrow \mathsf{pkeEnc}^*(EK^*, m||\hat{r})$
    • output $CT := (\hat{c}, CT^*)$

– $\mathsf{pkeDec}(DK, CT)$:
    • if $\mathsf{pkeAcc}(DK) \neq \textsc{dk}$ or $\mathsf{pkeAcc}(CT) \neq \textsc{ct}$, output $\perp$
    • parse $DK$ as $(r_0, r_1)$
    • parse $CT$ as $(\hat{c}, CT^*)$
    • $(DK^*, EK^*) \leftarrow \mathsf{pkeGen}^*(1^\kappa; r_0)$ using randomness $r_0$
    • $m||\hat{r} \leftarrow \mathsf{pkeDec}^*(DK^*, CT^*)$
    • if $\hat{c} \neq \mathsf{comCommit}(EK;\ \hat{r})$, output $\perp$
      else output $m$

– $\mathsf{pkeAcc}(obj)$: if $obj \in \mathcal{DK}/\ \mathcal{EK}/\ \mathcal{CT}$, output $\textsc{dk}/\ \textsc{ek}/\ \textsc{ct}$ respectively.
```

Fig. 6: COA-secure QD-PKE scheme.

Lemma 4. *If there exists a quasi-deterministic anon-CCA PKE scheme and a fully binding commitment scheme; then there exists a COA-secure quasi-deterministic PKE scheme.*

5.2 Building Block: ECAS

In this section, we define and instantiate a new primitive called Existentially Consistent Anonymous Signature (ECAS) that will be required in the main construction.

Definition 13 (Existentially Consistent Anonymous Signature). A signature scheme (sigSKGen, sigVKGen, sigSign, sigVerify) is Existentially Consistent Anonymous Signature if it has the following additional algorithm

– sigAcc: takes any string $obj \in \{0,1\}^{poly(\kappa)}$ and outputs a token $t \in \{\textsc{sk}, \textsc{vk}, \textsc{sig}, \perp\}$.

Where, sigAcc is a deterministic algorithm. We require the algorithms to satisfy the following additional properties:

1. **Correctness**: $\forall \kappa$, there exists a negligible function $\mathsf{negl}(.)$ such that, $\forall SK \in \mathcal{SK}$, $\forall m \in \mathcal{M}$, the following probabilities are negligible in κ

$$\Pr\Big[\mathsf{sigAcc}\big(\mathsf{sigSKGen}(1^\kappa)\big) \neq \textsc{sk}\Big]$$
$$\Pr\Big[\mathsf{sigAcc}(SK) = \textsc{sk}\ \wedge\ \mathsf{sigAcc}\Big(\mathsf{sigVKGen}(SK)\Big) \neq \textsc{vk}\Big]$$
$$\Pr\Big[\mathsf{sigAcc}(SK) = \textsc{sk}\ \wedge\ \mathsf{sigAcc}\Big(\mathsf{sigSign}(SK, m)\Big) \neq \textsc{sig}\Big]$$
$$\Pr\Big[\mathsf{sigAcc}(SK) = \textsc{sk}\ \wedge\ \mathsf{sigVerify}\Big(\mathsf{sigVKGen}(SK), m, \mathsf{sigSign}(SK, m)\Big) \neq 1\Big]$$

2. **Strong-Unforgeability**: For any PPT adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(.)$ such that for SigForgeExp in Figure 3:

$$\Pr\Big[\mathsf{SigForgeExp}(\mathcal{A}) = 1\Big] \leq \mathsf{negl}(\kappa)$$

3. **(Signer) Anonymity**: For any PPT adversary $\mathcal{A} = (A_0, A_1)$, there exists a negligible function $\mathsf{negl}(.)$ such that tfor SigAnonExp as in Figure 7:

$$\Pr\Big[\mathsf{SigAnonExp}(\mathcal{A}) = 1\Big] \leq \frac{1}{2} + \mathsf{negl}(\kappa)$$

```
                              Experiment SigAnonExp

Parameter: $\mathcal{A} = (A_0, A_1)$ is a 2-stage adversary and $\kappa$ is the security parameter.

– for each $b \in \{0,1\}$, sample $(SK_b, VK_b) \leftarrow \mathsf{sigGen}(1^\kappa)$.
▷ Let $\mathcal{S}$ be s.t. $\mathcal{S}(b', m') = \mathsf{sigSign}(SK_{b'}, m')$
– $(\mathsf{st}_{A_0}, m) \leftarrow A_0^{\mathcal{S}}(1^\kappa)$
– $b^* \leftarrow \{0,1\}$, $\sigma \leftarrow \mathsf{sigSign}(SK_{b^*}, m)$,
– $b^* \leftarrow A_1^{\mathcal{S}}(\mathsf{st}_{A_0}, \sigma)$
– Output 1 iff $b = b^*$.
```

Fig. 7: Experiment for Existentially Consistent Anonymous Signature .

4. **Existential Consistency**: There exist *computationally inefficient* deterministic extractor algorithms $\mathsf{sigVKId} : \Sigma \to \mathcal{VK} \cup \{\bot\}$, $\mathsf{sigSKId} : \mathcal{VK} \to \mathcal{SK} \cup \{\bot\}$ s.t. $\forall SK \in \mathcal{SK}$, $\forall VK \in \mathcal{VK}$, $\forall \sigma \in \Sigma$, the following probabilities are negligible in κ:

$$\begin{aligned} \mathsf{sigVKGen}(SK) = VK &\quad \Rightarrow \mathsf{sigSKId}(VK) = SK \\ \mathsf{sigSign}(SK, x) = \sigma &\quad \Rightarrow \mathsf{sigSKId}(\mathsf{sigVKId}(\sigma) = SK \\ \mathsf{sigVerify}(VK, x, \sigma) = 1 &\quad \Rightarrow \mathsf{sigVKId}(\sigma)) = VK \end{aligned} \qquad \lhd$$

Instantiating Existentially Consistent Anonymous Signature . Following the description in Section 2.2, we show how any COA-secure QD-PKE scheme and fully binding commitment can be used to construct an ECAS scheme (Figure 8). Please refer to Appendix B.2 for the proof.

```
Parameter: Let $\kappa$ be the security parameter.
Let $S^* = (\mathsf{sigGen}^*, \mathsf{sigSign}^*, \mathsf{sigVerify}^*)$ be a signature scheme.
Let $E = (\mathsf{pkeGen}, \mathsf{pkeEnc}, \mathsf{pkeDec}, \mathsf{pkeAcc}, \mathsf{pkeSKId}, \mathsf{pkePKId}, \mathsf{pkeMsgId})$ be a COA-secure QD-PKE scheme.
Let $C = (\mathsf{comGen}, \mathsf{comCommit}, \mathsf{comOpen})$ be a fully binding commitment scheme.

Existentially Consistent Anonymous Signature  scheme $S$:

  – $\mathsf{sigSKGen}(1^\kappa)$:
      • sample $r_0 \leftarrow \{0,1\}^{poly(\kappa)}$
      • sample $r_1 \leftarrow \{0,1\}^{poly(\kappa)}$
      • sample $DK^* \leftarrow \mathsf{pkeSKGen}(1^\kappa)$
      • output $SK := (r_0, r_1, DK^*)$

  – $\mathsf{sigVKGen}(SK)$:
      • if $\mathsf{sigAcc}(SK) \neq \textsc{sk}$, output $\bot$
      • parse $SK$ as $(r_0, r_1, DK^*)$
      • $(SK^*, VK^*) \leftarrow \mathsf{sigGen}^*(1^\kappa; r_0)$
      • $c \leftarrow \mathsf{comCommit}(r_0;\ r_1)$
      • output $VK := (VK^*, c, DK^*)$
```

```
– $\mathsf{sigSign}(SK, m)$:
    • if $\mathsf{sigAcc}(SK) \neq \textsc{sk}$, output $\perp$
    • parse $SK$ as $(r_0, r_1, DK^*)$ and $VK$ as $(VK^*, c, DK^*)$
    • sample $\hat{r}, r_3 \leftarrow \{0,1\}^{poly(\kappa)}$
    • $\hat{c} \leftarrow \mathsf{comCommit}(VK^*||c||EK^*;\ \hat{r})$
    • $(SK^*, VK^*) \leftarrow \mathsf{sigGen}^*(1^\kappa; r_0)$
    • $EK^* \leftarrow \mathsf{pkePKGen}(DK^*)$
    • $\tau \leftarrow \mathsf{pkeEnc}_1(EK^*; r_3)$
    • $\sigma^* \leftarrow \mathsf{sigSign}^*(SK^*, m||\tau)$
    • $CT \leftarrow \mathsf{pkeEnc}(EK^*, \sigma^*||\hat{r}; r_3)$
    • output $(\hat{c}, CT)$
```

```
– $\mathsf{sigVerify}(VK, m, \sigma)$:
    • if $\mathsf{sigAcc}(VK) \neq \textsc{vk}$ or $\mathsf{sigAcc}(\sigma) \neq \textsc{sig}$, output $\perp$
    • parse $VK$ as $(VK^*, c, DK^*)$
    • parse $\sigma$ as $(\hat{c}, CT)$ and $CT$ as $(\tau, CT')$
    • $EK^* \leftarrow \mathsf{pkePKGen}(DK^*)$
    • $\sigma^*||\hat{r} \leftarrow \mathsf{pkeDec}(DK^*, CT)$
    • if $\hat{c} \neq \mathsf{comCommit}(VK^*||c||EK^*;\ \hat{r})$, output $\perp$
      else output $\mathsf{sigVerify}^*(VK^*, m||\tau, \sigma^*)$
```

```
– $\mathsf{sigAcc}(obj)$:
    • if $obj \in \mathcal{SK}$, parse $obj$ as $(r_0, r_1, DK^*)$. If $\mathsf{pkeAcc}(DK^*) = \textsc{dk}$, output $\textsc{sk}$; else $\perp$.
    • if $obj \in \mathcal{VK}$, parse $obj$ as $(VK^*, c, DK^*)$. If $\mathsf{pkeAcc}(DK^*) = \textsc{dk}$, output $\textsc{vk}$; else $\perp$.
    • if $obj \in \Sigma$, parse $obj$ as $(CT, \hat{c})$. If $\mathsf{pkeAcc}(CT) = \textsc{ct}$, output $\textsc{sig}$; else $\perp$.
```

Fig. 8: Existentially Consistent Anonymous Signature scheme.

Lemma 5. *If there exists a signature scheme, a COA-secure QD-PKE scheme and a fully binding commitment scheme; then there exists a Existentially Consistent Anonymous Signature scheme.*

Compactness. Without loss of generality, we assume that our signature schemes have fixed length signatures independent of the size of the message (beyond the security parameter). To achieve compactness, we can start with any plain signature scheme and define a new scheme where the signature is actually on a hash of the message computed using a full-domain collision-resistant hash function.

5.3 Main Construction: COA-secure CASE

We now give the main construction.

Lemma 6. *If there exists a COA-secure QD-PKE scheme and an Existentially Consistent Anonymous Signature scheme, then there exists a COA-secure CASE scheme.*

Proof: Let E be a COA-secure QD-PKE scheme (Definition 12) and S be a ECAS scheme (Definition 13). We prove that the scheme in Figure 9 is a COA-secure CASE scheme (Definition 8).

– **Total Hiding:** we prove this via a reduction to the quasi-deterministic anon IND-CCA security of the underlying PKE scheme. Let $\mathcal{A}$ be an adversary with advantage α in the $\mathsf{distinguish\text{-}sans\text{-}DK}$ experiment. We build an adversary $\mathcal{A}^*$ for the $\mathsf{pkeQDAnonCCAExp}$ experiment as follows. It accepts (EK_0, EK_1, τ) from the experiment and forwards (EK_0, EK_1) to $\mathcal{A}$. For any polynomial oracle query of the form (VK', b', CP') from $\mathcal{A}$, it queries the experiment on (b', CT') (where $CT' = CP'$), receives the decryption $m'||\sigma'$, checks if the signature is valid w.r.t. VK' and returns m' to $\mathcal{A}$. It receives the challenge messages (SK_0, SK_1, m_0, m_1) from $\mathcal{A}$, constructs each m_b^* as $m_b^* = m_b||\sigma_b$, where $\sigma_b = \mathsf{sigSign}(SK_b, m||PK_b||\tau)$. It sends (m_0^*, m_1^*) to the experiment, receives the challenge ciphertext and forwards it to $\mathcal{A}$. Finally, it outputs $\mathcal{A}$'s output. Thus, $\mathcal{A}^*$ has advantage α, which from our assumption that E is a secure quasi-deterministic anon-PKE scheme, must be negligible.

– **Sender Anonymity:** we prove this via a reduction to the anonymity of the underlying signature scheme. Let $\mathcal{A}$ be an adversary with advantage α in the $\mathsf{distinguish\text{-}sans\text{-}VK}$ experiment. We build an adversary $\mathcal{A}^*$ for the $\mathsf{SigAnonExp}$ experiment as follows. For any polynomial oracle query of the form (b', EK', m') that it receives from $\mathcal{A}$, it samples randomness r', constructs $\tau' \leftarrow \mathsf{pkeEnc}_1(EK';\ r')$, queries the oracle on $(b', m'||EK'||\tau')$, gets back σ' and sends CT' =$\mathsf{pkeEnc}(EK', m'||\sigma';\ r')$ to $\mathcal{A}$. When $\mathcal{A}$ outputs the challenge

Parameter: Let κ be the security parameter.
Let $S = (\mathsf{sigGen}, \mathsf{sigSign}, \mathsf{sigVerify}, \mathsf{sigAcc}, \mathsf{sigSKId}, \mathsf{sigVKId})$ be a Existentially Consistent Anonymous Signature scheme.
Let $E = (\mathsf{pkeGen}, \mathsf{pkeEnc}_1, \mathsf{pkeEnc}, \mathsf{pkeDec}, \mathsf{pkeAcc}, \mathsf{pkeSKId}, \mathsf{pkePKId}, \mathsf{pkeMsgId})$ be a COA-secure QD-PKE scheme.

COA-secure CASE Scheme SE:

```
– skGen($1^\kappa$):
    output $SK \leftarrow \mathsf{sigSKGen}(1^\kappa)$
– vkGen($SK$):
    output $VK \leftarrow \mathsf{sigVKGen}(SK)$
– encase($SK, EK, m$):
    $\tau \leftarrow \mathsf{pkeEnc}_1(EK;\ r)$
    $\sigma \leftarrow \mathsf{sigSign}(SK, m||EK||\tau)$
    $CT \leftarrow \mathsf{pkeEnc}(EK, m||\sigma;\ r)$
    output $CP := CT$
– acc($obj$):
    if $obj \in \mathcal{SK} \cup \mathcal{VK}$, output $\mathsf{sigAcc}(obj)$
    else if $obj \in \mathcal{DK} \cup \mathcal{EK} \cup \mathcal{CP}$, output $\mathsf{pkeAcc}(obj)$
    else output $\perp$

– dkGen($1^\kappa$):
    output $DK \leftarrow \mathsf{pkeSKGen}(1^\kappa)$
– ekGen($DK$):
    output $EK \leftarrow \mathsf{pkePKGen}(DK)$
– decase-msg($DK, CP$):
    parse $CP$ as $CT$
    if $\mathsf{pkeDec}(DK, CT) = \perp$, output $\perp$
    $m||\sigma \leftarrow \mathsf{pkeDec}(DK, CT)$
    output $m$
– decase($VK, DK, CP$):
    parse $CP$ as $CT$
    if $\mathsf{pkeDec}(DK, CT) = \perp$, output $\perp$
    $m||\sigma \leftarrow \mathsf{pkeDec}(DK, CT)$
    $EK \leftarrow \mathsf{pkePKGen}(DK)$
    parse $CT$ as $(\tau, c)$
    output $\big(m, \mathsf{sigVerify}(VK, \sigma, m||EK||\tau)\big)$
```

Fig. 9: COA secure CASE

(EK, m), it samples randomness r, constructs $\tau \leftarrow \mathsf{pkeEnc}_1(EK;\ r)$, sends $m||EK||\tau$ as the challenge message to the experiment, receives σ as the challenge signature, sends $CT = \mathsf{pkeEnc}(EK, m||\sigma;\ r)$ as the challenge ciphertext to $\mathcal{A}$ and outputs $\mathcal{A}$'s output. Thus, $\mathcal{A}^*$ has advantage α, which from our assumption that S is a COA-secure signature scheme, must be negligible.

– **Strong-Unforgeability:** we prove this via a reduction to the unforgeability of the underlying signature scheme. Let $\mathcal{A}$ be an adversary with advantage α in the forge experiment. We build an adversary $\mathcal{A}^*$ for the $\mathsf{SigForgeExp}$ experiment as follows. It receives VK from the experiment and forwards it to $\mathcal{A}$. For any polynomial oracle query of the form (m', EK') that it receives from $\mathcal{A}$, it samples randomness r', constructs $\tau' \leftarrow \mathsf{pkeEnc}_1(EK';\ r')$, queries the oracle on $m'||EK'||\tau'$, gets back σ' and sends $CT' = \mathsf{pkeEnc}(EK', m'||\sigma';\ r')$ to $\mathcal{A}$. When $\mathcal{A}$ outputs the forgery (DK, CT), it gets $EK \leftarrow \mathsf{ekGen}(DK)$, parses CT as (τ, c), decrypts CT to get $m||\sigma \leftarrow \mathsf{decase\text{-}verify}(VK, DK, CT)$ and outputs $(m||EK||\tau, \sigma)$ as its forgery. Thus, $\mathcal{A}^*$ has advantage α, which from our assumption that S is a COA-secure signature scheme, must be negligible.

– **Unpredictability:** this follows trivially from the Quasi-Deterministic property of the PKE scheme. The PKE ciphertext is of the form (τ, CT'), but τ must have enough entropy so that IND-CCA holds.

– **Correctness and Existential Consistency:** the extractor algorithms are defined in Figure 10. Then, $\forall SK \in \mathcal{SK}, DK \in \mathcal{DK}, m \in \mathcal{M}$, let $VK \leftarrow \mathsf{vkGen}(SK)$, $EK \leftarrow \mathsf{ekGen}(DK)$, $CP \leftarrow \mathsf{encase}(SK, EK, m)$.

- From the correctness of the underlying primitives, it holds that the objects are accepted with probability $1 - \mathsf{negl}(\kappa)$. Further, $\mathsf{pkeDec}(DK, CP)$ outputs $m||\sigma$ and $\mathsf{sigVerify}(VK, \sigma, m||EK||\tau)$ outputs 1 with probability $1 - \mathsf{negl}(\kappa)$.
- From the existential consistency of the underlying primitives, it holds that $\mathsf{skId}(VK) = SK$, $\mathsf{dkId}(EK) = DK$. Further, for any $CP \in \mathcal{CP}$ s.t. $\mathsf{acc}(CP) = 1$, it holds that if $\mathsf{decase\text{-}msg}(DK, CP) \neq$

Existential Consistency:

```
• skId($VK$):
    output sigSKId($VK$)
• vkId($CP$):
    $m||\sigma \leftarrow$ pkeMsgId($CP$)
    output sigVKId($\sigma$)
• msgId($CP$):
    $m||\sigma \leftarrow$ pkeMsgId($CP$)
    output $m$
• dkId($EK$):
    output pkeSKId($EK$)
• ekId($CP$):
    output pkePKId($CP$)
```

Fig. 10: Existential consistency for scheme in Figure 9

$\perp$, then $\mathsf{ekId}(CP) = EK$. Similarly, if $\mathsf{decase\text{-}verify}(VK, DK, CP) \neq \perp$, it holds that $\mathsf{vkId}(CP) = VK$.

□

5.4 Improving the Efficiency of COA-secure CASE

We now show how to improve the efficiency of a COA-secure CASE scheme like the one above, by leveraging the efficiency of a CPA-secure SKE and a collision-resistant hash scheme, analogous to hybrid encryption.

Parameter: Let κ be the security parameter.
Let S = (skeGen, skeEnc, skeDec) be a CPA-secure symmetric-key encryption scheme.
Let H be a collision-resistant hash function family.
Let case = (skGen, vkGen, dkGen, ekGen, encase, decase) be a COA-secure CASE scheme.

COA-secure CASE Scheme case*:

```
– case$^\star$.skGen($1^\kappa$):
    output $SK \leftarrow$ case.skGen($1^\kappa$)
– case$^\star$.vkGen($SK$):
    output $VK \leftarrow$ case.vkGen($SK$)
– case$^\star$.encase($SK, EK, m$):
    sample $k_1 \leftarrow$ skeGen($1^\kappa$), $k_2 \leftarrow \{0,1\}^\kappa$
    $c_1 \leftarrow$ skeEnc($k_1, m$)
    $c_0 \leftarrow$ case.encase($SK, EK, k_1||k_2||H(k_2, c_1)$)
    output $(c_0, c_1)$
– case$^\star$.acc($obj$):
    output case.acc($obj$)
– case$^\star$.dkGen($1^\kappa$):
    output $DK \leftarrow$ case.dkGen($1^\kappa$)
– case$^\star$.ekGen($DK$):
    output $EK \leftarrow$ case.ekGen($DK$)
– case$^\star$.decase($VK, DK, CP$):
    parse $CP$ as $(c_0, c_1)$
    $(m', b) \leftarrow$ case.decase($VK, DK, c_0$)
    parse $m'$ as $k_1||k_2||h$
    $m \leftarrow$ skeDec($k_1, c_1$)
    if $H(k_2, c_1) = h$, output $(m, b)$; else output $\perp$
```

Fig. 11: Efficient COA secure CASE via hybrid encryption

Lemma 7. *If S is a CPA-secure SKE scheme, H is a CRHF scheme and* case *is a COA-secure CASE scheme; then the scheme* case* *in Figure 11 is a COA-secure CASE scheme.*

Please refer to Appendix B.3.1 for the proof.

6 Active Agents Framework

In this section, we present the active agents framework that we develop and use. In particular, it allows the adversary's cryptographic objects to also be modelled as transferable agents. Please refer to Section 6.3 for a summary of the substantial differences between our model and the original model of [2].

6.1 The Model

Agents are interactive Turing machines with tapes for input, output, incoming communication, outgoing communication, randomness and work-space and behave differently depending on the contents of their work-tape (Definition 1). Multiple agents can interact with one another in a *session* (Definition 2).

Ideal World Model (parameterized by a schema Σ) Formally, a schema Σ is described by an agent. The ideal system for a schema Σ consists of two parties Test and User and a fixed third party $\mathcal{B}[\Sigma]$ (for "black-box"). All three parties are probabilistic polynomial time (PPT) interactive turing-machines with a built in security parameter κ. Test and User may be non-uniform. Test receives a *test-bit* b as input and User produces an output bit b'.

$\mathcal{B}[\Sigma]$ maintains two lists of handles $\mathsf{R}^{\mathsf{Test}}$ and $\mathsf{R}^{\mathsf{User}}$, which contain the set of handles belonging to Test and User respectively. Each handle in these lists is mapped to an agent. At the beginning of an execution, both the lists are empty. While Test and User can arbitrarily talk to each other, their interaction with $\mathcal{B}[\Sigma]$ can be summarized as follows:

- **Creating agents.** Test and User can, at any point, request $\mathcal{B}[\Sigma]$ to create a new agent. We describe the process when Test requests creating an agent; the process for User is symmetric.
Test can send a command $(\mathsf{init}, \mathtt{string})$ to $\mathcal{B}[\Sigma]$. $\mathcal{B}[\Sigma]$ then instantiates the agent (with an empty work-tape) and runs it with $\mathtt{string}$ and security parameter as inputs. It assigns a handle number h to the agent (for example, the next available number in the list), gets the agent's configuration $\mathtt{config}$ at the end of the execution and stores $(h, \mathtt{config})$ in $\mathsf{R}^{\mathsf{Test}}$. It finally returns h to Test.

- **Request for Session Execution.** Test or User can, at any point, request an execution of a session. We describe the process when Test requests a session execution; the process for User is symmetric.
Test can send a command $(\mathsf{run}, (h_1, x_1) \ldots, (h_t, x_t))$, where h_i are handles obtained in the list $\mathsf{R}^{\mathsf{Test}}$, and x_i are input strings for the corresponding agents.[14] $\mathcal{B}[\Sigma]$ executes a session with the agents with starting configurations in $\mathsf{R}^{\mathsf{Test}}$, corresponding to the specified handles, with their respective inputs, till it terminates. It obtains a collection of outputs $(y_1, \ldots, y_t)$ and updated configurations of agents. It generates new handles $h'_1, \ldots, h'_t$ corresponding to the updated configurations, adds them to $\mathsf{R}^{\mathsf{Test}}$, and returns $(h'_1, \ldots, h'_t, y_1, \ldots, y_t)$ to Test. If an agent halts in a session, no new handle h'_i is given out for that agent. After a session, the old handles for the agents are not invalidated; so a party can access a configuration of an agent any number of times, by using the same handle.

- **Transferring agents.** Test can send a command $(\mathsf{transfer}, h)$ to $\mathcal{B}[\Sigma]$ upon which it looks up the entry $(h, \mathtt{config})$ from $\mathsf{R}^{\mathsf{Test}}$ (if such an entry exists) and adds an entry $(h', \mathtt{config})$ to $\mathsf{R}^{\mathsf{User}}$, where h' is a new handle, and sends the handle h' to User. Symmetrically, User can transfer an agent to Test using the $\mathsf{transfer}$ command.

We define the random variable $\text{IDEAL}\langle \mathsf{Test}(b) \mid \Sigma \mid \mathsf{User} \rangle$ to be the output of User in an execution of the above system, when Test gets b as the test-bit. We write $\text{IDEAL}\langle \mathsf{Test} \mid \Sigma \mid \mathsf{User} \rangle$ to denote the output when the test-bit is a uniformly random bit. We also define $\text{TIME}\langle \mathsf{Test} \mid \Sigma \mid \mathsf{User} \rangle$ as the maximum number of steps taken by Test (with a random input), $\mathcal{B}[\Sigma]$ and User in total.

In this work, we use the notion of *statistical* hiding in the ideal world as introduced in [3], rather than the original notion used in [2]. (This still results in a security definition that subsumes the traditional definitions, as they involve tests that are statistically hiding.)

Definition 14 ((Statistical) Ideal world hiding). A Test is *s-hiding w.r.t. a schema Σ* if, for all unbounded users User who make at most a polynomial number of queries,

$$\text{IDEAL}\langle \mathsf{Test}(0) \mid \Sigma \mid \mathsf{User} \rangle \approx \text{IDEAL}\langle \mathsf{Test}(1) \mid \Sigma \mid \mathsf{User} \rangle. \qquad \lhd$$

Real World Model (parameterized by a scheme Π) The real world for a schema Σ consists of two parties Test and User that interact with each other arbitrarily, as in the ideal world. However, the third party $\mathcal{B}[\Sigma]$ in the ideal world is replaced by two other parties $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ and $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$ (when User

[14] If a handle appears more than once among $h_1, \ldots, h_t$, it is interpreted as separate agents with the same configuration (but possibly different inputs). In our use-case of CASE, this scenario is not relevant.

is honest), which run the algorithms specified by a *cryptographic scheme* Π. A cryptographic scheme (or simply scheme) Π is a collection of stateless (possibly randomized) algorithms $\Pi.\mathsf{init}, \Pi.\mathsf{run}$ and $\Pi.\mathsf{receive}$, which use a repository Repo to store a mapping from handles to objects. More precisely, the repository is a table with entries of the form (h, obj), where h is a unique handle (say, a non-negative integer) and obj is a cryptographic object (represented, for instance, as a binary string). At the start of an execution, Repo is empty.

If a scheme implementation ($\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ or $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$) receives input $(\mathsf{init}, \mathtt{string})$, then it runs $\Pi.\mathsf{init}(\mathtt{string})$ to obtain an object obj which is added to Repo and a handle is returned. If it receives the command $(\mathsf{run}, (h_1, x_1), \cdots, (h_t, x_t))$, then objects $(\mathit{obj}_1, \ldots, \mathit{obj}_t)$ corresponding to $(h_1, \ldots, h_t)$ are retrieved from Repo and $\Pi.\mathsf{run}((\mathit{obj}_1, x_1), \ldots, (\mathit{obj}_t, x_t))$ is evaluated to obtain $((\mathit{obj}'_1, y_1), \ldots, (\mathit{obj}'_t, y_t))$ where obj'_i are new objects and y_i are output strings; the objects are added to Repo, with a new handle for each, and the new handles, along with the outputs, are returned. (If an obj'_i is empty, then no new handle is added; this corresponds to an agent having halted.)

$\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ and $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$ do not interact with each other, except when one of them receives a $\mathsf{transfer}$ command. If Test sends a command $(\mathsf{transfer}, h)$ to $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$, it looks for an entry (h, obj) in $\mathsf{Repo}_{\mathsf{Test}}$ and sends obj to $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$; on receiving obj from $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$, $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$ will run $\Pi.\mathsf{receive}(\mathit{obj})$ which outputs (a possibly modified) object obj' and if $\mathit{obj}' \neq \bot$, $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$ will add (h', obj') to $\mathsf{Repo}_{\mathsf{User}}$, where h' is a new handle, and outputs h' to User. The process of User transferring an object to Test is symmetric.

When an object is transferred to $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$, the $\mathsf{receive}$ algorithm can be used to accept or reject the object. This check is performed only once, rather than each time the object is used: aside from the inefficiency of repeating this operation, note that the check may be probabilistic and an object may pass sometimes and fail at other times. Since this is not captured in the ideal world, an object is tested and received once and for all.

Note that we *do not* allow Test direct access to the cryptographic objects stored in its repository. In particular, it cannot look up the object associated with a handle in $\mathsf{Repo}_{\mathsf{Test}}$. Also observe that if User is corrupt, which we denote by $\mathcal{A}$, it may not run the scheme it is supposed to. It can run any arbitrary algorithm and send any object of its choice directly to $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$.

We define the random variable $\textsc{real}\langle \mathsf{Test}(b) \mid \Pi \mid \mathcal{A} \rangle$ to be the output of $\mathcal{A}$ in an execution of the above system involving Test with test-bit b, $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$ and $\mathcal{A}$; as before, we omit b from the notation to indicate a random bit. Also, as before, $\textsc{Time}\langle \mathsf{Test} \mid \Pi \mid \mathcal{A} \rangle$ is the maximum number of steps taken by Test (with a random input), $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$ and $\mathcal{A}$ in total.

Definition 15 (Real world hiding). Test is said to be *hiding w.r.t.* Π if $\forall$ PPT party $\mathcal{A}$,

$$\textsc{real}\langle \mathsf{Test}(0) \mid \Pi \mid \mathcal{A} \rangle \approx \textsc{real}\langle \mathsf{Test}(1) \mid \Pi \mid \mathcal{A} \rangle. \qquad \triangleleft$$

6.2 Security Definition

We are ready to present the security definition of a cryptographic agent scheme Π implementing a schema Σ. Below, the *honest real-world user*, corresponding to an ideal-world user User, is defined as the composite program $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}] \circ \mathsf{User}$ as shown in Figure 12.

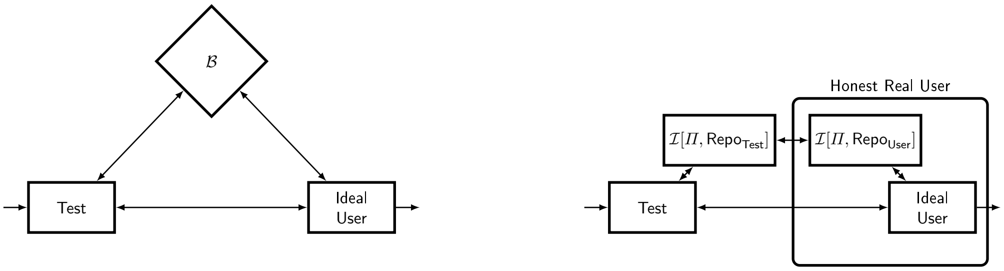
$\mathcal{B}$
Test
Ideal User
Honest Real User
$\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$
$\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{User}}]$
Test
Ideal User

Fig. 12: $\textsc{ideal}$ world (left) and $\textsc{real}$ world with an honest user.

Test Families. We write $\varGamma_{\mathsf{ppt}}$ to denote the family of all PPT Test. We also define a test-family $\varDelta$ as follows: $\mathsf{Test} \in \varDelta$ iff it behaves as follows: every init and run command it sends to $\mathcal{B}[\varSigma]$ is also reported to User. For transfer commands, it picks two handles h_0, h_1 and sends a message $(\mathsf{transfer}, h_0, h_1)$ to User and sends $\mathsf{transfer}[h_b]$ to $\mathcal{B}[\varSigma]$, where b is the test-bit.

Now we define our security notion, $\varDelta$-s-IND-PRE. Note that below the correctness and efficiency requirements are w.r.t. all PPT Test, but indistinguishability-preservation is only for $\mathsf{Test} \in \varDelta$.

Definition 16. A cryptographic agent scheme $\varPi$ is said to be a *$\varDelta$-s-IND-PRE-secure scheme* for a schema $\varSigma$ if the following conditions hold.

- *Correctness.* $\forall$ PPT User, $\forall$ $\mathsf{Test} \in \varGamma_{\mathsf{ppt}}$,

$$\text{IDEAL}\langle \mathsf{Test} \mid \varSigma \mid \mathsf{User}\rangle \approx \text{REAL}\langle \mathsf{Test} \mid \varPi \mid \mathcal{I}[\varPi, \mathsf{Repo}_{\mathsf{User}}] \circ \mathsf{User}\rangle.$$

- *Efficiency.* There exists a polynomial poly s.t. $\forall$ PPT User, $\forall$ $\mathsf{Test} \in \varGamma_{\mathsf{ppt}}$,
$\text{TIME}\langle \mathsf{Test} \mid \varPi \mid \mathcal{I}[\varPi, \mathsf{Repo}_{\mathsf{User}}] \circ \mathsf{User}\rangle \leq \text{poly}(\text{TIME}\langle \mathsf{Test} \mid \varSigma \mid \mathsf{User}\rangle, \kappa)$.
- *(Statistical) Indistinguishability Preservation.* $\forall$ $\mathsf{Test} \in \varDelta$,

$$\mathsf{Test} \text{ is s-hiding w.r.t. } \varSigma \Rightarrow \mathsf{Test} \text{ is hiding w.r.t. } \varPi. \qquad \lhd$$

6.3 A Comparison with the Original Framework

We make several technical extensions to the Cryptographic Agents framework of [2]. For readers familiar with the model of [2], we summarize the important changes below.

• Firstly, we use an execution model that treats Test and User symmetrically, allowing both parties to create and transfer agents in the ideal world (or objects in the real world). This automatically allows for the possibility that the objects in the real world – including secret-keys as well as public-keys and ciphertexts – could be created maliciously (by an actively corrupt User).

• Secondly, we allow the two parties to locally act on the agents in their possession, and only selectively transfer agents to each other. In contrast, in [2], all agents created by Test were automatically transferred to User. This models, in particular, various operations that can be executed by honest parties on objects received from the adversary.

• In [2] encryption-like primitives were modeled so that only a single key-agent existed in the system. In our formulation, we model the agents in an encryption scheme as evolving from a secret-key agent, which is initialized using a randomized initialization step. Such an initialization, which was not part of the original framework, allows us to model multiple keys in the system in a sound manner (by including random tags generated during initialization, which are not controlled by Test or User). This would correspond to randomized key objects in the real world.

• In our new model, we introduce a mechanism to "vet" an object before accepting it. This opens up new avenues in constructing schemes that securely implement various schemas.

• Following [3], we slightly relax the security definitions in [2] so that computational indistinguishability is required to hold in the real world only if statistical indistinguishability holds in the ideal world (against a computationally unbounded adversary). This relaxation will be crucial later in exploiting existential consistency of COA security to argue that COA security implies a $\varDelta$-s-IND-PRE secure implementation of the CASE schema.

6.4 Impossibility of $\varGamma_{\mathsf{ppt}}$-IND-PRE Security

In Definition 16, security was defined w.r.t. a restricted class of tests $\varDelta$ that report $(\mathsf{transfer}, h_0, h_1)$ to User and transfers h_b to User via $\mathcal{B}[\varSigma]$ s.t. the test bit b must remain hidden from User. One can consider a more general general class $\varGamma_{\mathsf{ppt}}$ of all PPT Tests. In [2] it was pointed out that obfuscation does not have a $\varGamma_{\mathsf{ppt}}$-IND-PRE secure implementation. Here we point out that, even in the original model of [2] (i.e., without our extension), public-key encryption – and even symmetric-key encryption – cannot have a $\varGamma_{\mathsf{ppt}}$-IND-PRE secure implementation.

We point out that impossibility of $\mathit{\Gamma}_{\mathsf{ppt}}$-IND-PRE security implies impossibility of simulation-based security too (as the latter implies $\mathit{\Gamma}_{\mathsf{ppt}}$-IND-PRE security).

Schema $\mathit{\Sigma}_{\mathsf{SKE}}$ for SKE

$\mathit{\Sigma}_{\mathsf{SKE}}$ consists of an agent which behaves as follows.

- **Initialization.** When run with an empty work-tape and input κ, it samples $\mathit{sk\text{-}tag} \leftarrow \{0,1\}^{\kappa}$ and records ($\mathtt{sk}$, $\mathit{sk\text{-}tag}$) on its work-tape.
- **Encrypting a message.** When a key agent with work-tape contents ($\mathtt{sk}$, $\mathit{sk\text{-}tag}$) is run with input ($\mathtt{enc}$, m), it samples $\mathit{ct\text{-}tag} \leftarrow \{0,1\}^{\kappa}$ and updates its work-tape as ($\mathtt{ct}$, m, $\mathit{sk\text{-}tag}$, $\mathit{ct\text{-}tag}$).
- **Decrypting a message.** When two agents are run in a session with input $\mathtt{dec}$:
  - the key agent with work-tape contents ($\mathtt{sk}$, $\mathit{sk\text{-}tag}$) accepts ($\mathtt{ct}$, m, $\mathit{sk\text{-}tag}$’, $\mathit{ct\text{-}tag}$) from the other agent. It outputs m if $\mathit{sk\text{-}tag} = \mathit{sk\text{-}tag}'$, else it outputs $\perp$.
  - the ciphertext agent with work-tape contents ($\mathtt{ct}$, m, $\mathit{sk\text{-}tag}$, $\mathit{ct\text{-}tag}$) sends its work-tape contents to the first agent.
- **Type of agent:** When run with input $\mathtt{type}$, it behaves as follows:
  - if the work-tape has $(\mathtt{sk}, \mathit{sk\text{-}tag})$, output DK.
  - if the work-tape has $(\mathtt{ct}, m, \mathit{sk\text{-}tag}, \mathit{ct\text{-}tag})$, output CT.
- **Comparing agents:** When two agents are run in a session with input $\mathtt{compare}$, the second agent sends the contents of it's work tape to the first agent. The first agent waits for a message from the other agent in the session and if the message is identical to its own tape's contents, it outputs true, otherwise it outputs false.

Fig. 13: Schema for Symmetric-Key Encryption.

Lemma 8. *There does not exist a $\mathit{\Gamma}_{\mathsf{ppt}}$-IND-PRE secure scheme for a schema corresponding to* SKE*.*

Proof: Consider a simple schema for SKE be as in Figure 13. Suppose we are given a candidate scheme $\mathit{\Pi}_{\mathsf{SKE}}$ that purportedly is a $\mathit{\Gamma}_{\mathsf{ppt}}$-IND-PRE secure implementation of $\mathit{\Sigma}_{\mathsf{SKE}}$. Let $\ell(\kappa)$ be an upper bound on the key-length in this scheme.

Then, consider $\mathsf{Test} \in \mathit{\Gamma}_{\mathsf{ppt}}$ that behaves as follows. It initializes a key agent and gets a handle h_{SK}. It uniformly picks $m \leftarrow \{0,1\}^{\ell(\kappa)+\kappa}$, creates a ciphertext agent using the key agent and message m, gets handle h_{CT} and automatically transfers it to User. Then, it expects User to send back a (polynomially long) program σ. After receiving σ, Test transfers the key agent h_{SK}. Next, it expects user to send back an $\ell(\kappa)$ bit input x for σ. If x is such that $\sigma(x) = m$, Test sends the test-bit b to the User and otherwise it halts.

In the ideal world, User cannot access m until after it sends σ, and hence information-theoretically it is unlikely that $\sigma(x) = m$, since $|m| \gg |x|$. However, in the real execution with $\mathit{\Pi}_{\mathsf{SKE}}$, an adversary can set σ to be a program which will take as input a decryption key, and use it to decrypt the ciphertexts that the adversary received in the first step (which are hardwired into σ). Further, on receiving the decryption key, the adversary sends it to Test as x, so that, by the requisite correctness properties of $\mathit{\Pi}_{\mathsf{SKE}}$, with high probability, $\sigma(x) = m$ and learns b exactly. This violates indistinguishability preservation. □

Extensions. In the above attack, Test is hiding even against a computationally unbounded adversary in the ideal world. As such, the impossibility extends to the weaker definition of $\mathit{\Gamma}_{\mathsf{ppt}}$-$s$-IND-PRE as well.

Also note that simulation-based security implies $\mathit{\Gamma}_{\mathsf{ppt}}$-IND-PRE security (and the weaker security of unbounded simulation implies $\mathit{\Gamma}_{\mathsf{ppt}}$-$s$-IND-PRE security). Hence the above attack rules out (unbounded) simulation-based security for SKE if the decryption key can be transferred.

We note that simulation-based security against key exposure is possible for one-time encryption [16]. While this suffices in the context of secure computation protocols, this is unsatisfactory for PKE wherein the same secret-key should allow decrypting an *a priori* unbounded number of ciphertexts (possibly sent by different parties).

7 CASE in the Active Agents Framework

Figure 14 gives a simple and intuitive schema Σ_{case} for CASE in the active agents framework. At a high level, we want to capture the following properties:

- Public Keys: the verification key agent h_{VK} should be fixed and computable given the signing key agent h_{SK}. Similarly, h_{EK} should be fixed and computable given the decryption key agent h_{DK}.
- Encasing: to encase a message m, a signing key agent h_{SK}, an encryption key agent h_{EK} are required, and give a case-packet agent h_{CP}.
- Decasing: to get the message, a case-packet agent h_{CP}, corresponding verification key agent h_{VK} and decryption key agent h_{DK} are required. We allow partial decryption (specifically, extraction) of the message given only the decryption key agent h_{DK}.
- Randomized Agents: we want agents h_{SK}, h_{DK} and h_{CP} to be randomized. In particular, this ensures that encasing the same message again results in a fresh agent $h_{CP'}$ that does not compare with h_{CP}. To enable this, we use random tags.

Recall that, the black-box $\mathcal{B}[\Sigma_{\mathsf{case}}]$ automatically creates new agents and gives out handles for them at the end of any session.

Schema Σ_{case}:
Σ_{case} consists of an agent which behaves as follows.

- **Initialization.** When run with an empty work-tape and input ($\texttt{key-type}$, κ):
  - if $\texttt{key-type} = \textsc{sk}$, it samples $\textit{sk-tag} \leftarrow \{0,1\}^\kappa$ and records ($\texttt{sk}$, *sk-tag*) on its work-tape
  - if $\texttt{key-type} = \textsc{dk}$, it samples $\textit{dk-tag} \leftarrow \{0,1\}^\kappa$ and records ($\texttt{dk}$, *dk-tag*) on its work-tape
- **Deriving a verification-key.** When run with ($\texttt{sk}$, *sk-tag*) on its work-tape and input $\texttt{vkGen}$, it updates its work-tape as ($\texttt{vk}$, *sk-tag*)
- **Deriving an encryption-key.** When run with ($\texttt{dk}$, *dk-tag*) on its work-tape and input $\texttt{ekGen}$, it updates its work-tape as ($\texttt{ek}$, *dk-tag*)
- **Encasing a message.** When two agents are run in a session with input ($\texttt{encase}$, m):
  - if work-tape of agent has ($\texttt{sk}$, *sk-tag*), it receives ($\texttt{ek}$, *dk-tag*) from the other agent, samples $\textit{cp-tag} \leftarrow \{0,1\}^\kappa$ and updates its work-tape as ($\texttt{cp}$, m, *sk-tag*, *dk-tag*, *cp-tag*)
  - if work-tape of agent has ($\texttt{ek}$, *dk-tag*), it sends it's work-tape contents to the first agent
- **Decasing a message.** When three agents are run in a session with input $\texttt{decase-verify}$:
  - if work-tape of agent has ($\texttt{dk}$, *dk-tag*), it accepts ($\texttt{vk}$, $\textit{sk-tag}^*$) and ($\texttt{cp}$, m, *sk-tag*, $\textit{dk-tag}^*$, *cp-tag*) from the other agents. It outputs $\perp$ if $\textit{dk-tag} \neq \textit{dk-tag}^*$, outputs $(m, 1)$ if $\textit{sk-tag} = \textit{sk-tag}^*$ and $(m, 0)$ else.
  - if work-tape of agent has ($\texttt{vk}$, *sk-tag*), it sends it to first agent
  - if work-tape of agent has ($\texttt{cp}$, m, *sk-tag*, *dk-tag*, *cp-tag*), it sends it to first agent.
- **Extracting the message.** When two agents are run in a session with input $\texttt{decase-msg}$:
  - if work-tape of agent has ($\texttt{dk}$, *dk-tag*), it accepts ($\texttt{cp}$, m, *sk-tag*, $\textit{dk-tag}^*$, *cp-tag*) from the other agent. It outputs $\perp$ if $\textit{dk-tag} \neq \textit{dk-tag}^*$, else outputs m.
  - if work-tape of agent has ($\texttt{cp}$, m, *sk-tag*, *dk-tag*, *cp-tag*), it sends it to first agent.
- **Type of agent:** When run with input $\texttt{type}$, it behaves as follows:
  - if the work-tape has ($\texttt{sk}$, *sk-tag*), output $\textsc{sk}$.
  - if the work-tape has ($\texttt{vk}$, *sk-tag*), output $\textsc{vk}$.
  - if the work-tape has ($\texttt{dk}$, *dk-tag*), output $\textsc{dk}$.
  - if the work-tape has ($\texttt{ek}$, *dk-tag*), output $\textsc{ek}$.
  - if the work-tape has ($\texttt{cp}, m$, *sk-tag*, *dk-tag*, *cp-tag*), output $\textsc{cp}$.

- **Comparing agents:** When two agents are run in a session with input `compare`, the second agent sends the contents of it's work tape to the first agent. The first agent waits for a message from the other agent in the session and if the message is identical to its own tape's contents, it outputs true, otherwise it outputs false.

Fig. 14: Schema Σ_{case} for CASE.

In Figure 15, we build a Δ-s-IND-PRE secure scheme Π_{case} for CASE from any COA-secure scheme for CASE.

Scheme Π_{case}:
Let $\mathsf{case} = (\mathsf{skGen}, \mathsf{vkGen}, \mathsf{dkGen}, \mathsf{ekGen}, \mathsf{encase}, \mathsf{decase}, \mathsf{acc})$ be a COA-secure CASE scheme.

- **Initialization.** $\Pi_{\mathsf{case}}.\mathsf{init}$ (`key-type`, κ)
  - if `key-type` = SK: sample $SK \leftarrow \mathsf{case.skGen}(1^\kappa)$ and output SK
  - if `key-type` = DK: sample $DK \leftarrow \mathsf{case.dkGen}(1^\kappa)$ and output DK
- **Deriving a verification key.** $\Pi_{\mathsf{case}}.\mathsf{run}$ (obj,`vkGen`) outputs $(\mathsf{case.vkGen}(obj), \perp)$.
- **Deriving an encryption key.** $\Pi_{\mathsf{case}}.\mathsf{run}$ (obj,`ekGen`) outputs $(\mathsf{case.ekGen}(obj), \perp)$.
- **Encasing a message.** $\Pi_{\mathsf{case}}.\mathsf{run}$ ((obj_{sk},(`encase`,m)),(obj_{pk},(`encase`,m))) outputs $((\mathsf{case.encase}(obj_{sk}, obj_{pk}, m), \perp), (\perp, \perp))$.
- **Decasing a message.** $\Pi_{\mathsf{case}}.\mathsf{run}$ ((obj_{dk},`decase-verify`),(obj_{vk},`decase-verify`), (obj,`decase-verify`)) outputs $((\perp, \mathsf{case.decase}(obj_{dk}, obj_{vk}, obj)), (\perp, \perp), (\perp, \perp))$.
- **Extracting a message.** $\Pi_{\mathsf{case}}.\mathsf{run}$ ((obj_{dk},`decase-msg`), (obj,`decase-msg`)) outputs $((\perp, \mathsf{case.decase\text{-}msg}(obj_{dk}, obj)), (\perp, \perp), (\perp, \perp))$.
- **Type of agent.** $\Pi_{\mathsf{case}}.\mathsf{run}$ (obj, `type`) outputs $(\perp, \mathsf{case.acc}(obj))$.
- **Comparing agents:** $\Pi_{\mathsf{case}}.\mathsf{run}$ ((obj_1,`compare`),(obj_2,`compare`)) outputs $((\perp, \mathsf{true}), (\perp, \perp))$ if $obj_1 = obj_2$ and $((\perp, \mathsf{false}), (\perp, \perp))$ otherwise.
- **Receiving agents:** $\Pi_{\mathsf{case}}.\mathsf{receive}$ (obj) outputs obj if $\mathsf{case.acc}(obj) \neq \perp$ else outputs $\perp$.

Fig. 15: Schema Π_{case} for CASE.

We now prove Theorem 1, i.e., that a COA secure CASE scheme implies a Δ-s-IND-PRE secure implementation of Σ_{case}.

Theorem 1 (Restated). *A Δ-s-IND-PRE secure implementation of Σ_{case} exists if a COA secure CASE scheme exists.*

Proof sketch: We show that the Π_{case} in Figure 15 is a Δ-s-IND-PRE secure implementation of Σ_{case}. Given any $\mathsf{Test} \in \Delta$ that is hiding w.r.t. Σ_{case}, we need to argue that for all PPT adversary $\mathcal{A}$,

$$\textsc{real}\langle \mathsf{Test}(0) \mid \Pi \mid \mathcal{A} \rangle \approx \textsc{real}\langle \mathsf{Test}(1) \mid \Pi \mid \mathcal{A} \rangle.$$

The proof uses guarantees such as unforgeability, total hiding and encasing resistance from the underlying COA-Secure CASE scheme case, along with the statistical guarantees of existential consistency, given in terms of computationally unbounded algorithms like $\mathsf{case.skId}$, $\mathsf{case.ekId}$ and $\mathsf{case.msgId}$. The argument uses a sequence of hybrid random variables to prove Δ-s-IND-PRE security, H_i for $i = 0$ to 7:

H_0: $\textsc{real}\langle \mathsf{Test}(0) \mid \Pi_{\mathsf{case}} \mid \mathcal{A} \rangle$	H_7: $\textsc{real}\langle \mathsf{Test}(1) \mid \Pi_{\mathsf{case}} \mid \mathcal{A} \rangle$
H_1: $\textsc{ideal}\langle \mathsf{Test}(0) \mid \Sigma^{\ddagger}_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^{\dagger}_0 \circ \mathcal{A} \rangle$	H_6: $\textsc{ideal}\langle \mathsf{Test}(1) \mid \Sigma^{\ddagger}_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^{\dagger}_1 \circ \mathcal{A} \rangle$
H_2: $\textsc{ideal}\langle \mathsf{Test}(0) \mid \Sigma^{\ddagger}_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^{\ddagger} \circ \mathcal{A} \rangle$	H_5: $\textsc{ideal}\langle \mathsf{Test}(1) \mid \Sigma^{\ddagger}_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^{\ddagger} \circ \mathcal{A} \rangle$
H_3: $\textsc{ideal}\langle \mathsf{Test}(0) \mid \Sigma_{\mathsf{case}} \mid \mathcal{S}^{*} \circ \mathcal{S}^{\ddagger} \circ \mathcal{A} \rangle$	H_4: $\textsc{ideal}\langle \mathsf{Test}(1) \mid \Sigma_{\mathsf{case}} \mid \mathcal{S}^{*} \circ \mathcal{S}^{\ddagger} \circ \mathcal{A} \rangle$

Hybrids H_0 and H_7 correspond to the output of $\mathcal{A}$ in the real world with test bits $b = 0$ and $b = 1$ respectively. The simulators $\mathcal{S}_b^\dagger$ (for $b \in \{0,1\}$), $\mathcal{S}^\ddagger$ are computationally bounded while $\mathcal{S}^* \circ \mathcal{S}^\ddagger$ is a computationally unbounded simulator due to $\mathcal{S}^*$.

When $\mathsf{Test} \in \Delta$ is s-hiding w.r.t. Σ_{case}, we show:

1. Firstly, $\mathsf{H}_3 \approx \mathsf{H}_4$, even though they involve a computationally unbounded simulator $\mathcal{S}^*$ (by definition of s-hiding of Test).
2. We rely on the existential consistency of the underlying signature scheme to show that $\mathsf{H}_2 \approx \mathsf{H}_3$ and (symmetrically) $\mathsf{H}_4 \approx \mathsf{H}_5$.
3. We use the augmented security guarantees of the underlying CASE scheme to establish that $\mathsf{H}_1 \approx \mathsf{H}_2$ and (symmetrically) $\mathsf{H}_5 \approx \mathsf{H}_6$.
4. Finally, we argue that $\mathsf{H}_0 \approx \mathsf{H}_1$ and $\mathsf{H}_6 \approx \mathsf{H}_7$. This follows from the construction of $\mathcal{S}_0^\dagger$ and $\mathcal{S}_1^\dagger$, conditioned on some "bad events" not occurring. We prove that these bad events occur with negligible probability using the guarantees - strong-unforgeability, total hiding, sender anonymity, unpredictability and encasing resistance from the underlying COA-Secure CASE scheme case (see Lemma 10) and statistical guarantees of sampling from a uniform distribution (sampling of tags in Σ_{case} and $\Sigma^\ddagger_{\Pi_{\mathsf{case}}}$).

Together, these steps show that any $\mathsf{Test} \in \Delta$ that is s-hiding w.r.t. Σ_{case} is also hiding w.r.t. Σ_{case}. Please refer to Appendix C for the full proof.

□

References

[1] Michel Abdalla, Mihir Bellare, and Gregory Neven. "Robust Encryption". In: *TCC*. Ed. by Daniele Micciancio. 2010.

[2] Shashank Agrawal, Shweta Agrawal, and Manoj Prabhakaran. "Cryptographic Agents: Towards a Unified Theory of Computing on Encrypted Data". In: *EUROCRYPT*. Ed. by Elisabeth Oswald and Marc Fischlin. 2015.

[3] Shashank Agrawal, Manoj Prabhakaran, and Ching-Hua Yu. "Virtual Grey-Boxes Beyond Obfuscation: A Statistical Security Notion for Cryptographic Agents". In: *TCC 2016-B*. 2016.

[4] Joël Alwen et al. "Analysing the HPKE standard". In: *EUROCRYPT*. Springer. 2021, pp. 87–116.

[5] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. "On the security of joint signature and encryption". In: *EUROCRYPT*. Springer. 2002, pp. 83–107.

[6] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. "On the security of joint signature and encryption". In: *EUROCRYPT*. Springer. 2002, pp. 83–107.

[7] Christian Badertscher, Fabio Banfi, and Ueli Maurer. "A constructive perspective on signcryption security". In: *Security and Cryptography for Networks: 11th International Conference, SCN 2018, Amalfi, Italy, September 5–7, 2018, Proceedings 11*. Springer. 2018, pp. 102–120.

[8] Joonsang Baek, Ron Steinfeld, and Yuliang Zheng. "Formal proofs for the security of signcryption". In: *PKC*. Springer. 2002, pp. 80–98.

[9] Manuel Barbosa and Pooya Farshim. "Certificateless signcryption". In: *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. 2008, pp. 369–372.

[10] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz. "Subtleties in the Definition of IND-CCA: When and How Should Challenge Decryption Be Disallowed?" In: *J. Cryptology* 28.1 (2015), pp. 29–48. DOI: 10.1007/s00145-013-9167-4. URL: https://doi.org/10.1007/s00145-013-9167-4.

[11] Mihir Bellare and Igors Stepanovs. "Security under message-derived keys: Signcryption in iMessage". In: *EUROCRYPT*. Springer. 2020, pp. 507–537.

[12] Mihir Bellare et al. "Key-privacy in public-key encryption". In: *ASIACRYPT*. Springer. 2001, pp. 566–582.

[13] Tor E Bjørstad and Alexander W Dent. "Building better signcryption schemes with tag-KEMs". In: *PKC*. Springer. 2006, pp. 491–507.

[14] Xavier Boyen. "Multipurpose identity-based signcryption: A swiss army knife for identity-based cryptography". In: *CRYPTO*. Springer. 2003, pp. 383–399.

[15] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*. FOCS '01. 2001.

[16] Ran Canetti et al. "Adaptively Secure Multi-Party Computation". In: *STOC*. 1996, pp. 639–648.

[17] Ronald Cramer and Victor Shoup. "A Practical Public Key Cryptosystem Provably Secure against Adaptive Chosen Ciphertext Attack". In: *CRYPTO*. Vol. 1462. Lecture Notes in Computer Science. Springer, 1998.

[18] Pratish Datta, Ratna Dutta, and Sourav Mukhopadhyay. "Compact attribute-based encryption and signcryption for general circuits from multilinear maps". In: *Progress in Cryptology–INDOCRYPT 2015*. Springer. 2015, pp. 3–24.

[19] Pratish Datta, Ratna Dutta, and Sourav Mukhopadhyay. "Functional signcryption: notion, construction, and applications". In: *Provable Security: 9th International Conference, ProvSec 2015, Kanazawa, Japan, November 24-26, 2015, Proceedings 9*. Springer. 2015, pp. 268–288.

[20] Alexander W Dent. "Hybrid signcryption schemes with insider security". In: *Information Security and Privacy: 10th Australasian Conference, ACISP 2005, Brisbane, Australia, July 4-6, 2005. Proceedings 10*. Springer. 2005, pp. 253–266.

[21] Danny Dolev, Cynthia Dwork, and Moni Naor. "Nonmalleable cryptography". In: *SICOMP* 30.2 (2000). Preliminary version in STOC 1991., 391–437 (electronic). ISSN: 1095-7111.

[22] Pooya Farshim et al. "Robust Encryption, Revisited". In: *PKC*. Ed. by Kaoru Kurosawa and Goichiro Hanaoka. 2013.

[23] Martin Gagné, Shivaramakrishnan Narayan, and Reihaneh Safavi-Naini. "Threshold attribute-based signcryption". In: *Security and Cryptography for Networks: 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings 7*. Springer. 2010, pp. 154–171.

[24] Kristian Gjøsteen and Lillian Kråkmo. "Universally composable signcryption". In: *Public Key Infrastructure: 4th European PKI Workshop: Theory and Practice, EuroPKI 2007*. Springer. 2007, pp. 346–353.

[25] Oded Goldreich. *Foundations of Cryptography: Volume 1*. New York, NY, USA: Cambridge University Press, 2006.

[26] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption". In: *JCSS* 28.2 (Apr. 1984). Preliminary version appeared in STOC' 82., pp. 270–299.

[27] Benoit Libert and Jean-Jacques Quisquater. "A new identity based signcryption scheme from pairings". In: *Proceedings 2003 IEEE Information Theory Workshop (Cat. No. 03EX674)*. IEEE. 2003, pp. 155–158.

[28] Benoît Libert and Jean-Jacques Quisquater. "Efficient signcryption with key privacy from gap Diffie-Hellman groups". In: *Public Key Cryptography*. Vol. 2947. Springer. 2004, pp. 187–200.

[29] Joseph K Liu, Joonsang Baek, and Jianying Zhou. "Online/Offline Identity-Based Signcryption Revisited." In: *Inscrypt*. Springer. 2010, pp. 36–51.

[30] John Malone-Lee. "Identity-based signcryption". In: *Cryptology ePrint Archive* (2002).

[31] Ueli Maurer. "Constructive Cryptography - A New Paradigm for Security Definitions and Proofs". In: *Theory of Security and Applications - Joint Workshop, TOSCA 2011*. 2011, pp. 33–56. DOI: 10.1007/978-3-642-27375-9_3. URL: https://doi.org/10.1007/978-3-642-27375-9_3.

[32] Ueli Maurer, Christopher Portmann, and Guilherme Rito. "Multi-designated receiver signed public key encryption". In: *EUROCRYPT*. Springer. 2022, pp. 644–673.

[33] Payman Mohassel. "A Closer Look at Anonymity and Robustness in Encryption Schemes". In: *ASIACRYPT*. 2010, pp. 501–518.

[34] Moni Naor and Moti Yung. "Public-key Cryptosystems Provably Secure against Chosen Ciphertext Attacks". In: *STOC*. 1990, pp. 427–437.

[35] Jesper Buus Nielsen. "Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case". In: *CRYPTO 2002, Proceedings*. 2002, pp. 111–126. URL: https://doi.org/10.1007/3-540-45708-9_8.

[36] Kenneth G Paterson et al. "On the joint security of encryption and signature, revisited". In: *ASIACRYPT*. Springer. 2011, pp. 161–178.

[37] Charles Rackoff and Daniel R. Simon. "Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack". In: *Advances in Cryptology - CRYPTO '91, Proceedings*. 1991, pp. 433–444.

[38] S Sharmila Deva Selvi, S Sree Vivek, and C Pandu Rangan. "Identity based public verifiable signcryption scheme". In: *Provable Security: 4th International Conference, ProvSec 2010, Malacca, Malaysia, October 13-15, 2010. Proceedings 4*. Springer. 2010, pp. 244–260.

[39] S Sharmila Deva Selvi et al. "ID based signcryption scheme in standard model". In: *Provable Security: 6th International Conference, ProvSec 2012, Chengdu, China, September 26-28, 2012. Proceedings 6*. Springer. 2012, pp. 35–52.

[40] Ron Steinfeld and Yuliang Zheng. "A signcryption scheme based on integer factorization". In: *ISW*. Vol. 1975. 2000, pp. 308–322.

[41] Yang Wang et al. "Relations among privacy notions for signcryption and key invisible "sign-then-encrypt"". In: *Information Security and Privacy: 18th Australasian Conference, ACISP 2013, Brisbane, Australia, July 1-3, 2013. Proceedings 18*. Springer. 2013, pp. 187–202.

[42] Moti Yung, Alexander Dent, and Yuliang Zheng. *Practical signcryption*. Springer Science & Business Media, 2010.

[43] Yuliang Zheng. "Digital signcryption or how to achieve cost (signature & encryption)≪ cost (signature)+ cost (encryption)". In: *CRYPTO*. Springer. 1997, pp. 165–179.

[44] Yuliang Zheng and Hideki Imai. "How to construct efficient signcryption schemes on elliptic curves". In: *Information processing letters* 68.5 (1998), pp. 227–233.

Appendix

A Details omitted from Section 4

A.1 Encasing Resistance

Lemma 1 (Restated)**.** *Any COA-secure CASE scheme satisfies encasing-resistance.*

Proof: We first prove that the following are negligible in κ.

$$\max_{DK^* \in \mathcal{DK}} \Pr_{DK \leftarrow \mathsf{dkGen}(1^\kappa)} \Big[DK = DK^* \Big] \tag{1}$$

$$\max_{CP^* \in \mathcal{CP}} \Pr_{DK \leftarrow \mathsf{dkGen}(1^\kappa)} \Big[\mathsf{decase\text{-}msg}(DK, CP^*) \neq \perp \Big] \tag{2}$$

It is easy to see that (1) must be negligible from the total hiding of the CASE scheme. Indeed, otherwise an adversary in experiment distinguish-sans-DK will find that, with non-negligible probability, at least one of the two keys EK_0, EK_1 (in fact, even both) corresponds to a fixed decryption key DK^* that maximizes the probability in (1). In that case, the bit b^* can be learnt exactly, by choosing $m_0 \neq m_1$[15] for the challenge, and $\mathsf{decase\text{-}msg}(DK^*, CP^*)$ has different outcomes depending on the bit b^* in the experiment: If both $EK_0 = EK_1 = \mathsf{ekGen}(DK^*)$, then the outcome will be m_{b^*}; if only $EK_b = \mathsf{ekGen}(DK^*)$, the outcome will be m_b when $b = b^*$ and $\perp$ otherwise.

To upper bound (2), recall that from existential consistency of the CASE scheme, we have

$$\mathsf{decase\text{-}msg}(DK, CP^*) \neq \perp \Rightarrow DK = \mathsf{dkId}(\mathsf{ekId}(CP^*)).$$

So, $\forall CP^* \in \mathcal{CP}$, we have

$$\Pr_{DK \leftarrow \mathsf{dkGen}(1^\kappa)} [\mathsf{decase\text{-}msg}(DK, CP^*) \neq \perp] \leq \Pr_{DK \leftarrow \mathsf{dkGen}(1^\kappa)} [DK = \mathsf{dkId}(\mathsf{ekId}(CP^*))].$$

[15] This assumes $|\mathcal{M}| > 1$. Alternately, $SK_0 \neq SK_1$ can be used, and decase-verify can be used instead of decase-msg with a similar effect.

But the latter probability is negligible from (1).

Finally, to prove our lemma, consider a hybrid experiment derived from $\mathsf{encase\text{-}sans\text{-}EK}$, in which the adversary is given oracle access using encryption keys (DK', EK') generated independently. From the total hiding requirement on CASE, the two hybrids are indistinguishable. Secondly, in this hybrid since DK used to determine the outcome of the experiment is independent of the rest of the experiment, we can use the bound of (2) to conclude that the experiment outputs 1 with negligible probability. Thus, it holds in the original experiment as well. □

A.2 Augmented Security

We define an augmented security experiment where an adversary adaptively attacks either the sender anonymity or the receiver anonymity of the CASE scheme. The experiment maintains the following types of lists to ensure that it never sends any object to the adversary that would reveal the challenge bit:

– T_t lists: these contain objects of token type t in the experiment
– R list: this contains pairs (t, i) s.t. $T_t[i]$ was transferred to/from $\mathcal{A}$

Experiment aug

– Receive n from $\mathcal{A}$. Initialise the following lists:

$$T_{\textsc{sk}} := \{(i, SK_i) \mid i \in [n], SK_i \leftarrow \mathsf{skGen}(1^\kappa)\} \qquad T_{\textsc{vk}} := \{(i, VK_i) \mid i \in [n], VK_i \leftarrow \mathsf{vkGen}(SK_i)\}$$
$$T_{\textsc{dk}} := \{(i, DK_i) \mid i \in [n], DK_i \leftarrow \mathsf{dkGen}(1^\kappa)\} \qquad T_{\textsc{ek}} := \{(i, EK_i) \mid i \in [n], EK_i \leftarrow \mathsf{ekGen}(DK_i)\}$$

and $T_{\textsc{cp}} = \{\}$, $R = \{\}$

– Query phase - receive (polynomial number of) either of the following queries:
- Adversarial objects: for each (i, obj) received from $\mathcal{A}$, let $t = \mathsf{acc}(obj)$; if $t \in \{\textsc{sk}, \textsc{vk}, \textsc{dk}, \textsc{ek}, \textsc{cp}\}$ and $i > n$, add (i, obj) to T_t and add (t, i) to R.
- Key Queries: for each (t, i) received from $\mathcal{A}$, if $t \in \{\textsc{sk}, \textsc{vk}, \textsc{dk}, \textsc{ek}\}$ and $i < n$, then send $T_t[i]$ to $\mathcal{A}$ and add (t, i) to R.
- Encryption Query: for each (k, l, m) received from $\mathcal{A}$, send $\mathsf{encase}(T_{\textsc{sk}}[k],\ T_{\textsc{ek}}[l],\ m)$ to $\mathcal{A}$.
- Decryption Query: for each (k, l, c) received from $\mathcal{A}$; if $k = 0$, send $\mathsf{decase}(\perp,\ T_{\textsc{dk}}[l],\ T_{\textsc{cp}}[c])$; else, send $\mathsf{decase}(T_{\textsc{vk}}[k],\ T_{\textsc{dk}}[l],\ T_{\textsc{cp}}[c])$ to $\mathcal{A}$.

– Challenge phase - sample a bit $b \leftarrow \{0, 1\}$, receive either of the following challenges:
- key-challenge: receive (t, i_0, i_1) from $\mathcal{A}$; abort if any of the following hold:
  ∗ if $t \in \{\textsc{sk}, \textsc{vk}\}$, $i_0 \neq i_1$, $\exists b'$ s.t. $(\textsc{sk}, i_{b'}) \in R$ or $(\textsc{vk}, i_{b'}) \in R$
  ∗ if $t \in \{\textsc{dk}, \textsc{ek}\}$, $i_0 \neq i_1$, $\exists b'$ s.t. $(\textsc{dk}, i_{b'}) \in R$ or $(\textsc{ek}, i_{b'}) \in R$
  else, send $T_t[i_b]$ to $\mathcal{A}$
- case-packet-challenge: receive $(k_0, k_1, l_0, l_1, m_0, m_1)$ from $\mathcal{A}$. $\forall b' \in \{0, 1\}$, let $SK_{b'} = T_{\textsc{sk}}[k_{b'}]$ and $EK_{b'} = T_{\textsc{ek}}[l_{b'}]$; abort if any of the following hold:
  ∗ if $l_0 \neq l_1$ and $\exists b'$ s.t. $(l_{b'} > n$ or $(\textsc{dk}, l_{b'}) \in R)$
  ∗ if $l_0 = l_1 = l$ and $(\textsc{dk}, l) \in R$ and $m_0 \neq m_1$
  ∗ if $l_0 = l_1 = l$ and $(\textsc{dk}, l) \in R$ and $k_0 \neq k_1$ and $\exists b'$ s.t. $(\textsc{sk}, k_{b'}) \in R$ or $(\textsc{vk}, k_{b'}) \in R$
  else, send $CP = \mathsf{encase}(SK_b, EK_b, m_b)$ to $\mathcal{A}$.

– Query phase - receive (polynomial number of) either of the following queries:
  - Adversarial objects: for each (i, obj) received from $\mathcal{A}$, let $t = \mathsf{acc}(obj)$; if $t \in \{\textsc{sk}, \textsc{vk}, \textsc{dk}, \textsc{ek}, \textsc{cp}\}$ and $i > n$, add (i, obj) to T_t and add (t, i) to R.
  - Key Queries: for each (t, i) received from $\mathcal{A}$, abort if $t > n$

  ▷ abort if $\mathcal{A}$ sent a key-challenge (t^*, i_0^*, i_1^*) in the previous phase and
    - ∗ if $t \in \{\textsc{sk}, \textsc{vk}\}$, $i \in \{i_0^*, i_1^*\}$ and $t^* \in \{\textsc{sk}, \textsc{vk}\}$
    - ∗ if $t \in \{\textsc{dk}, \textsc{ek}\}$, $i \in \{i_0^*, i_1^*\}$ and $t^* \in \{\textsc{dk}, \textsc{ek}\}$

  ▷ abort if $\mathcal{A}$ sent a case-packet-challenge $(k_0^*, k_1^*, l_0^*, l_1^*, m_0^*, m_1^*)$ in the previous phase, got case-packet-response CP^* and
    - ∗ if $t = \textsc{dk}$, $i \in \{l_0^*, l_1^*\}$ and $l_0^* \neq l_1^*$ or $m_0^* \neq m_1^*$ or $\Big(k_0^* \neq k_1^*$ and $\exists b' \in \{0, 1\}$, s.t. $(\textsc{sk}, k_{b'}^*) \in R$ or $(\textsc{vk}, k_{b'}^*) \in R\Big)$
    - ∗ if $t \in \{\textsc{sk}, \textsc{vk}\}$, $i \in \{k_0^*, k_1^*\}$, $l_0^* = l_1^* = l$ and $(\textsc{dk}, l) \in R$ and $k_0^* \neq k_1^*$

  else, send $T_t[i]$ to $\mathcal{A}$ and add (t, i) to R.

  - Encryption Query: for each (k, l, m) received from $\mathcal{A}$,

  ▷ abort if $\mathcal{A}$ sent a key-challenge (t^*, i_0^*, i_1^*) in the previous phase and
    - ∗ if $t^* \in \{\textsc{sk}, \textsc{vk}\}$, $i_0^* \neq i_1^*$, $k \in \{i_0^*, i_1^*)$ and $(\textsc{dk}, l) \in R$
    - ∗ if $t^* = \textsc{dk}$, $i_0^* \neq i_1^*$

    else, send $\mathsf{encase}\big(T_{\textsc{sk}}[k],\ T_{\textsc{ek}}[l],\ m\big)$ to $\mathcal{A}$.
  - Decryption Query: for each (k, l, c) received from $\mathcal{A}$; if $T_{\textsc{cp}}[c] = CP^*$, abort; else if $k = 0$, send $\mathsf{decase}\big(\perp,\ T_{\textsc{dk}}[l],\ T_{\textsc{cp}}[c]\big)$; else, send $\mathsf{decase}\big(T_{\textsc{vk}}[k],\ T_{\textsc{dk}}[l],\ T_{\textsc{cp}}[c]\big)$ to $\mathcal{A}$.

– Final phase - $\mathcal{A}$ outputs a bit b^*. Output 1 if $b = b^*$, else 0.

Fig. 16: Augmented Security Experiment for COA-secure CASE.

Lemma 2 (Restated)**.** *Any COA-secure CASE scheme satisfies augmented security.*

Proof: We prove this via a reduction to the COA-security. Without loss of generality, let the following be adversaries that behave as follows. We argue that the advantage in each case must be negligible.

- $\mathcal{A}_0$ only sends key-challenge (t, i_0, i_1) s.t. $i_0 = i_1$:
  the advantage in this case is trivially 0, since the challenge response is independent of the bit b
- $\mathcal{A}_1$ only sends key-challenge (t, i_0, i_1) s.t. $i_0 \neq i_1$:
  - if $t \in \{\textsc{sk}, \textsc{vk}, \textsc{ek}\}$: this can be reduced to a corresponding adversary $\mathcal{A}^*$ for the total hiding experiment $\mathsf{distinguish\text{-}sans\text{-}DK}$ with the same advantage. Thus, advantage of $\mathcal{A}_1$ in this case must be negligible.
  - if $t = \textsc{dk}$: the advantage in this case is trivially 0, since neither the encryption keys nor any ciphertexts using these keys were queried by adversary.
- $\mathcal{A}_2$ only sends case-packet-challenge $(k_0, k_1, l_0, l_1, m_0, m_1)$ s.t. $l_0 \neq l_1$:
  this can be reduced to a corresponding adversary $\mathcal{A}^*$ for the total hiding experiment $\mathsf{distinguish\text{-}sans\text{-}DK}$ with the same advantage. Thus, advantage of $\mathcal{A}_2$ in this case must be negligible.
- $\mathcal{A}_3$ only sends case-packet-challenge $(k_0, k_1, l_0, l_1, m_0, m_1)$ s.t. $l_0 = l_1$, $m_0 \neq m_1$
  this can be reduced to a corresponding adversary $\mathcal{A}^*$ for the IND-CCA experiment ($\mathsf{distinguish\text{-}sans\text{-}DK}$ with $DK_0 = DK_1$, please refer Appendix **??**) with the same advantage. Thus, advantage of $\mathcal{A}_1$ in this case must be negligible.
- $\mathcal{A}_4$ only sends case-packet-challenge $(k_0, k_1, l_0, l_1, m_0, m_1)$ s.t. $l_0 = l_1$, $m_0 = m_1$ and $k_0 \neq k_1$ and $(\textsc{dk}, l_0) \in R$
  this can be reduced to a corresponding adversary $\mathcal{A}^*$ for the sender hiding experiment $\mathsf{distinguish\text{-}sans\text{-}VK}$ with the same advantage. Thus, advantage of $\mathcal{A}_4$ in this case must be negligible.

- $\mathcal{A}_5$ only sends case-packet-challenge $(k_0, k_1, l_0, l_1, m_0, m_1)$ s.t. $l_0 = l_1$, $m_0 = m_1$ and $k_0 \neq k_1$ and $(\textsc{dk}, l_0) \notin R$
  this can be reduced to a corresponding adversary $\mathcal{A}^*$ for the IND-CCA experiment (distinguish-sans-DK with $DK_0 = DK_1$, please refer Appendix **??**) with the same advantage. Thus, advantage of $\mathcal{A}_5$ in this case must be negligible.
- $\mathcal{A}_6$ only sends case-packet-challenge $(k_0, k_1, l_0, l_1, m_0, m_1)$ s.t. $l_0 = l_1$, $m_0 = m_1$ and $k_0 = k_1$
  the advantage in this case must be 0, since the challenge response is independent of the bit b

Now, for any $\mathcal{A}$ with non-negligible advantage, there exists an adversary in one of the above types which also has non-negligible advantage. Thus, advantage of $\mathcal{A}$ must be negligible. □

B Details omitted from Section 5

B.1 COA-secure Quasi-Deterministic PKE

Implementing QD-PKE : We now show that the Cramer-Shoup PKE scheme based on the DDH assumption, with the modification of [1] is already quasi-deterministic anon-CCA PKE.

Parameter: Let κ be the security parameter.
Let H be a collision-resistant hash function (CRHF)

PKE Primitive P**:**

- pkeSKGen(1^κ):
  - pick a cyclic group G of order q with distinct random generators g_1, g_2
  - sample y_1, y_2, w_1, w_2, z_1, z_2 from $[0, q-1]$
  - output $DK := (G,\ q,\ g_1,\ g_2,\ y_1,\ y_2,\ w_1,\ w_2,\ z_1,\ z_2)$
- pkePKGen(DK):
  - parse SK as $(G,\ q,\ g_1,\ g_2,\ y_1,\ y_2,\ w_1,\ w_2,\ z_1,\ z_2)$
  - $Y = g_1^{y_1} g_2^{y_2}$, $W = g_1^{w_1} g_2^{w_2}$, $Z = g_1^{z_1} g_2^{z_2}$
  - output $EK := (G, q, g_1, g_2, Y, W, Z)$
- pkeEnc(EK, m):
  - parse EK as $(G,\ q,\ g_1,\ g_2,\ Y,\ W,\ Z)$
  - sample $x \leftarrow [q-1]$
  - $c = (g_1^x, g_2^x, mY^x)$ and $v = W^x Z^{xH(c)}$
  - output $CT := (c, v)$
- pkeDec(DK, CT):
  - parse DK as $(G,\ q,\ g_1,\ g_2,\ y_1,\ y_2,\ w_1,\ w_2,\ z_1,\ z_2)$
  - parse CT as (c, v), parse c as (X_1, X_2, C)
  - If $X_1^{w_1} X_2^{w_2} (X_1^{z_1} X_2^{z_2})^{H(c)} \neq v$, output $\bot$
    else output $m = C(X_1^{y_1} X_2^{y_2})^{-1}$

Quasi-Deterministic property:

- pkeEnc$_1(PK; r)$:
  - parse EK as $(G,\ q,\ g_1,\ g_2,\ Y,\ W,\ Z)$
  - output $\tau := (g_1^x, g_2^x)$, where $x = r$
- pkeEnc$_2(EK, \tau, m)$:
  - inefficiently extract r from τ
  - output $CT :=$ pkeEnc$(EK, m; r)$

Fig. 17: Cramer-Shoup construction for Quasi-Deterministic Anon-CCA PKE.

Lemma 3 (Restated). *Assuming the Decisional Diffie-Hellman assumption (DDH), there exists a Quasi-Deterministic Anon-CCA PKE scheme.*

Proof: We show that the scheme P (with the quasi-deterministic algorithms pkeEnc$_1$, pkeEnc$_2$) in Figure 17 satisfies Definition 11 via a sequence of hybrids.

- Let $\mathcal{H}_b^0$ be the real experiment for challenge bit b. Let the b^{th} challenge message be m_b and the b^{th} PKE keys be $DK_b = (G, q, g_1, g_2, y_1, y_2, w_1, w_2, z_1, z_2)$, $EK_b = (G, q, g_1, g_2, Y, W, Z)$.

- $\mathcal{H}_b^1$: in this hybrid, the challenge ciphertext (c, v) is modified to (c', v), where c' is constructed using x_2 independent of x as $c' = \big(g_1^x,\, g_2^{x_2},\, m_b(g_1^x)^{y_1}(g_2^{x_2})^{y_2}\big)$.

  Indistinguishability with hybrid $\mathcal{H}_b^0$ follows via a reduction to the DDH assumption. Let $\mathcal{A}$ be an adversary that distinguishes between $\mathcal{H}_0$ and $\mathcal{H}_1$ with advantage α. Then, we define adversary $\mathcal{A}^*$ for the DDH experiment, that on input (g_1, g_1^x, g_2, g_2^d) [16] sets $X_1 = g_1^x$, $X_d = g_2^d$, runs $\mathcal{A}$ internally in a straightline black-box way, samples a PKE key pair (DK, EK) conditioned on (g_1, g_2), constructs $c = \big(X_1,\, X_d,\, m_b X_1^{y_1} X_d^{y_2}\big)$, sends (EK, c) to $\mathcal{A}$, responds to oracle queries of the form CT from $\mathcal{A}$ with $\mathsf{pkeDec}(DK, CT)$. It accepts the challenge messages (m_0, m_1) from $\mathcal{A}$, constructs $v = X_1^{w_1} X_d^{w_2}(X_1^{z_1} X_d^{z_2})^{H(c)}$ and sends $CT_b = (c, v)$ to $\mathcal{A}$. It finally outputs $\mathcal{A}$'s output. Thus, $\mathcal{A}^*$ also has advantage α; but from the DDH assumption, this must be negligible.
- $\mathcal{H}_b^2$: in this hybrid, the challenge ciphertext (c', v) is further modified to (c'', v), where c'' is constructed using x_1, x_2 independent of x as $c'' = \big(g_1^{x_1},\, g_2^{x_2},\, m_b(g_1^{x_1})^{y_1}(g_2^{x_2})^{y_2}\big)$.

  Indistinguishability with hybrid $\mathcal{H}_b^1$ follows via a similar reduction to the DDH assumption.
- We now note that $\mathcal{H}_b^2$ corresponds to the IND-CCA experiment (and an extra communication of $\tau = c''$ that is independent of the experiment), thus $\mathcal{H}_0^2 \approx \mathcal{H}_1^2$.

□

Implementing COA-secure QD-PKE: We now show that a COA-secure QD-PKE scheme can be constructed from any QD anon-CCA PKE scheme (Definition 11) and fully binding commitment scheme.

Lemma 4 (Restated)**.** *If there exists a quasi-deterministic anon-CCA PKE scheme and a fully binding commitment scheme; then there exists a COA-secure quasi-deterministic PKE scheme.*

Proof: We prove that the scheme in Figure 6 is a COA-secure QD-PKE primitive (Definition 12).

- Correctness holds directly from the correctness of the underlying primitives.
- Quasi-Deterministic: we define the algorithms as follows from the quasi-deterministic property of the underlying PKE scheme

```
$\mathsf{pkeEnc}_1(EK)$ :
* if $\mathsf{pkeAcc}(EK) \neq$ EK, output $\perp$
* parse $EK$ as $(PK^*, c)$
* sample $\hat{r} \leftarrow \{0,1\}^{poly(\kappa)}$
* $\hat{c} \leftarrow \mathsf{comCommit}^*(EK; \hat{r})$
* output $\tau := (\hat{c}, \mathsf{pkeEnc}_1^*(EK^*))$
```

```
$\mathsf{pkeEnc}_2(EK, \tau, m)$ :
* parse $\tau$ as $(\hat{c}, \tau^*)$
* output $\mathsf{pkeEnc}_2^*(EK^*, \tau^*, m)$
```

- IND-CCA security: this holds from a reduction to the IND-CCA security of the underlying PKE scheme and the computational hiding of the commitment scheme. Let $\mathcal{A}$ be an adversary that has advantage α in the experiment $\mathsf{pkeCCAExp}$ for P. We define a sequence of hybrids as follows:
  - $\mathcal{H}_1$: in this hybrid, the experiment uses a modified public key, where the commitment is to 0. That is, $PK' = (PK^*, \mathsf{comCommit}(0))$. Indistinguishability holds from the computational hiding of the commitment primitive.
  - $\mathcal{H}_2$: in this hybrid, the experiment is replaced by the experiment $\mathsf{pkeCCAExp}$ for P^* with an adversary $\mathcal{A}^*$ that runs $\mathcal{A}$ in a straightline black-box way and behaves as follows. It gets EK^* from the experiment and sends $EK' = \big(EK^*, \mathsf{comCommit}(0)\big)$ to $\mathcal{A}$. For each polynomial query CT_j that $\mathcal{A}$ makes, it parses CT_j as $(\hat{c_j}, CT_j^*)$, queries the experiment on CT_j^*, receives $m_j || \hat{r_j}$, verifies that $\hat{c_j} = \mathsf{comCommit}(EK'; \hat{r_j})$ and sends m_j to $\mathcal{A}$. When $\mathcal{A}$ outputs the challenge messages (m_0, m_1), it constructs $m_0^* = m_0 || \hat{r}$ and

[16] where, if challenge bit is 0, then $d = x$, else $d = x_2$

$m_1^* = m_1 || \hat{r}$ for a uniformly sampled $\hat{r}$ and sends (m_0^*, m_1^*) as the challenge to the experiment. It receives CT^* as the challenge ciphertext, sends $CT = \big(\mathsf{comCommit}(EK'; \hat{r}), CT^*\big)$ to $\mathcal{A}$ and outputs $\mathcal{A}$'s output. Indistinguishability holds trivially since the view of $\mathcal{A}$ in $\mathcal{H}_1$ is identical to its view in $\mathcal{H}_2$.

But, hybrid $\mathcal{H}_2$ corresponds to the IND-CCA experiment for P^* , thus the advantage of $\mathcal{A}$ must be negligible.

– QD anon-CCA: We define a sequence of hybrids similar to the previous case:

- $\mathcal{H}_1$: in this hybrid, the experiment uses modified public keys, where the commitment is to 0. That is, for $b \in \{0,1\}$, $EK'_b = (PK^*_b, \mathsf{comCommit}(0))$. Indistinguishability holds from the computational hiding of the commitment primitive.
- $\mathcal{H}_2$: in this hybrid, the experiment is replaced by the experiment $\mathsf{pkeQDAnonCCAExp}$ for P^* with an adversary $\mathcal{A}^*_b$ that runs $\mathcal{A}$ in a straightline black-box way and behaves as follows. It gets (EK^*_0, PK^*_1, τ) from the experiment and sends $\{EK'_b = \big(PK^*_b, \mathsf{comCommit}(0)\big)\}_{b \in \{0,1\}}$ to $\mathcal{A}$. For each polynomial oracle query (b_j, CT_j) that $\mathcal{A}$ makes, it parses CT_j as $(\hat{c}_j, CT^*_j)$, queries the oracle on (b_j, CT^*_j), receives $m_j || \hat{r}_j$, verifies that $\hat{c}_j = \mathsf{comCommit}(EK'_b; \hat{r}_j)$ and sends m_j to $\mathcal{A}$. When $\mathcal{A}$ outputs the challenge message m, it constructs $m^* = m || \hat{r}$ for a uniformly sampled $\hat{r}$ and sends m^* as the challenge to the experiment. It receives CT^* as the challenge ciphertext and the challenge bit b, sends $CT = \big(\mathsf{comCommit}(EK'_b; \hat{r}), CT^*\big)$ to $\mathcal{A}$ and outputs $\mathcal{A}$'s output.

  The view of $\mathcal{A}$ in $\mathcal{H}_1$ is identical to its view in $\mathcal{H}_2$.
- $\mathcal{H}_3$: in this hybrid, we replace $\mathcal{A}^*_b$ (that gets the challenge bit b) with an adversary $\mathcal{A}^*$ that behaves exactly as $\mathcal{A}^*_b$, except that it does not get b and instead constructs the challenge ciphertext as follows. It sends $CT = \big(\mathsf{comCommit}(0; \hat{r}), CT^*\big)$ to $\mathcal{A}$ and outputs $\mathcal{A}$'s output.

  We now argue indistinguishability between hybrids $\mathcal{H}_2$ and $\mathcal{H}_3$ via intermediate hybrids. We first replace the ciphertext CT^* with a dummy ciphertext $CT^* = \mathsf{pkeEnc}(DK^*_b, 0)$, indistinguishability holds from the IND-CCA of P^*. Then, we replace $\mathsf{comCommit}(EK'_b)$ with $\mathsf{comCommit}(0)$, indistinguishability holds from the computational binding of the commitment primitive. Finally, we replace the dummy ciphertext with the real ciphertext.

But, hybrid $\mathcal{H}_3$ corresponds to the anonymity experiment for P^*, thus the advantage of $\mathcal{A}$ must be negligible.

– Existential Consistency: the extractor algorithms $\mathsf{pkeSKId}$, $\mathsf{pkePKId}$ and $\mathsf{pkeMsgId}$ are defined in Figure 18.

Existential Consistency:

- $\mathsf{pkePKId}(CT)$:
  - ∗ parse CT as $(\hat{c}, CT^*)$
  - ∗ inefficiently extract EK from $\hat{c}$
  - ∗ output EK

- $\mathsf{pkeMsgId}(CT)$:
  - ∗ $EK \leftarrow \mathsf{pkePKId}(CT)$
  - ∗ $DK \leftarrow \mathsf{pkeSKId}(EK)$
  - ∗ $m \leftarrow \mathsf{pkeDec}(DK, m)$
  - ∗ output m
- $\mathsf{pkeSKId}(PK)$:
  - ∗ parse EK as (EK^*, c)
  - ∗ inefficiently extract (r_0, r_1) from c
  - ∗ output $DK = (r_0, r_1)$

Fig. 18: Existential Consistency for COA-secure QD-PKE in Figure 6.

We now prove that the constraints are satisfied.

1. for any $DK \in \mathcal{DK}$, an honest execution of $\mathsf{pkePKGen}(DK)$ will output a correct commitment to DK and from the full binding property, $\mathsf{pkeSKId}$ extracts DK with probability 1.
2. for any $EK \in \mathcal{EK}$, an honest execution of $\mathsf{pkeEnc}(EK, m)$ will output a correct commitment to EK, and from the full binding property, $\mathsf{pkePKId}$ extracts this EK with probability 1.
3. this is true by construction. The decrypt algorithm outputs a message $m \neq \bot$ if and only if $\mathsf{pkePKId}(CT) = EK$ (where $EK = \mathsf{pkePKGen}(DK)$). And from the previous constraints, DK can uniquely be extracted as $\mathsf{pkeSKId}(EK)$.

4. from the previous constraints, $\mathsf{pkeMsgId}$ correctly extracts EK using $\mathsf{pkePKId}$, from which it correctly extracts DK using $\mathsf{pkeSKId}$. For a PK generated honestly from any $DK \in \mathcal{DK}$, perfect correctness of the underlying anon-PKE primitive P^* guarantees that $\mathsf{pkeDec}^*(DK^*, CT) = m$.

□

B.2 Existentially Consistent Anonymous Signature

Instantiating Existentially Consistent Anonymous Signature scheme. We now show that a Existentially Consistent Anonymous Signature scheme can be constructed from any signature scheme (Definition 6), COA-secure QD-PKE scheme (Definition 12) and fully binding commitment scheme.

Lemma 5 (Restated)**.** *If there exists a signature scheme, a COA-secure QD-PKE scheme and a fully binding commitment scheme; then there exists a Existentially Consistent Anonymous Signature scheme.*

Proof: We prove that the scheme S in Figure 8 is a Existentially Consistent Anonymous Signature scheme.

- Correctness of Verification: this holds trivially from the perfect correctness of the underlying schemes.
- Correctness of Accept: the accept algorithm is as described in Figure 8
  1. the output of $\mathsf{sigSKGen}(1^\kappa)$ will be in $\mathcal{SK}$. Then, from the correctness of the underlying PKE, $\mathsf{pkeAcc}(DK^*)$ outputs $\textsc{dk}$ and thus sigAcc outputs $\textsc{sk}$.
  2. for any $SK \in \mathcal{SK}$, the output of $\mathsf{sigVKGen}(SK)$ will be in $\mathcal{VK}$.
  3. for any $SK \in \mathcal{SK}$, the output of $\mathsf{sigSign}(SK, m)$ will be in Σ. Then, from the correctness of the underlying PKE, if $\mathsf{pkeAcc}(DK^*) = \textsc{dk}$, then $\mathsf{pkeAcc}(\mathsf{pkeEnc}(EK^*, m))$ outputs $\textsc{ct}$ with all but negligible probability (where $\hat{EK} = \mathsf{pkePKGen}(\hat{DK})$) and thus sigAcc outputs $\textsc{sig}$.
  4. this follows trivially from the perfect correctness of the underlying PKE, signature and commitment schemes.
- Strong-Unforgeability: this holds from a reduction to the strong-unforgeability of the underlying signature scheme and the computational hiding of the commitment scheme. Let $\mathcal{A}$ be an adversary that has advantage α in the experiment $\mathsf{SigForgeExp}$ for S. We define a sequence of hybrids as follows:
  - $\mathcal{H}_1$: in this hybrid, we modify the verification key, so that the commitment is to 0. That is, $VK' = (VK^*, \mathsf{comCommit}(0), DK^*)$. Indistinguishability holds from the computational hiding of the commitment scheme.
  - $\mathcal{H}_2$: in this hybrid, the experiment is replaced by the experiment $\mathsf{SigForgeExp}$ for S^* with an adversary $\mathcal{A}^*$ that runs $\mathcal{A}$ in a straightline black-box way and behaves as follows. It gets VK^* from the experiment, samples a PKE key $DK^* \leftarrow \mathsf{pkeSKGen}(1^\kappa)$, sets $c = \mathsf{comCommit}(0)$ and sends $VK' = (VK^*, c, DK^*)$ to $\mathcal{A}$. For each polynomial query m_j that $\mathcal{A}$ makes, it samples $r_{3,j}$, constructs $\tau_j \leftarrow \mathsf{pkeEnc}_1(EK^*; r_{3,j})$, queries the experiment on $m_j || \tau_j$, receives σ_j^*, samples $\hat{r}_j$, constructs $CT_j = \mathsf{pkeEnc}(EK^*, \sigma_j || \hat{r}; r_{3,j})$, $\hat{c}_j = \mathsf{comCommit}(VK^* || c || EK^*)$ and sends $(CT_j, \hat{c}_j)$ to $\mathcal{A}$. When $\mathcal{A}$ outputs (m, σ) as the forgery, it parses σ as $(\hat{c}, CT)$, parses CT as (τ, CT'), decrypts CT to get $\sigma^* || \hat{r}$ and outputs $(m || \tau, \sigma^*)$ as its forgery. Indistinguishability holds trivially since the view of $\mathcal{A}$ in $\mathcal{H}_1$ is identical to its view in $\mathcal{H}_2$.

  But, hybrid $\mathcal{H}_2$ corresponds to the unforgeability experiment for S^* , thus the advantage of $\mathcal{A}$ must be negligible. Note that, from the quasi-deterministic property of the PKE scheme, if CT is different from any response from oracle, it holds that either τ or the message must be different. But, the underlying signature is to $m || \tau$. Thus, it is a valid forgery on the underlying signature.
- (Signer) Anonymity: this holds from a reduction to the anonymous security of the underlying PKE scheme and the computational hiding of the commitment scheme. Let $\mathcal{A}$ be an adversary that has advantage α in the experiment $\mathsf{SigAnonExp}$ for S. We define a sequence of hybrids similar to the previous case:
  - $\mathcal{H}_1$: in this hybrid, we modify the public keys, so that the commitments are to 0. That is, for $b \in \{0, 1\}$, $VK'_b = (VK_b^*, \mathsf{comCommit}(0), DK_b^*)$. Indistinguishability holds from the computational hiding of the commitment scheme.

- $\mathcal{H}_2$: in this hybrid, we modify each signature in the experiment (oracle queries as well as challenge signature) as follows. Suppose the real signature of any m be $\sigma = (CT, \hat{c})$, then the modified signature is $\sigma' = (\mathsf{pkeEnc}(\widehat{EK}, 0), \hat{c})$. Indistinguishability holds from the QD anon-CCA security of the PKE scheme.
- $\mathcal{H}_3$: in this hybrid, we modify each signature in the experiment as follows. Suppose the signature of any m be $\sigma' = (\mathsf{pkeEnc}(EK^*, 0), \hat{c})$, then the modified signature is $\sigma'' = (\mathsf{pkeEnc}(EK^*, 0), \mathsf{comCommit}(0))$. Indistinguishability holds from the computational hiding of the commitment scheme.
- $\mathcal{H}_4$: in this hybrid, the experiment is replaced by the experiment pkeQDAnonCCAExp for P^* with an adversary $\mathcal{A}^*$ that runs $\mathcal{A}$ in a straightline black-box way and behaves as follows. For each $b \in \{0, 1\}$, it gets (EK_0^*, EK_1^*, τ) from the experiment. For each polynomial query (b_j, m_j) that $\mathcal{A}$ makes, it sends $\sigma_j'' = (\mathsf{pkeEnc}(EK_{b_j}^*, 0), \mathsf{comCommit}(0))$ to $\mathcal{A}$. When $\mathcal{A}$ outputs the challenge message m, it sends (m, m) as the challenge messages to the experiment, receives CT as the challenge ciphertext, sends $\sigma'' = (CT, \mathsf{comCommit}(0))$ to $\mathcal{A}$ and outputs $\mathcal{A}$'s output. Indistinguishability holds trivially since the view of $\mathcal{A}$ in $\mathcal{H}_3$ is identical to its view in $\mathcal{H}_4$.

But, hybrid $\mathcal{H}_4$ corresponds to the anonymity experiment for P^*, thus the advantage of $\mathcal{A}$ must be negligible.

– Existential Consistency: the extractor algorithms sigVKId and sigSKId are defined in Figure 19.

Existential Consistency:

- $\mathsf{sigSKId}(VK)$:
  - ∗ parse VK as (VK^*, c, DK^*)
  - ∗ inefficiently extract (r_0, r_1) from c
  - ∗ output $SK = r_0 || r_1 || DK^*$
- $\mathsf{sigVKId}(\sigma)$:
  - ∗ parse σ as $(CT, \hat{c})$
  - ∗ inefficiently extract $(VK^* || c || EK^*, \hat{r})$ from $\hat{c}$
  - ∗ inefficiently extract $DK^* \leftarrow \mathsf{pkeSKId}(EK^*)$
  - ∗ output $VK := (VK^*, c, DK^*)$

Fig. 19: Existential consistency for scheme in Figure 8.

We now prove that the constraints are satisfied:

1. let $SK \in \mathcal{SK}$, $VK \leftarrow \mathsf{sigVKGen}(SK)$; then from the full binding of the commitment scheme, $\mathsf{sigSKId}(VK)$ outputs SK with probability 1
2. let $SK \in \mathcal{SK}$, $m \in \mathcal{M}$, $\sigma \leftarrow \mathsf{sigSign}(SK, m)$;then from the full binding of the commitment scheme and the COA existential consistency of the PKE scheme, $\mathsf{sigSKId}(\mathsf{sigVKId}(\sigma))$ outputs SK with probability 1
3. this holds trivially by construction, sigVerify outputs 1 if and only if the commitment to VK in the signature σ matches.

□

B.3 Main Construction

B.3.1 Efficient COA-secure CASE

Lemma 7 (Restated)**.** *If S is a CPA-secure SKE scheme, H is a CRHF scheme and* case *is a COA-secure CASE scheme; then the scheme* case* *in Figure 11 is a COA-secure CASE scheme.*

Proof: We verify the properties in Definition 8. The arguments below refer to the experiments in Figure 4.

– **Total Hiding:** Let CP_{b^*} be the challenge ciphertext in the experiment distinguish-sans-DK with challenge bit $b^* \in \{0, 1\}$. We prove that $CP_0 \approx CP_1$ via a sequence of hybrids as follows. Let the adversary, given (EK_0, EK_1) output (m_0, m_1, SK_0, SK_1), and the challenge ciphertext be $CP_{b^*} = (c_0, c_1)$.

- In first hybrid, $b^* = 0$, but we replace c_0 with a case-packet $\mathsf{encase}(SK_0, EK_0, 0)$ (where 0 denotes a dummy message). Indistinguishability with R_0 follows from the total hiding of the CASE scheme case.

- In the next hybrid, we replace the message in c_1 from m_0 to m_1. That is, $c_1 = \mathsf{skeEnc}(k_1, m_0)$ is replaced with $c_1' = \mathsf{skeEnc}(k_1, m_1)$. Indistinguishability with the previous hybrid follows from the semantic-security of the SKE scheme S.
- We finally replace c_0 (encase to 0) with $c_0' = \mathsf{encase}(SK_1, EK_1, k_1||k_2||H(k_2, m_1))$ (that is, the case-packet for challenge bit 1). Indistinguishability with $\mathcal{H}^2$ follows from the total hiding of the CASE scheme case. But, this hybrid is exactly the experiment R_1. Hence, proved.

– **Sender Anonymity:** This follows from a reduction to the underlying CASE scheme case. The reduction involves implementing the oracles $\mathcal{E}$, $\mathcal{D}$, and $\mathcal{D}'$ for the sender anonymity experiment for $\mathsf{case}^\star$, given access to the same oracles for case. While this is straightforward for $\mathcal{E}$ and $\mathcal{D}$, for $\mathcal{D}'$ we need to rely on the collision-resistance of H. If the adversary received a challenge ciphertext $CP^* = (c_0^*, c_1^*)$ and later queried $\mathcal{D}'$ with (b, VK, CP) where $CP = (c_0^*, c_1)$ for $c_1 \neq c_1^*$, the reduction cannot get CP decased using the version of the oracle $\mathcal{D}'$ that it has access to. However, it is guaranteed that $\mathsf{case}^\star.\mathsf{decase\text{-}verify}$ will reject CP unless the hash of c_1 equals that of c_1^*, which happens only with negligible probability thanks to the collision resistance of H. So in this case, the reduction's implementation of $\mathcal{D}'$ simply returns $\perp$.

– **Strong Unforgeability:** holds via a reduction to the underlying CASE scheme case and CRHF scheme H. Let $\mathcal{A}$ be an adversary with advantage α in the experiment forge for $\mathsf{case}^\star$. We build adversary $\mathcal{A}_1^*$ for the experiment forge for case that internally runs $\mathcal{A}$ in a black-box straightline way and interacts as follows.

- For any query (EK, m) of $\mathcal{A}$ to $\mathcal{E}$, $\mathcal{A}_1^*$ samples k_1, k_2, constructs $c_1 = \mathsf{skeEnc}(k_1, m)$, $h = H(k_2, c_1)$, queries the oracle $\mathcal{E}$ it has access to with $(EK, k_1||k_2||h)$, receives c_0 back, and sends $CP = (c_0, c_1)$ to $\mathcal{A}$.
- Finally, $\mathcal{A}$ outputs (DK, CP). It parses CP as (c_0, c_1) and outputs (DK, c_0).
  If c_0 matches a response that $\mathcal{A}_1^*$ got from a query to its oracle $\mathcal{E}$, but (c_0, c_1) does not match it aborts. Else, it outputs (DK, c_0) as its output.

Note that $\mathcal{A}$ succeeds in its forgery experiment while $\mathcal{A}_1^*$ fails iff (c_0, c_1) passes decasing for $\mathsf{case}^\star$ (and hence so does c_0 for case), and further (c_0, c_1) was not obtained by $\mathcal{A}$ as reponse for a query to its $\mathcal{E}$, *but* c_0 was obtained by $\mathcal{A}_1^*$ from $\mathcal{E}$ that it accesses. Let $k_1\|k_2\|h$ be the message that $\mathcal{A}_1^*$ queried $\mathcal{E}$ with to get c_0, where $h = H(k_2, c_1')$ for some $c_1' \neq c_1$. Since (c_0, c_1) passes decasing, it must be the case that $H(c_1) = h$ thereby yielding a hash collision. We capture this as an adversary $\mathcal{A}_2^*$ for the CRHF primitive H, as follows:

- $\mathcal{A}_2^*$ internally runs the challenger and $\mathcal{A}$ and has it interact as per the experiment forge for $\mathsf{case}^\star$.
- Finally, $\mathcal{A}$ outputs (DK, CP). It parses CP as (c_0, c_1). If c_0 matches some query to oracle $\mathcal{E}$, let the query be (EK, m) and response be (c_0, c_1'). It outputs (c_1, c_1') as the collision. If no such query matches, it aborts.

Now, if advantage α of $\mathcal{A}$ is non-negligible, then one of $\mathcal{A}_1^*$ and $\mathcal{A}_2^*$ will not abort and must have non-negligible advantage.

– **Unpredictability:** This holds directly from the unpredictability of the underlying CASE scheme case.

– **Correctness and Existential Consistency:** $\forall SK \in \mathcal{SK}, DK \in \mathcal{DK}, m \in \mathcal{M}$, let $VK \leftarrow \mathsf{case}^\star.\mathsf{vkGen}(SK)$, $EK \leftarrow \mathsf{case}^\star.\mathsf{ekGen}(DK)$, $CP \leftarrow \mathsf{case}^\star.\mathsf{encase}(SK, EK, m)$.

- Correctness: From the correctness of the underlying CASE scheme case, it holds that the objects are accepted with probability $1 - \mathsf{negl}(\kappa)$. Further, it holds that c_0 decrypts to $k_1||k_2||h$ with probability $1 - \mathsf{negl}(\kappa)$ and from the perfect correctness of the SKE scheme, c_1 decrypts to m with probability 1.
- Existential Consistency: the extrator algorithms are defined in Figure 20.
  From the existential consistency of the underlying primitives, it holds that $\mathsf{case}^\star.\mathsf{skId}(VK) = SK$, $\mathsf{case}^\star.\mathsf{dkId}(EK) = DK$. Further, for any $CP \in \mathcal{CP}$ s.t. $\mathsf{case}^\star.\mathsf{acc}(CP) = 1$, it holds that if $\mathsf{case}^\star.\mathsf{decase\text{-}msg}(DK, CP) \neq \perp$, then $\mathsf{case}^\star.\mathsf{ekId}(CP) = EK$. Similarly, if $\mathsf{case}^\star.\mathsf{decase\text{-}verify}$ $(VK, DK, CP) \neq \perp$, it holds that $\mathsf{case}^\star.\mathsf{vkId}(CP) = VK$ and $\mathsf{msgId}(c_0) = k_1||k_2||h$. Finally, from the correctness of the SKE scheme, c_1 decrypts to m using key k_1 with probability 1 and from the correctness of CRHF scheme, verifies with probability 1.

□

```
Existential Consistency:
* $\mathsf{case}^\star.\mathsf{dkId}(EK)$:
    output $\mathsf{case}.\mathsf{dkId}(EK)$
* $\mathsf{case}^\star.\mathsf{skId}(VK)$:
    output $\mathsf{case}.\mathsf{skId}(VK)$
* $\mathsf{case}^\star.\mathsf{vkId}(CP)$:
    parse $CP$ as $(c_0, c_1)$
    output $\mathsf{case}.\mathsf{vkId}(c_0)$
* $\mathsf{case}^\star.\mathsf{msgId}(CP)$:
    parse $CP$ as $(c_0, c_1)$
    $k_1||k_2||h \leftarrow \mathsf{case}.\mathsf{msgId}(c_0)$, $m \leftarrow \mathsf{skeDec}(k_1, c_1)$
    output $m$
* $\mathsf{case}^\star.\mathsf{ekId}(CP)$:
    parse $CP$ as $(c_0, c_1)$
    output $\mathsf{case}.\mathsf{ekId}(c_0)$
```

Fig. 20: Efficient COA secure CASE via hybrid encryption

C Details omitted from Section 7

Theorem 1 (Restated). *A Δ-s-IND-PRE secure implementation of Σ_{case} exists if a COA secure CASE scheme exists.*

In the rest of the section, we prove that Π_{case} in Figure 15 is a Δ-s-IND–secure implementation of Σ_{case}.

C.1 Extended Schema $\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}$

We will be using a schema $\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}$, which combines the schema Σ_{case} with the scheme Π_{case} as follows: The agent in $\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}$ behaves like the agent in Σ_{case}, but in addition provides additional operations for the User (exploited by our simulators $\mathcal{S}^{\dagger}_b, \mathcal{S}^{\ddagger}_b$ and $\mathcal{S}^{\ddagger}$). These additional operations allow incorporating objects into the handles (referred to as "patching"), so that some of the sessions among handles are carried out using the algorithms in Π_{case} and such objects. Figure 21 describes this extended schema.

$\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}$ has an agent which behaves as follows, when invoked in a session.

- **Patching a signing-key.** If the input is the $(\texttt{patch}, \mathit{obj})$ and the work-tape has $(\texttt{sk}, \mathit{sk\text{-}tag})$, then the agent changes the work-tape entry to $(\texttt{sk}, \mathit{sk\text{-}tag}, \mathit{obj})$.
- **Patching a verification-key.** If the input is the $(\texttt{patch}, \mathit{obj})$ and the work-tape has $(\texttt{vk}, \mathit{sk\text{-}tag})$, then the agent changes the work-tape entry to $(\texttt{vk}, \mathit{sk\text{-}tag}, \mathit{obj})$.
- **Patching a decryption-key** While patching a decryption-key, multiple agents are run as follows in a session with the first agent being a decryption-key agent and the other agents being signing-key agents.
  - If the work-tape entry of the agent is $(\texttt{sk}, \mathit{sk\text{-}tag})$ and the command received is $(\texttt{dkPatch}, \{CP_i\})$, it sends $(\mathit{sk\text{-}tag}, \{CP_i\})$ to the first agent in the session.
  - If the work-tape entry of the agent is $(\texttt{dk}, \mathit{dk\text{-}tag})$ and the command received is $(\texttt{dkPatch}, \mathit{obj})$, then it waits for messages from the other agents in the session. After receiving all messages, the first agent in the session, collects all messages of the form $(\mathit{sk\text{-}tag}, \{CP_i\})$ received into a list L (possibly empty). It then changes the work-tape entry to $(\texttt{dk}, \mathit{dk\text{-}tag}, \mathit{obj}, L)$.
- **Patching an encryption-key.** If the input is the $(\texttt{patch}, \mathit{obj})$ and the work-tape has $(\texttt{ek}, \mathit{dk\text{-}tag})$, then the agent runs in a session without any changes.[17]
- **Creating a case-packet agent with an associated decryption-key** If the input is the $(\texttt{CPgen}, (CP, DK))$ and the work-tape has $(\texttt{ek}, \mathit{dk\text{-}tag})$, then the agent changes the work-tape entry to $(\texttt{cp}, CP, DK, \mathit{dk\text{-}tag})$.

[17] Note that this operation is functionally redundant but the command itself is required by the simulator $\mathcal{S}^*$ to associate ideal handles to objects.

- **Creating an extended case-packet agent** When run with an empty work-tape (when the init command is sent to $\mathcal{B}[\Sigma_{\texttt{case}}]$) and with the tuple $(\texttt{CPgen}, \mathit{obj})$ as the input, the agent is initialized as an extended case-packet agent: i.e., the agent records the tuple $(\texttt{cp}, \mathit{obj})$ on its work-tape.
- **Decasing and verifying a message** While decrypting and verifying, three agents are run in a session, all having the keyword `decase-verify` as input. For each of the three agents, the following is done.
  - If its work-tape contents are $(\texttt{cp}, m, \mathit{sk\text{-}tag}, \mathit{dk\text{-}tag}, \mathit{cp\text{-}tag})$ or $(\texttt{cp}, \mathit{obj})$ or $(\texttt{cp}, \mathit{CP}, \mathit{DK}, \mathit{dk\text{-}tag})$, it sends the contents to the first agent in the session.
  - If its work-tape contents are $(\texttt{vk}, \mathit{sk\text{-}tag})$ or $(\texttt{vk}, \mathit{sk\text{-}tag}, \mathit{obj})$, it sends the contents to the first agent in the session.
  - If its work-tape contents are $(\texttt{dk}, \mathit{dk\text{-}tag}, \mathit{obj}, L)$, it waits for messages from the second and third agent in the session.
    - * If it receives messages of the form $(\texttt{cp}, m, \mathit{sk\text{-}tag}^\star, \mathit{dk\text{-}tag}^\star, \mathit{cp\text{-}tag})$ and $((\texttt{vk}, \mathit{sk\text{-}tag})$ or $(\texttt{vk}, \mathit{sk\text{-}tag}, \mathit{VK}))$ such that $\mathit{sk\text{-}tag} = \mathit{sk\text{-}tag}^\star$ and $\mathit{dk\text{-}tag} = \mathit{dk\text{-}tag}^\star$, then it writes m on the output tape.
    - * If it receives messages of the form $(\texttt{cp}, \mathit{CP})$ and $(\texttt{vk}, \mathit{sk\text{-}tag})$ such that $(\mathit{sk\text{-}tag}, \mathit{CP}) \in L$, write output of $\mathsf{case.decase\text{-}msg}(\mathit{obj}, \mathit{CP})$ to the output tape.
    - * If it receives messages of the form $(\texttt{cp}, \mathit{CP})$ and $(\texttt{vk}, \mathit{sk\text{-}tag}, \mathit{VK})$, write output of $\mathsf{case.decase\text{-}verify}(\mathit{obj}, \mathit{VK}, \mathit{CP})$ to the output tape.
  - If its work-tape contents are $(\texttt{dk}, \mathit{dk\text{-}tag})$, it waits for messages from the second and third agent in the session.
    - * If it receives messages of the form $(\texttt{cp}, m, \mathit{sk\text{-}tag}^\star, \mathit{dk\text{-}tag}^\star, \mathit{cp\text{-}tag})$ and $((\texttt{vk}, \mathit{sk\text{-}tag})$ or $(\texttt{vk}, \mathit{sk\text{-}tag}, \mathit{VK}))$ such that $\mathit{sk\text{-}tag} = \mathit{sk\text{-}tag}^\star$ and $\mathit{dk\text{-}tag} = \mathit{dk\text{-}tag}^\star$, then it writes m on the output tape.
    - * If it receives messages of the form $(\texttt{cp}, \mathit{CP}, \mathit{DK}, \mathit{dk\text{-}tag}^\star)$ and $(\texttt{vk}, \mathit{sk\text{-}tag}, \mathit{VK})$ such that $\mathit{dk\text{-}tag} = \mathit{dk\text{-}tag}^\star$, write output of $\mathsf{case.decase\text{-}verify}(\mathit{DK}, \mathit{VK}, \mathit{CP})$ to the output tape.
- **Extracting the message:** To extract the message, two agents are run in a session all having the input keyword as `decase-msg`.
  - If the work-tape entry of the agent is $(\texttt{dk}, \mathit{dk\text{-}tag})$ or $(\texttt{dk}, \mathit{dk\text{-}tag}, \mathit{obj}, L)$, the agent sends the contents of its work-tape to the second agent in the session.
  - If the work-tape entry of the agent is $(\texttt{cp}, m, \mathit{sk\text{-}tag}, \mathit{dk\text{-}tag}, \mathit{cp\text{-}tag})$, the agent waits for a message from the other agent. If it receives a message of the form $(\texttt{dk}, \mathit{dk\text{-}tag}^\star)$ or $(\texttt{dk}, \mathit{dk\text{-}tag}^\star, \mathit{obj}, L)$ such that $\mathit{dk\text{-}tag} = \mathit{dk\text{-}tag}^\star$, it writes m to its output tape.
  - If the work-tape entry of the agent is $(\texttt{cp}, \mathit{CP}, \mathit{DK}, \mathit{dk\text{-}tag})$, the agent waits for a message from the other agent. If it receives a message of the form $(\texttt{dk}, \mathit{dk\text{-}tag}^\star)$ such that $\mathit{dk\text{-}tag} = \mathit{dk\text{-}tag}^\star$, it writes $\mathsf{case.decase\text{-}msg}(\mathit{DK}, \mathit{CP})$ to its output tape.
  - If the work-tape entry of the agent is $(\texttt{cp}, \mathit{CP})$, the agent waits for a message from the other agent. If it receives a message of the form $(\texttt{dk}, \mathit{dk\text{-}tag}^\star, \mathit{obj}, L)$, it writes $\mathsf{case.decase\text{-}msg}(\mathit{obj}, \mathit{CP})$ to its output tape.
- **Differentiating type of agent:** If an agent is invoked with the keyword `type` as input, it behaves as follows, depending on the contents of its work-tape:
  - if the work-tape has $(\texttt{sk}, \mathit{sk\text{-}tag}, \mathit{obj})$ or $(\texttt{sk}, \mathit{sk\text{-}tag})$, output `sk`.
  - if the work-tape has $(\texttt{vk}, \mathit{sk\text{-}tag}, \mathit{obj})$ or $(\texttt{vk}, \mathit{sk\text{-}tag})$, output `vk`.
  - if the work-tape has $(\texttt{dk}, \mathit{dk\text{-}tag}, \mathit{obj}, L)$ where L is a list or $(\texttt{dk}, \mathit{dk\text{-}tag})$, output `dk`.
  - if the work-tape has $(\texttt{ek}, \mathit{dk\text{-}tag})$, output `ek`.
  - if the work-tape has $(\texttt{cp}, \mathit{obj})$ or $(\texttt{cp}, \mathit{obj}, \mathit{DK}, \mathit{dk\text{-}tag})$ or $(\texttt{cp}, m, \mathit{sk\text{-}tag}, \mathit{dk\text{-}tag}, \mathit{cp\text{-}tag})$, output $(\texttt{cp}, \ell)$, where $\ell = \mathit{len}(m)$.

Fig. 21: Extended schema $\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}$.

C.2 Handle derivation graph

We introduce some notation that will be used throughout the proof.

Basic Handle Notations.

- We denote the handle-space for Test as $\widehat{\mathcal{H}}$. Handles in $\widehat{\mathcal{H}}$ are denoted as $\widehat{h}$ (generic handle), specific handles such as $\widehat{dk}$ denote a decryption-key type handle, etc.
- We denote the handle-space for User as $\overline{\mathcal{H}}$. Handles in $\overline{\mathcal{H}}$ are denoted as $\overline{h}$ (generic handle), specific handles such as $\overline{dk}$ denote a decryption-key type handle, etc.
- Variables like h denote generic handles that can either be in $\widehat{\mathcal{H}}$ or $\overline{\mathcal{H}}$.
- We write $\circ \overset{\mathsf{init}}{\to} h$ to denote that on input a command $(\mathsf{init}, \texttt{key-type}, \kappa)$ to $\mathcal{B}[\Sigma]$, it output the handle h.

  We write $h_0 \xrightarrow{inp} h_1$ to denote that on input a command $(\mathsf{run}, \{h_0\}, \{inp\})$ to $\mathcal{B}[\Sigma]$, it output the handle h_1.

  We write $h_0 \to h_1$ to denote that on input a command $(\mathsf{run}, \{h_0\}, \{\mathsf{transfer}\})$ to $\mathcal{B}[\Sigma]$, it output the handle h_1 to the other side.
- We write $h_0 \rightsquigarrow h_1$ to denote that h_1 was derived from h_0 via a sequence of commands and transfers.
- We write $h_0 \equiv h_1$ to denote that $\texttt{compare}(h_0, h_1) = 1$.

Handle Derivation Graph. Throughout the proof, the simulator in each hybrid is defined in terms of a handle derivation graph $\mathbb{G}$. While the nodes in the graphs in the various hybrids have different (more) state information, we describe the basic structure and notation that is common across them. A graph $\mathbb{G}$ is defined as a pair $(\mathbb{V}, \mathbb{E})$, where $\mathbb{V} = \mathbb{V}_{\mathsf{Test}} \cup \mathbb{V}_{\mathsf{User}}$ denotes the set of vertices and $\mathbb{E}$ denotes the set of edges.

- We denote $\mathbb{G}.\mathbb{V}$ as the vertext set $\mathbb{V}$ and $\mathbb{G}.\mathbb{E}$ as the edge set $\mathbb{E}$ of graph $\mathbb{G}$.
- Each vertex $v \in \mathbb{V}_{\mathsf{Test}}$ is of the form $(obj, \widehat{\mathbf{L}}, \ldots)$, where obj is an object, $\widehat{\mathbf{L}}$ is a list of Test handles s.t. $\forall \widehat{h_0}, \widehat{h_1} \in \widehat{\mathbf{L}}$, it holds that $\texttt{compare}(\widehat{h_0}, \widehat{h_1}) = 1$. Vertices in $\mathbb{V}_{\mathsf{Test}}$ correspond to handles that Test has access to. We use the notation $v.\mathsf{obj}$ to represent the object obj and $v.\mathbf{L}$ to represent the list $\widehat{\mathbf{L}}$.
- Each vertex $v \in \mathbb{V}_{\mathsf{User}}$ is of the form $(obj, \mathring{\mathbf{L}}, \ldots)$, where obj is an object and $\mathring{\mathbf{L}}$ is a list of round numbers (corresponding to the communication transcript of User). These nodes correspond to objects transferred to/from $\mathcal{A}$. Again, we use the notation $v.\mathsf{obj}$ to represent the object obj and $v.\mathring{\mathbf{L}}$ to represent the list $\mathring{\mathbf{L}}$.
- We use $\texttt{node}_{\mathbb{G}}(\widehat{h})$ to denote the unique vertex $v \in \mathbb{V}_{\mathsf{Test}}$ s.t. $\widehat{h} \in v.\mathbf{L}$.
- In the various hybrids, we also use $\texttt{node}_{\mathbb{G}}(.)$ as a function that takes some state information and returns the vertex in $\mathbb{G}$ with that state information.
- There are 7 kinds of edges in the graph. The different types of edges are identified by a string stored in the edges. The different types of such strings are -
  1. ekGen
  2. vkGen
  3. (dk-ct, $\texttt{encase}$, m) where $m \in \mathcal{M}$
  4. (sk-ct,$\texttt{encase}$, m) where $m \in \mathcal{M}$
  5. (pk-ct, $\texttt{encase}$, m) where $m \in \mathcal{M}$
  6. (vk-ct, $\texttt{encase}$, m) where $m \in \mathcal{M}$
  7. transfer
- $v_1 \dashrightarrow v_2$ denotes that either $v_1 = v_2$ or $v_1 \xrightarrow{\text{transfer}} v_2$ exists
- $\texttt{root}$ (v) is the *unique* node $v^\star$ such that there is a path with zero or more edges from $v^\star$ to v and $\nexists v'$ *s.t.* $v' \to v^\star$.
- $\texttt{dk-root}$ (v) is the *unique* node $v^\star$ such that there is a path of the form $v^\star \dashrightarrow v_1 \xrightarrow{(dk-ct,\mathsf{encase},m)} v_2 \dashrightarrow v$ or $v^\star \dashrightarrow v_1 \xrightarrow{\text{ekGen}} v_2 \dashrightarrow v_3 \xrightarrow{(pk-ct,\mathsf{encase},m)} v_5 \dashrightarrow v$ and $\nexists v'$ *s.t.* $v' \to v^\star$. Returns $\perp$ if no such $v^\star$ exists.

- `sk-root` (v) - The unique node $v^\star$ such that there is a path of the form $v^\star \dashrightarrow v_1 \xrightarrow{(sk-ct,\mathsf{encase},m)} v_2 \dashrightarrow v$ or $v^\star \dashrightarrow v_1 \xrightarrow{\mathrm{vkGen}} v_2 \dashrightarrow v_3 \xrightarrow{(vk-ct,\mathsf{encase},m)} v_5 \dashrightarrow v$ and $\nexists v'$ *s.t.* $v' \to v^\star$. Returns $\perp$ if no such $v^\star$ exists.
- `vk-root` (v) - The unique node $v^\star$ such that there is a path of the form $v^\star \dashrightarrow v_3 \xrightarrow{(vk-ct,\mathsf{encase},m)} v_5 \dashrightarrow v$ and $\nexists v'$ *s.t.* $v' \to v^\star$. Returns $\perp$ if no such $v^\star$ exists.

C.3 Proof of Security: Indistinguishability of Hybrids

C.3.1 Hybrids H_0 and $\mathsf{H}_{0|1}$ H_0 corresponds to the real execution $\textsc{real}\langle\mathsf{Test}(0) \mid \Pi_{\mathsf{case}} \mid \mathcal{A}\rangle$ with test bit $b = 0$. It uses $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ to execute the commands and transfers from Test and $\mathcal{A}$. The joint view of Test and $\mathcal{A}$ can be captured by an implicit handle derivation graph $\mathbb{G}_0$ and get view function (Figure 22). We prove this by showing indistinguishability with an intermediate hybrid $\mathsf{H}_{0|1}$ in which the handle derivation graph is made explicit. That is, we replace $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ by $\mathcal{I}'[\Pi, \mathbb{G}_0]$ which simulates $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ and stores relations between handles in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ using a graph $\mathbb{G}_0$. On obtaining a command which produces a new handle, $\mathcal{I}'[\Pi, \mathbb{G}_0]$ runs $\mathtt{update}_{\mathsf{Test}}(\mathbb{G}_0)$ and runs $\mathtt{update}_{\mathcal{A}}(\mathbb{G}_0)$ on transfers from $\mathcal{A}$ (see Figure 23). It uses the functions $\mathtt{getView}_{\mathsf{Test}}$ to return outputs for commands `decase-verify`, `decase-msg`, `compare`, `type` and $\mathtt{getView}_{\mathcal{A}}$ for transferring objects to $\mathcal{A}$ (see Figure 22). Please refer to Lemma 9 for the proof of indistinguishability.

The function $\mathtt{getView}_{\mathsf{Test}}$ takes a handle derivation graph and a command as an input and generates the output of the command using structural properties of the graph.

$\underline{\mathtt{getView}_{\mathsf{Test}}(\mathbb{G}, \mathtt{command})}$

- $\mathtt{command} = (\mathsf{run}, (\widehat{dk}, \mathtt{decase\text{-}verify}), (\widehat{vk}, \mathtt{decase\text{-}verify}), (\widehat{cp}, \mathtt{decase\text{-}verify}))$
  - Let $v_1 = \mathtt{node}_{\mathbb{G}}(\widehat{cp})$, $v_2 = \mathtt{node}_{\mathbb{G}}(\widehat{dk})$ and $v_3 = \mathtt{node}_{\mathbb{G}}(\widehat{vk})$.
  - If $\mathsf{case.acc}(v_1.\mathsf{obj}) \neq \textsc{cp}$ or $\mathsf{case.acc}(v_2.\mathsf{obj}) \neq \textsc{dk}$ or $\mathsf{case.acc}(v_3.\mathsf{obj}) \neq \textsc{vk}$, return $\perp$.
  - Set $v_4 = \mathtt{dk\text{-}root}\ (v_1)$ and $v_5 = \mathtt{sk\text{-}root}\ (v_1)$. If $\mathtt{sk\text{-}root}\ (v_1) = \perp$, $v_5 = \mathtt{vk\text{-}root}\ (v_1)$
  - If $v_4 \neq \perp$, $v_5 \neq \perp$ and $v_4 = \mathtt{root}(v_2)$ and $v_5 = \mathtt{root}(v_3)$, return m.
  - Else, return $\perp$
- $\mathtt{command} = (\mathsf{run}, (\widehat{h}_1, \mathtt{compare}), (\widehat{h}_2, \mathtt{compare}))$
  - Let $v_1 = \mathtt{node}_{\mathbb{G}}(\widehat{h}_1)$ and $v_2 = \mathtt{node}_{\mathbb{G}}(\widehat{h}_2)$
  - If $v_1 = v_2$, return true. Else, return false.
- $\mathtt{command} = (\mathsf{run}, (\widehat{dk}, \mathtt{decase\text{-}msg}), (\widehat{cp}, \mathtt{decase\text{-}msg}))$
  - Let $v_1 = \mathtt{node}_{\mathbb{G}}(\widehat{cp})$ and $v_2 = \mathtt{node}_{\mathbb{G}}(\widehat{dk})$
  - If $\mathsf{case.acc}(v_1.\mathsf{obj}) \neq \textsc{cp}$ or $\mathsf{case.acc}(v_2.\mathsf{obj}) \neq \textsc{dk}$, return $\perp$.
  - Set $v_3 = \mathtt{dk\text{-}root}\ (v_1)$.
  - If $v_3 \neq \perp$ and $v_3 = \mathtt{root}(v_2)$, return m.
  - Else, return $\perp$.
- $\mathtt{command} = (\mathsf{run}, (\widehat{h}, \mathtt{type}))$
  - Find $v_1 = \mathtt{node}_{\mathbb{G}}(\widehat{h})$
  - Return the output of $\mathsf{case.acc}(v_1.\mathsf{obj})$
- Else, return $\perp$

The function $\mathtt{getView}_{\mathcal{A}}$ take a handle derivation graph and a "round" number as input and outputs the object received/transferred by the adversary in that round.

$\underline{\texttt{getView}_{\mathcal{A}}(\mathbb{G}, \mathring{r})}$

Find $v \in \mathbb{V}_{\mathcal{A}}$ such that $\mathring{r} \in v.\mathring{\mathbf{L}}$. Return $v.\mathsf{obj}$.

Fig. 22: Description of $\texttt{getView}_{\mathsf{Test}}$ and $\texttt{getView}_{\mathcal{A}}$

In $\mathsf{H}_{0|1}$ and $\mathsf{H}_{6|7}$, $\mathcal{I}'[\Pi, \mathbb{G}_b]$ maintains a handle derivation graph $\mathbb{G}_b$ which stores the relationships between the handles and the underlying objects. The graph is updated with commands sent by Test or objects sent by $\mathcal{A}$. The construction of the graph is given below. Here, $\mathring{r}$ refers to the current round. Refer to Section C.2 for description of nodes and edges.

$\underline{\texttt{update}_{\mathsf{Test}}(\mathbb{G}_b, \texttt{str}, \widehat{h})}$

Let $\widehat{h}$ be the handle to be generated by the command $\texttt{str}$. We consider different cases based on the command sent.

- $\texttt{str} = (\mathsf{init}, (\texttt{key-type}, \kappa))$
  - If $\texttt{key-type} = \textsc{sk}$, generate $SK^\star \leftarrow \mathsf{case.skGen}$ and set $obj = SK^\star$ as the initialized object. Else, if $\texttt{key-type} = \textsc{dk}$, generate $DK^\star \leftarrow \mathsf{case.dkGen}$ and set $obj = DK^\star$ as the initialized object.
  - If $\exists v \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v.\mathsf{obj} = obj$, abort execution. Else, add node $v^\star = (obj, \{\widehat{h}\})$ to $\mathbb{V}_{\mathsf{Test}}$.
- $\texttt{str} = (\mathsf{run}, (dk, \texttt{ekGen}))$ Let $v' \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $dk \in v'.\mathbf{L}$. Let $v'.\mathsf{obj} = DK$. Generate $EK = \mathsf{case.ekGen}(DK)$.
  - If $\exists v \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v.\mathsf{obj} = EK$, update $v.\mathbf{L} \leftarrow v.\mathbf{L} \cup \widehat{h}$. Else, add node $v^\star = (EK, \{\widehat{h}\})$ to $\mathbb{V}_{\mathsf{Test}}$. Add the edge $v' \xrightarrow{\text{ekGen}} v^\star$
- $\texttt{str} = (\mathsf{run}, (sk, \texttt{vkGen}))$ Let $v' \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $sk \in v'.\mathbf{L}$. Let $v'.\mathsf{obj} = SK$. Generate $VK = \mathsf{case.vkGen}(SK)$.
  - If $\exists v \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v.\mathsf{obj} = VK$, update $v.\mathbf{L} \leftarrow v.\mathbf{L} \cup \widehat{h}$. Else, add node $v^\star = (VK, \{\widehat{h}\})$ to $\mathbb{V}_{\mathsf{Test}}$. Add the edge $v' \xrightarrow{\text{vkGen}} v^\star$
- $\texttt{str} = (\mathsf{run}, ((sk, (\texttt{encase}, m)), (ek, (\texttt{encase}, m))))$ Let $v_1 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $sk \in v'.\mathbf{L}$ and $v_2 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $dk \in v'.\mathbf{L}$. Let $v_1.\mathsf{obj} = SK$ and $v_2.\mathsf{obj} = EK$.
  - Generate $obj = \mathsf{case.encase}(SK, EK, m)$.
  - If $\exists v \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v.\mathsf{obj} = obj$, abort execution. Else, add node $v^\star = (obj, \{\widehat{h}\})$ to $\mathbb{V}_{\mathsf{Test}}$. Add the edge $v_2 \xrightarrow{(pk-ct, \texttt{encase}, m)} v^\star$. Add the edge $v_1 \xrightarrow{(sk-ct, \texttt{encase}, m)} v^\star$.
- $\texttt{str} = (\mathsf{transfer}, \widehat{h}_0, \widehat{h}_1)$ Let $v' \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $\widehat{h}_b \in v'.\mathbf{L}$.
  - If $\exists v \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $v'.\mathsf{obj} = v.\mathsf{obj}$, update $v.\mathring{\mathbf{L}} \leftarrow v.\mathring{\mathbf{L}} \cup \{\mathring{r}\}$. Else, add node $v^\star = (v'.\mathsf{obj}, \{\mathring{r}\})$ to $\mathbb{V}_{\mathcal{A}}$ and add edge $v' \xrightarrow{\text{transfer}} v^\star$

$\underline{\texttt{update}_{\mathcal{A}}(\mathbb{G}_b, obj)}$

Let $t = \mathsf{acc}(obj)$. Let $\widehat{h}$ be the next handle to be received by Test. We consider different cases based on value of t.

- **If** $t \neq \bot$ **and** $\exists v$ $s.t.$ $v.\mathsf{obj} = obj$
  Update $v.\mathring{\mathbf{L}} \leftarrow v.\mathring{\mathbf{L}} \cup \{\mathring{r}\}$ in the matched node v. If $\exists v' \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v'.\mathsf{obj} = obj$, update $v'.\mathbf{L} \leftarrow v'.\mathbf{L} \cup \{\widehat{h}\}$.
- **Else, if** $t \neq \bot$
  Add node $v^{\star} = (obj, \{\mathring{r}\})$ to $\mathbb{V}_{\mathcal{A}}$.
  If $t = \text{EK} \vee t = \text{VK}$ and $\exists v \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v.\mathsf{obj} = obj$, update $v.\mathbf{L} \leftarrow v.\mathbf{L} \cup \{\widehat{h}\}$. Else, add node $v' = (obj, \{\widehat{h}\})$ to $\mathbb{V}_{\mathsf{Test}}$. Add the edge $v^{\star} \xrightarrow{\text{transfer}} v'$ to $\mathbb{G}_0$.
  Now do the following based on the value of t.
  - $t =$ **DK**
    - If $\exists v_1, v_2 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v_1.\mathsf{obj} = obj \wedge v_2.\mathsf{obj} = obj$, abort execution.
    - $\forall v \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{decase\text{-}msg}(obj, v.\mathsf{obj}) = m \neq \bot$, add edge $v^{\star} \xrightarrow{(dk-ct, \mathsf{encase}, m)} v$.
    - $\forall v \in \mathbb{V}_{\mathcal{A}}$ $s.t.\mathsf{ekGen}(obj) = v.\mathsf{obj}$, add edge $v^{\star} \xrightarrow{\text{ekGen}} v$.
    - $\forall v_1, v_2 \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{decase\text{-}verify}(obj, \mathsf{vkGen}(v_1.\mathsf{obj}), v_2.\mathsf{obj}) = m \neq \bot$, add edge $v_1 \xrightarrow{(sk-ct, \mathsf{encase}, m)} v_2$.
    - $\forall v_1, v_2 \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{decase\text{-}verify}(obj, v_1.\mathsf{obj}, v_2.\mathsf{obj}) = m \neq \bot$, add edge $v_1 \xrightarrow{(vk-ct, \mathsf{encase}, m)} v_2$.
  - $t =$ **EK**
    - If $(\exists v_1 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $\mathsf{ekGen}(v_1.\mathsf{obj}) = obj) \wedge (\nexists v_2 \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{ekGen}(v_2.\mathsf{obj}) = obj)$, abort execution.
    - $\forall v \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{ekGen}(v.\mathsf{obj}) = obj$, add edge $v \xrightarrow{\text{ekGen}} v^{\star}$.
  - $t =$ **SK**
    - If $\exists v_1, v_2 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v_1.\mathsf{obj} = obj \wedge v_2.\mathsf{obj} = obj$, abort execution.
    - $\forall v \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{vkGen}(obj) = v.\mathsf{obj}$, add edge $v^{\star} \xrightarrow{\text{vkGen}} v$.
    - $\forall v_1, v_2 \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{decase\text{-}verify}(v_1.\mathsf{obj}, \mathsf{vkGen}(obj), v_2.\mathsf{obj}) = m \neq \bot$, add edge $v^{\star} \xrightarrow{(sk-ct, \mathsf{encase}, m)} v_2$. Else if no such edge can be added, $\forall v_1, v_2 \in \mathbb{V}_{\mathcal{A}}, v_3 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v_3 \xrightarrow{\text{ekGen}} v_s \xrightarrow{\text{transfer}} v_1 \wedge \mathsf{decase\text{-}verify}(v_3.\mathsf{obj}, \mathsf{vkGen}(obj), v_2.\mathsf{obj}) = m \neq \bot$, add edge $v^{\star} \xrightarrow{(sk-ct, \mathsf{encase}, m)} v_2$.
  - $t =$ **VK**
    - If $(\exists v_1 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $\mathsf{vkGen}(v_1.\mathsf{obj}) = obj) \wedge (\nexists v_2 \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{vkGen}(v_2.\mathsf{obj}) = obj)$, abort execution.
    - $\forall v \in \mathbb{V}_{\mathcal{A}}$ $s.t.\mathsf{vkGen}(v.\mathsf{obj}) = obj$, add edge $v \xrightarrow{\text{vkGen}} v^{\star}$.
    - $\forall v_1, v_2 \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{decase\text{-}verify}(v_1.\mathsf{obj}, obj, v_2.\mathsf{obj}) = m \neq \bot$, add edge $v^{\star} \xrightarrow{(vk-ct, \mathsf{encase}, m)} v_2$. Else if no such edge can be added, $\forall v_1, v_2 \in \mathbb{V}_{\mathcal{A}}, v_3 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v_3 \xrightarrow{\text{ekGen}} v_s \xrightarrow{\text{transfer}} v_1 \wedge \mathsf{decase\text{-}verify}(v_3.\mathsf{obj}, obj, v_2.\mathsf{obj}) = m \neq \bot$, add edge $v^{\star} \xrightarrow{(vk-ct, \mathsf{encase}, m)} v_2$.

- $t = \mathbf{CP}$
  - $*$ If $\exists v_1, v_2 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $v_1.\mathsf{obj} = obj \wedge v_2.\mathsf{obj} = obj$, abort execution.
  - $*$ If $\exists v_1 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $\mathsf{decase\text{-}msg}(v_1.\mathsf{obj}, obj) = m \neq \perp \wedge (\nexists v_2 \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $v_2.\mathsf{obj} = v_1.\mathsf{obj} \vee v_2.\mathsf{obj} = \mathsf{ekGen}(v_1.\mathsf{obj}))$, abort execution.
  - $*$ If $\exists v_1, v_2 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $\mathsf{decase\text{-}verify}(v_1.\mathsf{obj}, \mathsf{vkGen}(v_2.\mathsf{obj}), obj) = m \neq \perp \wedge (\nexists v_3 \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $v_3.\mathsf{obj} = v_2.\mathsf{obj})$, abort execution.
  - $*$ $\forall v \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $\mathsf{decase\text{-}msg}(v.\mathsf{obj}, obj) = m \neq \perp$, add edge $v \xrightarrow{(dk-ct,\mathsf{encase},m)} v^{\star}$ and set $DK^{\star} = v.\mathsf{obj}$. Else if no such edge can be added, $\forall v_1 \in \mathbb{V}_{\mathsf{Test}}$ $s.t.$ $\mathsf{decase\text{-}msg}(v_1.\mathsf{obj}, obj) = m \neq \perp \wedge (\exists v_2 \in \mathbb{V}_{\mathcal{A}}$ $s.t.$ $v_1 \xrightarrow{\mathsf{ekGen}} v_s \xrightarrow{\mathsf{transfer}} v_2)$, add edge $v_0 \xrightarrow{(pk-ct,\mathsf{encase},m)} v^{\star}$ and set $DK^{\star} = v_1.\mathsf{obj}$. Else, set $DK^{\star} = \perp$.
  - $*$ If $DK^{\star} \neq \perp$, $\forall v \in \mathbb{V}_{\mathcal{A}}$ $s.t.\mathsf{decase\text{-}verify}(DK^{\star}, \mathsf{vkGen}(v.\mathsf{obj}), obj) = m \neq \perp$, add edge $v \xrightarrow{(sk-ct,\mathsf{encase},m)} v^{\star}$.
  - $*$ If $DK^{\star} \neq \perp$, $\forall v \in \mathbb{V}_{\mathcal{A}}$ $s.t.\mathsf{decase\text{-}verify}(DK^{\star}, v.\mathsf{obj}, obj) = m \neq \perp$, add edge $v \xrightarrow{(vk-ct,\mathsf{encase},m)} v^{\star}$.

Fig. 23: Description of $\mathbb{G}_0$

We also note some properties of the graph $\mathbb{G}_0$ that hold at every round conditioned on $\mathbb{G}_0$ not aborting.

1. If $v_1, v_2 \in \mathbb{V}_{\mathsf{Test}}$ and $v_1 \neq v_2$, then $obj_1 \neq obj_2$.
2. If $\mathsf{acc}(obj_1) = \text{EK}$ (resp. VK), $\mathsf{acc}(obj_2) = \text{EK}$ (resp. VK) and $obj_1 = obj_2$, then $\texttt{root}\ (v_1) = \texttt{root}\ (v_2)$.
3. If $\mathsf{acc}(obj_1) = \text{DK}$ (resp. SK), $\mathsf{acc}(obj_2) = \text{EK}$ (resp. VK) and $obj_2 = \mathsf{ekGen}(obj_1)$ (resp. $obj_2 = \mathsf{vkGen}(obj_1)$), then $\texttt{root}\ (v_1) = \texttt{root}\ (v_2)$
4. If $\mathsf{acc}(obj_1) = \text{DK}$, $\mathsf{acc}(obj_2) = \text{CP}$ and $\mathsf{decase\text{-}msg}(obj_1, obj_2) \neq \perp$, then $\texttt{root}\ (v_1) = \texttt{dk-root}\ (v_2)$
5. If $\mathsf{acc}(obj_1) = \text{DK}$, $\mathsf{acc}(obj_2) = \text{VK}$ (resp. SK), $\mathsf{acc}(obj_3) = \text{CP}$ and $\mathsf{decase\text{-}verify}(obj_1, obj_2, obj_3) \neq \perp$ (resp. $\mathsf{decase\text{-}verify}(obj_1, \mathsf{vkGen}(obj_2), obj_3)$), then $\texttt{root}\ (v_2) = \texttt{sk-root}\ (v_2)$

where v_1, v_2, v_3 denote arbitrary nodes chosen from $\mathbb{G}_0$ and obj_1, obj_2, obj_3 represent the objects present inside them.

C.3.2 Hybrids $\mathsf{H}_{0|1}$ and H_1 Hybrid H_1 corresponds to $\text{IDEAL}\langle \mathsf{Test}(0) \mid \Sigma^{\ddagger}_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^{\dagger}_0 \circ \mathcal{A} \rangle$, where the implementation $\mathcal{I}'[\Pi, \mathbb{G}_0]$ from previous hybrid $\mathsf{H}_{0|1}$ is replaced by the ideal extended schema $\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}$ (as in Figure 21) and a simulator $\mathcal{S}^{\dagger}_0$ (as in Figure 26).

Bad events We now specify the "bad events" in H_0 which can cause $\mathsf{H}_{0|1}$ to abort in Figure 24. Note that, all abort conditions in $\texttt{update}_{\mathsf{Test}}$ and $\texttt{update}_{\mathcal{A}}$ correspond to one the the events above. Thus, conditioned on bad events not occurring, $\mathsf{H}_{0|1}$ does not abort.

We also specify the a "bad event" for H_1 in Figure 25 which can cause executions of $\mathsf{H}_{0|1}$ and H_1 to diverge due to "tag collisions". The probability of this event occuring is negligible as a polynomial number of tags are sampled uniformly from $\{0,1\}^{\kappa}$.

1. In H_0, Test generates a signing-key SK (resp. decryption-key DK) which is equal to another obj which exists in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ or is such that $SK = \mathsf{skId}\ (obj)$ or $SK = \mathsf{skId}\ (\mathsf{vkId}\ (obj))$ (resp. $DK = \mathsf{dkId}\ (obj)$ or $DK = \mathsf{dkId}\ (\mathsf{ekId}\ (obj)))$ for an obj which exists in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$.
2. In H_0, Test generates a CP which is equal to another obj which exists in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$.
3. In H_0, a signing-key SK (resp. decryption-key DK or CP) transferred by the user is equal to an object obj which exists in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ but was not transferred by or to Test.
4. In H_0, a verification-key VK (resp. public-key EK) transferred by the user is such that there exists a signing-key SK (resp. decryption-key DK) inside the work-tape contents of a handle $\widehat{h}$ that was created through the init command in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ such that $VK = \mathsf{vkGen}(SK)$ (resp. $EK = \mathsf{ekGen}\ (DK)$) and $\widehat{h}$ or a verification-key (resp. encryption-key) handle derived from $\widehat{h}$ was never transferred to $\mathcal{A}$.
5. In H_0, $\mathcal{A}$ transfers CP such that there is a decryption-key DK generated by $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ which satisfies $\mathsf{decase\text{-}msg}(DK, CP) \neq \perp$ and DK or $EK = \mathsf{ekGen}\ (DK)$ has not been transferred to $\mathcal{A}$.
6. In H_0, $\mathcal{A}$ transfers CP such that there is a decryption-key DK in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ and a signing-key SK generated by $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ which satisfies $\mathsf{decase\text{-}verify}(DK, \mathsf{vkGen}(SK), CP) \neq \perp$ and SK has not been transferred to $\mathcal{A}$.

Fig. 24: Bad events in H_0

1. In H_1, creation or evolution of a $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ agent results in the same *sk-tag*, *dk-tag* or *cp-tag* as in a previous command.

Fig. 25: Bad events in H_1

$\mathcal{S}^{\dagger}_{b}$: Processing objects and commands

Processing objects transferred by $\mathcal{A}$ When $\mathcal{A}$ attempts to transfer object obj to Test, it calls a subroutine $\mathtt{update}^{\dagger}\mathcal{A}(\mathbb{G}^{\dagger}_{b}, obj)$ which returns a handle $\overline{h}$. Send the command $(\mathsf{transfer}, \overline{h})$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$.

$\underline{\mathtt{update}^{\dagger}_{\mathcal{A}}(\mathbb{G}^{\dagger}_{b}, obj)}$

– Run $\mathtt{update}_{\mathcal{A}}(\mathbb{G}^{\dagger}_{b}, obj)$

– Let v' be the node of the form $(obj, \widehat{\mathbf{L}})$ added to $\mathbb{V}_{\mathsf{Test}}$ by $\mathtt{update}$. Let $v^{\star}$ be the node of the form $(obj, \mathring{\mathbf{L}})$ added to $\mathbb{V}_{\mathcal{A}}$ by $\mathtt{update}$. If any of these nodes are not added, $v^{\star} = \perp$ and/or $v' = \perp$. Let $t = \mathsf{acc}(obj)$ during the execution of $\mathtt{update}$.

– If $v^{\star} = \perp$, find a node $v \in \mathbb{V}_{\mathcal{A}}$ such that $v.\mathsf{obj} = obj$ and return any $h \in v.\overline{h}$.

– Else, if $t = \textsc{sk}$, proceed as follows

- Find a node $v_1 \in \mathbb{V}_{\mathcal{A}}$ such that the edge $v^\star \xrightarrow{\text{vkGen}} v_1$ exists.
- If v_1 does not exist, send a command $(\mathsf{init}, (\textsc{sk}, \kappa))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ and obtain handle h'. Add node $v_2 = (\bot, \bot, h')$ to $\mathbb{V}_{\mathcal{A}}$. Now send the command $(\mathsf{run}, (h', (\mathtt{patch}, \mathit{obj})))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ to obtain $\overline{h}$. Update node $v^\star$ to $(\mathit{obj}, \{\mathring{r}\}, \{\overline{h}\})$. Add edge $v_2 \xrightarrow{\text{patch}} v^\star$.
- Else, find node $v_2 = (\bot, \bot, h') \in \mathbb{V}_{\mathcal{A}}$ such that the path $v_2 \xrightarrow{\text{vkGen}} v_s \xrightarrow{\text{patch}} v_1$ exists. Send the command $(\mathsf{run}, (h', (\mathtt{patch}, \mathit{obj})))$ to obtain $\overline{h}$. Update node $v^\star$ to $(\mathit{obj}, \{\mathring{r}\}, \{\overline{h}\})$. Add edge $v_2 \xrightarrow{\text{patch}} v^\star$.

– Else, if $t = \textsc{dk}$, proceed as follows

- Construct a list of node-pairs $\{(v_{i1}, v_{i2})\}$ such that $v_{i1} \xrightarrow{(sk-ct, \mathsf{encase}, m)} v_{i2}$ and $v^\star \xrightarrow{(dk-ct, \mathsf{encase}, m)} v_{i2}$ where $v_{i1}, v_{i2} \in \mathbb{V}_{\mathcal{A}}$.
- Now, construct the list $S = \{(h_i, \mathit{CP}_i)\}$ where $h_i \in v_{i1}.\overline{h}$ and $\mathit{CP}_i = v_{i2}.\mathsf{obj}$.
- If $\exists v_1 \in \mathbb{V}_{\mathcal{A}}$ *s.t.* $v^\star \xrightarrow{\text{ekGen}} v_1$, find $v_2 = (\bot, \bot, h')$ such that the path $v_2 \xrightarrow{\text{ekGen}} v_s \xrightarrow{\text{patch}} v_1$ exists. Send $(\mathsf{run}, (h', (\mathtt{dkPatch}, \mathit{obj})), \{(h_i, (\mathtt{dkPatch}, \mathit{CP}_i))\}_{(h_i, \mathit{CP}_i) \in S})$ to obtain $\overline{h}$. Update node $v^\star$ to $(\mathit{obj}, \{\mathring{r}\}, \{\overline{h}\})$. Add edge $v_2 \xrightarrow{\text{patch}} v^\star$.
- Else, send a command $(\mathsf{init}, (\textsc{dk}, \kappa))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ and obtain handle h'. Add node $v_2 = (\bot, \bot, h')$ to $\mathbb{V}_{\mathcal{A}}$. Now send the command $(\mathsf{run}, (h', (\mathtt{patch}, \mathit{obj})), \{(h_i, (\mathtt{dkPatch}, \mathit{CP}_i))\}_{(h_i, \mathit{CP}_i) \in S})$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ to obtain $\overline{h}$. Update node $v^\star$ to $(\mathit{obj}, \{\mathring{r}\}, \{\overline{h}\})$. Add edge $v_2 \xrightarrow{\text{patch}} v^\star$.

– Else if $t = \textsc{ek}$, proceed as follows,

- If $\exists v_1 \in \mathbb{V}_{\mathcal{A}}$ *s.t.* $v_1 \xrightarrow{\text{ekGen}} v^\star$, send the command $(\mathsf{run}, (\overline{h}_1, \mathtt{ekGen}))$ to obtain $\overline{h}$ where $\overline{h}_1 \in v_1.\overline{h}$. Update node $v^\star$ to $(\mathit{obj}, \{\mathring{r}\}, \{\overline{h}\})$.
- Else, send a command $(\mathsf{init}, (\textsc{dk}, \kappa))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ and obtain handle h'. Add node $v_1 = (\bot, \bot, h')$ to $\mathbb{V}_{\mathcal{A}}$. The, send the command $(\mathsf{run}, (h', \mathtt{ekGen}))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ and obtain handle h''.Add node $v_2 = (\bot, \bot, h'')$ to $\mathbb{V}_{\mathcal{A}}$. Send the command $(\mathsf{run}, (h'', (\mathtt{patch}, \mathit{obj})))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ and obtain handle h''. Update node $v^\star$ to $(\mathit{obj}, \{\mathring{r}\}, \{\overline{h}\})$. Add edges $v_1 \xrightarrow{\text{ekGen}} v_2$ and $v_2 \xrightarrow{\text{patch}} v^\star$.

– Else if $t = \textsc{vk}$, proceed as follows,

- If $\exists v_1 \in \mathbb{V}_{\mathcal{A}}$ *s.t.* $v_1 \xrightarrow{\text{vkGen}} v^\star$, send the command $(\mathsf{run}, (\overline{h}_1, \mathtt{vkGen}))$ to obtain $\overline{h}$ where $\overline{h}_1 \in v_1.\overline{h}$. Update node $v^\star$ to $(\mathit{obj}, \{\mathring{r}\}, \{\overline{h}\})$.
- Else, send a command $(\mathsf{init}, (\textsc{sk}, \kappa))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ and obtain handle h'. Add node $v_1 = (\bot, \bot, h')$ to $\mathbb{V}_{\mathcal{A}}$. The, send the command $(\mathsf{run}, (h', \mathtt{ekGen}))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ and obtain handle h''. Add node $v_2 = (\bot, \bot, h'')$ to $\mathbb{V}_{\mathcal{A}}$. Send the command $(\mathsf{run}, (h'', (\mathtt{patch}, \mathit{obj})))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ to obtain $\overline{h}$. Update node $v^\star$ to $(\mathit{obj}, \{\mathring{r}\}, \{\overline{h}\})$. Add edges $v_1 \xrightarrow{\text{vkGen}} v_2$ and $v_2 \xrightarrow{\text{patch}} v^\star$.

– Else if $t = \textsc{cp}$, proceed as follows,

- Execute the following steps to find a "matching" encryption-key handle $\mathit{ek}^\star$ and a decryption-key object $\mathit{DK}^\star$
  1. If $\exists v_1 \in \mathbb{V}_{adv}$ *s.t.* $v_1 \xrightarrow{(dk-ct, \mathsf{encase}, m)} v^\star$, send the command $(\mathsf{run}, (\overline{h}_1, \mathtt{ekGen}))$ to obtain $\mathit{ek}^\star$ where $\overline{h}_1 \in v_1.\overline{h}$ and set $\mathit{DK}^\star = v_1.\mathsf{obj}$.
  2. Else if, $\exists v_1 \in \mathbb{V}_{\mathcal{A}}$ *s.t.* $v_1 \xrightarrow{(pk-ct, \mathsf{encase}, m)} v'$, find the node $v_2 \in \mathbb{V}_{\mathsf{Test}}$ such that the path $v_2 \xrightarrow{\text{ekGen}} v_s \xrightarrow{\text{transfer}} v_1$ exists. Set $\mathit{ek}^\star = \overline{h}_1$ where $\overline{h}_1 \in v_1.\overline{h}$ and $\mathit{DK}^\star = v_2.\mathsf{obj}$.
  3. Else, $\mathit{ek}^\star = \bot$ and $\mathit{DK}^\star = \bot$.

- Execute the following steps to find a matching signing-key handle handle $sk^\star$ and message $m^\star$
  1. If $\exists v_1 \in \mathbb{V}_{\mathcal{A}}$ *s.t.* $v_1 \xrightarrow{(sk-ct,\mathsf{encase},m)} v^\star$, set $sk^\star = \overline{h}_1$ where $\overline{h}_1 \in v_1.\overline{h}$ and $m^\star = m$.
  2. Else, set $sk^\star = \perp$ and $m^\star = \perp$.
- Execute the following steps to obtain the a handle $\overline{h}$.
  1. If $sk^\star \neq \perp \wedge ek^\star \neq \perp$, send the command $(\mathsf{run}, (sk^\star, (\mathtt{encase}, m)), (ek^\star, \perp))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ to obtain $\overline{h}$.
  2. Else if $ek^\star \neq \perp$, send the command $(\mathsf{run}, (ek^\star, (\mathtt{CPgen}, obj, DK^\star)))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ to obtain $\overline{h}$.
  3. Else, send the command $(\mathsf{init}, (\mathtt{CPgen}, obj))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ to obtain $\overline{h}$.
- Finally, update node $v^\star$ to $(obj, \{\mathring{r}\}, \{\overline{h}\})$.

– In all of the above cases with $v^\star \neq \perp$, return $\overline{h}$.

Processing reports of commands by Test **(except** $\mathsf{transfer}$**)** On obtaining the report of a command c which output a handle $\widehat{h}$, $\mathcal{S}^{\dagger}_b$ updates $\mathbb{G}^{\dagger}_b$ by executing $\mathtt{update}_{\mathsf{Test}}(\mathbb{G}^{\dagger}_b, \mathsf{c}, \widehat{h})$.

Processing transfers by Test Let $\overline{h}$ be the handle obtained by $\mathcal{S}^{\dagger}_b$ from $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ as a result of a transfer command $(\mathsf{transfer}, h_0, h_1)$ by Test. $\mathcal{S}^{\dagger}_b$ updates $\mathbb{G}^{\dagger}_b$ by executing $\mathtt{update}_{\mathsf{Test}}(\mathbb{G}^{\dagger}_b, \mathsf{c}, \widehat{h})$. Let $v^\star \in \mathbb{V}_{\mathcal{A}}$ such that $v^\star.\mathsf{obj} = obj$. Update $v^\star.\overline{h} \leftarrow v^\star.\overline{h} \cup \{\overline{h}\}$ (or $v^\star.\overline{h} = \{\overline{h}\}$ if $v^\star.\overline{h}$ did not exist)

Fig. 26: $\mathcal{S}^{\dagger}_b$: Processing objects transferred by $\mathcal{A}$

We *couple* the executions of H_0, $\mathsf{H}_{0|1}$, H_1 by considering a single experiment which runs all three executions using a common random-tape. The randomness used by $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ for operations of Π_{case} in H_0 are identified with the randomness used by $\mathcal{I}'[\Pi, \mathbb{G}_0]$ for operations of Π_{case} in $\mathsf{H}_{0|1}$ and the randomness used by $\mathcal{S}^{\dagger}_0$ in H_1. The randomness used in H_1 by $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ to sample the tags (*sk-tag*, *dk-tag*, or *cp-tag*) are not used in H_0 or $\mathsf{H}_{0|1}$. The random-tapes of the adversary and Test are the same in all three parts of the coupled execution.

A coupled execution does not diverge if the view of the adversary and Test is identical in H_0, $\mathsf{H}_{0|1}$ and H_1. Conditioned on the bad events not occurring, we claim that a coupled execution does not diverge which is formally stated in the lemma below.

Lemma 9. *Conditioned on bad events in Figure 24 and in Figure 25 not occurring in a coupled execution of* H_0*,* $\mathsf{H}_{0|1}$ *and* H_1*, the joint view of* $(\mathsf{Test}, \mathcal{A})$ *is the same in* H_0*,* $\mathsf{H}_{0|1}$ *and* H_1*.*

Proof:

This is verified inductively, over each message from the adversary or Test. Indeed, as long as there have been no divergence previously, the objects sent to the adversary – created by $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ or $\mathcal{I}'[\Pi, \mathbb{G}_0]$ or $\mathcal{S}^{\dagger}_0$ – and the outputs received by Test are identical in H_0, $\mathsf{H}_{0|1}$ and H_1.

First we note that, by construction of $\mathbb{G}^{\dagger}_0$, the induced subgraph $\mathbb{G}^{\dagger}_{0p}$ over all nodes NOT of the form $(\perp, \perp, h)$ in $\mathbb{G}^{\dagger}_0$ in H_1 is structurally equivalent to $\mathbb{G}_0$ in $\mathsf{H}_{0|1}$.

Now, we prove that conditioned on the bad events not occurring, some invariants hold at every round. Note that the proof of each invariant for the current round follows an inductive argument and assumes that all invariants hold till the current round.

Claim 1. *The objects received by* $\mathcal{A}$ *corresponding to a transfer command* c *by* Test *is same in both* H_0*,* $\mathsf{H}_{0|1}$ *and* H_1*.*

Proof: As $\mathbb{G}^{\dagger}_{0p}$ is equivalent to $\mathbb{G}_0$, thus $\mathtt{node}_{\mathbb{G}^{\dagger}_{0p}}(\widehat{h}).\mathsf{obj} = \mathtt{node}_{\mathbb{G}_0}(\widehat{h}).\mathsf{obj}$. Moreover, by the coupling of randomness of $\mathsf{H}_{0|1}$ and H_0 and by construction of $\mathsf{H}_{0|1}$, the object inside the work-tape of a handle $\widehat{h}$ in H_0 is equal to $\mathtt{node}_{\mathbb{G}_0}(\widehat{h}).\mathsf{obj}$. Thus, the object transferred to adversary on invocation of a $\mathsf{transfer}$ command is same in all three hybrids. □

Claim 2. *For every command* c *sent by* Test*, the output of* $\mathtt{getView}_{\mathsf{Test}}(\mathbb{G}_0,\ \mathsf{c})$ *is equal to the output received by* Test *in* H_t *for* $t \in \{0, 1\}$

Proof: We will prove this invariant for each value of c.

- $\underline{\mathsf{c} = (\mathsf{run}, (h, \mathtt{type}))}$

  For H_0, the invariant is easy to verify. $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ in H_0 runs the algorithm $\mathsf{case.acc}$ on the underlying object corresponding to h. $\mathtt{getView}_{\mathsf{Test}}(\mathbb{G}_0,\ \mathsf{c})$ also runs $\mathsf{case.acc}$ on the object stored in the graph which is equal to the underlying object corresponding to h.

  In H_1, the construction of the $\mathbb{G}_0^{\dagger}$ ensures that the type of the handle inserted in a node corresponds to the value of $\mathsf{case.acc}(obj)$ where obj is the object inserted in the node. Thus, as $\mathbb{G}_{0p}^{\dagger}$ is structurally equivalent to $\mathbb{G}_0$, $\mathtt{getView}_{\mathsf{Test}}(\mathbb{G}_0,\ \mathsf{c})$ returns the same value as $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ in H_1.
- $\underline{\mathsf{c} = (\mathsf{run}, (h_1, \mathtt{compare}), (h_2, \mathtt{compare}))}$

  For H_0, the invariant is easy to verify. By the graph invariant 1, each node in $\mathbb{V}_{\mathsf{Test}}$ has a different object present inside it which is, by construction, the object associated with every handle belonging to the node. Thus, the command c returns true in H_0 iff the h_1 and h_2 belong to the same node i.e. $\mathtt{getView}_{\mathsf{Test}}(\mathbb{G}_0,\ \mathsf{c})$ returns true.

  In H_1, we prove the invariant in two parts.

  First, we prove that if $\mathtt{getView}_{\mathsf{Test}}(\mathbb{G}_0',\ \mathsf{c})$ returns true, then c returns true in H_1. To this end, we consider all cases when a new handle $h^{\star}$ is added to a list $v^{\star}.\mathbf{L}$ for a node $v^{\star} \in \mathbb{G}_0^{\dagger}$ and prove that $h^{\star}$ returns true on running $\mathtt{compare}$ with other handles in $v^{\star}.\mathbf{L}$. By construction of the simulator, $h^{\star}$ can be added to $v^{\star}.\mathbf{L}$ only if it is a transfer by $\mathcal{A}$ or it is obtained by a command $(\mathsf{run}, (h', \mathtt{ekGen}))$ or $(\mathsf{run}, (h', \mathtt{vkGen}))$.

  If $h^{\star}$ is added by a transfer of handle $\overline{h}$ which existed before receiving the command c, then $\exists v_1 \in \mathbb{V}_{\mathcal{A}}$ *s.t.* $(v_1 \xrightarrow{\text{transfer}} v^{\star} \vee v^{\star} \xrightarrow{\text{transfer}} v_1) \wedge \overline{h} \in v_1.\overline{h}$ and $\overline{h}$ has the same work-tape contents as with other handles in $v^{\star}.\mathbf{L}$ by semantics of $\mathsf{transfer}$ and induction on Claim 2. Thus, $h^{\star}$ returns true on running $\mathtt{compare}$ with all handles in $v^{\star}.\mathbf{L}$.

  Now we consider the case when $h^{\star}$ is added by a transfer of handle $\overline{h}$ which did not exist before receiving the command c. If $h^{\star}$ is added to a new node $v^{\star}$, the $h^{\star}$ is the only handle in $v^{\star}.\mathbf{L}$ and we are done. Otherwise, let $v^{\star}$ be the existing node to which $h^{\star}$ is added. Then, by the graph invariants 1 and 2, the object transferred by $\mathcal{A}$ can only be a encryption-key or a verification key whose corresponding decryption-key or signing-key has been transferred before. Thus, there exists a path $v^{\star} \xleftarrow{\text{ekGen}} v_r \xrightarrow{\text{transfer}} v_s \xrightarrow{\text{ekGen}} v_1$ or $v^{\star} \xleftarrow{\text{ekGen}} v_s \xleftarrow{\text{transfer}} v_r \xrightarrow{\text{ekGen}} v_1$ such that $\overline{h} \in v_1.\overline{h}$. Thus, $\overline{h}$ compares with all handles in $v^{\star}.\mathbf{L}$ by the semantics of $\mathsf{transfer}$ and $\mathtt{ekGen}$ and induction on Claim 2. Thus, $h^{\star}$ compares with all handles in $v^{\star}.\mathbf{L}$.

  If $h^{\star}$ is obtained through a command $(\mathsf{run}, (h', \mathtt{ekGen}))$ or $(\mathsf{run}, (h', \mathtt{vkGen}))$, let v_2 such that $h' \in v_2.\mathbf{L}$.

  If $v_2 \xrightarrow{\text{ekGen}} v^{\star}$ exists, then it is easy to see that $h^{\star}$ will compare with all handles in $v_2.\mathbf{L}$ using semantics of $\mathtt{ekGen}$. Else, the graph invariant 3 implies there must exist a path $v_2 \xrightarrow{\text{transfer}} \xrightarrow{\text{ekGen}} \xrightarrow{\text{transfer}} v^{\star}$ or $v_2 \xleftarrow{\text{transfer}} v_r \xrightarrow{\text{ekGen}} \xrightarrow{\text{transfer}} v$. Thus, using the semantics of $\mathtt{ekGen}$ and $\mathsf{transfer}$, $h^{\star}$ will compare with all handles in $v^{\star}.\mathbf{L}$.

  A similar argument holds for $\mathtt{vkGen}$.

  This proves that if two handles h_1 and h_2 belong to the same node then c returns true in H_1.

  Now, we prove that if two handles h_1 and h_2 *do not* belong to the same node then c returns false in H_1. To this end, using the equivalence property of $\mathtt{compare}$, we only need consider cases when a new handle $h^{\star}$ is added to a list $v^{\star}.\mathbf{L}$ in a *new* node $v^{\star}$ and prove that $h^{\star}$ returns false on running $\mathtt{compare}$ with all other handles. By construction of the simulator, a new node is added during $\mathtt{skGen}$, $\mathtt{vkGen}$, $\mathtt{encase}$ and may be added during $\texttt{vkGen}$, $\texttt{ekGen}$ or transfers be $\mathcal{A}$.

During a run of $\texttt{skGen}$, $\texttt{vkGen}$, $\texttt{encase}$ by Test, a new *sk-tag*, *dk-tag* or *cp-tag* is generated by $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$. Thus, conditioned on non-occurrence of bad event 1, $h^\star$ returns false on running $\texttt{compare}$ with all other handles.

Next, let's consider the case new node is created during $\texttt{vkGen}$. Let the command leading to creation of $h^\star$ be $(\mathsf{run}, (h', \texttt{ekGen}))$. Suppose there exists a node v_3 such that $\exists \widehat{h}_1 \in v_3.\mathbf{L}$ where $\widehat{h}_1$ returns true on $\texttt{compare}$ with $h^\star$.

Now, $v^\star.\mathsf{obj} \neq v_3.\mathsf{obj}$, as a result of the graph invariant 1. Let $v_4 = \texttt{root}\ (v^\star)$ and $v_5 = \texttt{root}\ (v_3)$. As $v^\star.\mathsf{obj} \neq v_3.\mathsf{obj}$, thus $v_4.\mathsf{obj} \neq v_4.\mathsf{obj}$ but *dk-tag* in handles of v_4 and v_5 is the same by semantics of $\texttt{ekGen}$ and $\mathsf{transfer}$. The graph invariant 1 also implies $v_4 \neq v_5$. But, conditioned on non-occurrence of 1, this is a contradiction because creation of v_4 and v_5 involved independent sampling of *dk-tag*. A similar argument holds for $\texttt{vkGen}$.

A new node may be created when $\mathcal{A}$ transfers objects that have never been transferred previously. In a transfer of new object obj' by $\mathcal{A}$, let $\overline{h}$ be the handle transferred by $\mathcal{S}_b^\dagger$. If either the new object is present inside the work-tape (in case of *CP*) or a new *sk-tag*, *dk-tag*, *cp-tag* is sampled, then conditioned on non-occurrence of 1 as a result of the graph invariant 1, $h^\star$ returns false on running $\texttt{compare}$ with any other handle in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$. Else, an analysis similar to that for the command $\texttt{vkGen}$ yields a contradiction conditioned on non-occurrence of 1.

This proves that if two handles h_1 and h_2 DO NOT belong to the same node then c returns $\perp$ in H_1. Thus, the invariant is proved.

- $\underline{\mathsf{c} = (\mathsf{run}, (h_{dk}, \texttt{decase-verify}), (h_{vk}, \texttt{decase-verify}), (h, \texttt{decase-verify}))}$

By the construction of $\mathbb{G}_0$, the following properties hold. The relations between the objects are easy to verify. The relations between the handles are also easy to verify by the construction of $\mathbb{G}_0^\dagger$ and specification of $\Sigma^\ddagger_{\Pi_{\mathsf{case}}}$. Thus as, $\mathbb{G}_{0p}^\dagger$ in H_1 is structurally equivalent to $\mathbb{G}_0$ in $\mathsf{H}_{0|1}$, the properties hold for $\mathbb{G}_0$.

1. If two nodes $v_i = (obj_i, \widehat{\mathbf{L}}_i)$ (resp. (obj_i, $\mathring{\mathbf{L}}_i, \overline{h}_i$)) for $i \in \{1,2\}$ and $v_1 \xrightarrow{\text{ekGen}} v_2$, then $\mathsf{case.ekGen}(obj_1)$ $= obj_2$ and $dk\text{-}tag_1 = dk\text{-}tag_2$ where $dk\text{-}tag_i$ is the decryption-key tag on the work tape of handles in v_i
2. If two nodes $v_i = (obj_i, \widehat{\mathbf{L}}_i)$ (resp. (obj_i, $\mathring{\mathbf{L}}_i, \overline{h}_i$)) for $i \in \{1,2\}$ and $v_1 \xrightarrow{\text{vkGen}} v_2$, then $\mathsf{case.vkGen}(obj_1)$ $= obj_2$ and $sk\text{-}tag_1 = sk\text{-}tag_2$ where $sk\text{-}tag_i$ is the signing-key tag on the work tape of handles in v_i
3. If two nodes $v_i = (obj_i, \widehat{\mathbf{L}}_i)$ (resp. (obj_i, $\mathring{\mathbf{L}}_i, \overline{h}_i$)) for $i \in \{1,2\}$ and $v_1 \xrightarrow{\text{transfer}} v_2$, then $obj_1 = obj_2$ and work-tape contents of all handles in v_1 and v_2 are equal
4. If two nodes $v_i = (obj_i, \widehat{\mathbf{L}}_i)$ (resp. (obj_i, $\mathring{\mathbf{L}}_i, \overline{h}_i$)) for $i \in \{1,2\}$and $v_1 \xrightarrow{(dk-ct,\texttt{encase},m)} v_2$, then $\mathsf{case}.\mathsf{decase}\text{-}\texttt{msg}(obj_1,obj_2) = \mathrm{m} =$ output of $(\mathsf{run}, (h_1, \mathsf{decase\text{-}msg}), (h_2, \mathsf{decase\text{-}msg}))$.
5. If two nodes $v_i = (obj_i, \widehat{\mathbf{L}}_i)$ (resp. (obj_i, $\mathring{\mathbf{L}}_i, \overline{h}_i$)) for $i \in \{1,2\}$ and $v_1 \xrightarrow{(pk-ct,\texttt{encase},m)} v_2$, then $\mathsf{case.ekId}(obj_2) = obj_1$ and $dk\text{-}tag_1 = dk\text{-}tag_2$ where $dk\text{-}tag_i$ is the decryption-key tag on the work tape of handles in v_i and m is the message on the work-tape of handles in v_2
6. If two nodes $v_i = (obj_i, \widehat{\mathbf{L}}_i)$ (resp. (obj_i, $\mathring{\mathbf{L}}_i, \overline{h}_i$)) for $i \in \{1,2\}$ and $v_1 \xrightarrow{(sk-ct,\texttt{encase},m)} v_2$, then $\mathsf{case.skId}(obj_2) = obj_1$ and there exists $v_r = \texttt{dk-root}\ (obj_2) = (obj_r,\widehat{\mathbf{L}}_r)$ (resp. $(obj_r,\mathring{\mathbf{L}}_r,\ \overline{h}_r)$) such that either $sk\text{-}tag_1 = sk\text{-}tag_2$ where $sk\text{-}tag_i$ is the signing-key tag on the work tape of handles in v_i or obj_i are present on the work-tape contents of v_i or the pair $(obj_2,\ sk\text{-}tag_1)$ is present on work-tape of the handles of v_r.
7. If two nodes $v_i = (obj_i, \widehat{\mathbf{L}}_i)$ (resp. (obj_i, $\mathring{\mathbf{L}}_i, \overline{h}_i$)) for $i \in \{1,2\}$ and $v_1 \xrightarrow{(vk-ct,\texttt{encase},m)} v_2$, then $\mathsf{case.vkId}(obj_2) = obj_1$ and there exists $v_r = \texttt{dk-root}\ (obj_2) = (obj_r,\widehat{\mathbf{L}}_r)$ (resp. $(obj_r,\mathring{\mathbf{L}}_r,\overline{h}_r)$) such that either $sk\text{-}tag_1 = sk\text{-}tag_2$ where $sk\text{-}tag_i$ is the signing-key tag on the work tape of handles in v_i or obj_i are present on the work-tape contents of v_i or the pair $(obj_2,\ sk\text{-}tag_1)$ is present on work-tape of the handles of v_r.

If $\texttt{getView}_{\mathsf{Test}}(\mathbb{G}_0, \mathtt{c}) \neq \perp$, then using the above lemmas and composition of existential consistency guarantees, the output received by Test in H_0 will be the same.

Similarly for H_1, if $\texttt{getView}_{\mathsf{Test}}(\mathbb{G}'_1, \mathtt{c}) \neq \perp$, the above lemmas and the specification of $\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}$ imply that the output received by Test in H_1 will be the same.

We now consider all cases when $\texttt{getView}_{\mathsf{Test}}(\mathbb{G}_0, \mathtt{c}) = \perp$ for H_0. We refer to variables from the body of $\texttt{getView}_{\mathsf{Test}}$ during the proof. If the conditions involving acc fail, then by definition of decase-verify, $\mathtt{c}$ returns $\perp$ in H_0. If $\texttt{dk-root}(v_1) = \perp$, then, as a result of the graph invariant 4, either i) *obj* is derived by Test from a public-key *EK* transferred by $\mathcal{A}$ whose corresponding decryption-key has not been transferred or ii) *obj* is transferred by $\mathcal{A}$ such that there is no transferred *DK* or there is no *DK* created by Test whose encryption-key ekGen (DK) has been transferred which satisfies $\mathsf{decase\text{-}msg}(DK, obj) \neq \perp$. Thus, $\mathtt{c}$ returns $\perp$ in H_0. Otherwise, $v_4 = \texttt{dk-root}(v_1)$ and $DK^\star$ be the object associated with it. If $\texttt{sk-root}(v_1) = \perp$ and $\texttt{vk-root}(v_1) = \perp$, then *obj* is an object transferred by $\mathcal{A}$ such that there are no transferred *VK* or *SK* satisfying $\mathsf{decase\text{-}verify}(DK^\star, VK, obj) \neq \perp$ or $\mathsf{decase\text{-}verify}(DK^\star, \mathsf{vkGen}(SK), obj) \neq \perp$. Then, as a result of the graph invariant 5, the output of $\mathtt{c}$ in H_0 is $\perp$. Let the object inside $v_4 = obj_4$ and the object within $\texttt{root}$ $(v_2) = obj_{dk}$. If $v_4 \neq \texttt{root}(v_2)$, then by the graph invariant 1, $obj_4 \neq obj_{dk}$. Using the semantics of edges in $\mathbb{G}_0$, we know that $obj_4 = \mathsf{dkId}(obj)$ and thus, $\mathsf{decase\text{-}msg}(obj_{dk}, obj) = \perp$ and output of $\mathtt{c}$ is $\perp$ in H_0. Similarly if $v_5 \neq \texttt{root}(v_3)$, by the graph invariants 2, 1 and 3, output of $\mathtt{c}$ is $\perp$ in H_0.

We now consider all cases when $\texttt{getView}_{\mathsf{Test}}(\mathbb{G}'_0, \mathtt{c}) = \perp$ for H_1. If the conditions involving acc fail, then by specification either the type token in $h_{dk} \neq \mathtt{dk}$ or token in $h_{vk} \neq \mathtt{vk}$ or token in $h \neq \mathtt{cp}$ and thus, $\mathtt{c}$ returns $\perp$ in H_1. If $\texttt{dk-root}(v_1) = \perp$, then either i) *obj* is derived by Test from a public-key *EK* transferred by $\mathcal{A}$ whose corresponding decryption-key has not been transferred or ii) *obj* is transferred by $\mathcal{A}$ such that there is no transferred *DK* which satisfies $\mathsf{decase\text{-}msg}(DK, obj) \neq \perp$. In case i), the work-contents of h contain an *dk-tag* that does not exist in any decryption-key handle. In case ii), the work-tape contents of h are $(\textsc{cp}, obj)$ and no transferred *DK* which satisfies $\mathsf{decase\text{-}msg}(DK, obj) \neq \perp$. Thus, $\mathtt{c}$ returns $\perp$ as output. h cannot return non-$\perp$ output with handles created by Test because of the form of its work-tape contents. Otherwise, $v_4 = \texttt{dk-root}(v_1)$ and $DK^\star$ be the object associated with it. If $\texttt{sk-root}(v_1) = \perp$ and $\texttt{vk-root}(v_1) = \perp$, then *obj* is an object transferred by $\mathcal{A}$ such that there are no transferred *VK* or *SK* satisfying $\mathsf{decase\text{-}verify}(DK^\star, VK, obj) \neq \perp$ or $\mathsf{decase\text{-}verify}(DK^\star, \mathsf{vkGen}(SK), obj) \neq \perp$. In this case, the work-tape contents of h are $(\textsc{cp}, obj)$ or $(\textsc{cp}, CP, DK^\star, dk\text{-}tag)$. As there is no matching *VK* or *SK*, thus $\mathtt{c}$ returns $\perp$ as output in H_1 as well. In this, h cannot return non-$\perp$ output with verification-key handles created by Test because of the form of its work-tape contents. If $v_4 \neq \texttt{root}(v_2)$, then handles in v_4 and $\texttt{root}$ (v_2) contained different objects (by the proof for H_0) and different tags (conditioned on 1 not occurring). From the semantic properties of edges (4, 5, 1, 3), we can conclude that object or tag in the content of handles in v_4 matches that in the content of h, which cannot be the case for handles in v_2. Thus, $\mathtt{c}$ return $\perp$ in H_1. Similarly, conditioned on 1 not occurring and using semantic properties of edges (6, 7,2, 3), we can prove that $\mathtt{c}$ return $\perp$ in H_1 if $v_5 \neq \texttt{root}(v_3)$

- $\underline{\mathtt{c} = (\mathsf{run}, (h_{dk}, \texttt{decase-msg}), (h, \texttt{decase-msg}))}$

  The proof for this command is similar to the proof when $\mathtt{c} = (\mathsf{run}, (h_{dk}, \texttt{decase-verify}), (h_{vk}, \texttt{decase-verify}), (h, \texttt{decase-verify}))$.

□

Claim 3. *Views of the $\mathcal{A}$ and* Test *are same in H_0, $\mathsf{H}_{0|1}$ and H_1.*

Proof:

Claim 4. *In every round, the same command is sent by* Test *or the same object is transferred by $\mathcal{A}$ in H_0, $\mathsf{H}_{0|1}$ and H_1. Moreover,* Test *obtains a handle in H_0 iff* Test *obtains a handle in $\mathsf{H}_{0|1}$ iff* Test *obtains a handle in H_1. These two handles are referred to as "corresponding handles" hereafter because they will share the same handle identifier.*

Proof: The view of the Test and $\mathcal{A}$ consists of outputs of commands to $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ in H_0, $\mathcal{I}'[\Pi, \mathbb{G}_0]$ in $\mathsf{H}_{0|1}$ and $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ in H_1, objects received by adversary and the common communication channel. Using Claim 3 itself by induction and coupling the randomness used in all hybrids, we ensure that the same command is sent by Test or the same object is transferred by $\mathcal{A}$. Thus, Test obtains a handle in H_0 iff Test obtains a handle in $\mathsf{H}_{0|1}$ iff Test obtains a handle in H_1. □

Using Claim 4 and Claim 1 and Claim 2 coupled with the equivalence of $\mathbb{G}^{\dagger}_{0p}$ and $\mathbb{G}_0$, we can prove that the views of the $\mathcal{A}$ and Test are same in H_0, $\mathsf{H}_{0|1}$ and H_1. □

□

Lemma 10. *The probability of occurrence of bad events in* H_0 *(listed in Figure 24) is negligible.*

Proof:

1. In H_0, Test generates a signing-key SK (resp. decryption-key DK) which is equal to another obj which exists in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ or is such that $SK = \mathsf{skId}\ (obj)$ or $SK = \mathsf{skId}\ (\mathsf{vkId}\ (obj))$ (resp. $DK = \mathsf{dkId}\ (obj)$ or $DK = \mathsf{dkId}\ (\mathsf{ekId}\ (obj))$) for an obj which exists in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$.

   Proof: We prove that the following are negligible in κ.

   $$\max_{DK^* \in \mathcal{DK}} \Pr_{DK \leftarrow \mathsf{dkGen}(1^\kappa)} \Big[DK = DK^*\Big] \tag{3}$$

   $$\max_{SK^* \in \mathcal{SK}} \Pr_{SK \leftarrow \mathsf{skGen}(1^\kappa)} \Big[SK = SK^*\Big] \tag{4}$$

   $$\max_{PK^* \in \mathcal{EK}} \Pr_{PK \leftarrow \mathsf{ekGen}(\mathsf{dkGen}(1^\kappa))} \Big[PK = PK^*\Big] \tag{5}$$

   $$\max_{VK^* \in \mathcal{VK}} \Pr_{VK \leftarrow \mathsf{vkGen}(\mathsf{skGen}(1^\kappa))} \Big[VK = VK^*\Big] \tag{6}$$

   (3) is negligible from the total hiding of the CASE primitive as shown in the proof for Lemma 1.
   The value in (4) is negligible from the sender anonymity of the CASE primitive. Otherwise, we can create an adversary $\mathcal{A}^\star$ for the $\mathsf{distinguish\text{-}sans\text{-}VK}$ with a non-negligible probability of success. The adversary A_1 in the experiment uses the SK^* which maximizes (4) to create a CP using SK^* and EK and then, adversary outputs $b \in \{0, 1\}$ such that $\mathcal{D}(b, DK, CP) \neq \perp$.
   (4) is negligible directly from the unpredictability property of the CASE primitive.
   (3) and (4) imply that (5) and (6) are negligible due to existential consistency guarantees of the CASE primitive. □

2. In H_0, Test generates a CP which is equal to another obj which exists in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$.

   Proof: We prove that the following is negligible in κ.

   $$\max_{CP^* \in \mathcal{CP}} \Pr_{CP \leftarrow \mathsf{encase}(SK, PK, m)} \Big[CP = CP^*\Big] \quad \forall\ SK \in \mathcal{SK}, PK \in \mathcal{EK}, m \in \mathcal{M} \tag{7}$$

   (7) is negligible directly using the unpredictability of the CASE primitive.

   □

3. In H_0, a signing-key SK (resp. decryption-key DK or CP) transferred by the user is equal to an object obj which exists in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ but was not transferred by or to Test.

   Proof: The probability of such an SK being transferred is negligible from the unforgeability of the CASE primitive. Otherwise, we can construct an adversary $\mathcal{A}'$ for the forge with a non-negligible probability of success.

Let $\textsc{System}\langle \mathsf{Test}^\star \mid \Pi_{\mathsf{case}} \mid \mathcal{A}^\star \rangle$ such that the bad-event occurs in the system with non-negligible probability. Then, we construct $\mathcal{A}'$ such that it runs $\textsc{System}\langle \mathsf{Test}^\star \mid \Pi''_{\mathsf{case}} \mid \mathcal{A}^\star \rangle$ internally where Π''_{case} behaves as follows -

During the execution of $\textsc{System}\langle \mathsf{Test}^\star \mid \Pi''_{\mathsf{case}} \mid \mathcal{A}^\star \rangle$, it chooses a command of the form $(\mathsf{init}, (\textsc{sk}, \kappa))$ sent by $\mathsf{Test}^\star$ uniformly at random. Let $\widehat{sk}$ be the next handle expected by $\mathsf{Test}^\star$. Π''_{case} does NOT run the init command and sends $\widehat{sk}$ to $\mathsf{Test}^\star$. If a command of the form $(\mathsf{run}, (\widehat{h}, \mathtt{vkGen}))$ is received such that $\widehat{sk} \rightsquigarrow \widehat{h} \wedge \mathsf{type}(\widehat{h}) = \textsc{sk}$, then it generates $\widehat{h}_1$ using the VK given by the experiment where $\widehat{h}_1$ is the next handle expected by $\mathsf{Test}^\star$. Similarly, an $\mathtt{encase}$ command is simulated using the oracle $\mathcal{E}$. All other operations are handled as in Π_{case}. Note that $\textsc{System}\langle \mathsf{Test}^\star \mid \Pi_{\mathsf{case}} \mid \mathcal{A}^\star \rangle$ is indistinguishable from $\textsc{System}\langle \mathsf{Test}^\star \mid \Pi''_{\mathsf{case}} \mid \mathcal{A}^\star \rangle$ in the view of $\mathsf{Test}^\star$ and $\mathcal{A}^\star$. After the execution of $\textsc{System}\langle \mathsf{Test}^\star \mid \Pi''_{\mathsf{case}} \mid \mathcal{A}^\star \rangle$, $\mathcal{A}'$ chooses an SK_g transferred by $\mathcal{A}$ at random. It generates $DK \leftarrow \mathsf{dkGen}(1^\kappa)$ and chooses $m \in \mathcal{M}$. It then sends $(DK, \mathsf{encase}(SK_g, \mathsf{ekGen}(DK), m))$ as the challenge to the experiment.

As the probability that the bad event occurs in $\textsc{System}\langle \mathsf{Test}^\star \mid \Pi_{\mathsf{case}} \mid \mathcal{A}^\star \rangle$ is non-negligible and the probability that $\widehat{sk}$ and SK_g correspond to "guessed" signing-keys is non-negligible (number of operations in the system is polynomially bounded), thus, the probability of success in the experiment is non-negligible.

All adversaries in the following proofs can be constructed similarly for their respective experiments. We do not give details of the construction for further proofs but specify the property of the CASE primitive that is violated.

The probability of such an DK being transferred is negligible from the total hiding property of the CASE primitive. Otherwise, we can construct an adversary $\mathcal{A}^\star$ for the $\mathsf{distinguish\text{-}sans\text{-}DK}$ with a non-negligible probability of success.
The probability of such an CP being transferred is negligible due to the unpredictability property of the CASE primitive. □

4. In H_0, a verification-key VK (resp. public-key EK) transferred by the user is such that there exists a signing-key SK (resp. decryption-key DK) inside the work-tape contents of a handle $\widehat{h}$ that was created through the init command in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ such that $VK = \mathsf{vkGen}(SK)$ (resp. $EK = \mathsf{ekGen}\ (DK)$) and $\widehat{h}$ or a verification-key (resp. encryption-key) handle derived from $\widehat{h}$ was never transferred to $\mathcal{A}$.

   Proof: The probability of such an VK being transferred is negligible from the sender anonymity of the CASE primitive. Otherwise, we can construct an adversary $\mathcal{A}^\star$ for the $\mathsf{distinguish\text{-}sans\text{-}VK}$ with a non-negligible probability of success.
   The probability of such an EK being transferred is negligible from the encasing resistance of the CASE primitive. Otherwise, we can construct an adversary $\mathcal{A}^\star$ for the $\mathsf{encase\text{-}sans\text{-}EK}$ with a non-negligible probability of success.

   □

5. In H_0, $\mathcal{A}$ transfers CP such that there is a decryption-key DK generated by $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ which satisfies $\mathsf{decase\text{-}msg}(DK, CP) \neq \perp$ and DK or $EK = \mathsf{ekGen}\ (DK)$ has not been transferred to $\mathcal{A}$.

   Proof: The probability of such an CP being transferred is negligible from the encasing resistance of the CASE primitive. Otherwise, we can construct an adversary $\mathcal{A}^\star$ for the $\mathsf{encase\text{-}sans\text{-}EK}$ with a non-negligible probability of success.

   □

6. In H_0, $\mathcal{A}$ transfers CP such that there is a decryption-key DK in $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ and a signing-key SK generated by $\mathcal{I}[\Pi, \mathsf{Repo}_{\mathsf{Test}}]$ which satisfies $\mathsf{decase\text{-}verify}\ (DK,\ \mathsf{vkGen}\ (SK),\ CP) \neq \perp$ and SK has not been transferred to $\mathcal{A}$.

Proof: The probability of such an *CP* being transferred is negligible from the unforgeability of the CASE primitive. Otherwise, we can construct an adversary $\mathcal{A}^\star$ for the forge with a non-negligible probability of success. □

□

Thus, using Lemma 10 and Lemma 9, we can see that $\mathsf{H}_{0|1} \approx \mathsf{H}_0$ and $\mathsf{H}_{0|1} \approx \mathsf{H}_1$. This implies $\mathsf{H}_0 \approx \mathsf{H}_1$.

C.3.3 Hybrid $\mathsf{H}_{1|2}$ and $\mathsf{H}_{5|6}$ In this hybrid, we run the experiment $\textsc{ideal}\langle \mathsf{Test}(b) \mid \Sigma^{\ddagger}_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^{\ddagger}_b \circ \mathcal{A} \rangle$ with test bit $b = 0$ for $\mathsf{H}_{1|2}$ and $b = 1$ for $\mathsf{H}_{5|6}$, where $\mathcal{S}^{\ddagger}_b$ is as described in Figure 27. We list the main differences between $\mathcal{S}^{\dagger}_b$ (in H_1 and H_6) and $\mathcal{S}^{\ddagger}_0$:

1. **Handle Derivation Graph:** $\mathcal{S}^{\ddagger}_b$ maintains a graph $\mathbb{G}^{\ddagger}_b$ that it uses to simulate the view of $\mathcal{A}$. This graph is similar to $\mathbb{G}^{\dagger}_b$, except that each node contain an extra state st [18]. In addition, $\mathcal{S}^{\ddagger}_b$ also maintains a graph $\mathbb{G}^{\ddagger}_{1-b}$ corresponding to the bit $1 - b$.
2. **Lazy Assignment:** $\mathcal{S}^{\ddagger}_b$ only assigns an object to a test handle if it is needed to construct and send an object to $\mathcal{A}$ (please refer to Figure 28). A node in $\mathbb{V}_{\mathsf{Test}}$ has $\mathsf{st} = \bot$ if it is unassigned, $\mathsf{st} = \mathtt{T}$ if tentatively assigned and $\mathsf{st} = \mathtt{R}$ if it is assigned and transferred to $\mathcal{A}$. Further, $\mathcal{S}^{\ddagger}_b$ uses the same randomness to update both graphs $\mathbb{G}^{\ddagger}_0$ and $\mathbb{G}^{\ddagger}_1$. This is a key idea that will be useful later to simulate without using the bit b.
3. **Delta Test Check:** $\mathcal{S}^{\ddagger}_b$ aborts if a transfers from Test would result in revealing the bit b to $\mathcal{A}$ (please refer to Figure 30). As we show later, the function $\mathtt{checkDeltaHiding}^{\ddagger}$ returns false with negligible probability if Test is a hiding-test.

Simulator $\mathcal{S}^{\ddagger}_b$:

It maintains graphs $\mathbb{G}^{\ddagger}_0$, $\mathbb{G}^{\ddagger}_1$.

- **Processing objects transferred by $\mathcal{A}$:**
  Let the object from $\mathcal{A}$ be *obj* and the handle to be received by Test be $\widehat{h}$
  - sample $r \leftarrow \{0,1\}^{\kappa}$
  - $\forall b' \in \{0,1\}$, $\overline{h}_{b'} \leftarrow \mathtt{update}^{\dagger}_{\mathcal{A}}(\mathbb{G}^{\ddagger}_{b'}, obj;\ r)$ using randomness r
  - set $\mathtt{node}_{\mathbb{G}^{\ddagger}_0}(\widehat{h}).\mathsf{st} = \mathtt{R}$, $\mathtt{node}_{\mathbb{G}^{\ddagger}_1}(\widehat{h}).\mathsf{st} = \mathtt{R}$
  - send $\overline{h}_b$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$

- **Processing commands by Test:**
  Let the report from Test be $\mathtt{report}$ and handle received from $\mathcal{B}[\Sigma]$ be $\overline{h}$
  - sample $r \leftarrow \{0,1\}^{poly(\kappa)}$
  - $\forall b' \in \{0,1\}$, run $\mathtt{update}^{\ddagger}_{\mathsf{Test}}(\mathbb{G}^{\ddagger}_{b'}, b', \mathtt{report}, \overline{h};\ r)$ using randomness r
  - if $\mathtt{report} = (\mathsf{transfer}, \widehat{h_0}, \widehat{h_1})$:
    - ∗ if $\mathtt{checkDeltaHiding}^{\ddagger}12(\mathbb{G}^{\ddagger}_0, \mathbb{G}^{\ddagger}_1, \widehat{h_0}, \widehat{h_1}, \overline{h}) = \mathsf{false}$, abort
    - ∗ set $\mathtt{node}_{\mathbb{G}^{\ddagger}_0}(\widehat{h_0}).\mathsf{st} = \mathtt{R}$, $\mathtt{node}_{\mathbb{G}^{\ddagger}_1}(\widehat{h_1}).\mathsf{st} = \mathtt{R}$
    - ∗ send $\mathtt{node}_{\mathbb{G}^{\ddagger}_b}(\overline{h}).obj$ to $\mathcal{A}$.

Fig. 27: Simulator $\mathcal{S}^{\ddagger}_b$ in hybrid $\mathsf{H}_{1|2}$.

[18] that is, $\forall v \in \mathbb{G}^{\ddagger}_b.\mathbb{V}_{\mathsf{Test}}$, $v = (obj, \widehat{\mathbf{L}}, \mathsf{st})$

Function $\texttt{update}^{\ddagger}_{\textsf{Test}}\ (\mathbb{G}^{\ddagger}, b', \texttt{report}, \overline{h})$ **:**
Let the handle to be generated for command $\texttt{report}$ be $\widehat{h}$ [19]. Proceed as follows depending on $\texttt{report}$.

- $\texttt{str} = (\textsf{init}, (\texttt{key-type}, \kappa))$
  add node $\big(\bot, \{\ \widehat{h}\ \}, \bot\big)$ to $\mathbb{G}^{\ddagger}.\mathbb{V}_{\textsf{Test}}$

- $\texttt{str} = (\textsf{run}, (\widehat{g}, \texttt{vkGen}))$
  - if $\exists v \in \mathbb{G}^{\ddagger}.\mathbb{V}_{\textsf{Test}}$ s.t. $\texttt{Type}(v) = \textsc{vk}$ and $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{g}) \rightsquigarrow v$, then update $v.\widehat{\mathbf{L}} = v.\widehat{\mathbf{L}} \cup \widehat{h}$
  - else, add node $\big(\bot, \{\ \widehat{h}\ \}, \bot\big)$ to $\mathbb{G}^{\ddagger}.\mathbb{V}_{\textsf{Test}}$ and add edge $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{g}) \xrightarrow{\text{vkGen}} \texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h})$ to $\mathbb{G}^{\ddagger}$

- $\texttt{str} = (\textsf{run}, (\widehat{g}, \texttt{ekGen}))$
  - if $\exists v \in \mathbb{G}^{\ddagger}.\mathbb{V}_{\textsf{Test}}$ s.t. $\texttt{Type}(v) = \textsc{ek}$ and $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{g}) \rightsquigarrow v$, then update $v.\widehat{\mathbf{L}} = v.\widehat{\mathbf{L}} \cup \widehat{h}$
  - else, add node $\big(\bot, \{\ \widehat{h}\ \}, \bot\big)$ to $\mathbb{G}^{\ddagger}.\mathbb{V}_{\textsf{Test}}$ and add edge $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{g}) \xrightarrow{\text{ekGen}} \texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h})$ to $\mathbb{G}^{\ddagger}$

- $\texttt{str} = (\textsf{run}, ((\widehat{g_0}, (\texttt{encase}, m)), (\widehat{g_1}, (\texttt{encase}, m))))$
  - add node $\big(\bot, \{\ \widehat{h}\ \}, \bot\big)$ to $\mathbb{G}^{\ddagger}.\mathbb{V}_{\textsf{Test}}$
    add edge $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{g_0}) \xrightarrow{(sk-ct, \text{encase}, m)} \texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h})$ to $\mathbb{G}^{\ddagger}$
    add edge $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{g_1}) \xrightarrow{(pk-ct, \text{encase}, m)} \texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h})$ to $\mathbb{G}^{\ddagger}$

- $\texttt{str} = (\textsf{transfer}, \widehat{h_0}, \widehat{h_1})$
  If $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h_{b'}}).obj = \bot$, sample $r \leftarrow \{0,1\}^{poly(\kappa)}$ and run $\texttt{lazyAssign}^{\ddagger}(\mathbb{G}^{\ddagger}, \widehat{h_{b'}};\ r)$ using randomness r.
  Let the handle received from $\mathcal{B}\big[\Sigma\big]$ be $\overline{h}$ and the round number be $\mathring{r}$.
  - if $\exists v \in \mathbb{G}^{\ddagger}.\mathbb{V}_{\mathcal{A}}$ s.t. $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h_{b'}}) \rightarrow v$, then update $v.\overline{\mathbf{L}} = v.\overline{\mathbf{L}} \cup \{\overline{h}\}$, $v.\mathring{\mathbf{L}} = v.\mathring{\mathbf{L}} \cup \mathring{r}$
  - Else, add node $\big(\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h_{b'}}).obj,\ \{\ \overline{h}\ \},\ \{\ \mathring{r}\ \}\big)$ to $\mathbb{G}^{\ddagger}.\mathbb{V}_{\mathcal{A}}$ and add edge $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h_{b'}}) \rightarrow \texttt{node}_{\mathbb{G}^{\ddagger}}(\overline{h})$

Fig. 28: Function $\texttt{update}^{\ddagger}_{\textsf{Test}}$ used by simulator $\mathcal{S}^{\ddagger}_{b}$ in hybrid $\mathsf{H}_{1|2}$.

Function $\texttt{lazyAssign}^{\ddagger}\ (\mathbb{G}^{\ddagger}, \widehat{h})$ **:**
If $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h}).obj \neq \bot$, return $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h}).obj$. Else, proceed as follows:

- if $\texttt{Type}(\widehat{h}) = \textsc{sk}$:
  sample $SK \leftarrow \textsf{skGen}(1^{\kappa})$ and set $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h}).obj = SK$, $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h}).\textsf{st} = \texttt{T}$
- if $\texttt{Type}(\widehat{h}) = \textsc{dk}$:
  sample $DK \leftarrow \textsf{dkGen}(1^{\kappa})$ and set $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h}).obj = DK$, $\texttt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h}).\textsf{st} = \texttt{T}$

[19] recall that, $\widehat{h}$ is simply a number that is implicitly fixed from the execution so far

- if $\texttt{Type}(\widehat{h}) = \textsc{vk}$:
  run $\texttt{lazyAssign}^{\ddagger}\ \big(\texttt{sk-root}(\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}))\big)$, $VK \leftarrow \mathsf{vkGen}\big(\texttt{sk-root}(\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h})).obj\big)$ and set $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).obj = VK$, $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).\mathsf{st} = \texttt{T}$
- if $\texttt{Type}(\widehat{h}) = \textsc{ek}$:
  run $\texttt{lazyAssign}^{\ddagger}\ \big(\texttt{dk-root}(\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}))\big)$, $EK \leftarrow \mathsf{ekGen}\big(\texttt{dk-root}(\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h})).obj\big)$ and set $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).obj = EK$, $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).\mathsf{st} = \texttt{T}$

- if $\texttt{Type}(\widehat{h}) = \textsc{cp}$:
  let $v, w \in \mathbb{G}^{\ddagger}.\mathbb{V}_{\mathsf{Test}}$ s.t. $\texttt{Type}(v) = \textsc{sk}$, $\texttt{Type}(w) = \textsc{ek}$, $v \underset{m}{\rightsquigarrow} \texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h})$ and $w \underset{m}{\rightsquigarrow} \texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h})$
  $SK = \texttt{lazyAssign}^{\ddagger}\ (v)$, $EK = \texttt{lazyAssign}^{\ddagger}\ (w)$
  sample $CP \leftarrow \mathsf{encase}(SK, EK, m)$ and set $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).obj = CP$, $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).\mathsf{st} = \texttt{T}$

Return $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).obj$

Fig. 29: Function $\texttt{lazyAssign}^{\ddagger}$ used by simulator $\mathcal{S}_b^{\ddagger}$ in hybrid $\mathsf{H}_{1|2}$.

Function $\texttt{checkDeltaHiding}^{\ddagger}\ (\mathbb{G}_0^{\ddagger}, \mathbb{G}_1^{\ddagger}, \widehat{h_0}, \widehat{h_1}, \overline{h})$:
$\forall b \in \{0,1\}$, let $v_b = \texttt{node}_{\mathbb{G}_b^\ddagger}(\widehat{h_b})$

- Return false if one of the following conditions hold:
  - $\texttt{Type}(v_0) \neq \texttt{Type}(v_1)$
  - $\exists b$ s.t. $v_b.\mathsf{st} = \texttt{R}$ and $v_{1-b}.\mathsf{st} \neq \texttt{R}$
  - $v_0.\mathsf{st} = \texttt{R}$, $v_1.\mathsf{st} = \texttt{R}$ and $\texttt{node}_{\mathbb{G}_0^\ddagger}(\overline{h}).\overset{\circ}{\mathbf{L}} \neq \texttt{node}_{\mathbb{G}_1^\ddagger}(\overline{h}).\overset{\circ}{\mathbf{L}}$
- Else, if $v_0.\mathsf{st} = \texttt{R}$, $v_1.\mathsf{st} = \texttt{R}$ and $\texttt{node}_{\mathbb{G}_0^\ddagger}(\overline{h}).\overset{\circ}{\mathbf{L}} = \texttt{node}_{\mathbb{G}_1^\ddagger}(\overline{h}).\overset{\circ}{\mathbf{L}}$, return true
- Else, proceed as follows depending on $\texttt{Type}(\widehat{h_0}) = \texttt{Type}(\widehat{h_1})$ (note that, in all these cases: $v_0.\mathsf{st} = \texttt{T}$, $v_1.\mathsf{st} = \texttt{T}$)

  Case $\texttt{Type}(\widehat{h_0}) = \texttt{Type}(\widehat{h_1}) = \textsc{sk}$.
  - if $\exists \overline{g}$, $\exists b \in \{0,1\}$ s.t. $\texttt{Type}(\overline{g}) = \textsc{vk}$, $v_b \rightsquigarrow \texttt{node}_{\mathbb{G}_b^\ddagger}(\overline{g})$ but $v_{1-b} \not\rightsquigarrow \texttt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{g})$, return false
  - else, if $\exists \overline{f}, \overline{g}$, $\exists b \in \{0,1\}$ s.t. $\texttt{Type}(\overline{f}) = \textsc{cp}$, $\texttt{Type}(\overline{g}) = \textsc{dk}$, $\texttt{dk-root}(\texttt{node}_{\mathbb{G}_0^\ddagger}(\overline{f})) \rightsquigarrow \texttt{node}_{\mathbb{G}_0^\ddagger}(\overline{g})$, $\texttt{dk-root}(\texttt{node}_{\mathbb{G}_1^\ddagger}(\overline{f})) \rightsquigarrow \texttt{node}_{\mathbb{G}_1^\ddagger}(\overline{g})$, $\texttt{sk-root}(\texttt{node}_{\mathbb{G}_b^\ddagger}(\overline{f})) \rightsquigarrow v_b$ but $\texttt{sk-root}(\texttt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{f})) \not\rightsquigarrow v_{1-b}$, return false
  - else, return true

  Case $\texttt{Type}(\widehat{h_0}) = \texttt{Type}(\widehat{h_1}) = \textsc{vk}$.
  - if $\exists \overline{g}$, $\exists b \in \{0,1\}$ s.t. $\texttt{Type}(\overline{g}) = \textsc{sk}$, $\texttt{root}(\texttt{node}_{\mathbb{G}_b^\ddagger}(\overline{g})) \rightsquigarrow v_b$ but $\texttt{root}(\texttt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{g})) \not\rightsquigarrow v_{1-b}$, return false
  - else, if $\exists \overline{f}, \overline{g}$, $\exists b \in \{0,1\}$ s.t. $\texttt{Type}(\overline{f}) = \textsc{cp}$, $\texttt{Type}(\overline{g}) = \textsc{dk}$, $\texttt{dk-root}(\texttt{node}_{\mathbb{G}_0^\ddagger}(\overline{f})) \rightsquigarrow \texttt{node}_{\mathbb{G}_0^\ddagger}(\overline{g})$, $\texttt{dk-root}(\texttt{node}_{\mathbb{G}_1^\ddagger}(\overline{f})) \rightsquigarrow \texttt{node}_{\mathbb{G}_1^\ddagger}(\overline{g})$, $\texttt{sk-root}(\texttt{node}_{\mathbb{G}_b^\ddagger}(\overline{f})) \rightsquigarrow v_b$ but $\texttt{sk-root}(\texttt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{f})) \not\rightsquigarrow v_{1-b}$, return false
  - else, return true

Case $\mathtt{Type}(\widehat{h}_0) = \mathtt{Type}(\widehat{h}_1) = \textsc{dk}$.

- if $\exists \overline{g}$, $\exists b \in \{0,1\}$ s.t. $\mathtt{Type}(\overline{g}) \in \{\textsc{ek}, \textsc{cp}\}$, $v_b \rightsquigarrow \mathtt{node}_{\mathbb{G}_b^\ddagger}(\overline{b})$ but $v_{1-b} \not\rightsquigarrow \mathtt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{g})$, return false
- else, if $\exists \overline{g}$, s.t. $\mathtt{Type}(\overline{g}) = \textsc{cp}$, $\forall b \in \{0,1\}$, $v_b \underset{m_b}{\rightsquigarrow} \mathtt{node}_{\mathbb{G}_b^\ddagger}(\overline{g})$ but $m_0 \neq m_1$, return false
- else, if $\exists \overline{f}, \overline{g}$, $\exists b \in \{0,1\}$ s.t. $\mathtt{Type}(\overline{f}) = \textsc{cp}$, $\mathtt{Type}(\overline{g}) \in \{\textsc{sk}, \textsc{vk}\}$, $v_0 \rightsquigarrow \mathtt{node}_{\mathbb{G}_0^\ddagger}(\overline{f})$, $v_1 \rightsquigarrow \mathtt{node}_{\mathbb{G}_1^\ddagger}(\overline{f})$, $\mathtt{sk\text{-}root}(\mathtt{node}_{\mathbb{G}_b^\ddagger}(\overline{f})) \rightsquigarrow \mathtt{node}_{\mathbb{G}_b^\ddagger}(\overline{g})$ but $\mathtt{sk\text{-}root}(\mathtt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{f})) \not\rightsquigarrow \mathtt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{g})$, return false
- else, return true

Case $\mathtt{Type}(\widehat{h}_0) = \mathtt{Type}(\widehat{h}_1) = \textsc{ek}$.

- if $\exists \overline{g}$, $\exists b \in \{0,1\}$ s.t. $\mathtt{Type}(\overline{g}) = \textsc{dk}$, $\mathtt{root}(\mathtt{node}_{\mathbb{G}_b^\ddagger}(\overline{g})) \rightsquigarrow v_b$ but $\mathtt{root}(\mathtt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{g})) \not\rightsquigarrow v_{1-b}$, return false
- else, return true

Case $\mathtt{Type}(\widehat{h}_0) = \mathtt{Type}(\widehat{h}_1) = \textsc{cp}$.

- if $\exists \overline{g}$, $\exists b \in \{0,1\}$ s.t. $\mathtt{Type}(\overline{g}) = \textsc{dk}$, $\mathtt{dk\text{-}root}(v_b) \rightsquigarrow \mathtt{node}_{\mathbb{G}_b^\ddagger}(\overline{g})$ but $\mathtt{dk\text{-}root}(v_{1-b}) \not\rightsquigarrow \mathtt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{g})$, return false
- else, if $\exists \overline{g}, m_0, m_1$ s.t. $\mathtt{Type}(\overline{g}) = \textsc{dk}$, $\mathtt{dk\text{-}root}(v_0) \rightsquigarrow \mathtt{node}_{\mathbb{G}_0^\ddagger}(\overline{g})$, $\mathtt{dk\text{-}root}(v_1) \rightsquigarrow \mathtt{node}_{\mathbb{G}_1^\ddagger}(\overline{g})$
  - ∗ if $\mathtt{dk\text{-}root}(v_0) \underset{m_0}{\rightsquigarrow} v_0$ and $\mathtt{dk\text{-}root}(v_1) \underset{m_1}{\rightsquigarrow} v_1$, s.t. $m_0 \neq m_1$, return false
  - ∗ else, if $\exists \overline{f}$, $\exists b \in \{0,1\}$ s.t. $\mathtt{Type}(\overline{f}) \in \{\textsc{sk}, \textsc{vk}\}$, $\mathtt{sk\text{-}root}(v_b) \rightsquigarrow \mathtt{node}_{\mathbb{G}_b^\ddagger}(\overline{f})$ but $\mathtt{sk\text{-}root}(v_{1-b}) \not\rightsquigarrow \mathtt{node}_{\mathbb{G}_{1-b}^\ddagger}(\overline{f})$
- else, return true

Fig. 30: Function $\mathtt{checkDeltaHiding}^\ddagger$ used by simulator $\mathcal{S}_b^\ddagger$ in hybrid $\mathsf{H}_{1|2}$.

Indistinguishability between H_1 and $\mathsf{H}_{1|2}$ (and similarly between H_6 and $\mathsf{H}_{5|6}$). Conditioned on the function $\mathtt{checkDeltaHiding}^\ddagger$ not returning false in $\mathsf{H}_{1|2}$, the two hybrids are trivially indistinguishable. Later, we show that $\mathtt{checkDeltaHiding}^\ddagger$ returns false with negligible probability if Test is hiding (Lemma 14).

Lemma 11. *Conditioned on* $\mathtt{checkDeltaHiding}^\ddagger$ *not returning* false *in* $\mathsf{H}_{1|2}$*, the two hybrids* H_1 *and* $\mathsf{H}_{1|2}$ *are indistinguishable.*

Proof: Note that the bad-events corresponding to object collisions are already negligible probability events. The only difference in the behaviour of the simulators in the two hybrids is that $\mathcal{S}_0^\ddagger$ samples objects in a lazy manner for handles transferred by Test, but it is easy to see that this does not affect the transcript of interaction of $\mathcal{S}_0^\ddagger$ with $\mathcal{B}[\Sigma^\ddagger_{\Pi_{\mathsf{case}}}]$ and $\mathcal{A}$. Inductively, at the end of each transfer of the form $(\mathsf{transfer}, \widehat{h}_0, \widehat{h}_1)$ from Test to $\mathcal{A}$, it holds that the distribution from which the object for $\overline{h}$ is sampled are the same, and thus the view of $\mathcal{A}$ are the same. □

C.3.4 Hybrid H_2 and H_5 In this hybrid, we run the experiment $\textsc{ideal}\langle \mathsf{Test}(b) \mid \Sigma^\ddagger_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^\ddagger \circ \mathcal{A}\rangle$ with test bit $b = 0$ for H_2 and $b = 1$ for H_5, where $\mathcal{S}^\ddagger$ is as described in Figure 31. We list the main differences between $\mathcal{S}_b^\ddagger$ (in $\mathsf{H}_{1|2}$ and $\mathsf{H}_{5|6}$) and $\mathcal{S}^\ddagger$:

1. **Challenge bit b:** Both simulators maintain graphs $\mathbb{G}_0^\ddagger$, $\mathbb{G}_1^\ddagger$; but $\mathcal{S}_b^\ddagger$ (that gets the challenge bit b) only uses $\mathbb{G}_b^\ddagger$ to send object to $\mathcal{A}$. That is, it sends $\mathtt{node}_{\mathbb{G}_b^\ddagger}(\overline{h}).obj$ to $\mathcal{A}$ corresponding to some user handle $\overline{h}$ transferred by $\mathcal{B}[\Sigma^\ddagger_{\Pi_{\mathsf{case}}}]$ from Test. $\mathcal{S}^\ddagger$ on the other hand does not get the challenge bit b and instead uses both graphs. Recall that, updates to $\mathbb{G}_0^\ddagger$ and $\mathbb{G}_1^\ddagger$ are made using the same randomness. $\mathcal{S}^\ddagger$ sends an object to $\mathcal{A}$ only if both graphs can be made consistent with this object. Please refer to Figure 32 for the full description.

2. **Resolve Object:** $\mathcal{S}^{\ddagger}$ uses a resolve object function (please refer Figure 32) if the object in $\mathbb{G}_0^{\ddagger}$ and $\mathbb{G}_1^{\ddagger}$ are not consistent for some transfer from Test. At a high level, $\mathcal{S}^{\ddagger}$ simply samples a fresh object (consistent with all previous transfers) and sends it to $\mathcal{A}$. We show below that if $\texttt{checkDeltaHiding}^{\ddagger}$ returns true, then this is a valid simulation (via a reduction to the augmented security experiment aug of the COA-secure scheme). We show later that $\texttt{checkDeltaHiding}^{\ddagger}$ returns false with negligible probability if Test is hiding (Lemma 14).

```
Simulator $\mathcal{S}^{\ddagger}$:

It maintains graphs $\mathbb{G}_0^{\ddagger}$, $\mathbb{G}_1^{\ddagger}$.

  – Processing objects transferred by $\mathcal{A}$:
      Let the object from $\mathcal{A}$ be $obj$ and the handle to be received by Test be $\widehat{h}$
    • sample $r \leftarrow \{0,1\}^{\kappa}$
    • $\forall b' \in \{0,1\}$, $\overline{h}_{b'} \leftarrow \texttt{update}^{\dagger}_{\mathcal{A}}(\mathbb{G}_{b'}^{\ddagger}, obj;\ r)$ using randomness $r$
    • set $\texttt{node}_{\mathbb{G}_0^{\ddagger}}(\widehat{h}).\texttt{st} = \texttt{R}$, $\texttt{node}_{\mathbb{G}_1^{\ddagger}}(\widehat{h}).\texttt{st} = \texttt{R}$
    • send $\overline{h}_b$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$

  – Processing commands by Test:
    Let the report from Test be $\texttt{report}$ and handle received from $\mathcal{B}[\Sigma]$ be $\overline{h}$
    • sample $r \leftarrow \{0,1\}^{\kappa}$
    • $\forall b' \in \{0,1\}$, run $\texttt{update}^{\ddagger}_{\mathsf{Test}}(\mathbb{G}_{b'}^{\ddagger}, b', \texttt{report}, \overline{h};\ r)$ using randomness $r$
    • if $\texttt{report} = (\mathsf{transfer}, \widehat{h}_0, \widehat{h}_1)$:
      ∗ if $\texttt{checkDeltaHiding}^{\ddagger}12(\mathbb{G}_0^{\ddagger}, \mathbb{G}_1^{\ddagger}, \widehat{h}_0, \widehat{h}_1, \overline{h}) = \perp$, abort
      ∗ set $\texttt{node}_{\mathbb{G}_0^{\ddagger}}(\widehat{h}_0).\texttt{st} = \texttt{R}$, $\texttt{node}_{\mathbb{G}_1^{\ddagger}}(\widehat{h}_1).\texttt{st} = \texttt{R}$
      ∗ if $\texttt{node}_{\mathbb{G}_0^{\ddagger}}(\overline{h}).obj = \texttt{node}_{\mathbb{G}_1^{\ddagger}}(\overline{h}).obj = obj$, return $obj$ to $\mathcal{A}$
      ∗ else,
        · $obj = \texttt{resolveObject}^{\ddagger}\ (\mathbb{G}_0^{\ddagger}, \mathbb{G}_1^{\ddagger}, \overline{h})$
        · set $\texttt{node}_{\mathbb{G}_0^{\ddagger}}(\overline{h}).obj = obj$, $\texttt{node}_{\mathbb{G}_1^{\ddagger}}(\overline{h}).obj = obj$
        · return $obj$ to $\mathcal{A}$
```

Fig. 31: Simulator $\mathcal{S}^{\ddagger}$ in hybrid H_2.

```
Function $\texttt{resolveObject}^{\ddagger}\ (\mathbb{G}_0^{\ddagger}, \mathbb{G}_1^{\ddagger}, \overline{h})$ :

  – if $\texttt{Type}(\overline{h}) = \textsc{sk}$: sample $SK \leftarrow \mathsf{skGen}(1^{\kappa})$ and return $SK$
  – if $\texttt{Type}(\overline{h}) = \textsc{vk}$: sample $SK \leftarrow \mathsf{skGen}(1^{\kappa})$, $VK \leftarrow \mathsf{vkGen}(DK)$ and return $VK$
  – if $\texttt{Type}(\overline{h}) = \textsc{ek}$: sample $DK \leftarrow \mathsf{dkGen}(1^{\kappa})$, $EK \leftarrow \mathsf{ekGen}(DK)$ and return $EK$
```

```
- if $\mathtt{Type}(\overline{h}) = \textsc{cp}$:
  • if $\mathtt{sk\text{-}root}(\mathtt{node}_{\mathbb{G}_0^\ddagger}(\overline{h})).obj = \mathtt{sk\text{-}root}(\mathtt{node}_{\mathbb{G}_1^\ddagger}(\overline{h})).obj$, set $SK = \mathtt{sk\text{-}root}(\mathtt{node}_{\mathbb{G}_0^\ddagger}(\overline{h})).obj$
    else, $SK \leftarrow \mathsf{skGen}(1^\kappa)$
  • $\forall b \in \{0,1\}$, let $w_b \in \mathbb{G}_b^\ddagger.\mathbb{V}_{\mathsf{Test}}$ s.t. $\exists m_b$, $\exists x_b \in \mathbb{G}_b^\ddagger.\mathbb{V}_{\mathsf{Test}}$ and $w_b \xrightarrow{(pk-ct,\mathsf{encase},m_b)} x_b \rightarrow \mathtt{node}_{\mathbb{G}_b^\ddagger}(\overline{h})$
      * if $w_0.obj = w_1.obj$, set $EK = w_0.obj$
        else, $DK \leftarrow \mathsf{dkGen}(1^\kappa)$, $EK \leftarrow \mathsf{ekGen}(DK)$
      * if $m_0 = m_1$, set $m = m_0$
        else, $m = 0$
  return $\mathsf{encase}(SK, EK, m)$
```

Fig. 32: Function $\mathtt{resolveObject}^\ddagger$ used by simulator $\mathcal{S}^\ddagger$ in hybrid H_2.

Indistinguishability between $\mathsf{H}_{1|2}$ and H_2 (and similarly between $\mathsf{H}_{5|6}$ and H_5). We prove this via a reduction to the augmented security experiment aug (please refer to Section 4.1) of the CASE primitive.

Lemma 12. *The hybrids $\mathsf{H}_{1|2}$ and H_2 are indistinguishable.*

Proof: We first note that, if $\mathtt{checkDeltaHiding}^\ddagger$ returns false, both hybrids abort. We now argue for the case that $\mathtt{checkDeltaHiding}^\ddagger$ does not return false. Let Test and $\mathcal{A}$ be s.t. they have advantage α, that is:

$$\left| \Pr[\textsc{ideal}\langle \mathsf{Test}(0) \mid \Sigma^\ddagger_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^\ddagger_b \circ \mathcal{A}\rangle = b] - \Pr[\textsc{ideal}\langle \mathsf{Test}(0) \mid \Sigma^\ddagger_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^\ddagger \circ \mathcal{A}\rangle = b] \right| \geq \frac{1}{2} + \alpha$$

We define a sequence of intermediate hybrids ${\mathsf{H}_j}^*$ corresponding to the experiment aug, where in each hybrid, the adversary $\mathcal{A}^*_j$ internally runs Test, $\mathcal{A}$, $\mathcal{B}[\Sigma^\ddagger_{\Pi_{\mathsf{case}}}]$, feeds them inputs appropriately similar to $\mathcal{S}^\ddagger$ and uses the j^{th} transfer command from Test to construct the case-packet-challenge to be sent to the experiment. Similar to the simulators, $\mathcal{A}^*_j$ also maintains the graphs $\mathbb{G}_0^\ddagger$, $\mathbb{G}_1^\ddagger$, except that instead of sampling objects, it instead indexes them to objects in the experiment. That is, for any node $v \in \mathbb{G}^\ddagger.\mathbb{V}_{\mathsf{Test}}$, it parses $v.obj$ as an index to $T_{\mathsf{Type}(v)}[v.obj]$. Correspondingly, it uses modified functions $\mathtt{lazyAssign}^\ddagger_{\mathcal{A}^*}$ (Figure 34) and $\mathtt{resolveObject}^\ddagger_{\mathcal{A}^*}$ (Figure 35) that simply assign an index.

Note that, if the experiment aborts, then $\mathcal{A}^*$ also aborts (since, $\mathtt{checkDeltaHiding}^\ddagger$ returns false). Thus, the advantage of $\mathcal{A}^*$ in the experiment aug is also α. But, from the COA-security of the primitive, it must be negligible. Thus, $\forall j$, it holds that ${\mathsf{H}_j}^* \approx {\mathsf{H}_{j+1}}^*$ and in particular, $\mathsf{H}_{1|2} \approx \mathsf{H}_2$. □

Adversary $\mathcal{A}^*_j$:

```
- it sends $n = |\mathsf{Test} + \mathcal{A}|$ (bound on the runtime of $\mathsf{Test}$ and $\mathcal{A}$) to the experiment
- it internally runs $\mathsf{Test}$, $\mathcal{A}$ and $\mathcal{B}[\Sigma^\ddagger_{\Pi_{\mathsf{case}}}]$ in a straightline black-box way and maintains graphs $\mathbb{G}_0^\ddagger$, $\mathbb{G}_1^\ddagger$
```

```
**Processing objects transferred by $\mathcal{A}$:**
Let the object from $\mathcal{A}$ be $obj$ and the handle to be received by Test be $\widehat{h}$
• sample $r \leftarrow \{0,1\}^\kappa$
• $\forall b' \in \{0,1\}$, $\overline{h}_{b'} \leftarrow \texttt{update}^\dagger_{\mathcal{A}}(\mathbb{G}^\ddagger_{b'}, obj;\ r)$ using randomness $r$
• let $i = \min\big(\texttt{node}_{\mathbb{G}^\ddagger_0}(\overline{h}).\mathbf{\overline{L}}\big)$, send $(n+i, obj)$ to experiment
• set $\texttt{node}_{\mathbb{G}^\ddagger_0}(\overline{h}).obj = n+i$, $\texttt{node}_{\mathbb{G}^\ddagger_1}(\overline{h}).obj = n+i$
• set $\texttt{node}_{\mathbb{G}^\ddagger_0}(\widehat{h}).\mathsf{st} = \mathtt{R}$, $\texttt{node}_{\mathbb{G}^\ddagger_1}(\widehat{h}).\mathsf{st} = \mathtt{R}$
• send $\overline{h}_b$ to $\mathcal{B}\big[\Sigma^\ddagger_{\Pi_{\mathsf{case}}}\big]$ and $\widehat{h}$ to Test

**Processing commands by Test:**
Let the $i^{th}$ report from Test be $\texttt{report}$ and handle received from $\mathcal{B}\big[\Sigma\big]$ be $\overline{h}$
• sample $r \leftarrow \{0,1\}^\kappa$
• $\forall b' \in \{0,1\}$, run $\texttt{update}^\ddagger_{\mathsf{Test}}(\mathbb{G}^\ddagger_{b'}, b', \texttt{report}, \overline{h};\ r)$ using randomness $r$ and modified function $\texttt{lazyAssign}^\ddagger_{\mathcal{A}^*}$
• if $\texttt{report} = (\mathsf{transfer}, \widehat{h}_0, \widehat{h}_1)$:
    * if $\texttt{checkDeltaHiding}^\ddagger 12(\mathbb{G}^\ddagger_0, \mathbb{G}^\ddagger_1, \widehat{h}_0, \widehat{h}_1, \overline{h}) = \perp$, abort
    * set $\texttt{node}_{\mathbb{G}^\ddagger_0}(\widehat{h}_0).\mathsf{st} = \mathtt{R}$, $\texttt{node}_{\mathbb{G}^\ddagger_1}(\widehat{h}_1).\mathsf{st} = \mathtt{R}$
    * if $\texttt{node}_{\mathbb{G}^\ddagger_0}(\overline{h}).obj = \texttt{node}_{\mathbb{G}^\ddagger_1}(\overline{h}).obj$ or $i < j$, send key-query $\big(\texttt{Type}(\overline{h}), \texttt{node}_{\mathbb{G}^\ddagger_0}(\overline{h}).obj\big)$ to experiment and forward its response to $\mathcal{A}$
    * else, if $i = j$:
      if $\texttt{Type}(\overline{h}) \in \{\textsc{sk}, \textsc{vk}, \textsc{dk}, \textsc{ek}\}$, send key-challenge $\big(\texttt{Type}(\widehat{h}_0), \texttt{node}_{\mathbb{G}^\ddagger_0}(\widehat{h}_0).obj, \texttt{node}_{\mathbb{G}^\ddagger_1}(\widehat{h}_1).obj\big)$ to experiment and forward its response to $\mathcal{A}$
      else, if $\texttt{Type}(\overline{h}) = \textsc{cp}$,
      · let $v, w \in \mathbb{G}^\ddagger_0.\mathbb{V}_{\mathsf{Test}}$ s.t. $\texttt{Type}(v) = \textsc{sk}$, $\texttt{Type}(w) = \textsc{ek}$, $v \underset{m_0}{\rightsquigarrow} \texttt{node}_{\mathbb{G}^\ddagger_0}(\widehat{h}_0)$ and $w \underset{m_0}{\rightsquigarrow} \texttt{node}_{\mathbb{G}^\ddagger_0}(\widehat{h}_0)$
      · $(k, l, m) = \texttt{resolveObject}^\ddagger_{\mathcal{A}^*}\ (\mathbb{G}^\ddagger_0, \mathbb{G}^\ddagger_1, \overline{h}, n)$
      · send case-packet-challenge $(v.obj, w.obj, m_0, k, l, m)$ to experiment, get response $CP$, send $(\overline{h}, CP)$ to the experiment (add object to index $\overline{h}$)
      · set $\texttt{node}_{\mathbb{G}^\ddagger_0}(\overline{h}).obj = \overline{h}$, $\texttt{node}_{\mathbb{G}^\ddagger_1}(\overline{h}).obj = \overline{h}$
    * else, $i > j$:
      if $\texttt{Type}(\overline{h}) = \textsc{cp}$,
      · $(k, l, m) = \texttt{resolveObject}^\ddagger_{\mathcal{A}^*}\ (\mathbb{G}^\ddagger_0, \mathbb{G}^\ddagger_1, \overline{h}, n)$
      · send encryption-query $(k, l, m)$ to experiment, get response $CP$, send $(2n + \overline{h}, CP)$ to the experiment (add object to index $\overline{h}$)
      set $\texttt{node}_{\mathbb{G}^\ddagger_0}(\overline{h}).obj = 2n + \overline{h}$, $\texttt{node}_{\mathbb{G}^\ddagger_1}(\overline{h}).obj = 2n + \overline{h}$
      send key-query $\big(\texttt{Type}(\overline{h}), 2n + \overline{h}\big)$ to experiment and forward its response to $\mathcal{A}$
```

Fig. 33: Adversary $\mathcal{A}_j^*$ in hybrid $\mathsf{H}_j{}^*$ interacting with experiment aug.

```
**Function $\texttt{lazyAssign}^\ddagger_{\mathcal{A}^*}\ (\mathbb{G}^\ddagger, \widehat{h})$ :**
If $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).obj \neq \perp$, return $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).obj$. Else, proceed as follows:

  – if $\texttt{Type}(\widehat{h}) \in \{\textsc{sk}, \textsc{vk}, \textsc{dk}, \textsc{ek}\}$:
    set $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).obj = \min\big(\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).\mathbf{\overline{L}}\big)$ and $\texttt{node}_{\mathbb{G}^\ddagger}(\widehat{h}).\mathsf{st} = \mathtt{T}$
```

```
– if $\mathtt{Type}(\widehat{h}) = \textsc{cp}$:
    • let $v, w \in \mathbb{G}^{\ddagger}.\mathbb{V}_{\mathsf{Test}}$ s.t. $\mathtt{Type}(v) = \textsc{sk}$, $\mathtt{Type}(w) = \textsc{ek}$, $v \underset{m}{\rightsquigarrow} \mathtt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h})$ and $w \underset{m}{\rightsquigarrow} \mathtt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h})$
    • $k = \mathtt{lazyAssign}^{\ddagger}_{\mathcal{A}^*}(v)$, $l = \mathtt{lazyAssign}^{\ddagger}_{\mathcal{A}^*}(w)$
    • send encryption-query $(k, l, m)$ to the experiment and get response $CP$, send $(\widehat{h}, CP)$ to the experiment
      (add object to index $\widehat{h}$)
    • set $\mathtt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h}).obj = \widehat{h}$ and $\mathtt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h}).\mathsf{st} = \mathtt{T}$

Return $\mathtt{node}_{\mathbb{G}^{\ddagger}}(\widehat{h}).obj$
```

Fig. 34: Function $\mathtt{lazyAssign}^{\ddagger}_{\mathcal{A}^*}$ used by $\mathcal{A}^*_j$ in hybrid $\mathsf{H}_j{}^*$.

```
Function $\mathtt{resolveObject}^{\ddagger}_{\mathcal{A}^*}(\mathbb{G}^{\ddagger}_0, \mathbb{G}^{\ddagger}_1, \overline{h}, n)$ :

  – if $\mathtt{Type}(\overline{h}) = \textsc{cp}$:
    • if $\mathtt{sk\text{-}root}(\mathtt{node}_{\mathbb{G}^{\ddagger}_0}(\overline{h})).obj = \mathtt{sk\text{-}root}(\mathtt{node}_{\mathbb{G}^{\ddagger}_1}(\overline{h})).obj$, set $k = \mathtt{sk\text{-}root}(\mathtt{node}_{\mathbb{G}^{\ddagger}_0}(\overline{h})).obj$; else, $k = 2n + \overline{h}$
    • $\forall b \in \{0,1\}$, let $w_b \in \mathbb{G}^{\ddagger}_b.\mathbb{V}_{\mathsf{Test}}$ s.t. $\mathtt{Type}(w_b) = \textsc{ek}$, $w_b \underset{m_b}{\rightsquigarrow} \mathtt{node}_{\mathbb{G}^{\ddagger}_b}(\overline{h})$
      ∗ if $w_0.obj = w_1.obj$, set $l = w_0.obj$; else, $l = 2n + \overline{h}$
      ∗ if $m_0 = m_1$, set $m = m_0$; else, $m = 0$
    return $(k, l, m)$
```

Fig. 35: Function $\mathtt{resolveObject}^{\ddagger}_{\mathcal{A}^*}$ used by $\mathcal{A}^*_j$ in hybrid $\mathsf{H}_j{}^*$.

C.3.5 $\mathsf{H_2}$ and $\mathsf{H_3}$

H_3 uses a computationally unbounded simulator $\mathcal{S}^*$ to remove the need for $\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}$. It replaces non-ideal handles generated by $\mathcal{S}^{\ddagger}$ by forcing open objects using the (inefficient) algorithms - skId, dkId, vkId, ekId, msgId- guaranteed by existential consistency. Existential consistency and the construction of $\mathcal{S}^{\ddagger}$ ensures that The system $\mathcal{B}[\Sigma_{\mathsf{case}}] \circ \mathcal{S}^*$ and $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ behave identically for all $\mathcal{A}$. It assigns objects corresponding to non-ideal handles with ideal handles and the algorithms mentioned above are used to ensure that the relations betweens handles are maintained.

$\mathcal{S}^*$ (as a wrapper over a $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$-adversary $\mathcal{S}^{\ddagger}$)

$\mathcal{S}^*$ interacts with $\mathcal{B}[\Sigma_{\mathsf{case}}]$, while simulating to $\mathcal{S}^{\ddagger}$ the interface to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$, using super-polynomial computational power. It maintains two tables, Z_1 to map handles received from $\mathcal{B}[\Sigma_{\mathsf{case}}]$ (denoted as $\widetilde{h}$ etc.) to objects and Z_2 to map them to handles that it sends to $\mathcal{S}^{\ddagger}$ (denoted as $\overline{h}$ etc.). Some subroutines are used by $\mathcal{S}^*$ to interact with $\mathcal{B}[\Sigma_{\mathsf{case}}]$, and to read and update Z_1 which are also defined below.

Commands from $\mathcal{S}^{\ddagger}$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$: $\mathcal{S}^*$ processes commands according to the following cases.

– When $\mathcal{S}^{\ddagger}$ sends a command $(\mathsf{init}, (\mathtt{CPgen}, obj))$ (i.e., an init command for a case-packet agent in $\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}$) to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$, let $\widetilde{h} \leftarrow \mathsf{makeCT}(obj)$. Add $(\widetilde{h}, \overline{h})$ to Z_2 where $\overline{h}$ denotes the next handle to be returned by $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ (being simulated). Send $\overline{h}$ to $\mathcal{S}^{\ddagger}$.

- When $\mathcal{S}^{\ddagger}$ sends a command $(\mathsf{run}, (\overline{ek}, (\mathtt{CPgen}, (obj, DK))))$ to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$, let $m = \mathsf{decase\text{-}msg}(DK, obj)$.

  If $m \neq \perp \wedge \mathsf{tryAssign}(\overline{ek}, \mathsf{ekGen}(DK))$, let $\widetilde{h} \leftarrow \mathsf{makeCT}(obj)$. Add $(\widetilde{h}, \overline{h})$ to Z_2 where $\overline{h}$ denotes the next handle to be returned by $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ (being simulated). Send $\overline{h}$ to $\mathcal{S}^{\ddagger}$.
  Else, abort execution.

- When $\mathcal{S}^{\ddagger}$ sends a command $(\mathsf{init}, (\mathtt{key\text{-}type}, \kappa))$, send $\overline{h}$ to $\mathcal{S}^{\ddagger}$ where $\overline{h}$ denotes the next handle to be returned by the simulated $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$.
  - If $\mathtt{key\text{-}type} = \textsc{dk}$ in the init command and the next command sent by $\mathcal{S}^{\ddagger}$ is $(\mathsf{run}, (\overline{h}, (\mathtt{dkPatch}, obj)), \{(\overline{sk}_i, (\mathtt{dkPatch}, CP_i))\}_i)$, let $t = \mathsf{checkAssigned}(obj) \vee \mathsf{checkAssigned}(\mathsf{ekGen}(obj))$. Let $a = \wedge\{\mathsf{tryAssign}(\mathsf{skId}(\mathsf{ekId}(CP_i)), \overline{sk_i})\}_i$. If $t = \mathsf{true} \vee a = \mathsf{false}$, abort execution. Else, let $\widetilde{h} \leftarrow \mathsf{makeDK}(obj)$. Let $\overline{h}_1$ be the next handle to be returned by the simulated $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$. Add $(\widetilde{h}, \overline{h})$ and $(\widetilde{h}, \overline{h}_1)$ to Z_2 and send $\overline{h}_1$ to $\mathcal{S}^{\ddagger}$.
  - Else, if $\mathtt{key\text{-}type} = \textsc{sk}$ in the init command and the next command sent by $\mathcal{S}^{\ddagger}$ is $(\mathsf{run}, (\overline{h}, (\mathtt{patch}, obj)))$, let $t = \mathsf{checkAssigned}(obj) \vee \mathsf{checkAssigned}(\mathsf{vkGen}(obj))$. If $t = \mathsf{true}$, abort execution. Else, let $\widetilde{h} \leftarrow \mathsf{makeSK}(obj)$. Let $\overline{h}_1$ be the next handle to be returned by $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$. Add $(\widetilde{h}, \overline{h})$ and $(\widetilde{h}, \overline{h}_1)$ to Z_2 and send $\overline{h}_1$ to $\mathcal{S}^{\ddagger}$.
  - Else if, $\mathtt{key\text{-}type} = \textsc{dk}$ in the init command and the next command sent by $\mathcal{S}^{\ddagger}$ is $(\mathsf{run}, (\overline{h}, \mathtt{ekGen}))$, send the next handle $\overline{h}_1$ to $\mathcal{S}^{\ddagger}$.
    - If the next command is $(\mathsf{run}, (\overline{h}_1, (\mathtt{patch}, obj)))$, let $t = \mathsf{checkAssigned}(obj) \vee \mathsf{checkAssigned}(\mathsf{dkId}(obj))$. If $t = \mathsf{true}$, abort execution. Else, let $\widetilde{dk} \leftarrow \mathsf{makeDK}(\mathsf{dkId}(obj))$, $\widetilde{ek} \leftarrow \mathsf{makeEK}(obj)$. Let $\overline{h}_2$ be the next handle to be returned by $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$. Add $(\widetilde{dk}, \overline{h})$, $(\widetilde{ek}, \overline{h}_1)$ and $(\widetilde{ek}, \overline{h}_2)$ to Z_2 and send $\overline{h}_2$ to $\mathcal{S}^{\ddagger}$.
    - Else, abort.
  - Else if, $\mathtt{key\text{-}type} = \textsc{sk}$ in the init command and the next command sent by $\mathcal{S}^{\ddagger}$ is $(\mathsf{run}, (\overline{h}, \mathtt{vkGen}))$, send the next handle $\overline{h}_1$ to $\mathcal{S}^{\ddagger}$.
    - If the next command is $(\mathsf{run}, (\overline{h}_1, (\mathtt{patch}, obj)))$, let $t = \mathsf{checkAssigned}(obj) \vee \mathsf{checkAssigned}(\mathsf{skId}(obj))$. If $t = \mathsf{true}$, abort execution. Else, let $\widetilde{sk} \leftarrow \mathsf{makeSK}(\mathsf{skId}(obj))$, $\widetilde{vk} \leftarrow \mathsf{makeVK}(obj)$. Let $\overline{h}_2$ be the next handle to be returned by $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$. Add $(\widetilde{sk}, \overline{h})$, $(\widetilde{vk}, \overline{h}_1)$ and $(\widetilde{vk}, \overline{h}_2)$ to Z_2 and send $\overline{h}_2$ to $\mathcal{S}^{\ddagger}$.
    - Else, abort.
  - Else, abort execution.

- When $\mathcal{S}^{\ddagger}$ sends a command $(\mathsf{run}, (\overline{dk}, (\mathtt{dkPatch}, obj)), \{(\overline{sk}_i, (\mathtt{dkPatch}, CP_i))\}_i)$ (resp. $(\mathsf{run}, (\overline{sk}, (\mathtt{patch}, obj))))$) such that $\overline{dk}$ (resp. $\overline{sk}$) is not the handle transferred to $\mathcal{S}^{\ddagger}$ in the previous command, check if $\circ \overset{\mathsf{init}}{\to} \overline{dk}$ (resp. $\circ \overset{\mathsf{init}}{\to} \overline{sk}$) and if $\exists \widetilde{h}\ s.t.\ (\widetilde{h}, \overline{dk})\ (resp.\ \overline{sk}) \in Z_2 \wedge (\widetilde{h}, obj) \in Z_1$. When command is $(\mathsf{run}, (\overline{dk}, (\mathtt{dkPatch}, obj)), \{(\overline{sk}_i, (\mathtt{dkPatch}, CP_i))\}_i)$, we also check if $\mathsf{decase\text{-}msg}(obj, CP_i) \neq \perp$ and $\mathsf{tryAssign}(\mathsf{skId}(\mathsf{vkId}(CP_i)), \overline{sk}_i)$ holds $\forall\ i$.

  If all checks return true, obtain $\widetilde{h} \leftarrow \mathsf{makeDK}(obj)$ (resp. $\mathsf{makeSK}(obj)$). Let $\overline{h}$ be the next handle to be returned by $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$. Add $(\widetilde{h}, \overline{h})$ to Z_2 and send $\overline{h}$ to $\mathcal{S}^{\ddagger}$.
  Else, abort execution.

- When $\mathcal{S}^{\ddagger}$ sends any other run or $\mathsf{transfer}$ command to $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$, $\mathcal{S}^*$ simply relays the command to $\mathcal{B}[\Sigma_{\mathsf{case}}]$, but substitutes each handle $\overline{h}$ in the command with $\widetilde{h}$ using the Z_2 map. The response from $\mathcal{B}[\Sigma_{\mathsf{case}}]$ is relayed back to $\mathcal{S}^{\ddagger}$, but after replacing each new handle $\widetilde{h}$ in the response with a new handle $\overline{h}$ (i.e., the next handle to be returned by $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$), and adding an entry $(\widetilde{h}, \overline{h})$ to Z_2. (If a handle in the response is $\perp$, indicating that the agent halted, it isn't replaced with a new handle, but is kept as $\perp$.)

Transfers from Test**:** When $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ delivers a handle $\widetilde{h}$ corresponding to a transfer from Test, $\mathcal{S}^*$ will deliver send a new handle $\overline{h}$ to give to $\mathcal{S}^{\ddagger}$ and makes an entry $(\widetilde{h}, \overline{h})$ in Z_2.

Subroutine $\mathsf{makeSK}(obj)$
Precondition: $\mathsf{acc}(obj) = \textsc{sk}$, or $obj = \bot$
If $\exists \widetilde{sk}$ s.t. $(\widetilde{sk}, obj) \in Z_1$, then return $\widetilde{sk}$; else, send $(\mathsf{init}, (\textsc{sk}, \kappa))$ to $\mathcal{B}[\Sigma_{\mathsf{case}}]$. Let $\widetilde{sk}_1$ be the handle received in return. If $obj \neq \bot$, add $(\widetilde{sk}_1, obj)$ to Z_1. Return $\widetilde{sk}_1$.

Subroutine $\mathsf{makeVK}(obj)$
Precondition: $\mathsf{acc}(obj) = \textsc{vk}$ or $obj = \bot$
If $\exists \widetilde{vk}$ s.t. $(\widetilde{vk}, obj) \in Z_1$, then return $\widetilde{vk}$. Else, let $SK := \mathsf{skId}(obj)$ and $\widetilde{sk} := \mathsf{makeSK}(SK)$, and send $(\mathsf{run}, (\widetilde{sk}, \mathtt{vkGen}))$ to $\mathcal{B}[\Sigma_{\mathsf{case}}]$. Let $\widetilde{vk}$ be the handle received in return. If $obj \neq \bot$, add $(\widetilde{vk}, obj)$ to Z_1. Return $\widetilde{vk}$.

Subroutine $\mathsf{makeDK}(obj)$
Precondition: $\mathsf{acc}(obj) = \textsc{dk}$, or $obj = \bot$
If $\exists \widetilde{dk}$ s.t. $(\widetilde{dk}, obj) \in Z_1$, then return $\widetilde{dk}$; else, send $(\mathsf{init}, (\textsc{dk}, \kappa))$ to $\mathcal{B}[\Sigma_{\mathsf{case}}]$. Let $\widetilde{dk}_1$ be the handle received in return. If $obj \neq \bot$, add $(\widetilde{dk}_1, obj)$ to Z_1. Return $\widetilde{dk}_1$.

Subroutine $\mathsf{makeEK}(obj)$
Precondition: $\mathsf{acc}(obj) = \textsc{ek}$ or $obj = \bot$
If $\exists \widetilde{ek}$ s.t. $(\widetilde{ek}, obj) \in Z_1$, then return $\widetilde{ek}$. Else, let $DK := \mathsf{ekId}(obj)$ and $\widetilde{dk} := \mathsf{makeDK}(DK)$, and send $(\mathsf{run}, (\widetilde{dk}, \mathtt{ekGen}))$ to $\mathcal{B}[\Sigma_{\mathsf{case}}]$. Let $\widetilde{ek}$ be the handle received in return. If $obj \neq \bot$, add $(\widetilde{ek}, obj)$ to Z_1. Return $\widetilde{ek}$.

Subroutine $\mathsf{makeCT}(obj)$
Precondition: $\mathsf{acc}(obj) = \textsc{cp}$
If $\exists \widetilde{cp}$ s.t. $(\widetilde{cp}, obj) \in Z_1$, then return $\widetilde{cp}$. Else, let $m = \mathsf{msgId}(obj)$, $EK := \mathsf{ekId}(obj)$ and $SK := \mathsf{skId}(\mathsf{vkId}(obj))$. Get $\widetilde{ek} := \mathsf{makeEK}(EK)$ and $\widetilde{sk} := \mathsf{makeSK}(SK)$, and send $(\mathsf{run}, (\widetilde{sk}, (\mathtt{encase}, m)), (\widetilde{ek}, (\mathtt{encase}, m)))$ to $\mathcal{B}[\Sigma_{\mathsf{case}}]$. Let $\widetilde{cp}$ be the handle received in return. Add $(\widetilde{cp}, obj)$ to Z_1. Return $\widetilde{cp}$.

Subroutine $\mathsf{doCompare}(\widetilde{h}_1, \widetilde{h}_2)$
Send $(\mathsf{run}, (\widetilde{h}_1, \mathtt{compare}), (\widetilde{h}_2, \mathtt{compare}))$ to $\mathcal{B}[\Sigma_{\mathsf{case}}]$ and return the boolean output received.

Subroutine $\mathsf{checkAssigned}(obj)$
Return true if $\exists \overline{h}, \widetilde{h}\ s.t.\ (\widetilde{h}, \overline{h}) \in Z_2 \wedge (\widetilde{h}, obj) \in Z_1$. Else, return false.

Subroutine $\mathsf{tryAssign}(obj, \overline{h})$
Let $\widetilde{h}$ be the handle such that $(\widetilde{h}, \overline{h}) \in Z_2$.

– If $\mathtt{Type}\ (\overline{h}) = \textsc{dk}$ and $\mathsf{acc}\ (obj) = \textsc{dk}$
- If $\exists \widetilde{dk}$ s.t. $(\widetilde{dk}, obj) \in Z_1$, then return true if $\mathsf{doCompare}(\widetilde{dk}, \widetilde{h})$, else return false.
- Else if $\exists \widetilde{ek}\ s.t.\ (\widetilde{ek}, \mathsf{ekGen}(obj)) \in Z_1$, return true if $\mathsf{doCompare}(\widetilde{h}'', \widetilde{ek})$ where $\widetilde{h}'' \leftarrow (\mathsf{run}, (\widetilde{h}, \mathtt{ekGen}))$, else return false.
- Else if $\nexists \widetilde{dk}, obj'\ s.t.\ (\widetilde{dk}, obj') \in Z_1 \wedge \mathsf{doCompare}(\widetilde{dk}, \widetilde{h})$ and $\nexists \widetilde{ek}, obj'\ s.t.\ (\widetilde{ek}, obj') \in Z_1 \wedge \mathsf{doCompare}(\widetilde{ek}, \widetilde{h}'')$ where $\widetilde{h}'' \leftarrow (\mathsf{run}, (\widetilde{h}, \mathtt{ekGen}))$, add $(\widetilde{h},\ obj)$ to Z_1 and return true.
- Else, return false.

- If $\mathtt{Type}\ (\overline{h}) = \text{EK}$ and $\mathsf{acc}\ (obj) = \text{EK}$
  - If $\exists \widetilde{ek}$ s.t. $(\widetilde{ek}, obj) \in Z_1$, then return true if doCompare$(\widetilde{ek}, \widetilde{h})$, else return false.
  - Else if $\exists \widetilde{dk}$ *s.t.* $(\widetilde{dk}, \mathsf{dkId}(obj)) \in Z_1$, return true if doCompare$(\widetilde{h}'', \widetilde{h})$ where $\widetilde{h}'' \leftarrow (\mathsf{run}, (\widetilde{dk}, \mathtt{ekGen}))$, else return false.
  - Else if $\nexists \widetilde{ek}, obj'$ *s.t.* $(\widetilde{ek}, obj') \in Z_1 \wedge$ doCompare$(\widetilde{ek}, \widetilde{h})$ and $\nexists \widetilde{dk}, obj'$ *s.t.* $(\widetilde{dk}, obj') \in Z_1 \wedge$ doCompare$(\widetilde{h}, \widetilde{h}'')$ where $\widetilde{h}'' \leftarrow (\mathsf{run}, (\widetilde{dk}, \mathtt{ekGen}))$, add $(\widetilde{h},\ obj)$ to Z_1 and return true.
  - Else, return false.
- If $\mathtt{Type}\ (\overline{h}) = \text{SK}$ and $\mathsf{acc}\ (obj) = \text{SK}$
  - If $\exists \widetilde{sk}$ s.t. $(\widetilde{sk}, obj) \in Z_1$, then return true if doCompare$(\widetilde{sk}, \widetilde{h})$, else return false.
  - Else if $\exists \widetilde{vk}$ *s.t.* $(\widetilde{vk}, \mathsf{vkGen}(obj)) \in Z_1$, return true if doCompare$(\widetilde{h}'', \widetilde{vk})$ where $\widetilde{h}'' \leftarrow (\mathsf{run}, (\widetilde{h}, \mathtt{vkGen}))$, else return false.
  - Else if $\nexists \widetilde{sk}, obj'$ *s.t.* $(\widetilde{sk}, obj') \in Z_1 \wedge$ doCompare$(\widetilde{sk}, \widetilde{h})$ and $\nexists \widetilde{vk}, obj'$ *s.t.* $(\widetilde{vk}, obj') \in Z_1 \wedge$ doCompare$(\widetilde{vk}, \widetilde{h}'')$ where $\widetilde{h}'' \leftarrow (\mathsf{run}, (\widetilde{h}, \mathtt{vkGen}))$, add $(\widetilde{h},\ obj)$ to Z_1 and return true.
  - Else, return false.
- If $\mathtt{Type}\ (\overline{h}) = \text{VK}$ and $\mathsf{acc}\ (obj) = \text{VK}$
  - If $\exists \widetilde{vk}$ s.t. $(\widetilde{vk}, obj) \in Z_1$, then return true if doCompare$(\widetilde{vk}, \widetilde{h})$, else return false.
  - Else if $\exists \widetilde{sk}$ *s.t.* $(\widetilde{sk}, \mathsf{dkId}(obj)) \in Z_1$, return true if doCompare$(\widetilde{h}'', \widetilde{h})$ where $\widetilde{h}'' \leftarrow (\mathsf{run}, (\widetilde{sk}, \mathtt{vkGen}))$, else return false.
  - Else if $\nexists \widetilde{vk}, obj'$ *s.t.* $(\widetilde{vk}, obj') \in Z_1 \wedge$ doCompare$(\widetilde{vk}, \widetilde{h})$ and $\nexists \widetilde{sk}, obj'$ *s.t.* $(\widetilde{sk}, obj') \in Z_1 \wedge$ doCompare$(\widetilde{h}, \widetilde{h}'')$ where $\widetilde{h}'' \leftarrow (\mathsf{run}, (\widetilde{sk}, \mathtt{vkGen}))$, add $(\widetilde{h},\ obj)$ to Z_1 and return true.
  - Else, return false.

Fig. 36: Simulator $\mathcal{S}^*$

Indistinguishability between H_2 and H_3 (and similarly between H_5 and H_4).

Lemma 13. *The hybrids* H_2 *and* H_3 *are indistinguishable.*

Proof: We show that the view of Test + $\mathcal{S}^{\ddagger}$ remains the same in H_2 and H_3 conditioned on collisions of tags not occurring in $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$ and in $\mathcal{B}[\Sigma_{\mathsf{case}}]$ and $\mathcal{S}^*$ not aborting. This would ensure that H_2 and H_3 are indistinguishable.

We construct two graphs, $\mathbb{G}^{\ddagger}$ and $\mathbb{G}^*$, which represent the view of Test + $\mathcal{S}^{\ddagger}$ in H_2 and H_3 respectively and we show that the graphs are equivalent after removing "extra" nodes and edges which do not participate in the view.

Graph $\mathbb{G}^{\ddagger}$ The graph $\mathbb{G}^{\ddagger}$ is constructed using commands from Test and commands from $\mathcal{S}^{\ddagger}$. Again, the graph is split into two sets of nodes $\mathbb{V}_{\mathsf{Test}}$ and $\mathbb{V}_{\mathcal{A}}$. Commands from Test add new nodes to $\mathbb{V}_{\mathsf{Test}}$ or update a node v as $v.\mathbf{L} \leftarrow v.\mathbf{L} \cup \{\widehat{h}\}$ where $\widehat{h}$ is the handle to be $\mathcal{B}[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}]$. Equivalent handles are grouped together in the same node. For instance, a command $(\mathsf{run}, \widehat{dk}, \mathtt{ekGen}))$, either adds a node v to $\mathbb{V}_{\mathsf{Test}}$ such that $v.\mathbf{L} = \{\widehat{h}\}$ or updates a node v such that $v.\mathbf{L} = \{\widehat{h}\} \cup v.\mathbf{L}$ if a path $\mathtt{node}_{\mathbb{G}^{\ddagger}}(\widehat{dk}) \dashrightarrow v_r \xrightarrow{\text{ekGen}} v_s \dashrightarrow v$ exists.

Commands from $\mathcal{S}^{\ddagger}$ (except $\mathtt{patch}$, $\mathtt{dkPatch}$ and $\mathtt{CPgen}$ commands) are processed similarly. For remaining commands, we add a node v in $\mathbb{V}_{\mathcal{A}}$ such that $v.\mathbf{L} = \{\overline{h}\}$ where $\overline{h}$ is the next handle expected by $\mathcal{S}^{\ddagger}$. We also update $v.\mathsf{obj} = obj$, where obj is the object inside the patch command. We also add edges based on the command sent. For instance, assume the command is $(\mathsf{run}, (\overline{h}, (\mathtt{dkPatch}, obj))), \{(\overline{sk}_i, (\mathtt{dkPatch}, CP_i))\}_i)$ and the new node added is $v^{\star}$. Now first we add an edge, $\mathtt{node}_{\mathbb{G}^{\ddagger}}(\overline{h}) \xrightarrow{\text{patch}} v^{\star}$. Then, $\forall v'$ *s.t.* $\mathsf{decase\text{-}msg}(v^{\star}.\mathsf{obj}, v'.\mathsf{obj}) = m \neq \perp$, we add edges $v^{\star} \xrightarrow{(dk-ct, \mathsf{encase}, m)} v'$. We also add edges $\mathtt{node}_{\mathbb{G}^{\ddagger}}(\overline{sk}_i) \xrightarrow{(sk-ct, \mathsf{encase}, m)} v_i$ where $v_i.\mathsf{obj} = CP_i$.

We also add edges such that if $v_1 \xrightarrow{\text{ekGen}} v_2$ exists, then $v_1 \xrightarrow{(dk-ct, \mathsf{encase}, m)} v_3 \Leftrightarrow v_2 \xrightarrow{(pk-ct, \mathsf{encase}, m)} v_3$. Similarly, edges are added for signing-keys as well.

View of $\mathsf{Test} + \mathcal{S}^{\ddagger}$ in H_2

We define a procedure $\mathsf{prune}^{\ddagger}(\mathbb{G}^{\ddagger})$ which returns a graph $\mathbb{G}_p^{\ddagger}$ such that $\mathbb{G}_p^{\ddagger}$ is constructed as follows from $\mathbb{G}^{\ddagger}$:

1. Remove all nodes v such that $v.\overline{h} = \{\overline{h}\}$ and $\circ \overset{\mathsf{init}}{\to} \overline{h}$ or $\exists v'\ s.t. v' \xrightarrow{\text{ekGen}} v \vee v' \xrightarrow{\text{vkGen}} v$ where $v'.\overline{h} = \{\overline{h}\}$ and $\circ \overset{\mathsf{init}}{\to} \overline{h}$.
2. For all nodes v, set $v.\mathsf{obj} = \bot$.

Note that, conditioned on a tag collision not occurring, $\mathbb{G}_p^{\ddagger}$ contains the view of $\mathsf{Test} + \mathcal{S}^{\ddagger}$ in H_2 as the nodes removed from $\mathbb{G}_p^{\ddagger}$ correspond to "handles" that are never transferred to Test.

Graph $\mathbb{G}^*$ The graph $\mathbb{G}^*$ is constructed using reports from Test, commands from $\mathcal{S}^*$ and the list Z_2 maintained by $\mathcal{S}^*$. Again, the graph is split into two sets of nodes $\mathbb{V}_{\mathsf{Test}}$ and $\mathbb{V}_{\mathcal{A}}$. Updates to $\mathbb{G}^*$ by commands from Test are exactly the updates to $\mathbb{G}^{\ddagger}$.

The commands sent by $\mathcal{S}^*$ are handled similarly and result in updates to $\mathbb{V}_{\mathcal{A}}$. In addition, for every pair $(\widetilde{h}, \overline{h})$ added to Z_2, $\mathtt{node}_{\mathbb{G}^*}(\widetilde{h}).\overline{h} \leftarrow \mathtt{node}_{\mathbb{G}^*}(\widetilde{h}).\overline{h} \cup \{\overline{h}\}$.

View of $\mathsf{Test} + \mathcal{S}^{\ddagger}$ in H_3

We define a procedure $\mathsf{prune}^{\star}(\mathbb{G}^*)$ which returns a graph $\mathbb{G}_p^*$ such that $\mathbb{G}_p^*$ constructed as follows from $\mathbb{G}^*$:

1. Remove all nodes v such that $\nexists(\widetilde{h}, \overline{h}) \in Z_2\ s.t.\ \overline{h} \in v.\overline{h}$
2. Remove all edges $v_1 \xrightarrow{(sk-ct,\mathsf{encase},m)} v_2$ and $v_3 \xrightarrow{(vk-ct,\mathsf{encase},m)} v_2$ where $v_1, v_2, v_3 \in \mathbb{V}_{\mathcal{A}}$ if $\nexists v' \in \mathbb{V}_{\mathcal{A}}\ s.t.\ (v' \xrightarrow{(pk-ct,\mathsf{encase},m)} v_2 \wedge \exists v''\ s.t.\ v'' \xrightarrow{\text{ekGen}} v_s \xrightarrow{\text{transfer}} v') \vee (v' \xrightarrow{(dk-ct,\mathsf{encase},m)} v_2)$.

Note that, conditioned on a tag collision not occurring, $\mathbb{G}_p^*$ contains the view of $\mathsf{Test} + \mathcal{S}^{\ddagger}$ in H_3 as the nodes removed in $\mathbb{G}_p^*$ correspond to "handles" that are not visible to $\mathcal{S}^{\ddagger}$ yet and the edges removed correspond to relations that cannot be determined with the computational unboundedness of $\mathcal{S}^*$.

Equivalence of $\mathbb{G}_p^{\ddagger}$ and $\mathbb{G}_p^*$ Note that, by the definition of $\mathsf{prune}^{\ddagger}$ and $\mathsf{prune}^{\star}$ and the construction of $\mathcal{S}^{\ddagger}$ and $\mathcal{S}^*$, $\mathbb{G}_p^{\ddagger}$ and $\mathbb{G}_p^*$ are equal. It is easy to see that they both consist of the same nodes. Edges added by commands except $\mathtt{dkPatch}$, $\mathtt{patch}$, $\mathtt{CPgen}$ are also the same as $\mathcal{S}^*$ directly relays those commands to $\mathcal{B}[\Sigma_{\mathsf{case}}]$. The construction of $\mathcal{S}^{\ddagger}$ ensures that the checks involving the $\mathsf{tryAssign}$ and $\mathsf{checkAssigned}$ subroutines do not abort the execution of $\mathcal{S}^*$ and thus, edges added by $\mathtt{dkPatch}$, $\mathtt{patch}$ and $\mathtt{CPgen}$ are also the same by existential consistency guarantees of case.

Thus, we can say the the views of $\mathsf{Test} + \mathcal{S}^{\ddagger}$ in H_2 and H_3 are equal. □

C.3.6 Completing the Proof. We now show that function $\mathtt{checkDeltaHiding}^{\ddagger}$ returns false with negligible probability in H_3.

Lemma 14. *For any* $\mathsf{Test} \in \Delta$ *and adversary* $\mathcal{A}$*, let the simulator* $\mathcal{S}^{\ddagger}$ *be as in Figure 31. If* Test *is s-hiding w.r.t.* Σ*, then the function* $\mathtt{checkDeltaHiding}^{\ddagger}$ *returns* false *in the execution of* $\textsc{ideal}\langle\mathsf{Test}(0) \mid \Sigma^{\ddagger}_{\Pi_{\mathsf{case}}} \mid \mathcal{S}^{\ddagger} \circ \mathcal{A}\rangle$ *only with negligible probability.*

Proof: Note that there exists an extractor $\mathcal{E}$ s.t. if $\mathtt{checkDeltaHiding}^{\ddagger}$ returns false, it instead extracts and outputs the test bit. We demonstrate this for the case-packet case, the other cases can be similarly handled. Let $\mathtt{Type}(\widehat{h_0}) = \mathtt{Type}(\widehat{h_1}) = \textsc{cp}$ and the handle received from $\mathcal{B}\big[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}\big]$ be $\overline{h}$,

- if $\exists \overline{g}$, $\exists b \in \{0,1\}$ s.t. $\mathtt{Type}(\overline{g}) = \textsc{dk}$, $\mathtt{dk\text{-}root}(v_b) \rightsquigarrow \mathtt{node}_{\mathbb{G}_b^{\ddagger}}(\overline{g})$ but $\mathtt{dk\text{-}root}(v_{1-b}) \not\rightsquigarrow \mathtt{node}_{\mathbb{G}_{1-b}^{\ddagger}}(\overline{g})$; then $\mathcal{E}$ sends command $\big(\mathsf{run}, (\overline{g}, \{\mathtt{decase\text{-}msg}\}), (\overline{h}, \{\mathtt{decase\text{-}msg}\})\big)$ to $\mathcal{B}\big[\Sigma^{\ddagger}_{\Pi_{\mathsf{case}}}\big]$, if it receives output $m \neq \bot$, then it outputs b as the test bit, else it outputs $1 - b$.

Similarly, every case in $\mathtt{checkDeltaHiding}^{\ddagger}$ can be converted to a corresponding query that breaks the $s-$hiding. □

Proving Indistinguishability of all Hybrids. We now prove indistinguishability of all previous hybrids. The proof holds similarly for the hybrids for test bit $b = 1$. Finally, since $\mathsf{Test} \in \varDelta$ and is $s-$hiding, it holds that $\mathsf{H}_3 \approx \mathsf{H}_4$ (by definition). Thus, we get the overall proof that $\mathsf{H}_0 \approx \mathsf{H}_7$.

Lemma 15. *The hybrids* H_0*,* H_1*,* $\mathsf{H}_{1|2}$*,* H_2 *and* H_3 *are all indistinguishable.*

Proof:

– $\mathsf{H}_3 \approx \mathsf{H}_2$ from Lemma 13. From Lemma 14, this implies that $\texttt{checkDeltaHiding}^{\ddagger}$ returns false with negligible probability in H_2.

– $\mathsf{H}_2 \approx \mathsf{H}_{1|2}$ from Lemma 12. From above, this implies that $\texttt{checkDeltaHiding}^{\ddagger}$ returns false with negligible probability in $\mathsf{H}_{1|2}$.

– $\mathsf{H}_{1|2} \approx \mathsf{H}_1$ from Lemma 11 (conditioned on $\texttt{checkDeltaHiding}^{\ddagger}$ not returning false) and above: $\texttt{checkDeltaHiding}^{\ddagger}$ returns false with negligible probability.

– $\mathsf{H}_1 \approx \mathsf{H}_{0|1} \approx \mathsf{H}_0$ from Lemma 9 and Lemma 10

□