Password-Protected Key Retrieval with(out) HSM Protection[*]

Sebastian Faller
sebastian.faller@ibm.com
IBM Research Europe / ETH Zurich
Zurich, Switzerland

Tobias Handirk
tobias.handirk@uni-wuppertal.de
Bergische Universität Wuppertal
Wuppertal, Germany

Julia Hesse
juliahesse2@gmail.com
IBM Research Europe
Zurich, Switzerland

Máté Horváth
horvath@uni-wuppertal.de
Bergische Universität Wuppertal
Wuppertal, Germany

Anja Lehmann
anja.lehmann@hpi.de
Hasso Plattner Institute
Potsdam, Germany

Abstract

Password-protected key retrieval (PPKR) enables users to store and retrieve high-entropy keys from a server securely. The process is bootstrapped from a human-memorizable password only, addressing the challenge of how end-users can manage cryptographic key material. The core security requirement is protection against a corrupt server, which should not be able to learn the key or offline-attack it through the password protection. PPKR is deployed at a large scale with the WhatsApp Backup Protocol (WBP), allowing users to access their encrypted messaging history when switching to a new device. Davies et al. (Crypto'23) formally analyzed the WBP, proving that it satisfies most of the desired security. The WBP uses the OPAQUE protocol for password-based key exchange as a building block and relies on the server using a hardware security module (HSM) for most of its protection. In fact, the security analysis assumes that the HSM is incorruptible – rendering most of the heavy cryptography in the WBP obsolete.

In this work, we explore how provably secure and efficient PPKR can be built that either relies strongly on an HSM – but then takes full advantage of that – or requires less trust assumption for the price of more advanced cryptography. To this end, we expand the definitional work by Davies et al. to allow the analysis of PPKR with *fine-grained HSM corruption*, such as leakage of user records or attestation keys. For each scenario, we aim to give *minimal* PPKR solutions. For the strongest corruption setting, namely a fully corrupted HSM, we propose a protocol with a simpler design and better efficiency than the WBP. We also fix an attack related to client authentication that was identified by Davies et al.

1 Introduction

Secure messaging apps like Signal and WhatsApp have made end-to-end encrypted (E2EE) communication accessible to millions of end-users. With the deployment of E2EE on user-managed devices, new challenges arise. One is how to ensure that users maintain access to their data when they lose their devices or switch to a new phone. Protocols aiming at the secure backup of a user's chat history – or a user's cryptographic key in general – have been proposed by Signal [Lun19], Apple [Krs16], Google [Wal18] and WhatApp [Wha21]. The common idea in all approaches is to back up the user's high-entropy (encryption) key on a server and enable retrieval when the user correctly authenticates via a human-memorizable secret, like a password or PIN. In particular, when the backup service is offered by the same provider that handles the E2EE communication, extra care is necessary to not undermine the encryption security. This is done by relying on trusted hardware enclaves, such as hardware security modules (HSMs) that protect the user's key material and ensure protection even when the server that runs the HSM gets corrupted.

From the aforementioned approaches, the WhatsApp Backup protocol (WBP) has enjoyed the most attention due to its enhanced protocol design and widespread usage. The WBP deploys OPAQUE [JKX18], a strong asymmetric password-based key exchange protocol (saPAKE) [BM93, BMP00, GMR06], to ensure that no offline-attackable information is leaked as part of the protocol and security can be guaranteed even in the presence of a malicious server.

Password-Protected Key Retrieval. The security of WBP has been formally analyzed by Davies et al. [DFG$^+$23b]. Therein the problem is abstracted as *password-protected key retrieval* (PPKR) and the desirable security guarantees are expressed through an ideal functionality in the universal composability (UC) [Can01] model. A secure PPKR guarantees a multitude of security properties against malicious users and a corrupt server. For the sake of simplicity, we focus on the properties most relevant for our work in the remainder of our paper and refer to [DFG$^+$23b] for a full exposition of PPKR. The core properties (relevant to our work) are as follows:

Secret & Pseudorandom key: Honest users create pseudorandom keys, even if the server acts maliciously. Further, the key remains hidden from the server, as long as it does not correctly guess the password.

Security against Offline and Online Attacks: The user's key is protected through a human-memorizable password, which naturally has low entropy. Thus, it is crucial that the protocol ensures security against offline attacks on the user's password. In fact, PPKR even enforces an upper bound (set to 10 in WBP and [DFG$^+$23b]) that strictly limits the number of wrong password guesses through an online attack. If the limit has been reached through consecutive wrong retrieval attempts, the user's key gets securely deleted.

Authenticated Retrieval: As the user might use the key recovered via PPKR for further encryption, or other cryptographic operations, she should not be tricked into accepting a wrong key from the retrieval.

[*]An extended abstract [FHH$^+$24] of this article appears in the proceedings of the 31st ACM Conference on Computer and Communications Security (ACM CCS 2024). This is the full version.

Davies et al. [DFG+23b] have proven that the original WBP securely instantiates (most of) such an ideal *PPKR* and provides the desired strong security even when the server is corrupted – assuming that the core cryptographic parts of the server are handled by a trusted HSM.

The role of the HSM. The WBP protocol and analysis crucially rely on an HSM that is run within the server as an incorruptible entity. This was manifested in the ideal functionality of PPKR [DFG+23b] that maintains all its essential security guarantees even if the server is corrupt. (We will refer to this security level as Lev-1). Consequently, the WBP protocol has been proven secure under the assumption that the HSM is fully trusted, i.e., all keys used for the internally run OPAQUE protocol, as well as all user-specific files must be protected through the HSM.

While this correctly captures the initial design choice made by the WBP protocol, this is somewhat unsatisfactory from a security and protocol design perspective. First, if an incorruptible HSM can be assumed, the protocol could take more advantage of that – the core of the WBP protocol is the OPAQUE protocol that provides strong security guarantees even when the cryptographic state gets compromised, which wouldn't be needed if the assumption is that such an event can never occur. Second, relying on a perfectly secure sub-entity is a risky assumption. In fact, also trusted hardware modules have a history of getting breached or having to lower their security claims [VBMW+18, VBPS17, BC19, SRW22]. Thus, it would be desirable to clearly express and analyze the impact a partial or full corruption of the HSM has on the expected security guarantees of PPKR.

1.1 Our Contributions

In our work we revisit the design of password-protected key retrieval (PPKR), focusing on the impact of HSM-assurance. We first extend the security model of [DFG+23b] to include more corruption capabilities and express their impact on the expected security guarantees. Second, we present three provably secure PPKR protocols, each having a dedicated trade-off between the strength of the HSM assumption and simplicity. In more detail, we provide the following.

Security Model with Fine-Grained (HSM) Corruption. We extend the ideal functionality of [DFG+23b] to include two further corruption settings: The first is *file leakage*, expressing that the user-specific files that are stored on the server's side to provide the password-authenticated key retrieval can be leaked (repeatedly) to the adversary. We consider this as a partial corruption, and give the dynamically created key files to the adversary but still assume the HSM to be trusted in its core. In practice, an HSM's main task is secure attestation which we consider to be unaffected by this first type of attack. That is, the HSM only lost its protected files but still behaves honestly. The second setting of *full corruption* then models that all security of the HSM is compromised. This translates to an adversary getting access to all keys and files of the HSM, and from then on can deviate arbitrarily from the honest protocol execution.

Despite these strong corruption settings, we aim at a *graceful degradation* of the overall security properties. As in [DFG+23b], our security definition ensures that no offline attacks on the password and protected key are possible, even when the server, i.e., the entity controlling the interface to the HSM, is corrupt. When a user's file gets leaked, this security guarantee is lost, and offline attacks are inevitable – but are also still necessary. That is, the adversary must offline-attack the password of every key it wants to learn, and strong user passwords will still provide a line of defense. The impact of file leakage must also be strictly contained to only the users whose files got compromised . Further, even for users whose key file got lost, the security of active recovery sessions should remain, meaning that the authenticity of the recovered keys must still hold. Upon full corruption of the server, i.e., a complete break of the HSM, this property can no longer be guaranteed but is not fully lost either: it again requires the adversary to correctly guess the user's (login) password to make her accept an incorrect key.

We refer to the optimal security guarantees under server corruption and file leakage as Lev-2 security, and to Lev-3 security for the guarantees under full server corruption.

Table 1: Security achieved by our PPKR protocols and WBP. ✓= achieves optimal protection in that corruption setting, ✗= not secure, where Lev-1**: server corrupt,** Lev-2**: server corrupt + HSM leaks,** Lev-3**: server (and HSM) fully corrupt. Gray are conjectures. WBP was proven secure in a model that is slightly weaker than our** Lev-1 **model, denoted by ✓*, with similar relaxations likely needed for the higher levels, too. The last column indicates if the protocol relies on standard building blocks only.**

	Lev-1	Lev-2	Lev-3	Standard BB
encPw, Fig. 3	✓	✗	✗	✓
encPw+, Fig. 3	✓	✓	✗	✓
OPRF-PPKR, Fig. 4	✓	✓	✓	✗
WBP [DFG+23b]	✓*	✓*	✓*	✗

Simple Protocols for Different HSM-Levels. Having a security model that explicitly captures different levels of server corruption now allows to integrate internal HSM-protection more thoroughly in the protocol design. We propose three PPKR protocols, all relying on a server-internal HSM to protect the user's key files, but each aiming at a different trust level. We present two protocols that provide the same (or even stronger) *formally proven* security than WBP, yet are significantly simpler. Both fall short of achieving Lev-3 security, for which we design a protocol that uses an oblivious pseudorandom function (OPRF) as crucial building blocks, giving up on the reliance on basic primitives only.

Basic/enhanced encrypt-to-HSM: We start with a protocol that fully relies on a trusted HSM for most of its operation, and leverage this to simplify the design as much as possible. The resulting protocol merely uses standard symmetric and asymmetric encryption. We prove our simple protocol to be secure on Lev-1, which gives the same (or even slightly stronger, see discussion below) security guarantees as had been formally proven for WBP in [DFG+23b]. We further show how to upgrade the protocol to Lev-2 security, by relying on fresh encryption keys and salted hashing in the stored password files. The simplicity hinges on the trust in the HSM though, as the protocol loses its security if the HSM gets fully corrupted. Interestingly, the best attacks against all long-term user files are still offline attacks, but the login passwords that users send during an active retrieval session are now revealed in plain when the HSM is fully corrupted, which violates the Lev-3 guarantees.

OPRF-based: Our third protocol provides Lev-3 security, i.e., guarantees optimal protection even in the presence of full server/HSM corruption. This protocol partly resembles WBP, as it also relies on an OPRF as a core primitive. Recall that WBP relies on OPAQUE, an asymmetric PAKE that is built from an OPRF. By building our protocol directly from OPRFs and the clearly specified security guarantees as concrete target, our protocol enjoys a much cleaner design. In particular, we can omit all OPAQUE parts that are not needed for PPKR (which does not aim at fresh session keys as aPAKE). Overall, we propose an OPRF-based PPKR that has a simpler protocol design and better efficiency than WBP, and that fixes several attacks related to user authentication that were identified by Davies et al. [DFG+23b].

Strengthening Security. We also use the opportunity to remove unwanted (in)security artifacts that were present in the WBP and had to be included in the ideal functionality for a provable security treatment of that real-world protocol. These artifacts comprise the re-routing of logins to different files, whenever the server is corrupt. All our three protocols prevent that attack. In the Lev-2/3 setting, WBP has further undesirable – albeit not fatal – behavior, such as potentially leaking the information whether several honest users have the same password. Our protocols for these levels also improve on these shortcomings.

1.2 Related Works

The closest work to ours is the work by Davies et al. [DFG+23b], which we extend with a more fine-grained analysis of the HSM corruption settings. As we now distinguish between the "front-end" corruption of the server's networking part and further internal corruption (of the HSM), this might yield the question whether this is just a special 2-party version of the more general concept of *password-protected* (t, n)*-secret sharing* (PPSS) [BJSL11, JKKX16, CLLN14]. In PPSS, the user can store a password-protected secret (such as the backup key) with the help of n servers. The user can later recover the secret if she provides the correct password to at least t servers. Secure PPSS solutions ensure security against offline attacks on the user's password as long as less than the threshold gets corrupted. While a $(2, 2)$ version of PPSS is similar in spirit to PPKR, the main difference is in the overall setting: PPKR captures the single-server environment that is currently deployed by WhatsApp, and envisioned by other major E2EE providers too [Lun19, Krs16, Wal18]. Therein the user communicates with a single server only, which might run internal measures (such as an HSM) to enhance security. In contrast, PPSS requires the user to individually talk to all n parties, which must also run the equivalent code. In our HSM-supported setup, such direct communication between the user and the HSM is not desirable, and the server's networking part and HSM are clearly distinct. Thus, while PPSS appears more general, it also excludes the most common real-world deployment setting that we are targeting directly with PPKR.

The dedicated one-server setting of PPKR also distinguishes it from *password-protected encryption* protocols [AMMR18, WH20, CGMS21, DHL22] which work in a multi-server setting with n independent parties too. Further, these works target symmetric encryption directly, and not key-recovery that can be used for any key and subsequent cryptographic usage of it.

The recent works on Credential-less Secret Recovery [Sca19, OSV23] are for a somewhat similar single-server setting, as PPKR. Therein, a user stores a secret on some cloud storage and uses the additional power of a trusted execution environment (TEE) for secure recovery. In contrast to PPKR however, [OSV23] relies on a publicly accessible blockchain instead of passwords to authenticate users.

2 Preliminaries

In our protocols, we make use of standard cryptographic primitives, namely strongly EUF-CMA secure digital signatures (Sig), and IND-CPA and IND-CCA secure public key encryption (PKE), see e.g., [BS23]. We will also use *authenticated encryption* (AE), i.e., symmetric encryption (SE) that is IND-CPA secure and has ciphertext integrity in the following sense:

Definition 1 (Ciphertext integrity)**.** The advantage of an adversary $\mathcal{A}$ against the *integrity of ciphertexts* (INT-CTXT) of a symmetric encryption scheme SE = (KeyGen, Enc, Dec) is defined as

$$\mathbf{Adv}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{A},\mathsf{SE}}(\lambda) := \Pr[\mathsf{Dec}(k, c^*) \neq \bot : c^* \xleftarrow{\$} \mathcal{A}^{\mathsf{Enc}(k,\cdot)}(1^\lambda)],$$

where $k \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$ and c^* is fresh in the sense that it has never been output by the encryption oracle $\mathsf{Enc}(k, \cdot)$.

For simplicity, we will assume that KeyGen samples $k \xleftarrow{\$} \{0, 1\}^\lambda$. We define two additional properties of an authenticated encryption scheme that will be required in this work, namely *equivocability* and *random-key robustness.* Equivocability means that a simulator is able to produce ciphertexts of the scheme without committing to the plaintext. If the ciphertext must be decrypted then the simulator can output an appropriate key to open the ciphertext to a message of choice. Random-key robustness means that an adversary cannot come up with a ciphertext that simultaneously decrypts under *two* keys if the keys are chosen uniformly at random. According to [JKX18], this can be instantiated in the standard model with encrypt-then-MAC using HMAC or in the ROM by adding $H(k, c)$ to an authenticated ciphertext c.

Definition 2. A symmetric encryption scheme SE is *equivocable* if for any efficient algorithm $\mathcal{A}$, there exists an efficient stateful simulator $\textsc{Sim}_{\mathsf{EQV}}$ s.t. the distinguishing advantage $\mathbf{Adv}^{\mathsf{EQV}}_{\mathcal{A},\mathsf{SE}}$ of $\mathcal{A}$'s view in the following two games is a negligible function in λ:

- The real game: $\mathcal{A}$ outputs message m and computes its final output given (c, k), where $k \xleftarrow{\$} \{0, 1\}^\lambda$ and $c \xleftarrow{\$} \mathsf{Enc}(k, m)$.
- The ideal game: $\mathcal{A}$ outputs message m and computes its final output given (c, k), where $c \xleftarrow{\$} \textsc{Sim}_{\mathsf{EQV}}(|m|)$ and $k \xleftarrow{\$} \textsc{Sim}_{\mathsf{EQV}}(c, m)$.

Definition 3. A symmetric encryption scheme SE is *random-key robust* if for any efficient adversary $\mathcal{A}$

$$\mathbf{Adv}^{\mathsf{rkr}}_{\mathcal{A},\mathsf{SE}}(\lambda) := \Pr_{k,k' \xleftarrow{\$} \{0,1\}^\lambda}\left[\begin{array}{l}\mathsf{Dec}(k, c) \neq \bot \\ \wedge \mathsf{Dec}(k', c) \neq \bot\end{array} : c \xleftarrow{\$} \mathcal{A}(k, k')\right]$$

is a negligible function of λ.

$\mathcal{F}_{\mathsf{PPKR}}$ is parameterized with a security parameter λ. $\mathcal{F}_{\mathsf{PPKR}}$ talks to a server S where S is encoded in sid. S can have corruption state Honest, Corrupt, or FullyCorrupt. $\mathcal{F}_{\mathsf{PPKR}}$ further allows file leakage via the LeakFile corruption command. $\mathcal{F}_{\mathsf{PPKR}}$ also talks to the adversary $\mathcal{A}$, and arbitrary clients $\mathsf{ID_C}$. If the functionality tries to retrieve a record that does not exist, it ignores the incoming message. We write $\mathsf{tx}[\cdot]$ for a list of counters. For brevity, we omit session identifier sid from all inputs, outputs, and records. $\mathcal{F}_{\mathsf{PPKR}}$ ignores repeated inputs with the same ssid and in that case activates $\mathcal{A}$.

Corruption interfaces

```
On corruption command (LeakFile) from $\mathcal{A}$:
  LS.1 Create an empty list $\mathbf{ID_C}$.
  LS.2 For every record $\langle \textsc{File}, [\mathsf{ID_C}], [\mathsf{pw}], [K], [\mathsf{ctr}], * \rangle$:
         (1) Append $(\mathsf{ID_C}, \mathsf{ctr})$ to $\mathbf{ID_C}$.
         (2) If no record $\langle \textsc{leaked}, \mathsf{ID_C}, *, *, * \rangle$ exists, add a record
             $\langle \textsc{leaked}, \mathsf{ID_C}, \mathsf{pw}, K, 1 \rangle$.
         (3) Otherwise, add a record $\langle \textsc{leaked}, \mathsf{ID_C}, \mathsf{pw}, K, i+1 \rangle$, where $i \in \mathbb{N}$
             is the largest number, such that a record $\langle \textsc{leaked}, \mathsf{ID_C}, *, *, i \rangle$ exists.
  LS.3 Output $\mathbf{ID_C}$ to $\mathcal{A}$.

On corruption command (Corrupt, $P$) from $\mathcal{A}$:
  C.1 Mark $P$ as Corrupt.

On corruption command (FullyCorrupt, S) from $\mathcal{A}$:
  FC.1 Mark S as FullyCorrupt and run LS.1-LS.2
  FC.2 For each $(\mathsf{ID_C}, \cdot) \in \mathbf{ID_C}$, set $\mathsf{tx}[\mathsf{ID_C}] \leftarrow i$, where $i$ is the largest $i \in \mathbb{N}$
       such that a record $\langle \textsc{leaked}, \mathsf{ID_C}, \mathsf{pw}, K, i \rangle$ exists.
  FC.3 In every record $\langle \textsc{File}, *, *, *, [\mathsf{ctr}], * \rangle$ overwrite ctr with $\infty$.
  FC.4 Output $\mathbf{ID_C}$ to $\mathcal{A}$
```

Initialization phase

```
On input (InitC, ssid, pw) from $\mathsf{ID_C}$: // Client $\mathsf{ID_C}$ starts initialization of a password-protected key
  IC.1 Choose $K \xleftarrow{\$} \{0,1\}^\lambda$
  IC.2 If a record $\langle \textsc{Init}, \mathsf{ssid}, \mathsf{ID_C}, \perp, \perp, \mathtt{srvOk} \rangle$ exists, overwrite $(\perp, \perp)$ in it
       with $(\mathsf{pw}, K)$
  IC.3 Otherwise record $\langle \textsc{Init}, \mathsf{ssid}, \mathsf{ID_C}, \mathsf{pw}, K, \perp \rangle$
  IC.4 Send $(\textsc{InitC}, \mathsf{ssid}, \mathsf{ID_C})$ to $\mathcal{A}$

On input (InitS, ssid, $\mathsf{ID_C}$) from S: // Server agrees to initialize $\mathsf{ID_C}$
  IS.1 If a record $\langle \textsc{Init}, \mathsf{ssid}, \mathsf{ID_C}, *, *, \perp \rangle$ exists, overwrite $\perp$ with $\mathtt{srvOk}$
  IS.2 Otherwise record $\langle \textsc{Init}, \mathsf{ssid}, \mathsf{ID_C}, \perp, \perp, \mathtt{srvOk} \rangle$
  IS.3 Send $(\textsc{InitS}, \mathsf{ssid}, \mathsf{ID_C})$ to $\mathcal{A}$

On input (CompleteInitC, ssid, $b_C$) from $\mathcal{A}$: // Complete initialization for client
  CIC.1 Retrieve $\langle \textsc{Init}, \mathsf{ssid}, [\mathsf{ID_C}], *, [K], \mathtt{srvOk} \rangle$ and drop the query if $K = \perp$
  CIC.2 If $b_C = 1$, output $(\textsc{InitRes}, \mathsf{ssid}, K)$ to $\mathsf{ID_C}$
  CIC.3 If $b_C = 0$, output $(\textsc{InitRes}, \mathsf{ssid}, \textsc{Fail})$ to $\mathsf{ID_C}$

On input (CompleteInitS, ssid, $b_S$) from $\mathcal{A}$: // Complete initialization for server
  CIS.1 Retrieve $\langle \textsc{Init}, \mathsf{ssid}, [\mathsf{ID_C}], [\mathsf{pw}], [K], \mathtt{srvOk} \rangle$ and drop the query if
        $K = \perp$
  CIS.2 If $b_S = 0$, output $(\textsc{InitRes}, \mathsf{ssid}, \mathsf{ID_C}, \textsc{Fail})$ to S. Else continue.
  CIS.3 Store record $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{pw}, K, \mathsf{ctr}, \boxed{\textsc{Honest}} \rangle$ if $\mathsf{ID_C}$ is honest and
        $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{pw}, K, \mathsf{ctr}, \boxed{\textsc{Malicious}} \rangle$ otherwise, where $\mathsf{ctr} \leftarrow \infty$ if S
        is FullyCorrupt, and $\mathsf{ctr} \leftarrow 10$ else, overwriting any existing record
        $\langle \textsc{File}, \mathsf{ID_C}, *, *, *, * \rangle$
  CIS.4 If S is FullyCorrupt, record $\langle \textsc{leaked}, \mathsf{ID_C}, \mathsf{pw}, K, i+1 \rangle$, where $i \in \mathbb{N}$
        is the largest number, such that a record $\langle \textsc{leaked}, \mathsf{ID_C}, *, *, i \rangle$ exists, or
        $i = 1$ if no such record exists. If $\mathsf{tx}[\mathsf{ID_C}]$ is undefined set $\mathsf{tx}[\mathsf{ID_C}] \leftarrow 1$.
  CIS.5 Send $(\textsc{InitRes}, \mathsf{ssid}, \textsc{Succ})$ to S
```

Figure 1: Ideal functionality $\mathcal{F}_{\mathsf{PPKR}}$ for password-protected key retrieval, offline attacks, and initialization interfaces. The $\boxed{\text{boxed code}}$ can be dropped to strengthen $\mathcal{F}_{\mathsf{PPKR}}$ (see Section 3.2).

Recovery Phase

```
On input (RecC, ssid, pw') from $\mathsf{ID_C}$: // Client starts key recovery from password pw'
  RC.1 If a record $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, \perp, [\mathsf{pw}], * \rangle$ exists, overwrite $\perp$ with $\mathsf{pw}'$
       and set $match \leftarrow (\mathsf{pw} \overset{?}{=} \mathsf{pw}')$.
  RC.2 Otherwise record $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, \mathsf{pw}' \rangle$ and set $match \leftarrow \perp$
  RC.3 Send $(\textsc{RecC}, \mathsf{ssid}, \mathsf{ID_C}, match)$ to $\mathcal{A}$

On input (RecS, ssid, $\mathsf{ID_C}$, $\mathsf{pw}^*$, $K^*$, $i$) from S: // Server starts recovery session for $\mathsf{ID_C}$
  RS.1 Retrieve $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, [\mathsf{pw}'] \rangle$, otherwise set $\mathsf{pw}' \leftarrow \perp$ and record
       $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, \mathsf{pw}' \rangle$
  RS.2 If S is FullyCorrupt and $\mathsf{pw}^* \neq \perp$: set $\overline{\mathsf{pw}} \leftarrow \mathsf{pw}^*$, $\overline{K} \leftarrow K^*$
  RS.3 If S is FullyCorrupt, $\mathsf{pw}^* = \perp$, and $i \geq \mathsf{tx}[\mathsf{ID_C}]$, retrieve the record
       $\langle \textsc{leaked}, \mathsf{ID_C}, [\mathsf{pw}], [K], i \rangle$ and set $\mathsf{ctr} \leftarrow \infty$. Otherwise, retrieve the
       record $\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{pw}], [K], [\mathsf{ctr}], * \rangle$. Set $\overline{\mathsf{pw}}, \overline{K}$ as follows:
         (1) If no such record exists, set $\overline{\mathsf{pw}} \leftarrow \perp, \overline{K} \leftarrow \textsc{Fail}$.
         (2) If $\mathsf{ctr} = 0$, set $\overline{\mathsf{pw}} \leftarrow \perp, \overline{K} \leftarrow \textsc{DelRec}$ and delete
             $\langle \textsc{File}, \mathsf{ID_C}, *, *, *, * \rangle$.
         (3) Else, set $\overline{\mathsf{pw}} \leftarrow \mathsf{pw}, \overline{K} \leftarrow K$ and overwrite ctr with $\mathsf{ctr} - 1$ in the
             File record.
  RS.4 Append $\overline{\mathsf{pw}}$ and $\overline{K}$ to record $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, \mathsf{pw}' \rangle$
  RS.5 If $\mathsf{pw}' = \perp$, set $match \leftarrow \perp$, else set $match \leftarrow (\overline{\mathsf{pw}} \overset{?}{=} \mathsf{pw}')$
  RS.6 Send $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C}, match)$ to $\mathcal{A}$.

On input (CompleteRecC, ssid, $b_C$) from $\mathcal{A}$: // Complete recovery for client
  CRC.1 Retrieve record $\langle \textsc{Rec}, \mathsf{ssid}, [\mathsf{ID_C}], [\mathsf{pw}'], [\mathsf{pw}], [K] \rangle$ and drop the
        query if $\mathsf{pw}' = \perp$.
  CRC.2 Determine the output $K'$ as follows:
          (1) If $K \in \{\textsc{Fail}, \textsc{DelRec}\}$, set $K' \leftarrow K$
          (2) If $\mathsf{pw} = \mathsf{pw}'$ and $b_C = 1$, set $K' \leftarrow K$
          (3) In all other cases, set $K' \leftarrow \textsc{Fail}$
  CRC.3 Send $(\textsc{RecRes}, \mathsf{ssid}, K')$ to $\mathsf{ID_C}$.

On input (CompleteRecS, ssid, $b_S$) from $\mathcal{A}$: // Complete recovery for server
  CRS.1 Retrieve record $\langle \textsc{Rec}, \mathsf{ssid}, [\mathsf{ID_C}], [\mathsf{pw}'], [\mathsf{pw}], [K] \rangle$ and drop the
        query if $\mathsf{pw}' = \perp$.
  CRS.2 Determine the output $result$ as follows:
          (1) If $K \in \{\textsc{Fail}, \textsc{DelRec}\}$, set $result \leftarrow K$.
          (2) If $b_S = 1$ and either $\mathsf{pw} = \mathsf{pw}'$ $\boxed{\text{or a record}}$
              $\boxed{\langle \textsc{File}, \mathsf{ID_C}, *, *, *, \textsc{Malicious} \rangle \text{ exists}}$, set $result \leftarrow \textsc{Succ}$.
          (3) In all other cases, set $result \leftarrow \textsc{Fail}$.
  CRS.3 If $result = \textsc{Succ}$ and there exists a record $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{pw}, K, [\mathsf{ctr}], * \rangle$,
        overwrite ctr with 10 if S is not FullyCorrupt or with $\infty$ if S is
        FullyCorrupt.
  CRS.4 Send $(\textsc{RecRes}, \mathsf{ssid}, result)$ to S.
```

Attack interfaces

```
On input (MaliciousInit, $\mathsf{ID_C}$, $\mathsf{pw}^*$, $K^*$) from $\mathcal{A}$: // Malicious initialization with
adversarial password $\mathsf{pw}^*$ and key $K^*$
  MI.1 If S is Honest, ignore this input.
  MI.2 Record $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{pw}^*, K^*, 10, \boxed{\textsc{Malicious}} \rangle$, overwriting any existing
       record $\langle \textsc{File}, \mathsf{ID_C}, *, *, *, * \rangle$

On input (MaliciousRec, $\mathsf{ID_C}$, $\mathsf{pw}^*$) from $\mathcal{A}$: // Malicious server guesses $\mathsf{ID_C}$'s
password, subject to non-zero file counters
  MR.1 If S is Honest, ignore this input.
  MR.2 If no record $\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{pw}], [K], [\mathsf{ctr}], * \rangle$ exists, output Fail to $\mathcal{A}$.
  MR.3 If $\mathsf{ctr} = 0$, delete record $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{pw}, K, \mathsf{ctr}, * \rangle$ and output
       $(\textsc{DelRec}, \mathsf{ID_C})$ to $\mathcal{A}$
  MR.4 If $\mathsf{pw}^* = \mathsf{pw}$, overwrite ctr in the record with 10 and output $K$ to $\mathcal{A}$.
       Otherwise, overwrite ctr with $\mathsf{ctr} - 1$ and output Fail to $\mathcal{A}$

On input (OfflineAttack, $\mathsf{ID_C}$, $\mathsf{pw}^*$, $i$) from $\mathcal{A}$: // Offline attack on leaked key files
- unlimited number of guesses allowed
  OA.1 If a record $\langle \textsc{leaked}, \mathsf{ID_C}, \mathsf{pw}^*, [K], i \rangle$ exists, output $K$ to $\mathcal{A}$. Otherwise,
       output Fail to $\mathcal{A}$.
```

Figure 2: Ideal functionality $\mathcal{F}_{\mathsf{PPKR}}$, recovery interfaces. The $\boxed{\text{boxed code}}$ can be dropped to strengthen $\mathcal{F}_{\mathsf{PPKR}}$ (see Section 3.2).

3 Security model

Our ideal functionality is based on the definition from Davies et al. [DFG+23b], and models how a user – identified through a client identifier $\mathsf{ID_C}$ – can store and retrieve a strong cryptographic key

K with the help of an external server. The server S is encoded in the session id *sid* and is the central entity that interacts with many clients. At initialization, the client ID_C creates a random key K that it stores with the server, protected under a password pw. Later, the client can recover the key K from the server, but only if it correctly authenticates with the same password again.

The definition of Davies et al. [DFG+23b] provides very strong guarantees in case of server compromise: even if the server is corrupt, the client's key and password remain secure unless the adversary correctly guesses the password in a few attempts. In the realization, this essentially requires the server to internally run an *incorruptible* entity, like an HSM. The HSM is then responsible to protect the client's key and enforce strict access control to it.

In our work, we model stronger types of server corruption to avoid the need to assume an incorruptible sub-part in the constructions. Beyond the server compromise from [DFG+23b], our model captures *file leakage* of the stored client files and a *fully corrupt* server. The latter translates to a corruption of all parts of the server, including an internally hosted HSM. Beyond stronger corruption settings, we further make the definition more general to allow for a broader set of protocol flows and also remove some artifacts that had to be included in [DFG+23b] to capture the security of the concrete WhatsApp protocol. For an overview and explanation of these changes, we refer to Appendix A.

3.1 Ideal PPKR Functionality

We start by recalling the main functional interfaces of $\mathcal{F}_{\mathsf{PPKR}}$ (when all parties are honest) and then explain the different corruption settings and their impact on the guaranteed security properties.

Initialization. To use the key recovery service, the client ID_C and server S must first engage to create a password-protected account. This can be initiated by either party through the $\textsc{InitC}$ and $\textsc{InitS}$ interfaces, respectively. The client's interface takes the user's password pw as input and lets $\mathcal{F}_{\mathsf{PPKR}}$ create a random key K that gets stored along with ID_C and pw. If completion towards both parties is signaled by the adversary through the $\textsc{CompleteInitC}$ and $\textsc{CompleteInitS}$ interfaces, a key file $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID}_\mathsf{C}, \mathsf{pw}, K, \mathsf{ctr}\rangle$ is stored in the functionality. The counter is set to $\mathsf{ctr} \leftarrow 10$, which determines how many incorrect password recoveries will be tolerated before the file is deleted.

The initialization can be called repeatedly by the same client ID_C, which allows the renewal of the key and possibly the replacing of the password.

Recovery. If a client ID_C wishes to recover its key, it triggers the $\textsc{RecC}$ interface for a password attempt pw'. When the server initiates a recovery for ID_C too, and a key file $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID}_\mathsf{C}, \mathsf{pw}, K, \mathsf{ctr}\rangle$ for ID_C is stored within $\mathcal{F}_{\mathsf{PPKR}}$, the counter ctr is decremented by 1 and the adversary $\mathcal{A}$ learns whether $\mathsf{pw} \overset{?}{=} \mathsf{pw}'$. This bit is considered unavoidable leakage in password-based protocols, as the observable protocol behavior usually strongly differs depending on whether the password authentication was successful or not. If the counter reaches 0, the file $\textsc{File}$ for ID_C gets deleted, and the recovery "key" is set to $\textsc{DelRec}$.

If the adversary signals the completion of the recovery attempt towards the client via $\textsc{CompleteRecC}$ and the file hasn't been deleted, the client will receive the correct key K from its record if $\mathsf{pw} = \mathsf{pw}'$ or a failure notification $\textsc{Fail}$ otherwise. Likewise, if the adversary completes the session for ID_C towards the server via the $\textsc{CompleteRecS}$ interface, the server finally learns whether the password attempt was correct (or the file was deleted). Further, the counter ctr is reset to 10 when the password was correct and the session got completed.

Core Security Properties. In summary, the functionality $\mathcal{F}_{\mathsf{PPKR}}$ ensures that only the legitimate client ID_C can initiate sessions to setup or recover a key (*implicit client-to-server authentication*), and all generated keys are fresh and chosen at random (*pseudorandomness of* K). The server S neither learns the password pw nor key K (*secrecy of* pw *and* K), only if a recovery was run on an incorrect password attempt pw' or not. If the password attempt was correct and the client recovered a key K, she can be ensured that it is the correct key (*authenticity of* K). An adversary has at most 10 guessing attempts against the client's password (*no offline attacks, limited online attacks*). We give an overview of the core security properties of $\mathcal{F}_{\mathsf{PPKR}}$ – and their degradation with corruption – in Table 2.

3.2 Modelling Server (and HSM) Corruption

The central entity in our system is the server S, which provides the password-protected key retrieval service to its clients, and which we model to be corruptible in several ways. Note that the HSM does not appear as an explicit entity in a PPKR functionality, but is rather considered an artifact on how the server's code is deployed. The original PPKR definition [DFG+23b] required the server code to be split into a corruptible networking part, and an incorruptible cryptography part, e.g., deployed on a permanently secure HSM. In the protocol, the HSM was then responsible for performing the cryptographic operations and securely storing the users' password-protected keys. Our model additionally allows fine-grained corruption of the cryptography part and distinguishes whether the (HSM-protected) key files are leaked, or even all of the additional protection is lost.

We start with a brief overview of the three corruption settings, and then explain their impact on the guaranteed security properties. For the sake of clarity, we explain how the additional corruption settings translate to an HSM-supported protocol setup.

Server Corrupt: This is the original corruption status of [DFG+23b], which gives a corrupt server some, rather benign, attack capabilities, but still maintains most of the security guarantees. In the construction, this requires that the server is mainly a connecting interface between the clients and a secure HSM, where the key material and client records remain protected during this corruption thanks to being executed in an HSM, that can securely attest all outgoing messages it sends (via the server) to the clients.

File Leakage: We additionally model that the client files, containing the information to verify the recovery password and retrieve the client's key, can be leaked to the adversary. In the context of an HSM-supported setup, this means that the adaptively stored and maintained information by the HSM can get compromised. The file leakage can happen repeatedly, with the adversary obtaining snapshots of all the stored client files. This does neither

imply that the server nor HSM are corrupt though. In particular, the HSM still behaves honestly and its attestation remains secure.
We believe this realistically models that the core attestation (key) of the HSM enjoys particularly strong protection, whereas the HSM-protected database of possibly millions of client records is less secure and can be vulnerable to leakage attacks.

Server (and HSM) Fully Corrupt: We also want to capture the security (loss) when the server gets fully corrupted, meaning that all parts of the server – including the HSM – are under the control of the adversary. This implies the previous two corruption settings, i.e., normal server corruption and *continuous* compromise of all files, and also goes beyond as we now assume that the adversary is in full control of the HSM. That is, the attestation is no longer trusted, and the adversary can arbitrarily deviate from the original HSM protocol.

We now explain how each corruption impacts the security guarantees of $\mathcal{F}_{\mathrm{PPKR}}$. Note that the status of server corruption and file leakage are independent of each other, i.e., they can occur separately or jointly. In the latter case, the adversary's capabilities are simply the combination of both individual attacks.

Server Honest & No Leakage. If the server is honest and no files have been leaked, the functionality guarantees full protection of the clients' information. In particular, only the legitimate client $\mathsf{ID_C}$ can initiate the recovery of a previously stored key. This ensures that the account and key of an honest client cannot be deleted by the adversary through repeated incorrect authentication attempts.

An initially honest client can get corrupted through the corruption command (Corrupt, $\mathsf{ID_C}$) though. In that case, the adversary gets up to 10 guesses on the client's password and learns the key K when it provides the correct password.

An unavoidable attack even if all parties are honest, are network attacks, i.e., interrupting an honestly initiated session. This is modeled through the DoS-bit b_C and b_S, in the CompleteInitC and CompleteInitS interfaces for the initialization and the CompleteRecC, CompleteRecS interfaces for the recovery respectively. By providing $b_C = 0$ or $b_S = 0$, the adversary can make the session fail for either party. If it does so towards the server, this leads to a failed initialization, i.e., no file gets recorded in the functionality; or a failed recovery, i.e., the counter does not get reset to 10, even if the client provided the correct password.

Server Corrupt. To corrupt the server, the adversary sends the corruption command (Corrupt, S). From then on, the functionality no longer enforces proper client authentication – which is the main impact of this corruption type. As a consequence, the adversary can now replace records of honest clients with a malicious password and key (through the MaliciousInit interface). It still does not learn the passwords and keys of any honestly created or existing records but can try to recover a client's key from a stored record via password guessing (through the MaliciousRec interface). This password guessing is limited by the counter ctr though (which will be enforced through the honest HSM in the protocol), i.e., the adversary has at most 10 guesses for each honest account. Thus, there are still no unlimited offline attacks possible on any accounts.

Difference to [DFG⁺23b]: In the original functionality, a corrupt server was able to re-route sessions from an honest client $\mathsf{ID_C}$ to a different client $\mathsf{ID_C^*}$. This was modeled because the WhatsApp protocol allowed such an attack. In general, this can be prevented easily, and we therefore remove this weakness in our definition.

File Leakage. The adversary can repeatedly compromise the existing records, by sending (LeakFile) to $\mathcal{F}_{\mathrm{PPKR}}$. The functionality then provides the adversary with a list of all clients $\mathbf{ID_C}$ for which files are recorded. It creates leaked copies of all records and now permits *unlimited* offline attacks on these records via the OfflineAttack interface. As the initialization interface can be used by the client to renew its key and/or password, there can be different files for the same client $\mathsf{ID_C}$ at different times. We therefore attach a counter i to each leaked record and allow offline attacks on each version.

Note that the adversary can start using the OfflineAttack on records only *after* it compromised them. This resembles the notion of pre-computation attack resistance which was introduced as a strengthening of aPAKE with the OPAQUE protocol [JKX18].

Apart from allowing offline attacks on leaked key files, all other security guarantees of $\mathcal{F}_{\mathrm{PPKR}}$ are still intact. In particular, key authenticity is guaranteed and the deletion counter is still set honestly as instructed by our functionality.

Server (and HSM) Fully Corrupt. The strongest corruption level considers the server (and HSM) to be fully corrupted, which can be triggered through the (FullyCorrupt, S) interface. This marks the server as FullyCorrupt and leaks all existing and newly created key files to the adversary. This full corruption gives $\mathcal{A}$ all capabilities described above for the corrupt server and file leakage settings. In addition, the functionality now gives up on key authenticity and the strict enforcement of the deletion counter.

The latter is done in the FullyCorrupt interface, which sets all counters of stored and newly created files to ∞, such that they no longer get deleted automatically. When a client recovers her key, $\mathcal{A}$ can now freely decide if it wants to signal DelRec or not.

The full corruption also enables the adversary to online attack recovery sessions from honest clients, in two ways. The first is an *key planting and online guessing* attack, where the adversary can provide an arbitrary key K^* along with a password guess pw^* as optional inputs in the RecS interface. It then learns whether $\mathsf{pw}^* \overset{?}{=} \mathsf{pw}'$. If it guessed the password attempt correctly, K^* is returned to the client instead of the proper key. Note that this planting requires the correct guess of the client's password attempt, and also allows only a single password guess per recovery session. However, weak passwords might be known to the adversary through its offline attack capability on all key files.

The second enabled online attack is to re-use any of the previously leaked key files, which might have been replaced in the meantime. Here, $\mathcal{A}$ provides an additional input i in the RecS interface, that allows it to run the honest clients recovery against the old key file of version i. If the client's password attempt is pw' matches the pw contained in the old key file, the client gets tricked into accepting its old key.

Summary of Security Guarantees & Security Levels. Note that even in the full corruption setting, $\mathcal{F}_{\mathsf{PPKR}}$ still guarantees two important security properties: (1) pseudorandomness of keys – as the keys generated by honest clients are still chosen at random by the functionality and (2) the honest client's passwords and keys are never directly accessible to the adversary. For each key K that $\mathcal{A}$ wants to compromise, it has to correctly guess the user's password through an offline attack. Thus, users who have chosen a strong password can still hope that their key remains safe. Even the third core property of key authenticity is not entirely lost when S is fully corrupt, as this still requires the adversary to correctly guess the honest user's password. We give a summary and comparison of the main security properties in the different corruption settings in Table 2 below.

For brevity, we refer to security that is maintained when the server S is corrupt, but no files got leaked as Lev-1 security. Lev-2 security covers the optimal guarantees up to joint server corruption and file leakage, considering both attack capabilities combined. Finally, Lev-3 security refers to schemes that maintain the optimal security guarantees up to full server corruption.

Flaws in $\mathcal{F}_{\mathsf{PPKR}}$ from [DFG+23b]. In the WBP, if a corrupt party initializes and later runs a recovery, it can always reset the counter to 10, even if it used different passwords in initialization and recovery [1]. This weakness also exists in our protocol $\pi^{\mathsf{OPRF\text{-}PPKR}}$ aiming at Lev-3 security that we present in Section 4.3. The weakness was not captured in the original functionality in [DFG+23b] and we fix this in Figures 1 to 2. We remark that weakening $\mathcal{F}_{\mathsf{PPKR}}$ in this way gives only very limited additional capabilities as the corrupt party can always reset its file by simply using the correct password.

Interestingly, our protocols π^{encPw} (see Section 4.1) and $\pi^{\mathsf{encPw+}}$ (see Section 4.2), which only aim at Lev-1, resp. Lev-2, security, do not have this property. For this reason, we mark the corresponding code in Figures 1 to 2 in $\boxed{\text{boxes}}$, which allows us to strengthen $\mathcal{F}_{\mathsf{PPKR}}$ by dropping the boxed code.

Table 2: Overview of security properties guaranteed by $\mathcal{F}_{\mathsf{PPKR}}$ under different corruption settings. We refer to the following level: Lev-**1** = S **is corrupt,** Lev-**2** = S **is corrupt and files are leaked (both attacks combined),** Lev-**3** = **fully corrupt** S.

Property	Honest S	Lev-1	Lev-2	Lev-3
Pseudorandom key K	✓	✓	✓	✓
No direct leakage of pw, K	✓	✓	✓	✓
Limit file access to $\mathsf{ID_C}$	✓	✗	✗	✗
No offline attacks on pw, K	✓	✓	✗	✗
No precomputation for offline attacks	n.a.	n.a.	✓	✓
No offline attacks on pw′ (pw-attempt)	✓	✓	✓	✓
Upper limit on incorrect recoveries	✓	✓	✓	✗
Key authenticity	✓	✓	✓	✗

4 Constructions

In this section, we discuss three protocols for password-protected key retrieval that securely realize the functionality $\mathcal{F}_{\mathsf{PPKR}}$. To ensure security in the single-server setting, we adapt the approach from WhatsApp [Wha21] and let all our protocols rely on a Hardware Security Module (HSM). One can think of the HSM as a particularly secure part of the server or a dedicated module that only the server can access. We make the assumption that the code of the HSM is executed honestly and messages from the HSM are attested, i.e., signed such that anyone can verify the message's origin.

Each of the protocols in this section protects against a different level of attacks against the HSM. The first protocol, basic encrypt-to-HSM (Section 4.1), is based solely on public-key and symmetric encryption and is secure if the server gets corrupted, but the HSM is incorruptible and can be completely trusted (Lev-1). The second protocol, enhanced encrypt-to-HSM (Section 4.2), is a minor modification of the first protocol that provides additional protection of clear-text passwords and keys when the HSM leaks client account data (Lev-2). The third construction, OPRF-based PPKR (Section 4.3), is secure in the full corruption scenario (Lev-3), i.e., even when all files and all secret keys of the HSM are given to the adversary, an offline attack against the users' password is still necessary.

Like in [DFG+23b], we also assume client-to-server authenticated channels, e.g., through an SMS/email passcode authentication. This is merely to prevent an adversary from tricking an honest server into deleting accounts of honest users through repeated incorrect retrievals. Thus, this is used as a basic protection against deletion only, but not for actual user authentication — this is what the password is for. Importantly, however, the client-to-HSM channel is not authenticated, which is what enabled the attack from [DFG+23b].

Further, as we work in the UC-framework, all messages contain globally unique session- and subsession identifiers. This is a typical UC-artefact and there are standard ways to handle these ids when implementing the protocol in the real world, see [Can00, Sec. 3.3.2].

Modeling the HSM. All our protocols internally rely on the concept of an HSM (or similar trusted execution environments). We assume the HSM to attest every message, i.e., sign it under a public verification key that is known to all parties. Such verification key could be published on the HSM vendor's website, certified by some trusted authority.

In our model of leakage from the HSM, covered through Lev-2 security, HSM attestation is the *only* power kept from the adversary. Only in our strongest model, Lev-3 that considers full HSM corruption, we will provide the adversary with the attestation key. This distinction between the permanent attestation key and client-specific files is justified because attestation is a central capability of HSMs. Therefore, an HSM manufacturer might turn special attention to the protection of the single attestation key, such as more expensive hardware protection mechanisms.

For simplicity, we also assume that the HSM has some persistent memory where it stores file records. In practice, HSMs might securely outsource such storage as done, e.g. in the case of WhatsApp's solution [DFG+23b], but the actual realization of storage is irrelevant for our work. This HSM-protected storage is still considered secret upon server corruption (Lev-1) but entirely leaked to the adversary up from Lev-2.

When proving security in the UC-framework [Can01], we model the HSM as a subroutine of the server, similar to [PST17], instead of modeling it as an individual party. This captures the exclusive

[1]The corrupt client can remember the AKE secret keys from initialization and disregard the recovered keys, leading to a successful KE. In our protocol, this corresponds to remembering the original signing key.

access that the server has to the HSM. Concretely, the code of the HSM in Figure 3 and Figure 4 is the code of an entity that cannot be corrupted by the adversary (because it is not a protocol party) and that does not accept input from anybody else than the server. For the protocols that can handle certain types of HSM corruption, we equip our HSM functionalities with the respective corruption interfaces. Concretely, if we allow leakage of client files and all the protocol-specific long-term state, the HSM functionality will expose a LeakFile interface to the adversary that returns the client files. For the OPRF PPKR protocol, we add a FullyCorrupt interface to the HSM functionality that additionally gives the adversary the HSM's secret attestation key. A formal description of the respective HSM functionalities used by our protocols can be found in Figure 9 and Figure 10.

4.1 Lev-1 Protocol: Basic Encrypt-to-HSM

The security analysis of the WBP protocol of [DFG+23b] assumed that the HSM cannot be corrupted and never leaks information. A natural question is if this strong assumption allows for a simpler PPKR protocol. The basic encrypt-to-HSM protocol π^{encPw}, depicted in Figure 3, answers the question in the affirmative.

The central observation is that if no information is leaked by the HSM, then one can store the client's passwords and keys in the clear at the HSM. For recovery, the HSM is handed the clear-text password attempt and performs the comparison. To protect against network attackers and the potentially corrupt server, the client encrypts all messages to the HSM under a long-term encryption key $\mathsf{pk}_{\mathsf{Enc}}$. Like in WBP, all clients have access to $\mathsf{pk}_{\mathsf{Enc}}$, e.g., because the key is hard-coded into the client's source code.

To prevent the replay of messages by network attackers or corrupt servers, we let the client encrypt session identifier ssid and their own identity ID_C, and also include these items in the clear. By comparing the decrypted identifiers with the clear-text ones, the HSM will notice when a malicious server changes the identifiers of honest client requests, and abort. Hence, with this little check, π^{encPw} is stronger (on Lev-1) than the WBP, which admits such replay attacks [DFG+23b].

We show that π^{encPw} is a secure PPKR when only considering server corruptions, i.e., it provides Lev-1 security.

Theorem 1. *The protocol π^{encPw} from Figure 3 UC-realizes $\mathcal{F}_{\mathsf{PPKR}}$ in the $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$-hybrid model (i.e., assuming no corruption or leakage of the HSM), assuming HSM attested messages, and assuming that* PKE *is* IND-CCA *secure and* SE *is* IND-CPA *secure.*

More precisely, let $\mathbf{Dist}_{\mathcal{Z}}^{A,B}$ denote the advantage of $\mathcal{Z}$ to distinguish distributions A and B. We get that for every efficient real-world adversary $\mathcal{A}$ against π^{encPw}, there is an efficient simulator $\textsc{Sim}^{\mathsf{encPw}}$ (see Figures 11 to 12) that interacts with $\mathcal{F}_{\mathsf{PPKR}}$ such that for every efficient environment $\mathcal{Z}$ there exists an efficient adversaries $\mathcal{B}_1$ against the IND-CCA *security of* PKE *and $\mathcal{B}_2$ against the* IND-CPA *security of* SE *such that*

$$\begin{aligned}\mathbf{Dist}_{\mathcal{Z}}^{\pi^{\mathsf{encPw}},\{\mathcal{F}_{\mathsf{PPKR}},\textsc{Sim}^{\mathsf{encPw}}\}}(\lambda) \leq &(q_{\textsc{Init}} + q_{\textsc{Rec}})\mathbf{Adv}_{\mathsf{PKE},\mathcal{B}_1}^{\mathsf{IND\text{-}CCA}}(\lambda)\\ &+ q_{\textsc{Rec}}\mathbf{Adv}_{\mathsf{SE},\mathcal{B}_2}^{\mathsf{IND\text{-}CPA}}(\lambda)\\ &+ \mathbf{Adv}_{\mathsf{Sig},\mathcal{B}_4}^{\mathsf{sEUF\text{-}CMA}}(\lambda),\end{aligned}$$

where $q_{\textsc{Init}}$ is the number of initializations and $q_{\textsc{Rec}}$ is the number of recoveries.

Proof Sketch. The gist of the proof is that the incorruptible HSM holds the secret key $\mathsf{sk}_{\mathsf{Enc}}$ for all encryptions. As the HSM is modeled as a hybrid functionality, the simulator has access to the secret key. This allows for the following proof strategy:

To simulate *honest clients* without the password, the simulator can replace the PKE ciphertexts C by encryptions of $\perp$. The environment cannot detect this change by the IND-CCA security of the encryption scheme. Indeed, we require CCA security, as the HSM's Fail responses on mismatching ID_C and ssid give the adversary a very limited decryption oracle. The ciphertext C' is again replaced by an encryption of $\perp$. Note that this time we only need to reduce to IND-CPA security because the integrity of C' is ensured by the HSM attestation. Finally, the simulator can use $\mathcal{F}_{\mathsf{PPKR}}$ to provide clients with their correct output.

To extract the inputs of *corrupt clients* the simulator can use the secret key $\mathsf{sk}_{\mathsf{Enc}}$ of the HSM. The simulator knows this key because the HSM is modeled as a hybrid functionality. That means concretely that in the ideal-world execution, the simulator plays the role of the HSM (and thus, chooses $\mathsf{sk}_{\mathsf{Enc}}$ by itself). Using $\mathsf{sk}_{\mathsf{Enc}}$, the simulator can decrypt the PKE ciphertext C provided by the corrupt client and provide the used password pw^* to $\mathcal{F}_{\mathsf{PPKR}}$. If the password guess was correct the functionality will give the simulator the recovered backup key K. The simulator can also extract the corrupt client's symmetric key k_{sym} by using the secret key $\mathsf{sk}_{\mathsf{Enc}}$ of the HSM and respond to the client with an encrypted version of its backup key C'.

Simulating the server amounts to executing the rest of the protocol as usual because the server holds no private information. □

The full proof can be found in Appendix D.1.

4.2 Lev-2 Protocol: Enhanced Encrypt-to-HSM

When the HSM leaks permanently stored protocol data such as account information of clients, also called "password files", one cannot hope to prevent an adversary from, e.g., offline-attacking a PPKR protocol. That is because the file must contain enough information for the HSM to decide whether a recovery attempt is successful or not. Nonetheless, one can demand that an adversary still needs to guess a user's password and cannot read the password and/or the backup key immediately from the file. In other words, clients that choose very strong passwords should still have a certain level of security even if the HSM's files get leaked.

Clearly, the basic encrypt-to-HSM protocol from Section 4.1 falls short of this goal, and hence cannot reach Lev-2 security: The passwords and backup keys are stored in the clear and a compromise of the HSM's long-term state immediately gives them away. Further, the leakage includes all protocol-specific long-term state, i.e., the adversary also gets the HSM's decryption key $\mathsf{sk}_{\mathsf{Enc}}$, which leaks all the password (attempts) from sessions happening *after* the compromise to the attacker.

To reach Lev-2 security, we strengthen π^{encPw} in two ways: First, we store the password only in hashed form at the HSM, with a user-specific salt to prevent pre-computation attacks; and encrypt the backup key with another salted password hash. The salts are stored in the password file. Second, we let the HSM use ephemeral encryption keys for each session. Figure 3 including grayboxes and skipping dashed boxes shows our enhanced encrypt-to-HSM protocol $\pi^{\mathsf{encPw+}}$. We can formally prove in Theorem 2 that these relatively simple (although in the case of the encryption keys significantly more expensive) measures are enough to restrict an adversary in Lev-2 back to offline guessing, after getting access to the long-term storage of the HSM.

We stress that our HSM leakage model at Lev-2 returns all permanently stored user files and protocol-specific *long-term* state to the adversary, but not the temporary values that occur during execution of the protocol.

Theorem 2. *The protocol $\pi^{\mathsf{encPw+}}$ from Figure 3 UC-realizes $\mathcal{F}_{\mathsf{PPKR}}$ in the $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$-hybrid model (i.e., assuming no corruption but state leakage of the HSM) and assuming that H is modeled as random oracle, messages are HSM-attested,* PKE *is* IND-CCA *secure, and* SE *is* IND-CPA *secure.*

More precisely, let $\mathbf{Dist}_{\mathcal{Z}}^{A,B}$ denote the advantage of $\mathcal{Z}$ to distinguish distributions A and B. We get that for every efficient real-world adversary $\mathcal{A}$ against $\pi^{\mathsf{encPw+}}$, there is an efficient simulator $\textsc{Sim}^{\mathsf{encPw+}}$ (see Figure 13) that interacts with $\mathcal{F}_{\mathsf{PPKR}}$ such that for every efficient environment $\mathcal{Z}$ there exist efficient adversaries $\mathcal{B}_1$ against the IND-CCA *security of* PKE *and $\mathcal{B}_2$ against the* IND-CPA *security of* SE *such that*

$$\mathbf{Dist}_{\mathcal{Z}}^{\pi^{\mathsf{encPw+}},\{\mathcal{F}_{\mathsf{PPKR}},\textsc{Sim}^{\mathsf{encPw+}}\}}(\lambda) \leq (q_{\textsc{Init}} + q_{\textsc{Rec}})\mathbf{Adv}_{\mathsf{PKE},\mathcal{B}_1}^{\mathsf{IND\text{-}CCA}}(\lambda)$$
$$+ q_{\textsc{Rec}}\mathbf{Adv}_{\mathsf{SE},\mathcal{B}_2}^{\mathsf{IND\text{-}CPA}}(\lambda) + \mathbf{Adv}_{\mathsf{Sig},\mathcal{B}_4}^{\mathsf{sEUF\text{-}CMA}}$$
$$+ \frac{q_{\textsc{Init}}(2q_{\textsc{Init}} - 1)}{2^\lambda} + \frac{2q_{\textsc{Init}}q_H}{2^\lambda},$$

where $q_{\textsc{Init}}$ is the number of initializations, $q_{\textsc{Rec}}$ is the number of recoveries, and q_H is the number of H queries.

Proof Sketch. Because the formal proof of Theorem 2 shares most of its steps with the proof of Theorem 1, we focus on the changes to the proof that are required to also achieve security under leaked HSM files. We show the modified and additional interfaces that the simulator provides in Figure 13. Now we have to simulate leakage of HSM files, because $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$ allows the adversary to call its LeakFile interface. To that end, the simulator must observe and program the random oracle H and use the OfflineAttack interface of $\mathcal{F}_{\mathsf{PPKR}}$. The simulator now also has to generate fresh encryption keys $(\mathsf{pk}_{\mathsf{Enc}}, \mathsf{sk}_{\mathsf{Enc}})$ for every initialization and recovery and store them until they are used. Note that we can keep the rest of the simulation as in the proof of Theorem 1. That is because the leaked client file does not contain values that would allow impersonation of the HSM (as will be the case when we consider full corruption in Section 4.3).

Before the state of the HSM is leaked, an adversary has only a negligible chance to query $H(s_2, \mathsf{pw})$ to the random oracle. That is because s_2 has high entropy. Therefore, the simulator can store a uniformly random value c as the encoding of the backup key. But once the HSM state is leaked, the adversary knows s_1, s_2 and h and thus, can guess passwords and verify its guess using h and s_1. If one of the guesses is correct, the adversary will be able to check if c is indeed an encoding of K by using s_2. The simulator can use the OfflineAttack interface of $\mathcal{F}_{\mathsf{PPKR}}$ to see if the adversary's guess was correct. In case of a successful guess, the simulator learns K. Then, it can program $H(s_2, \mathsf{pw}) \leftarrow c \oplus K$ to equivocate c after the fact, and similarly Sim can program $H(s_1, \mathsf{pw}) \leftarrow h$. □

The full proof can be found in Appendix D.2.

Missing Lev-3 *Security of $\pi^{\mathsf{encPw+}}$.* Cleary, our enhanced encrypt-to-HSM protocol cannot satisfy Lev-3 security: if the adversary gets the HSM's attestation key, it has full control over the public keys for the encryption scheme under which the client encrypts the user's password. Thus, for all users that still use the PPKR service after the full compromise happened, the adversary immediately learns their plaintext passwords from the request – and consequently can recover their keys too.

To achieve Lev-3 security, we need to securely communicate some password-dependent data to a possibly entirely malicious party for authentication, and cannot rely on any server-held secret for the password's protection. This requires more advanced cryptographic techniques, for which we revert to an oblivious pseudorandom function in our third construction.

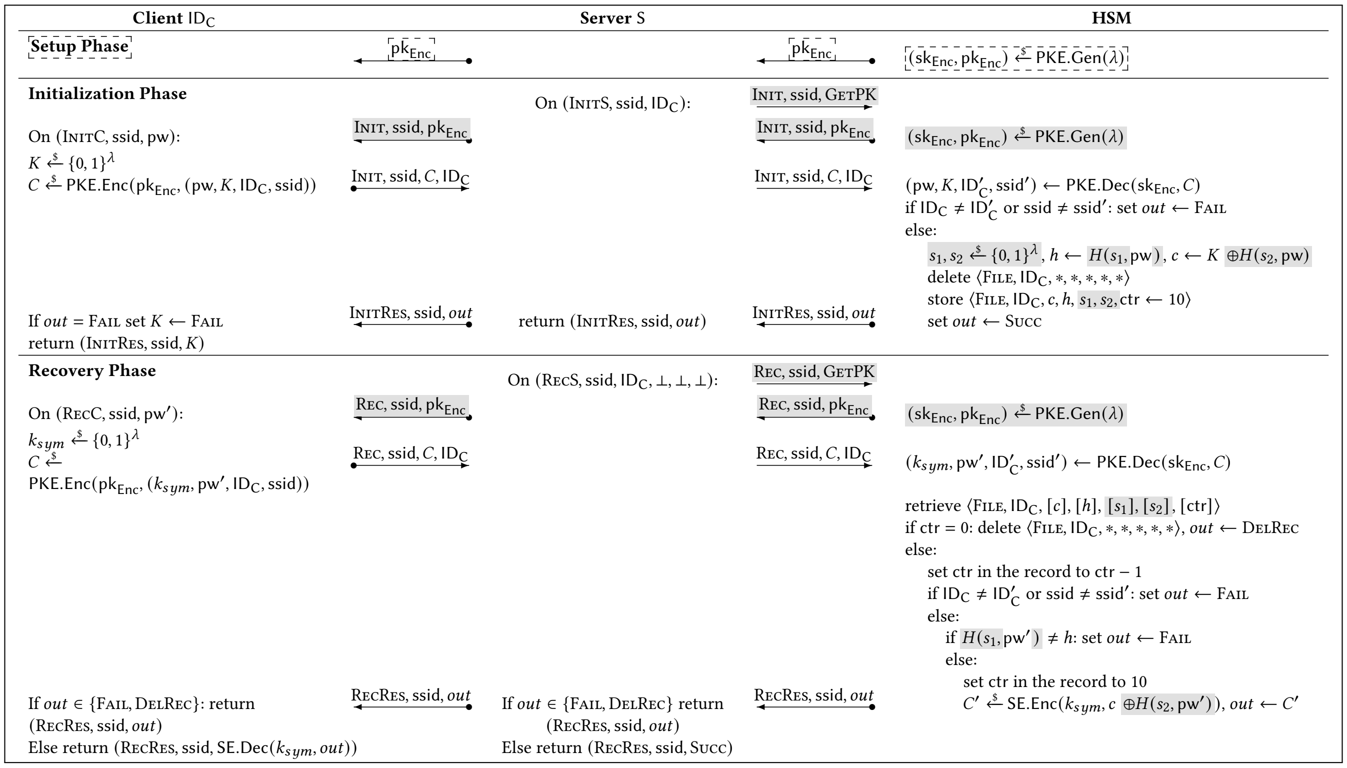
4.3 Lev-3 Protocol: OPRF-Based PPKR

In this section, we build PPKR that provides protection against full server and HSM corruption. That is, even when all keys and files are leaked, the best an adversary can do is an offline attack against each user's password. Our goal is to propose a simpler protocol than WBP that reaches such Lev-3 security.

Oblivious Pseudorandom Function. At the core of our construction is an oblivious pseudorandom function (OPRF). Such a function allows to deterministically compute $y = \mathsf{PRF}(k, x)$ through an interactive protocol between an evaluator and requester. The evaluator knows the PRF key k, but learns nothing about the input x or output y it computes. We will use such an OPRF to deterministically compute password-dependent key material, which allows the client to authenticate towards the HSM, and also derive the encryption key under which K gets wrapped.

High-Level Idea of $\pi^{\mathsf{OPRF\text{-}PPKR}}$. For initialization, the client executes the OPRF with the HSM (through the server) to obtain a value $\rho = \mathsf{PRF}(k_{\mathsf{OPRF}}, (\mathsf{pw}, \mathsf{ID_C}))$, depending on the user's password and $\mathsf{ID_C}$. The HSM chooses a client-specific key k_{OPRF}. The derived ρ then serves as the key for an authenticated encryption (AE) scheme, under which the client's randomly chosen key K gets encrypted. The client also generates a signature key pair $(\mathsf{sk}_C, \mathsf{pk}_C)$ for future re-authentication. It encrypts both sk_C and K under ρ, obtaining a ciphertext c. The HSM gets c and pk_C and stores them with $\mathsf{ID_C}$, k_{OPRF}, and a counter value ctr, initially set to 10, forming the password file for $\mathsf{ID_C}$.

For recovery, the client receives c from the HSM and runs the OPRF again. If the client uses the same password as in initialization, it obtains the same ρ as earlier. The client then uses ρ to decrypt c, obtaining (K, sk_C). Now the client can prove knowledge of the right password towards the HSM, by using the sk_C to sign the transcript of previous messages exchanged with the HSM. The HSM decreases the counter ctr at the beginning of every recovery, and if it receives such a signature that verifies under the pk_C in the stored user file, it considers the client as correctly authenticated and resets the ctr value to 10 again.



```
Client ID_C | Server S | HSM

Setup Phase
  HSM: (sk_Enc, pk_Enc) <-$ PKE.Gen(λ)
  HSM -> Server -> Client: pk_Enc

Initialization Phase
  Server: On (InitS, ssid, ID_C):
    Server -> HSM: Init, ssid, GetPK
  HSM: (sk_Enc, pk_Enc) <-$ PKE.Gen(λ)
    HSM -> Server -> Client: Init, ssid, pk_Enc
  Client: On (InitC, ssid, pw):
    K <-$ {0,1}^λ
    C <-$ PKE.Enc(pk_Enc, (pw, K, ID_C, ssid))
    Client -> Server -> HSM: Init, ssid, C, ID_C
  HSM:
    (pw, K, ID'_C, ssid') <- PKE.Dec(sk_Enc, C)
    if ID_C ≠ ID'_C or ssid ≠ ssid': set out <- Fail
    else:
        s_1, s_2 <-$ {0,1}^λ, h <- H(s_1, pw), c <- K ⊕ H(s_2, pw)
        delete ⟨File, ID_C, *, *, *, *, *⟩
        store ⟨File, ID_C, c, h, s_1, s_2, ctr <- 10⟩
        set out <- Succ
    HSM -> Server: InitRes, ssid, out
  Server: return (InitRes, ssid, out)
    Server -> Client: InitRes, ssid, out
  Client:
    If out = Fail set K <- Fail
    return (InitRes, ssid, K)

Recovery Phase
  Server: On (RecS, ssid, ID_C, ⊥, ⊥, ⊥):
    Server -> HSM: Rec, ssid, GetPK
  HSM: (sk_Enc, pk_Enc) <-$ PKE.Gen(λ)
    HSM -> Server -> Client: Rec, ssid, pk_Enc
  Client: On (RecC, ssid, pw'):
    k_sym <-$ {0,1}^λ
    C <-$
    PKE.Enc(pk_Enc, (k_sym, pw', ID_C, ssid))
    Client -> Server -> HSM: Rec, ssid, C, ID_C
  HSM:
    (k_sym, pw', ID'_C, ssid') <- PKE.Dec(sk_Enc, C)
    retrieve ⟨File, ID_C, [c], [h], [s_1], [s_2], [ctr]⟩
    if ctr = 0: delete ⟨File, ID_C, *, *, *, *, *⟩, out <- DelRec
    else:
        set ctr in the record to ctr − 1
        if ID_C ≠ ID'_C or ssid ≠ ssid': set out <- Fail
        else:
            if H(s_1, pw') ≠ h: set out <- Fail
            else:
                set ctr in the record to 10
                C' <-$ SE.Enc(k_sym, c ⊕ H(s_2, pw')), out <- C'
    HSM -> Server: RecRes, ssid, out
  Server:
    If out ∈ {Fail, DelRec} return
        (RecRes, ssid, out)
    Else return (RecRes, ssid, Succ)
    Server -> Client: RecRes, ssid, out
  Client:
    If out ∈ {Fail, DelRec}: return
    (RecRes, ssid, out)
    Else return (RecRes, ssid, SE.Dec(k_sym, out))
```

Figure 3: Protocols π^{encPw} and $\pi^{\mathsf{encPw+}}$. The code in dashbox is only executed in π^{encPw}, the code in gray boxes is only executed in $\pi^{\mathsf{encPw+}}$. $\xleftarrow{x}\bullet$ is the HSM-attested transmission of x. $\bullet\xrightarrow{y}$ is the client-to-server authenticated transmission of y. We implicitly assume that each message contains sid and the random oracle H takes sid as first input. $\mathsf{ID_C}$ and S output (InitRes, ssid, Fail) or (RecRes, ssid, Fail), whenever they receive an unexpected message (i.e. with mismatching ssid or $\mathsf{ID_C}$) or when attestation verification fails. If any party receives the same type of message twice with the same ssid, it ignores the second one. `Init` **and** `Rec` **are strings indicating which phase the client wishes to initiate. The server receives the input (RecS, ssid, $\mathsf{ID_C}$, $\perp, \perp, \perp$) instead of just (RecS, ssid, $\mathsf{ID_C}$) only for technical reasons as the syntax needs to match $\mathcal{F}_{\mathsf{PPKR}}$.**

Interestingly, we still need a fresh encryption key pair for communication towards the HSM – but only for the initialization. Therein, the client's values (pk_C, c) and the ssid get encrypted under the freshly chosen key. The purpose of this encryption is to prevent a malicious server from combining the ciphertext c of an honest user with a different public key pk_C. Such a mixing attack would allow the server to plant the honest c with some authentication information where it could easily circumvent the strict limit of failed retrievals imposed by the HSM. Thus, the purpose of the public-key encryption here is to bind the honest user's information together.

Comparison to WBP. Our protocol shares a lot of similarities with the WBP as stated by Davies et al. [DFG+23b], but simplifies the design enabling both a more efficient and more secure variant. We start by sketching the WBP core ideas and then explain the main differences to our protocol.

The overall idea of the WBP is to let the client and the HSM execute an *asymmetric PAKE* (aPAKE) protocol to exchange a fresh symmetric key K^{session} from a clear-text password of the client, and a password file stored by the HSM. K^{session} is subsequently used to prove the correctness of the client's password to the HSM using standard key confirmation techniques, to reset the file counter. The client's key K is encrypted under a static (i.e., depending only on the password) key K^{export} and stored by the HSM. The protocol is instantiated with OPAQUE [JKX18], which in turn crucially relies on an OPRF to produce the static export key.

In our protocol, we start from an OPRF instead of aPAKE, and implement the proof of password knowledge separately through digital signatures. This yields a simpler protocol layout and cleaner security proof. In terms of similarity, the AE ciphertext stored in the client's password files corresponds to the OPAQUE password files, and the OPRF is used to deterministically derive a key to decrypt this file and perform the re-authentication. How this authentication is done differs though: we save one message by using signatures instead of authenticated key exchange. Note that this exploits that we are not aiming at the key exchange, which was the goal of OPAQUE – and thus is more than what is needed for PPKR.

Apart from improving efficiency, our protocol provides better security then WBP. While the following attacks can all be considered minor, they are still in conflict with the desired security properties – and easily preventable as shown by our protocol. The security improvements are as follows:

(1) In the WBP, a corrupt server can prevent old password files of honest clients from being overwritten upon a client re-initializing with a new password. The corrupt server could then still use up all remaining password guesses against old password files, to recover previous keys of the client. We prevent this by having the HSM attest the client identity $\mathsf{ID_C}$ for each session ssid, and letting the client abort when it receives a mismatching $\mathsf{ID_C}$. This enforces an agreement between an honest client and honest HSM when the server is corrupt (needed for Lev-1 security).
(2) We fix a "rerouting" attack on the WBP that was discovered by [DFG+23b], and which allows a corrupt server to reroute honest recovery attempts to wrong password files. This results in honest Alice recovering Bob's key if both use the same password. We therefore invoke the OPRF not only on the user's password but also append the unique client identity $\mathsf{ID_C}$ to the input. This simple measure ensures that users derive unique wrapping keys ρ, which protects against this attack (again needed for Lev-1 security).
(3) The above two measures additionally prevent two attacks on the WBP that a fully corrupt server in the Lev-3 setting [2] can mount: (I) checking whether two honest clients use the same password, (II) resetting the counter in an honest client's password file if another client having the same password runs a recovery. Both attacks work by routing a recovery attempt of Alice to Bob's password file, which in the WBP goes unnoticed by Alice, and can succeed because the passwords of honest clients are not enforced to be unique. Both is prevented through measures (1) and (2).
(4) Considering the same HSM leakage model that we use (everything except the HSM's attestation key gets leaked on Lev-2), in the WBP, a malicious server can run offline password guessing attack not only against all leaked files but also against files that are created *after* the compromise. This is because the WBP relies on a long-term encryption key at the HSM for initialization. After a Lev-2 compromise, the malicious server can decrypt an honest user's initialization request, and re-encrypt the user's wrapped key K together with a maliciously chosen (AKE) public key. This allows the malicious server to get unlimited password guesses against the user's real password-wrapped key, as it can correctly complete key confirmation towards the HSM, even when performing the retrieval with the wrong passwords. Our protocol prevents that attack by using fresh encryption keys in initialization, achieving the necessary security for Lev-2.

Concrete OPRF for Efficiency. While our protocol can be securely realized with any 2-round OPRF, we build $\pi^{\mathsf{OPRF\text{-}PPKR}}$ from the 2HashDH OPRF [JKKX16] in a non-black-box way. This was mainly done for efficiency reasons. A generic approach relying on an ideal OPRF functionality would not allow binding the OPRF in- and outputs directly to other protocol values, nor do this efficiently in an HSM-attested way. However, even though our protocol is non-generic, our proof actually provides some modularity: we use the 2HashDH simulator as a step in our proof, and from then on rely on the abstracted security properties of a UC-secure OPRF. Nevertheless, our proof additionally relies on the internals of the 2HashDH simulator at several points, making the switch to another OPRF non-trivial.

Theorem 3. *The protocol $\pi^{\mathsf{OPRF\text{-}PPKR}}$ from Figure 4 UC-realizes $\mathcal{F}_{\mathsf{PPKR}}$ in the $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{OPRF\text{-}PPKR}}$-hybrid model (i.e., allowing file leakage and full corruption of the server), assuming adaptive corruptions outside of ongoing initialization and recovery phases, HSM-attested messages, client-to-server authenticated channels, that* AE *is* equivocable *and* random-key robust *and has* $\mathsf{INT\text{-}CTXT}$*-security,* Sig *is* $\mathsf{sEUF\text{-}CMA}$*-secure,* PKE *is* CCA*-secure, H_1 and H_2 are modeled as random oracles, and that the (N, Q)-*OMDH *assumption holds in $\mathbb{G}$.*

More precisely, let $\mathbf{Dist}_{\mathcal{Z}}^{A,B}$ denote the advantage of $\mathcal{Z}$ to distinguish distributions A and B. Then, for every efficient real-world adversary $\mathcal{A}$ against $\pi^{\mathsf{OPRF\text{-}PPKR}}$, there is an efficient simulator $\mathcal{S}\text{IM}^{\mathsf{OPRF\text{-}PPKR}}$ (see Figures 14 to 17) that interacts with $\mathcal{F}_{\mathsf{PPKR}}$ such that for every efficient environment $\mathcal{Z}$ there exists efficient adversaries $\mathcal{B}_1, \ldots, \mathcal{B}_7$ such that

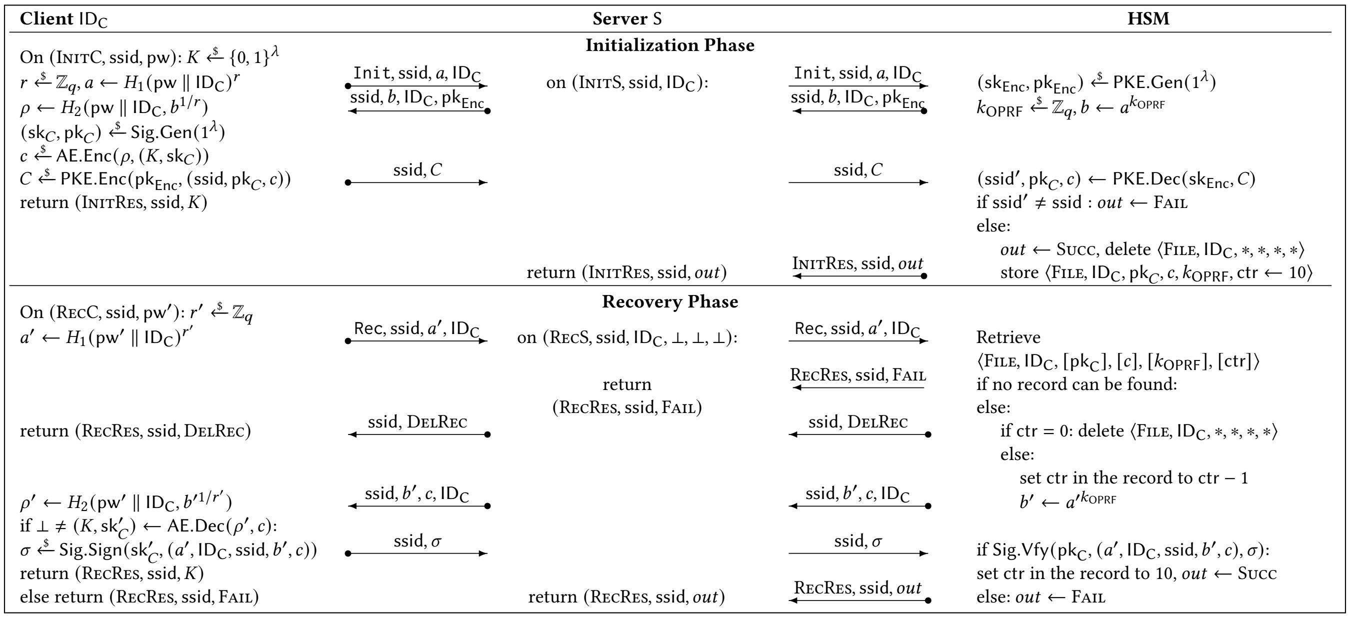
$$\mathbf{Dist}_{\mathcal{Z}}^{\pi^{\mathsf{OPRF\text{-}PPKR}},\{\mathcal{F}_{\mathsf{PPKR}},\mathcal{S}\text{IM}^{\mathsf{OPRF\text{-}PPKR}}\}}(\lambda) \leq \mathbf{Adv}_{\mathcal{B}_1,\mathsf{Sig}}^{\mathsf{sEUF\text{-}CMA}}(\lambda)$$
$$q_{\mathit{INIT}}\mathbf{Adv}_{\mathcal{B}_2,\mathbb{G}}^{(q_E+q_H,q_E)-\mathsf{OMDH}}(\lambda) + q_E^2\mathbf{Adv}_{\mathcal{B}_3,\mathsf{AE}}^{\mathsf{rkr}}(\lambda)$$
$$+ q_{\mathit{INIT}}\mathbf{Adv}_{\mathcal{B}_4,\mathsf{AE}}^{\mathsf{INT\text{-}CTXT}}(\lambda) + \mathbf{Adv}_{\mathcal{B}_5,\mathsf{AE}}^{\mathsf{EQV}}(\lambda) + q_{\mathit{INIT}}\mathbf{Adv}_{\mathcal{B}_6,\mathsf{Sig}}^{\mathsf{sEUF\text{-}CMA}}(\lambda)$$
$$+ q_{\mathit{INIT}}\mathbf{Adv}_{\mathcal{B}_7,\mathsf{PKE}}^{\mathsf{IND\text{-}CCA}}(\lambda) + (q_E + q_H)^2/q$$

where q_{INIT} is the number of initializations, q_{REC} is the number of recoveries, $q_E = q_{\mathit{INIT}} + q_{\mathit{REC}}$, q_H is the number of queries to H_1 and q is the order of $\mathbb{G}$.

Proof Sketch. The proof heavily relies on the security of the 2HashDH OPRF. We use the simulator $\mathrm{Sim}_{\mathsf{OPRF}}$ from Davies et al. [DFG+23b] (restated in Appendix B), which demonstrates that 2HashDH UC-realizes $\mathcal{F}_{\mathsf{OPRF}}$, in a non-black-box way throughout the proof. Whenever we have to simulate a message a, b, a', or b' for some honest party, we "outsource" the simulation to $\mathrm{Sim}_{\mathsf{OPRF}}$. This also means that we let $\mathrm{Sim}_{\mathsf{OPRF}}$ choose the key k_{OPRF}, which we then need to obtain from $\mathrm{Sim}_{\mathsf{OPRF}}$ whenever there is a LeakFile query from $\mathcal{A}$. Furthermore, from the random oracle queries to H_2 by $\mathcal{A}$, $\mathrm{Sim}_{\mathsf{OPRF}}$ is able to extract the password chosen by a corrupt party in an initialization, which allows us to install a corresponding file in $\mathcal{F}_{\mathsf{PPKR}}$.

A challenge that arises from allowing the LeakFile query is that $\mathcal{A}$ can do offline password guessing using k_{OPRF} obtained from leaked files. If $\mathcal{A}$ guesses the correct password pw of some $\mathsf{ID_C}$, it can decrypt the ciphertext c that was encrypted under the output ρ of the OPRF. Hence, c has to be simulated such that it decrypts to the key K chosen randomly by $\mathcal{F}_{\mathsf{PPKR}}$ for $\mathsf{ID_C}$, as otherwise, the simulation would be distinguishable from the real protocol. However, when we simulate c in the initialization phase, K is unknown, as $\mathcal{F}_{\mathsf{PPKR}}$ has not even given it to $\mathsf{ID_C}$, yet. To solve this, we require AE to be equivocable, which allows us to produce a simulated c and only later decide to which values c decrypts. More precisely, whenever $\mathcal{A}$ queries H_2 on an input of the form $(\mathsf{pw}' \,\|\, \mathsf{ID_C}, H_1(\mathsf{pw}' \,\|\, \mathsf{ID_C})^{k_{\mathsf{OPRF}}})$, where k_{OPRF} is some OPRF key chosen by $\mathrm{Sim}_{\mathsf{OPRF}}$, we submit pw' to the OfflineAttack interface of $\mathcal{F}_{\mathsf{PPKR}}$. If $\mathsf{pw} = \mathsf{pw}'$, then we obtains the key K and can equivocate c to obtain a ρ such that c decrypts to K under ρ and program the output of H_2 to ρ.

[2] We note that the fully corrupt server case was not analyzed in [DFG+23b], and hence they did not claim the protocol to prevent these attacks. It is nonetheless easy to verify that a fully corrupt server in the WBP can use the same OPRF key for Alice and Bob and perform these attacks.



Client $\mathsf{ID_C}$		**Server** S		**HSM**
		Initialization Phase		

On ($\textsc{InitC}$, ssid, pw): $K \xleftarrow{\$} \{0,1\}^\lambda$
$r \xleftarrow{\$} \mathbb{Z}_q, a \leftarrow H_1(\mathsf{pw} \parallel \mathsf{ID_C})^r$
$\rho \leftarrow H_2(\mathsf{pw} \parallel \mathsf{ID_C}, b^{1/r})$
$(\mathsf{sk}_C, \mathsf{pk}_C) \xleftarrow{\$} \mathsf{Sig.Gen}(1^\lambda)$
$c \xleftarrow{\$} \mathsf{AE.Enc}(\rho, (K, \mathsf{sk}_C))$
$C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, (\mathsf{ssid}, \mathsf{pk}_C, c))$
return ($\textsc{InitRes}$, ssid, K)
	$\bullet\!\xrightarrow{\mathtt{Init}, \mathsf{ssid}, a, \mathsf{ID_C}}$
	$\xleftarrow{\mathsf{ssid}, b, \mathsf{ID_C}, \mathsf{pk_{Enc}}}\!\bullet$
	$\bullet\!\xrightarrow{\mathsf{ssid}, C}$
		on ($\textsc{InitS}$, ssid, $\mathsf{ID_C}$):
		
		return ($\textsc{InitRes}$, ssid, out)
			$\xrightarrow{\mathtt{Init}, \mathsf{ssid}, a, \mathsf{ID_C}}$
			$\xleftarrow{\mathsf{ssid}, b, \mathsf{ID_C}, \mathsf{pk_{Enc}}}\!\bullet$
			$\xrightarrow{\mathsf{ssid}, C}$
			$\xleftarrow{\textsc{InitRes}, \mathsf{ssid}, out}\!\bullet$
				$(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \xleftarrow{\$} \mathsf{PKE.Gen}(1^\lambda)$
				$k_{\mathsf{OPRF}} \xleftarrow{\$} \mathbb{Z}_q, b \leftarrow a^{k_{\mathsf{OPRF}}}$
				$(\mathsf{ssid}', \mathsf{pk}_C, c) \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{Enc}}, C)$
				if ssid′ ≠ ssid : $out \leftarrow \textsc{Fail}$
				else:
				    $out \leftarrow \textsc{Succ}$, delete $\langle \textsc{File}, \mathsf{ID_C}, *, *, *, * \rangle$
				    store $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{pk}_C, c, k_{\mathsf{OPRF}}, \mathsf{ctr} \leftarrow 10 \rangle$

		Recovery Phase		

On ($\textsc{RecC}$, ssid, pw′): $r' \xleftarrow{\$} \mathbb{Z}_q$
$a' \leftarrow H_1(\mathsf{pw}' \parallel \mathsf{ID_C})^{r'}$
	$\bullet\!\xrightarrow{\mathsf{Rec}, \mathsf{ssid}, a', \mathsf{ID_C}}$	on ($\textsc{RecS}$, ssid, $\mathsf{ID_C}$, ⊥, ⊥, ⊥):
		
		return
		($\textsc{RecRes}$, ssid, $\textsc{Fail}$)
			$\xrightarrow{\mathsf{Rec}, \mathsf{ssid}, a', \mathsf{ID_C}}$
			$\xleftarrow{\textsc{RecRes}, \mathsf{ssid}, \textsc{Fail}}$
				Retrieve
				$\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{pk_C}], [c], [k_{\mathsf{OPRF}}], [\mathsf{ctr}] \rangle$
				if no record can be found:
				else:

return ($\textsc{RecRes}$, ssid, $\textsc{DelRec}$)	$\xleftarrow{\mathsf{ssid}, \textsc{DelRec}}\!\bullet$		$\xleftarrow{\mathsf{ssid}, \textsc{DelRec}}\!\bullet$	if ctr = 0: delete $\langle \textsc{File}, \mathsf{ID_C}, *, *, *, * \rangle$
				    else:
				        set ctr in the record to ctr − 1
				        $b' \leftarrow a'^{k_{\mathsf{OPRF}}}$

$\rho' \leftarrow H_2(\mathsf{pw}' \parallel \mathsf{ID_C}, b'^{1/r'})$
if $\perp \neq (K, \mathsf{sk}'_C) \leftarrow \mathsf{AE.Dec}(\rho', c)$:
$\sigma \xleftarrow{\$} \mathsf{Sig.Sign}(\mathsf{sk}'_C, (a', \mathsf{ID_C}, \mathsf{ssid}, b', c))$
return ($\textsc{RecRes}$, ssid, K)
else return ($\textsc{RecRes}$, ssid, $\textsc{Fail}$)
	$\xleftarrow{\mathsf{ssid}, b', c, \mathsf{ID_C}}\!\bullet$
	$\bullet\!\xrightarrow{\mathsf{ssid}, \sigma}$
		return ($\textsc{RecRes}$, ssid, out)	$\xleftarrow{\mathsf{ssid}, b', c, \mathsf{ID_C}}\!\bullet$
			$\xrightarrow{\mathsf{ssid}, \sigma}$
			$\xleftarrow{\textsc{RecRes}, \mathsf{ssid}, out}\!\bullet$
				if $\mathsf{Sig.Vfy}(\mathsf{pk_C}, (a', \mathsf{ID_C}, \mathsf{ssid}, b', c), \sigma)$:
				set ctr in the record to 10, $out \leftarrow \textsc{Succ}$
				else: $out \leftarrow \textsc{Fail}$

Figure 4: Protocol $\pi^{\mathsf{OPRF\text{-}PPKR}}$. $\xleftarrow{x}\!\bullet$ indicates that the message x is signed by the HSM; and $\bullet\!\xrightarrow{y}$ denotes the client-to-server authenticated transmission of y. $\mathsf{ID_C}$ outputs ($\textsc{InitRes}$, ssid, $\textsc{Fail}$) or ($\textsc{RecRes}$, ssid, $\textsc{Fail}$), whenever it receives an unexpected message (i.e. with mismatching ssid or $\mathsf{ID_C}$) or when signature verification fails for the received signed message. If any party receives the same type of message twice with the same ssid it ignores the second one. `Init` and `Rec` are strings indicating which phase the client wishes to initiate. The server receives the input ($\textsc{RecS}$, ssid, $\mathsf{ID_C}$, ⊥, ⊥, ⊥) instead of just ($\textsc{RecS}$, ssid, $\mathsf{ID_C}$) only for technical reasons as the syntax needs to match $\mathcal{F}_{\mathsf{PPKR}}$.

A similar challenge arises when S is fully corrupt. Then, S can make the critical query to H_2 even before we simulate c as it chooses the OPRF key k_{OPRF} used in the initialization. Thus, we have to check for all previous H_2 queries whether it was this critical query via the $\textsc{OfflineAttack}$ interface and if so, encrypt K under the corresponding output of H_2 instead of outputting an equivocable c.

The last major challenge is that a fully corrupt S can do key-planting attacks and can mix-and-match different files during a recovery, e.g., when $\mathsf{ID_C}$ recovers, S could use k_{OPRF} of some $\mathsf{ID}'_\mathsf{C} \neq \mathsf{ID_C}$ and an adversarial c. To deal with this, we check whether the password $\mathsf{pw}'' \parallel \mathsf{ID}''_\mathsf{C}$ and key k'_{oprf} used to derive the ρ, under which the adversarial c was encrypted, are the same as the password $\mathsf{pw} \parallel \mathsf{ID_C}$ and key k_{OPRF} used in the recovery by $\mathsf{ID_C}$. In the real world, $\mathsf{ID_C}$ would then output the key K decrypted from c. To simulate this, we use the key-planting capabilities in the $\textsc{RecS}$ interface. We can again find the password $\mathsf{pw}'' \| \mathsf{ID}''_\mathsf{C}$ used to derive ρ via the H_2 queries by $\mathcal{A}$ and check whether the same OPRF key was used with the help of $\textsc{Sim}_{\mathsf{OPRF}}$. We can then decrypt c to get K'' and submit pw'' and K'' to the $\textsc{RecS}$ interface. If $\mathsf{pw} = \mathsf{pw}''$, $\mathsf{ID_C}$ then outputs K''. We can proceed similarly if c instead is equivocable. □

The full proof can be found in Appendix D.3.

5 Evaluation & Discussion

In this section, we give an overview of the concrete efficiency of our protocols, compare them to WBP, and also discuss the respective advantages of our protocols.

Instantiations of Building Blocks. For the efficiency overview, we choose concrete instantiations of the required primitives, such that we can count the number of operations performed in each protocol. We only chose group-based public key primitives to keep the numbers comparable. But of course, one could use any other secure instantiation of the primitives, e.g., based on RSA or lattices.

Concretely, we chose Schnorr-Signatures, HMAC, HKDF, ElGamal encryption as a CPA secure encryption and DHIES as CCA secure encryption. For simplicity, we assumed that all hash evaluations cost a uniform unit "1 Hash"[3] and similarly that one AE encryption or decryption costs "1 AES". A detailed overview of the computation costs of each primitive can be found in Table 4. We did not list the HSM's attestation signatures as they are automatically produced by the HSM anyway.

Efficiency Comparison. As Table 3 shows, OPRF-PPKR is more efficient than the WBP in both, initialization and recovery. This comes mostly from WBP performing an authenticated key exchange where OPRF-PPKR uses a digital signature. In particular, in the recovery phase, which will be the more time-critical phase in deployment,

[3]Ignoring e.g., exponentiations that might be needed to hash into a group.

Table 3: Efficiency of PPKR realizations expressed in terms of the number of exponentiations (Exp), multiplications (Mult), hash evaluations (Hash) and AES encryptions/decryptions (AES). Since the server mostly just relays messages, we ignore its costs in the comparison. In case of encPw and WBP, we assume that the static encryption key of the HSM is hardcoded into the client, therefore no communication is needed for key sharing in Init and Rec and we neglect the costs of the one-time generation. For more details on the instantiations of the primitives used for the comparison, see Table 4.

		encPw **(Sec. 4.1)**	encPw+ **(Sec. 4.2)**	OPRF-PPKR **(Sec. 4.3)**	**WBP [DFG+23b]**
Init	Client	2 Exp, 3 Hash, 1 AES	2 Exp, 3 Hash, 1 AES	5 Exp, 5 Hash, 2 AES	7 Exp, 12 Hash, 1 AES, 1 Mult
	HSM	3 Hash, 1 AES	2 Exp, 5 Hash, 1 AES	2 Exp, 3 Hash, 1 AES	3 Exp, 3 Hash
	no. rounds	2	3	3	3
Rec	Client	2 Exp, 3 Hash, 2 AES	2 Exp, 3 Hash, 2 AES	3 Exp, 3 Hash, 1 AES	8 Exp, 27 Hash, 2 AES, 1 Mult
	HSM	3 Hash, 2 AES	2 Exp, 5 Hash, 2 AES	2 Exp, 1 Hash, 1 Mult	6 Exp, 15 Hash, 1 AES
	no. rounds	2	3	3	4

Table 4: Costs of concrete building blocks for the efficiency evaluation in Section 5.

	KeyGen	Enc	Dec
CPA Enc (ElGamal)	1 Exp	2 Exp, 1 Hash	1 Exp, 1 Hash
CCA Enc (DHIES)	1 Exp	2 Exp, 3 Hash, 1 AES	1 Exp, 3 Hash, 1 AES
AE (AES-GCM)	-	1 AES	1 AES
	KeyGen	**Sign**	**Vfy**
Signature (Schnorr)	1 Exp	1 Exp,1 Hash	2 Exp, 1 Mult, 1 Hash
MAC (HMAC)	-	2 Hash	2 Hash
KDF (n keys, HKDF)	$(2n + 2)$ Hash		

as it will be run more often than initialization, our protocol outperforms WBP: OPRF-PPKR reduces the round[4] complexity from 4 to 3, and uses roughly one-third of the operations required by WBP, for both the client and HSM.

If we compare the two enhanced encrypt-to-HSM protocols to OPRF-PPKR, they are more efficient and they save one round of communication, as they strongly rely on the HSM security, with the weakest protocol having the lowest computational requirements. Interestingly, in bare numbers, they are not significantly more efficient though.

Thus, considering that OPRF-PPKR provides much better security for almost the same costs, this might raise the question of whether there are any advantages in using our simpler protocols – which is what we answer next.

Advantages of Standard Primitives. The core benefit of our two basic/enhanced encrypt-to-HSM protocols is that they explore how the extended trust in the HSM can be traded for simplicity in the protocol design. Both protocols rely on standard and well-understood primitives only, which are public-key and symmetric encryption, and hash functions. In contrast, WBP and our OPRF-PPKR require (dedicated discrete-log-based) OPRFs.

In particular, when internally using an HSM, reliance on simple and standard building blocks is an advantage – established primitives are implemented in many well-tested frameworks and have been studied for resistance against side-channel attacks. OPRFs are still a somewhat more modern primitive that just recently came into focus of practitioners, i.e., it might require developers to implement low-level cryptographic procedures instead of merely invoking APIs of trusted libraries. Thus, the operations needed for OPRF-PPKR and WBP might be more prone to implementation errors or be simply not available or accessible through the shielded HSM APIs when using "off-the-shelf" HSMs. The clear downside of both simple protocols is that they lose security when the HSM is (fully) compromised. However, given that they require only well-understood operations by the HSM, such a simpler HSM might be easier to protect, making a security breach less likely.

Further, realizing an efficient *quantum-safe* OPRF is still an open problem. This is in contrast to the other standard building blocks for which quantum-safe options exist. Here the basic/enhanced encrypt-to-HSM protocols again have the advantage over OPRF-PPKR and WBP, as they rely on standard primitives only, and can easily benefit from a post-quantum "upgrade".

Lev*-3 Security for Offline Users in enhanced encrypt-to-HSM.* While we prove the enhanced encrypt-to-HSM protocol to satisfy at most Lev-2 security, it actually does preserve strong guarantees if the HSM is fully corrupted, but only for "offline" clients. Recall that we assumed the HSM's attestation key to be the single value that enjoys particularly strong protection – and thus is the only additional information the adversary gets upon full server corruption. Consequently, the impact of such a compromise on the protocol's security is then rather limited. In fact, for users who never use the PPKR *after* the full corruption happened, our enhanced encrypt-to-HSM protocol provides the same protection as OPRF-PPKR: their leaked files must still be cracked through individual offline attacks against each password. This might be a sufficient guarantee in reality. Especially in settings where it will be known which HSM is used by the server, and assuming that a breach of the HSM most critical operation would become public. Then either the service is stopped, and the server updates to a secure HSM; or the particularly cautious users would no longer log into the PPKR service – and their password and key would be just as secure as with the OPRF-PPKR protocol. Only the users who still engage in active sessions would lose their security. However, their security is fully compromised in enhanced encrypt-to-HSM, as the malicious server will learn their password and key in plain as soon as they start a new init or

[4]We count one round as a message from party A to party B (and not as a full round-trip A to B to A).

recovery session. Here, OPRF-PPKR still ensures the secrecy of the user's data.

What is the best protocol? Overall, there is no clear answer to what can be considered the "best" protocol. We believe that both our enhanced encrypt-to-HSM and OPRF-PPKR protocols have their individual strengths that can make each the right choice for a dedicated deployment setting. The enhanced encrypt-to-HSM is clearly superior to its unsalted variant, yet preserves all simplicity advantages. Thus, here we do not see a strong reason to favor the π^{encPw} protocol (Lev-1 security) and would recommend opting for the $\pi^{\mathsf{encPw+}}$ version (Lev-2 security) whenever the advantages of the simple and standard construction outweigh the concerns of a full HSM corruption. As just discussed, the guarantees of $\pi^{\mathsf{encPw+}}$ in case of full corruption are actually not lost entirely. For those users who never use the retrieval again after the full HSM corruption occurred, it provides the same security as $\pi^{\mathsf{OPRF\text{-}PPKR}}$. Nevertheless, for applications with very high-security requirements, the $\pi^{\mathsf{OPRF\text{-}PPKR}}$ construction provides the strongest (Lev-3) guarantees, but requires more care in the implementation.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions.

Tobias Handirk has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement 802823. Julia Hesse was supported by the Swiss National Science Foundation (SNSF) under the AMBIZIONE grant "Cryptographic Protocols for Human Authentication and the IoT". Máté Horváth has been supported by the German Research Foundation (DFG), project JA2445/6-1.

References

[AMMR18] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. DiSE: Distributed symmetric-key encryption. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1993–2010, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[BC19] Jean-Baptiste Bedrune and Gabriel Campana. Everybody be cool, this is a robbery! BlackHat USA 2019, 2019. http://i.blackhat.com/USA-19/Thursday/us-19-Campana-Everybody-Be-Cool-This-Is-A-Robbery.pdf, Accessed: 24.04.2024.

[BJSL11] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011: 18th Conference on Computer and Communications Security*, pages 433–444, Chicago, Illinois, USA, October 17–21, 2011. ACM Press.

[BM93] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 244–250, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.

[BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 156–171, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany.

[BS23] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. http://toc.cryptobook.us/, 2023.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://eprint.iacr.org/2000/067.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[CGMS21] Mihai Christodorescu, Sivanarayana Gaddam, Pratyay Mukherjee, and Rohit Sinha. Amortized threshold symmetric-key encryption. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2758–2779, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[CLLN14] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 256–275, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[DFG$^+$23a] Gareth T. Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the whatsapp end-to-end encrypted backup protocol. Cryptology ePrint Archive, Paper 2023/843, 2023. https://eprint.iacr.org/2023/843.

[DFG$^+$23b] Gareth T. Davies, Sebastian H. Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the WhatsApp end-to-end encrypted backup protocol. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 330–361, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Heidelberg, Germany.

[DHL22] Poulami Das, Julia Hesse, and Anja Lehmann. DPaSE: Distributed password-authenticated symmetric-key encryption, or how to get many keys from one password. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22: 17th ACM Symposium on Information, Computer and Communications Security*, pages 682–696, Nagasaki, Japan, May 30 – June 3, 2022. ACM Press.

[FHH$^+$24] Sebastian Faller, Tobias Handirk, Julia Hesse, Máté Horváth, and Anja Lehmann. Password-protected key retrieval with(out) HSM protection. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, 2024.

[GMR06] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 142–159, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany.

[JKKX16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 276–291. IEEE, 2016.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 456–486, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[Krs16] Ivan Krstic. Behind the scenes with ios security, 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf, Accessed: 18.04.2024.

[Lun19] Joshua Lund. Technology preview for secure value recovery, 2019. https://signal.org/blog/secure-value-recovery/, Accessed: 18.04.2024.

[OSV23] Chris Orsini, Alessandra Scafuro, and Tanner Verber. How to recover a cryptographic secret from the cloud. Cryptology ePrint Archive, Paper 2023/1308, 2023. https://eprint.iacr.org/2023/1308.

[PST17] Rafael Pass, Elaine Shi, and Florian Tramèr. Formal abstractions for attested execution secure processors. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 260–289, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[Sca19] Alessandra Scafuro. Break-glass encryption. In Dongdai Lin and Kazue Sako, editors, *PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 11443 of *Lecture Notes in Computer Science*, pages 34–62, Beijing, China, April 14–17, 2019. Springer, Heidelberg, Germany.

[SRW22] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on samsung's TrustZone keymaster design. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 251–268, Boston, MA, August 2022. USENIX Association.

[VBMW$^+$18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[VBPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[Wal18] Shabsi Walfish. Google cloud key vault service, 2018. https://developer.android.com/about/versions/pie/security/ckv-whitepaper, Accessed: 18.04.2024.

[WH20] Xunhua Wang and Ben Huson. Robust distributed symmetric-key encryption. Cryptology ePrint Archive, Report 2020/1001, 2020. https://eprint.iacr.org/2020/1001.

[Wha21] WhatsApp. Security of End-to-End Encrypted Backups, September 2021. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf, Accessed: 18.04.2024.

Appendix

A Comparison with $\mathcal{F}_{\mathsf{PPKR}}$ of [DFG⁺23b]

In this part, we summarize the changes that we introduced compared to the ideal functionality $\mathcal{F}_{\mathsf{PPKR}}$ in Figures 5 and 6, defined by [DFG⁺23b]. We can group the differences into the following categories.

Extension with HSM corruption.

- Added LeakFile interface to capture leakage of password files. The functionality keeps records $\langle \text{LEAKED}, \mathsf{ID_C}, \mathsf{pw}, K, i \rangle$ of leaked files (see LS.2), where the counter value i distinguishes between different leakages that affected the data of $\mathsf{ID_C}$. We note that running Init with a fully corrupt S also has the effect of leaking the password file (see CIS.4).
- Added OfflineAttack interface to model both offline password guessing against leaked records, and offline password guessing against currently and previously stored records by a fully corrupt HSM.
- Earlier the server had two corruption states, Honest, and Corrupt, that we extended with a third state called FullyCorrupt to model the case when the server has full control over the HSM (see the (FullyCorrupt, S) interface). This new corruption state also requires changes of the Init and Rec interfaces (see CIS.3, CIS.4, and RS.2, RS.3 respectively). For ease of expression, we set all ctr values to ∞ when the server gets FullyCorrupt.
- Added key planting in recovery in RS.2. A FullyCorrupt server may submit a password guess and key to be planted. If it submits a correct password guess, the client recovers the planted key. If it submits a wrong password guess, the client fails. If S is not FullyCorrupt or it doesn't submit a password guess, recovery works as before.

Changes affecting the security of PPKR.

- [DFG⁺23b] artificially weakened the PPKR functionality to be able to analyze WBP that allows a malicious server to prevent the erasure of password files when a client re-runs Init (see boxed code on figs. 5 to 6). We strengthened our functionality by disabling this attack.
- We identified and fixed a bug in functionality from Davies et al. [DFG⁺23b]. In particular, WBP and our OPRF-PPKR always enable resetting the counter in recoveries for files, stored in initialization by corrupt client or corrupt server. However, this was not captured in the ideal functionality of [DFG⁺23b] (see CRC.2 and CRS.3). We fix this problem by marking the file records either Honest or Malicious (see CIS.3) and allow for resetting the counter in case of a file record marked Malicious (see CRS.2).
- In the original PPKR functionality, S receives an output after invoking the InitC and RecC interfaces (see IC.3 and RC.2 respectively). When dealing with a corrupt client, the simulator needs early-extract the password to be able to send the proper input to the functionality. However, early extraction is impossible in any OPRF-based PPKR protocol that is instantiated with an OPRF that information-theoretically hides the PRF input (i.e. the password), such as 2HashDH, that is used by both WBP and our OPRF-PPKR. The problem is that we cannot extract the password from the client's first OPRF message (the security proof in [DFG⁺23b] missed this problem) so the simulation runs into a problem. We fix this issue by modifying the functionality so that S does not get output anymore (see IC.4 and RC.3 respectively), which is in line with our other modification that sessions can be initiated by both parties.

Broadening the applicability.

- Allowed arbitrary order of inputs in Init (see IC.2, IS.2) and Recovery (see RC.1, RS.1), instead of sessions always being initiated by the client (see IS.1 and IR.2 respectively).

Syntactical changes.

- Integrated former DoS interfaces called CompleteInitC-DoS, CompleteInitS-DoS, CompleteRecC-DoS and CompleteRecS-DoS into adversarial complete interfaces (see CIC.3, CIS.2, CRC.2 and CRS.2 respectively) via b_C, b_S bits indicating whether a DoS attack is successful (1) or not (0).

Recovery Phase

On input (RecC, sid, $\mathsf{ID_C}$, pw′) from $\mathsf{ID_C}$ (or $\mathcal{A}$ if $\mathsf{ID_C}$ is corrupt):

- RC.1 Record ⟨RecC, sid, $\mathsf{ID_C}$, pw′⟩, overwriting any existing record ⟨RecC, sid, $\mathsf{ID_C}$, ∗⟩ // Storing $\mathsf{ID_C}$'s current init state; a client can only be in one recovery session.
- RC.2 Send (RecC, sid, $\mathsf{ID_C}$) to $\mathcal{A}$ and S. // S learns which clients started recovery, with the guarantee that attempts by honest clients cannot be faked.

On input (RecS, sid, $\mathsf{ID_C}$, $\mathsf{ID_C}^*$) from S (or $\mathcal{A}$ if S is corrupt): // Server agrees to assist $\mathsf{ID_C}$ in recovery. If S is corrupt, $\mathcal{A}$ can reroute the recovery to a different $\mathsf{ID_C}^*$. Note that $\mathsf{ID_C}^*$ does not have to be the identity of an existing client and can be an arbitrary identity

- RS.1 If S is honest, set $\mathsf{ID_C}' \leftarrow \mathsf{ID_C}$, $\boxed{\text{otherwise set } \mathsf{ID_C}' \leftarrow \mathsf{ID_C}^*}$
- RS.2 Retrieve record ⟨RecC, sid, $\mathsf{ID_C}$, [pw′]⟩ // Continue only if $\mathsf{ID_C}$ started recovery already
- RS.3 If there exists no record ⟨File, $\mathsf{ID_C}'$, sid, [pw], [K]⟩ marked Stored, send (RecRes, aid, Fail) to S (or $\mathcal{A}$ if S is corrupt). Else retrieve the record. // The currently stored K and pw (for $\mathsf{ID_C}'$) are used. If $\mathsf{ID_C}'$ re-inits afterwards, it has no effect on this recovery session.
- RS.4 If $\mathsf{tx_{sid}}[\mathsf{ID_C}'] = 0$, delete record ⟨File, sid, $\mathsf{ID_C}'$, pw, K⟩ marked Stored and send (DelRec, sid, $\mathsf{ID_C}'$) to S and $\mathcal{A}$ Else continue.
- RS.5 Set $\mathsf{tx_{sid}}[\mathsf{ID_C}'] \leftarrow \mathsf{tx_{sid}}[\mathsf{ID_C}'] - 1$
- RS.6 Append pw and K to record ⟨RecC, sid, $\mathsf{ID_C}$, pw′⟩, overwriting any existing record ⟨RecC, sid, $\mathsf{ID_C}$, ∗, ∗, ∗⟩ // The recovery session of $\mathsf{ID_C}$ is used
- RS.7 Send (RecS, sid, $\mathsf{ID_C}'$, $\mathrm{pw} \overset{?}{=} \mathrm{pw}'$) to $\mathcal{A}$ // A ppKR protocol may not hide whether recovery was successful or not

On input (CompleteRecC, sid, $\mathsf{ID_C}$) from $\mathcal{A}$:

- CRC.1 Retrieve record ⟨RecC, sid, $\mathsf{ID_C}$, [pw′], [pw], [K]⟩. If record is marked recovered, delete it. Otherwise, mark it recovered. // Ensures that record can be retrieved twice before deletion.
- CRC.2 Determine the output as follows:
  - (1) If pw = pw′ then set $K' \leftarrow K$ // Recovering the key!
  - (2) In all other cases, set $K' \leftarrow$ Fail
- CRC.3 Send (RecRes, sid, K') to $\mathsf{ID_C}$ (or $\mathcal{A}$ if $\mathsf{ID_C}$ is corrupt).

On input (CompleteRecS, sid, $\mathsf{ID_C}$) from $\mathcal{A}$// Server finishes recovery session by learning whether the password was correct or not:

- CRS.1 Retrieve record ⟨RecC, sid, $\mathsf{ID_C}$, [pw], [pw′], [K]⟩. If record is marked recovered, delete it. Otherwise, mark it recovered. // Ensures that record can be retrieved twice before deletion.
- CRS.2 If pw = pw′, set $\mathsf{tx_{sid}}[\mathsf{ID_C}] \leftarrow 10$ and send (RecRes, sid, $\mathsf{ID_C}$, Succ) to S.
- CRS.3 If pw ≠ pw′, then send (RecRes, sid, $\mathsf{ID_C}$, Fail) to S (or $\mathcal{A}$ if S is corrupt).

Attacks on Recovery Phase

On input (CompleteRecC-DoS, sid, $\mathsf{ID_C}$) from $\mathcal{A}$ // Network attacker or malicious server can always make the client fail:

- CRCD.1 Retrieve record ⟨RecC, sid, $\mathsf{ID_C}$, ∗, ∗, ∗⟩ and delete it.
- CRCD.2 Send (RecRes, sid, Fail) to $\mathsf{ID_C}$ (or $\mathcal{A}$ if $\mathsf{ID_C}$ is corrupt).

On input (CompleteRecS-DoS, sid, $\mathsf{ID_C}$) from $\mathcal{A}$: // Server finishes with failure. In particular, it never learns if the password was correct

- CRSC.1 Retrieve record ⟨RecC, sid, $\mathsf{ID_C}$, ∗, ∗, ∗⟩ and delete it.
- CRSD.2 Output (RecRes, sid, $\mathsf{ID_C}$, Fail) to S (or $\mathcal{A}$ if S is corrupt).

Figure 6: Ideal functionality $\mathcal{F}_{\mathsf{PPKR}}$, original version from [DFG+23b], cont'd (recovery interfaces). Boxed code reflects an attack on WBP.

$\mathcal{F}_{\mathsf{PPKR}}$ is parameterized with a security parameter λ. $\mathcal{F}_{\mathsf{PPKR}}$ talks to a server S where S is encoded in sid. $\mathcal{F}_{\mathsf{PPKR}}$ also talks to the adversary $\mathcal{A}$, and arbitrary clients $\mathsf{ID_C}$. If the functionality tries to retrieve a record that does not exist, it ignores the incoming message. We write $\mathsf{tx_{sid}}[\cdot]$ for a list of counters.

Offline attacks

On input (MaliciousInit, sid, $\mathsf{ID_C}$, pw*, K^*) from $\mathcal{A}$: // A corrupt server can impersonate an either honest or corrupt $\mathsf{ID_C}$ and initialize on his behalf.

- MI.1 If S is honest ignore this input.
- MI.2 Record ⟨File, sid, $\mathsf{ID_C}$, pw*, K^*⟩, overwriting any existing record ⟨File, sid, $\mathsf{ID_C}$, ∗, ∗⟩. Set $\mathsf{tx_{sid}}[\mathsf{ID_C}] \leftarrow 10$

On input (MaliciousRec, sid, $\mathsf{ID_C}$, pw*) from $\mathcal{A}$: // Attacking an honest client's stored key: bury the key after 10 subsequent wrong password guesses.

- MR.1 If S is honest ignore this input. // Server needs to be corrupt to mount an offline attack.
- MR.2 Retrieve record ⟨File, sid, $\mathsf{ID_C}$, [pw], [K]⟩ marked Stored.
- MR.3 If $\mathsf{tx_{sid}}[\mathsf{ID_C}] = 0$, delete record ⟨File, sid, $\mathsf{ID_C}$, pw, K⟩ and output (DelRec, sid, $\mathsf{ID_C}$) to $\mathcal{A}$ // The key is buried if zero guesses remain.
- MR.4 If pw* = pw, set $\mathsf{tx_{sid}}[\mathsf{ID_C}] \leftarrow 10$ and output (sid, K) to $\mathcal{A}$. Otherwise, set $\mathsf{tx_{sid}}[\mathsf{ID_C}] \leftarrow \mathsf{tx_{sid}}[\mathsf{ID_C}] - 1$ an output (sid, Fail) to $\mathcal{A}$

Initialization phase

On input (InitC, sid, $\mathsf{ID_C}$, pw) from $\mathsf{ID_C}$ (or $\mathcal{A}$ if $\mathsf{ID_C}$ is corrupt): // Client always starts initialization

- IC.1 Choose $K \xleftarrow{\$} \{0,1\}^\lambda$
- IC.2 Record ⟨InitC, sid, $\mathsf{ID_C}$, pw, K⟩, overwriting any existing record ⟨InitC, sid, $\mathsf{ID_C}$, ∗, ∗⟩ // Storing $\mathsf{ID_C}$'s current init state; a client can only be in one initialization session
- IC.3 Send (InitC, sid, $\mathsf{ID_C}$) to $\mathcal{A}$ and to S

On input (InitS, sid, $\mathsf{ID_C}$, $\boxed{\mathsf{ID_C}^*}$) from S (or $\mathcal{A}$ if S is corrupt): // Server agrees to assist $\mathsf{ID_C}$ in initialization. If S is corrupt, $\mathcal{A}$ can reroute the initialization to a different $\mathsf{ID_C}^*$. Note that $\mathsf{ID_C}^*$ does not have to be the identity of an existing client and can be an arbitrary identity

- IS.1 Retrieve ⟨InitC, sid, $\mathsf{ID_C}$, [pw], [K]⟩ // Continue only if $\mathsf{ID_C}$ started initialization already
- IS.2 $\boxed{\text{If S is honest,}}$ record ⟨File, sid, $\mathsf{ID_C}$, pw, K⟩, overwriting any existing record ⟨File, sid, $\mathsf{ID_C}$, ∗, ∗⟩, and send (InitS, sid, $\mathsf{ID_C}$) to $\mathcal{A}$ // Storing S's current init state; the server can only be in one initialization session. Invariant: There is only one key stored
- IS.3 $\boxed{\text{Otherwise, record ⟨File, sid, } \mathsf{ID_C}^*\text{, pw, } K\text{⟩, overwriting any existing record ⟨File, sid, } \mathsf{ID_C}^*\text{, ∗, ∗⟩.}}$

On input (CompleteInitC, sid, $\mathsf{ID_C}$) from $\mathcal{A}$: // Client completes the protocol and outputs a key

- CIC.1 Retrieve record ⟨InitC, sid, $\mathsf{ID_C}$, ∗, [K]⟩ and delete it
- CIC.2 Output (InitRes, sid, K) to $\mathsf{ID_C}$ (or $\mathcal{A}$ if $\mathsf{ID_C}$ is corrupt)

On input (CompleteInitS, sid, $\mathsf{ID_C}$) from $\mathcal{A}$: // Server concludes initialization with file storage

- CIS.1 Retrieve record ⟨File, sid, $\mathsf{ID_C}$, ∗, ∗⟩ not marked Stored and mark it Stored // Note: there is only one such record thanks to overwriting in InitS interface. This becomes the stored key now!
- CIS.2 Set $\mathsf{tx_{sid}}[\mathsf{ID_C}] \leftarrow 10$
- CIS.3 Send (InitRes, sid, $\mathsf{ID_C}$, Succ) to S (or $\mathcal{A}$ if S is corrupt)

Attacks on Initialization Phase

On input (CompleteInitC-DoS, sid, $\mathsf{ID_C}$) from $\mathcal{A}$: // DoS attack against $\mathsf{ID_C}$, who concludes the initialization session with failure.

- CICD.1 Retrieve record ⟨InitC, sid, $\mathsf{ID_C}$, ∗, ∗⟩ and delete it
- CICD.2 Output (InitRes, sid, Fail) to $\mathsf{ID_C}$ (or $\mathcal{A}$ if $\mathsf{ID_C}$ is corrupt)

On input (CompleteInitS-DoS, sid, $\mathsf{ID_C}$) from $\mathcal{A}$: // DoS attack against the server, such that it cannot store a file

- CISD.1 Delete any record ⟨File, sid, $\mathsf{ID_C}$, ∗, ∗⟩ // Server's state in current initialization session no longer needed
- CISD.2 Send (InitRes, sid, $\mathsf{ID_C}$, Fail) to S (or $\mathcal{A}$ if S is corrupt)

Figure 5: Ideal functionality $\mathcal{F}_{\mathsf{PPKR}}$ for password-protected key retrieval, offline attacks and initialization interfaces, original version from [DFG+23b]. Boxed code reflects an attack on WBP.

Simulator $\textsc{Sim}_{\mathsf{OPRF}}(\mathsf{sid}, H_1, H_2, N)$

The simulator obtains as input a session identifier sid indicating which (multi-key) $\mathcal{F}_{\mathsf{OPRF}}$ instance it communicates with, the description of two hash functions $H_2 : \{0,1\}^* \times \mathbb{G} \to \{0,1\}^l, H_1 : \{0,1\}^* \to \mathbb{G}$ with $l \in \mathbb{N}$ and $\mathbb{G} = \langle g \rangle$ a group of order q, and a number $N \in \mathbb{N}$.

(1) Pick and record N random numbers $r_1, \ldots, r_N \in \mathbb{Z}_q$ and set $g_1 \leftarrow g^{r_1}, \ldots, g_N \leftarrow g^{r_N}$. Set counter $J \leftarrow 1$ and $I \leftarrow 1$.

(2) On $(\textsc{Init}, \mathsf{sid}, \mathsf{kid}, \mathsf{S})$ from $\mathcal{F}_{\mathsf{OPRF}}$, record $\langle F, \mathsf{S}, \mathsf{kid}, k, z = g^k \rangle$ for $k \xleftarrow{\$} \mathbb{Z}_q$ and record $\langle \mathsf{S}, \mathsf{kid} \rangle$.

(3) On $(\textsc{Compromise}, \mathsf{sid}, \mathsf{kid}, \mathsf{S})$ from $\mathcal{A}$, retrieve $\langle \mathsf{S}, \mathsf{kid} \rangle$ and declare it $\textsc{Compromised}$. Retrieve tuple $\langle F, \mathsf{S}, *, \mathsf{kid}, k, * \rangle$, send $(\textsc{Compromise}, \mathsf{sid}, \mathsf{kid})$ to $\mathcal{F}_{\mathsf{OPRF}}$, and send $(\mathsf{sid}, \mathsf{kid}, k)$ to $\mathcal{A}$.

(4) Every time when there is a fresh query x to $H_1(\cdot)$, answer it with g_J and record $\langle H_1, x, r_J \rangle$. Set $J \leftarrow J + 1$.

(5) Upon receiving $(\textsc{Eval}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S})$ from $\mathcal{F}_{\mathsf{OPRF}}$, send $(\mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, g_J)$ to $\mathcal{A}$ as C's message to S and record $\langle \mathsf{kid}, \mathsf{ssid}, \mathsf{C}, r_J \rangle$. Set $J \leftarrow J + 1$.

(6) Upon receiving $(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid})$ from $\mathcal{F}_{\mathsf{OPRF}}$ and $(\mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, a)$ from $\mathcal{A}$ as some client's C message to some honest server S:
  - If there is a record $\langle F, \mathsf{S}, \mathsf{kid}, k, * \rangle$, then send $(\mathsf{sid}, \mathsf{ssid}, a^k)$ as the response of S for client C to $\mathcal{A}$.

(7) Upon receiving $(\mathsf{sid}, \mathsf{ssid}, b)$ with $b \in \mathbb{G}$ from $\mathcal{A}$ as some server's S′ message to a client C, retrieve record $\langle *, \mathsf{ssid}, \mathsf{C}, r \rangle$ and g_j sent in step 5 for ssid, C.
  - [$\mathcal{A}$ delivers honestly.] If there is a record $\langle F, \mathsf{S}, \mathsf{kid}, k, * \rangle$ with $b = g_j^k$ and record $\langle \mathsf{S}, \mathsf{kid} \rangle$ is not marked $\textsc{Compromised}$, send $(\textsc{RcvComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S})$ to $\mathcal{F}_{\mathsf{OPRF}}$.
  - [$\mathcal{A}$ plays server using non-fresh adversarial key.] If there is a record $\langle M, \mathcal{A}, i, \perp, b^{1/r} \rangle$, send $(\textsc{SndrComplete}, \mathsf{sid}, i, \mathsf{ssid})$ and $(\textsc{RcvComplete}, \mathsf{sid}, i, \mathsf{ssid}, \mathsf{C}, \mathcal{A})$ to $\mathcal{F}_{\mathsf{OPRF}}$.
  - [$\mathcal{A}$ plays server with compromised key.] If there is a record $\langle F, \mathsf{S}, \mathsf{kid}, *, b^{1/r} \rangle$ and record $\langle \mathsf{S}, \mathsf{kid} \rangle$ is marked $\textsc{Compromised}$, send $(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid})$ and $(\textsc{RcvComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S})$ to $\mathcal{F}_{\mathsf{OPRF}}$.
  - [$\mathcal{A}$ uses fresh key.] If there is no such record $\langle T, *, *, *, b^{1/r} \rangle$, set $i \leftarrow I$, record $\langle M, \mathcal{A}, i, \perp, b^{1/r} \rangle$, and set $I++$. Send $(\textsc{SndrComplete}, \mathsf{sid}, i, \mathsf{ssid})$ and $(\textsc{RcvComplete}, \mathsf{sid}, i, \mathsf{ssid}, \mathsf{C}, \mathcal{A})$ to $\mathcal{F}_{\mathsf{OPRF}}$. // $b^{1/r} = g^{\bar{k}}$ serves as identifier of a malicious key $\bar{k}$ not known to $\textsc{Sim}_{\mathsf{OPRF}}$.

(8) Every time when there is a fresh query (x, u) to $H_2(\cdot, \cdot)$, retrieve record $\langle H_1, x, r \rangle$. If there is no such record, then pick $H_2(x, u) \xleftarrow{\$} \{0,1\}^l$. Otherwise, do the following:
  - [$u = H(x)^k$ for a server's key.] If some record $\langle F, \mathsf{S}, \mathsf{kid}, k, z \rangle$ satisfies $z = u^{1/r}$ do:
    - [Compute PRF value for k, x offline.] If S is $\textsc{Compromised}$ or corrupt, send $(\textsc{OfflineEval}, \mathsf{sid}, \mathsf{kid}, \mathsf{S}, x)$ to $\mathcal{F}_{\mathsf{OPRF}}$, and on response $(\textsc{OfflineEval}, \mathsf{sid}, \mathsf{kid}, \mathsf{S}, x, y)$, set $H_2(x, u) \leftarrow y$.
    - [Compute PRF value for k, x online, relying on a ticket $\mathsf{tx}[\mathsf{S}, \mathsf{kid}]$.] If S is not $\textsc{Compromised}$, pick a fresh identifier ssid* and send $(\textsc{Eval}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}^*, \perp, x)$ and $(\textsc{RcvComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}^*, \mathcal{A}, \mathsf{S})$ to $\mathcal{F}_{\mathsf{OPRF}}$. If $\mathcal{F}_{\mathsf{OPRF}}$ ignores the last message then abort. Else, on $\mathcal{F}_{\mathsf{OPRF}}$'s response $(\textsc{Eval}, \mathsf{sid}, \mathsf{ssid}^*, y)$, set $H_2(x, u) \leftarrow y$.
  - [$u = H(x)^k$ for an adversarial k.] Else, if there is a tuple $\langle M, \mathcal{A}, i, \perp, u^{1/r} \rangle$ then send $(\textsc{OfflineEval}, \mathsf{sid}, i, \mathcal{A}, x)$ to $\mathcal{F}_{\mathsf{OPRF}}$, and on response $(\textsc{OfflineEval}, \mathsf{sid}, i, \mathcal{A}, x, y)$ set $H_2(x, u) \leftarrow y$
  - [Fresh adversarial key.] Else, record $\langle M, \mathcal{A}, i, \perp, u^{1/r} \rangle$ for $i = I$, send $(\textsc{OfflineEval}, \mathsf{sid}, i, \mathcal{A}, x)$ to $\mathcal{F}_{\mathsf{OPRF}}$, and on response $(\textsc{OfflineEval}, \mathsf{sid}, i, \mathcal{A}, x, y)$ set $H_2(x, u) \leftarrow y$ and $I++$.

Figure 8: The simulator that demonstrates that "multi-key" 2HashDH UC-realizes our "multi-key" $\mathcal{F}_{\mathsf{OPRF}}$, restated from [DFG+23b].

Functionality $\mathcal{F}^{\ell}_{\mathsf{OPRF}}$

The functionality is parametrized by a PRF output-length ℓ. For every kid, x, value $F_{\mathsf{sid},\mathsf{S},\mathsf{kid}}(x)$ is initially undefined, and if an undefined value $F_{\mathsf{sid},\mathsf{S},\mathsf{kid}}(x)$ is referenced then $\mathcal{F}_{\mathsf{OPRF}}$ assigns $F_{\mathsf{sid},\mathsf{S},\mathsf{kid}}(x) \xleftarrow{\$} \{0,1\}^{\ell}$.

Initialization:
On $(\textsc{Init}, \mathsf{sid}, \mathsf{kid})$ from S, if this is the first $\textsc{Init}$ message for kid, set $\mathsf{tx}[\mathsf{S}, \mathsf{kid}] = 0$, store $\langle \mathsf{S}, \mathsf{kid} \rangle$ and send $(\textsc{Init}, \mathsf{sid}, \mathsf{kid}, \mathsf{S})$ to $\mathcal{A}$. Ignore all subsequent $\textsc{Init}$ messages for kid from S. // Unique key identifiers per server.

Server Compromise:
On $(\textsc{Compromise}, \mathsf{sid}, \mathsf{kid}, \mathsf{S})$ from $\mathcal{A}$, mark $\langle \mathsf{S}, \mathsf{kid} \rangle$ as $\textsc{Compromised}$. If S is corrupted, all key identifiers kid with records $\langle \mathsf{S}, \mathsf{kid} \rangle$ are marked as $\textsc{Compromised}$. *Note: Message* $(\textsc{Compromise}, \mathsf{sid}, \mathsf{kid}, \mathsf{S})$ *requires permission from the environment.* // Key-wise compromise is possible.

Offline Evaluation:
On $(\textsc{OfflineEval}, \mathsf{sid}, \mathsf{kid}^*, \mathsf{S}, x)$ from $\mathcal{A}$, send $(\textsc{OfflineEval}, \mathsf{sid}, \mathsf{kid}^*, \mathsf{S}, x, F_{\mathsf{sid},\mathsf{S},\mathsf{kid}}(x))$ to $\mathcal{A}$ if any of the following hold: (i) $\langle \mathsf{S}, \mathsf{kid}^* \rangle$ is marked $\textsc{Compromised}$, (ii) $\mathsf{kid}^* = \mathsf{kid}$ for a kid previously received via the $\textsc{Init}$ interface from S (iii) $\mathsf{kid}^* \neq \mathsf{kid}$ for all values kid previously received via the $\textsc{Init}$ interface from S.

Evaluation:
- On $(\textsc{Eval}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{S}, x)$ from $\mathsf{P} \in \{\mathsf{U}, \mathcal{A}\}$, record $\langle \mathsf{kid}, \mathsf{ssid}, \mathsf{P}, x \rangle$ and send $(\textsc{Eval}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{P}, \mathsf{S})$ to $\mathcal{A}$.
- On $(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{kid}', \mathsf{ssid})$ from $\mathsf{P} \in \{\mathsf{S}', \mathcal{A}\}$:
  - Ignore the message if $\mathsf{P} = \mathsf{S}'$ is honest and there is no record $\langle \mathsf{S}', \mathsf{kid}' \rangle$. // Honest servers do not use unknown keys.
  - If $\mathsf{P} = \mathcal{A}$ then record $\langle \mathcal{A}, \mathsf{kid}' \rangle$ (if it does not exist already) // Adversary can play server with its own keys.
  - Increment $\mathsf{tx}[\mathsf{S}', \mathsf{kid}']$.
  - Send $(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{kid}', \mathsf{ssid}, \mathsf{S}')$ to $\mathcal{A}$.
- On $(\textsc{RcvComplete}, \mathsf{sid}, \mathsf{kid}^*, \mathsf{ssid}, \mathsf{P}, \mathsf{S}^*)$ from $\mathcal{A}$:
  - Ignore this message if there is no record $\langle *, \mathsf{ssid}, \mathsf{P}, x \rangle$ or if $\mathsf{tx}[\mathsf{S}^*, \mathsf{kid}^*] = 0$.
  - Decrement $\mathsf{tx}[\mathsf{S}^*, \mathsf{kid}^*]$.
  - Send $(\textsc{EvalOut}, \mathsf{sid}, \mathsf{ssid}, F_{\mathsf{sid},\mathsf{S}^*,\mathsf{kid}^*}(x))$ to P.

Figure 7: A multi-key version of the ideal functionality $\mathcal{F}_{\mathsf{OPRF}}$ [DFG+23b].

B The 2HashDH simulator

We restate a result of Davies et al. [DFG+23b] about the multi-session security of the 2HashDH OPRF protocol of Jarecki et al [JKKX16]. The security is proven under the following assumption.

Definition 4 ((N, Q) one-more DH assumption [JKKX16])**.** The (N, Q) one-more Diffie–Hellman (DH) assumption holds in a cyclic group $\mathbb{G} = \langle g \rangle$ if for any polynomial-time adversary $\mathcal{A}$,

$$\mathbf{Adv}^{(N,Q)\text{-OMDH}}_{\mathcal{A},\mathbb{G}}(\lambda) := \Pr_{k \xleftarrow{\$} \mathbb{Z}_q, g_i \xleftarrow{\$} \mathbb{G}} \left[\mathcal{A}^{(\cdot)^k, \mathrm{DDH}(\cdot,\cdot,\cdot,\cdot)}(g, g^k, g_1, \ldots, g_N) = S \right]$$

is negligible, where $S = \{(g_{j_s}, g^k_{j_s}) \mid s = 1, \ldots, Q+1\}$ with $j_s \in [N]$ for $s \in [Q+1]$ and

- $(\cdot)^k$ is an exponentiation oracle that $\mathcal{A}$ can query Q times and on input $h \in \mathbb{G}$ it returns h^k;

- $\mathsf{DDH}(\cdot,\cdot,\cdot,\cdot)$ is a Diffie–Hellman oracle that takes as input (g, g^k, g^x, g^y) with $g \in \mathbb{G}$ and returns 1 if $y = kx$ and 0 otherwise.

Theorem 4. *Let $H_1 : \{0,1\}^* \to \mathbb{G}$ be a hash function into a group of order $q \in \mathbb{N}$, $H_2 : \{0,1\}^* \times \mathbb{G} \to \{0,1\}^\ell$ with $\ell \in \mathbb{N}$ be another hash function, and let $k \xleftarrow{\$} \mathbb{Z}_q$. Suppose the (N, Q) one-more DH assumption holds for $\mathbb{G}$, where $Q := q_E$ is the maximum number of $(\textsc{Eval}, *, \mathsf{kid}, \mathsf{S}, *)$ queries over all tuples $(\mathsf{kid}, \mathsf{S})$ made by the environment $\mathcal{Z}$, $N := q_E + q_H$, and q_H is the total number of H_1 queries made by $\mathcal{Z}$. Then the "multi-key" protocol 2HashDH UC-realizes the "multi-key" functionality $\mathcal{F}_{\mathsf{OPRF}}$ of Figure 7, with hash functions H_1, H_2 modeled as random oracles.*

More precisely, for any adversary against 2HashDH, there is a simulator $\textsc{Sim}_{\mathsf{OPRF}}$ that interacts with $\mathcal{F}_{\mathsf{OPRF}}$ and produces a view that no environment $\mathcal{Z}$ can distinguish with advantage better than

$$\Pr[\textsc{Fail}] \leq q_I \mathbf{Adv}^{(q_E+q_H,q_E)\text{-}\mathsf{OMDH}}_{\mathcal{A},\mathbb{G}}(\lambda) + (q_E + q_H)^2/q,$$

where q_I is the number of honestly initialized keys in the system.

The full proof can be found in [DFG+23b]. Because in our work we will make use of the simulator of this statement, we restate it in Figure 8.

```
Functionality $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{OPRF\text{-}PPKR}}$

The functionality initially computes $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{Sig.KeyGen}(1^\lambda)$
On $(\textsc{GetPK}, \mathsf{ssid})$ from anyone:
    • Output pk.
On $(\textsc{Init}, \mathsf{ssid}, a, \mathsf{ID_C})$ from S:
    • $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$
    • $k_{\mathsf{OPRF}} \xleftarrow{\$} \mathbb{Z}_q$, $b \leftarrow a^{k_{\mathsf{OPRF}}}$
    • send $(\mathsf{ssid}, b, \mathsf{ID_C}, \mathsf{pk_{Enc}})$ to S
On $(\mathsf{ssid}, C)$ from S:
    • $(\mathsf{ssid}', \mathsf{pk_C}, c) \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{Enc}}, C)$
    • if $\mathsf{ssid}' \neq \mathsf{ssid}$: $out \leftarrow \textsc{Fail}$
      else:
          store $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{pk_C}, c, k_{\mathsf{OPRF}}, \mathsf{ctr} \leftarrow 10 \rangle$, $out \leftarrow \textsc{Succ}$
    • send $(\textsc{InitRes}, \mathsf{ssid}, out)$ to S
On $(\textsc{Rec}, \mathsf{ssid}, a', \mathsf{ID_C})$ from S:
    • Retrieve $\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{pk_C}], [c], [k_{\mathsf{OPRF}}], [\mathsf{ctr}] \rangle$
    • if no record can be found: send $(\textsc{RecRes}, \mathsf{ssid}, \textsc{Fail})$ to S
      else:
          if $\mathsf{ctr} = 0$: delete $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{pk_C}, c, k_{\mathsf{OPRF}}, \mathsf{ctr} \rangle$
          else:
              set ctr in the record to $\mathsf{ctr} - 1$, $b' \leftarrow a'^{k_{\mathsf{OPRF}}}$
              send $(\mathsf{ssid}, b', c, \mathsf{ID_C})$ to S
On $(\mathsf{ssid}, \sigma)$ from S:
    • if $\mathsf{Sig.Vfy}(\mathsf{pk_C}, (a', \mathsf{ID_C}, \mathsf{ssid}, b', c), \sigma)$: set counter in the
      record to $\mathsf{ctr} \leftarrow 10$. Set $out \leftarrow \textsc{Succ}$
      else:
          $out \leftarrow \textsc{Fail}$
    • send $(\textsc{RecRes}, \mathsf{ssid}, out)$ to S
On $(\textsc{LeakFile}, \mathsf{sid})$ from $\mathcal{A}$:
    • Set $L \leftarrow \emptyset$
    • For every record $\langle \textsc{File}, [\mathsf{ID_C}], [\mathsf{pk_C}], [c], [k_{\mathsf{OPRF}}], [\mathsf{ctr}] \rangle$
      append $(\mathsf{ID_C}, \mathsf{pk_C}, c, k_{\mathsf{OPRF}}, \mathsf{ctr})$ to $\mathbf{ID_C}$.
    • Output $L$ to $\mathcal{A}$.
On $(\textsc{FullyCorrupt}, \mathsf{sid})$ from $\mathcal{A}$:
    • Run $(\textsc{LeakFile}, \mathsf{sid})$ to obtain $L$.
    • Output $L$ and sk.
```

Figure 10: The ideal functionality $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{OPRF\text{-}PPKR}}$.

```
Functionality $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ / $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$

The functionality initially computes $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$
and stores $\langle \mathsf{sk_{Enc}} \rangle$.
On $(\textsc{Init}, \textsc{GetPK}, \mathsf{ssid})$ from S:
    • $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \xleftarrow{\$} \mathsf{PKE.Gen}(1^\lambda)$
    • Store $\langle \mathsf{ssid}, \mathsf{sk_{Enc}} \rangle$.
    • Send $(\textsc{Init}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ to S.
On $(\textsc{Rec}, \textsc{GetPK}, \mathsf{ssid})$ from S:
    • $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \xleftarrow{\$} \mathsf{PKE.Gen}(1^\lambda)$
    • Store $\langle \mathsf{ssid}, \mathsf{sk_{Enc}} \rangle$.
    • Send $(\textsc{Rec}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ to S.
On $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ from S:
    • Retrieve $\langle \mathsf{ssid}, [\mathsf{sk_{Enc}}] \rangle$
    • $(\mathsf{pw}, K, \mathsf{ID_C}', \mathsf{ssid}') \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{Enc}}, C)$
    • If $\mathsf{ID_C}' \neq \mathsf{ID_C}$ or $\mathsf{ssid}' \neq \mathsf{ssid}$ send $(\mathsf{ssid}, \textsc{Fail})$ to S. Else
      continue.
    • $s_1, s_2 \xleftarrow{\$} \{0,1\}^\lambda$
    • $h \leftarrow H(s_1, \mathsf{pw})$
    • $c \leftarrow K \oplus H(s_2, \mathsf{pw})$
    • store $\langle \textsc{File}, \mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr} \leftarrow 10 \rangle$
    • Delete record $\langle \mathsf{ssid}, \mathsf{sk_{Enc}} \rangle$
On $(\textsc{Rec}, \mathsf{ssid}, C, \mathsf{ID_C})$ from S:
    • Retrieve $\langle \mathsf{ssid}, \mathsf{sk_{Enc}} \rangle$.
    • $(k_{sym}, \mathsf{pw}', \mathsf{ID_C}', \mathsf{ssid}') \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}, C)$
    • If $\mathsf{ID_C}' \neq \mathsf{ID_C}$ or $\mathsf{ssid}' \neq \mathsf{ssid}$: set $r \leftarrow \textsc{Fail}$. Else:
    • Retrieve $\langle \textsc{File}, \mathsf{ID_C}, [c], [h], [s_1], [s_2], [\mathsf{ctr}] \rangle$
    • If $\mathsf{ctr} = 0$: delete $\langle \textsc{File}, \mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr} \rangle$ and set $r \leftarrow$
      $\textsc{DelRec}$. Else:
    • set ctr in the record to $\mathsf{ctr} - 1$
    • If $h \neq H(s_1, \mathsf{pw}')$: set $r \leftarrow \textsc{Fail}$. Else:
    • set ctr in the record to 10
    • $C' \xleftarrow{\$} \mathsf{SE.Enc}(k_{sym}, c \oplus H(s_2, \mathsf{pw}))$. Set $r \leftarrow C'$.
    • **In any case:** Send $(\mathsf{ssid}, r)$ to S
On $(\textsc{LeakFile})$ from $\mathcal{A}$:
    • Set $L \leftarrow \emptyset$
    • For all $\langle \textsc{File}, [\mathsf{ID_C}], [c], [h], [s_1], [s_2], [\mathsf{ctr}] \rangle$ append
      $(\mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr})$ to $L$.
    • Return $L$.
```

Figure 9: The ideal functionalities $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ and $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$. The code in gray boxes is only executed in $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$.

C HSM Functionalities

We define the HSM functionalities used by our protocols in Figure 9 and Figure 10.

D Full proofs

In this section, we render the full proofs for Theorems 1 to 3.

D.1 Proof of Theorem 1

Proof. We construct a sequence of hybrid games $\mathbf{G}_0$ to $\mathbf{G}_{14}$ where we gradually change the real-world execution of the protocol π^{encPw} (interacting with the hybrid functionality $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$) to reach the ideal-world execution, where the environment interacts with the simulator from Figures 11 to 12 and the ideal functionality $\mathcal{F}_{\mathsf{PPKR}}$. We write $\Pr[\mathbf{G}_i]$ to denote the probability that the environment outputs 1 in the hybrid game $\mathbf{G}_i$.

Game $\mathbf{G}_0$: Real world. This is the real world.

Game $\mathbf{G}_1$: Create simulator. In this game we create two new entities called the ideal functionality $\mathcal{F}$ and the simulator Sim. Initially, $\mathcal{F}$ just forwards the input of the dummy parties to Sim and outputs what Sim instructs it to output. In particular, $\mathcal{F}$ has interfaces (InitC, ssid, pw), (InitS, ssid, $\mathsf{ID_C}$),(RecC, ssid, pw′), and (RecS, ssid, $\mathsf{ID_C}$, pw*, K^*, i) that just forward the input to Sim. The simulator executes the code of all honest parties of the protocol internally on the input that it is provided by $\mathcal{F}$ and it internally runs the code of the hybrid functionality $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$. Note that these are just syntactical changes and the protocol is still executed as in the real world. We have

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_0].$$

Game $\mathbf{G}_2$: Switch ciphertext in initialization. In this game, we change how Sim computes the ciphertext C produced by honest clients during initialization. Instead of computing $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, (\mathsf{pw}, K, \mathsf{ID_C}, \mathsf{ssid}))$, the simulator computes $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, \perp)$, where $\perp$ is an arbitrary string that is used nowhere else. Nonetheless, Sim still receives the secret input pw from $\mathcal{F}$ and chooses $K \leftarrow \{0,1\}^\lambda$, which it stores in a record ⟨Init, ssid, $\mathsf{ID_C}$, C, pw, K⟩ (see I.2). Note that storing pw and K in the record is only a temporary change and we remove this again in $\mathbf{G}_{11}$.

Further, we change how Sim acts when receiving a ciphertext C from an honest $\mathsf{ID_C}$ or from a corrupt server that honestly delivers C from an honest $\mathsf{ID_C}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ in an initialization. In more detail, Sim keeps track of all ciphertexts that it computes for honest $\mathsf{ID_C}$ as $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, \perp)$ in the ⟨Init, ssid, $\mathsf{ID_C}$, C, pw, K⟩ records. When Sim receives a message (Init, ssid, C, $\mathsf{ID_C}$), then Sim tries to retrieve ⟨Init, ssid, $\mathsf{ID_C}$, C, [pw], [K]⟩, i.e., it checks if C was computed by Sim for the honest client $\mathsf{ID_C}$ as an encryption of $\perp$ in subsession ssid. If that is the case, then Sim does not use $\mathsf{sk_{Enc}}$ to decrypt C (that would yield $\perp$ anyways) but instead stores a record ⟨File, $\mathsf{ID_C}$, pw, K, 10⟩ and sends (InitRes, ssid, Succ) to the server (see IC.2 and IC.6). Again, creating this record is only a temporary change that we remove again in games $\mathbf{G}_8$, $\mathbf{G}_{11}$ and $\mathbf{G}_{14}$. If C was not computed by Sim, indicated by the fact that no record ⟨Init, ssid, $\mathsf{ID_C}$, C, $*$, $*$⟩ exists, then Sim continues as in $\mathbf{G}_1$. Note that this may lead to Sim decrypting C to $\perp$, e.g., if $\mathcal{A}$ replays some C that was computed by Sim on behalf of some honest $\mathsf{ID_C}$. However, in that case, in $\mathbf{G}_1$, Sim would output Fail to S as there would be a mismatch between $\mathsf{ID_C}$ and ID'_C or ssid and ssid′. Thus, we let Sim output Fail to S if C is decrypted to $\perp$ (see IC.3(b) and IC.7).

It is easy to see that the outputs of the server and the client are just as in $\mathbf{G}_1$. Hence, the only difference is the distribution of C. If $\mathcal{Z}$ can distinguish $\mathbf{G}_2$ from $\mathbf{G}_1$, then we can construct an adversary $\mathcal{B}_1$ against the IND-CCA security of PKE as follows: First, $\mathcal{B}_1$ does not compute $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}})$ itself but uses the pk^* provided by its challenger. Let $q_{\textsc{Init}} \in \mathbb{N}$ be the number of initializations. We construct a sequence of games $\mathbf{G}_1^{(0)}, \ldots, \mathbf{G}_1^{(q_{\textsc{Init}})}$, where in $\mathbf{G}_1^{(i)}$ the first i ciphertexts are computed as encryptions of $\perp$ if simulated for an honest $\mathsf{ID_C}$ and the remaining ciphertexts are encrypted as in $\mathbf{G}_1$ (except that pk^* is used). We have $\mathbf{G}_1 = \mathbf{G}_1^{(0)}$ and $\mathbf{G}_2 = \mathbf{G}_1^{(q_{\textsc{Init}})}$. Because $\mathcal{Z}$ can distinguish $\mathbf{G}_1$ from $\mathbf{G}_2$, there must be an index $i^* \in [q_{\textsc{Init}}]$ such that $\mathcal{Z}$ has a non-negligible advantage in distinguishing $\mathbf{G}_1^{(i^*)}$ and $\mathbf{G}_1^{(i^*-1)}$. Now, in the i^*-th initialization the reduction $\mathcal{B}_1$ gives $m_0 := (k_{sym}, \mathsf{pw}, \mathsf{ID_C}, \mathsf{ssid})$ and $m_1 := \perp$ to the challenger and uses the returned C^* as ciphertext for the i^*-th initialization. Additionally, whenever $\mathcal{B}_1$ receives a message (Init, ssid, C^*, $\mathsf{ID_C}$) from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ on behalf of a corrupt server such that there is no record ⟨Init, ssid, C^*, $\mathsf{ID_C}$, [pw], [K]⟩, then $\mathcal{B}_1$ uses its decryption oracle to decrypt C^*. The oracle answers with (pw*, K^*, $\mathsf{ID_C}^*$, ssid*) and $\mathcal{B}_1$ then proceeds as in $\mathbf{G}_1$.

Now, if C^* encrypts m_0, the game is distributed as in $\mathbf{G}_1^{(i^*-1)}$ and if it encrypts m_1, then the game is distributed as in $\mathbf{G}_1^{(i^*)}$. We get

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq q_{\textsc{Init}} \mathbf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE},\mathcal{B}_1}(\lambda).$$

Game $\mathbf{G}_3$: Inform $\mathcal{F}$ of malicious initializations. In this game, we add the MaliciousInit interface as in $\mathcal{F}_{\mathsf{PPKR}}$ to $\mathcal{F}$. We also let Sim use MaliciousInit to make $\mathcal{F}$ create records for malicious initializations. More precisely, when $\mathcal{A}$ instructs a corrupted server to send a message (InitC, ssid, C^*, $\mathsf{ID_C}$) to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$, where C^* was never sent by an honest client before, then Sim first proceeds just as in $\mathbf{G}_2$, i.e., it executes the code of $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$, but additionally gives input (MaliciousInit, ssid, $\mathsf{ID_C}$, pw*, K^*) to $\mathcal{F}$ (see IC.7).

Note that on receiving the MaliciousInit input $\mathcal{F}$ only responds to Sim. Therefore, $\mathcal{Z}$'s view did not change and we get

$$\Pr[\mathbf{G}_3] = \Pr[\mathbf{G}_2].$$

Game $\mathbf{G}_4$: Provide $\mathcal{F}$ with input of corrupted clients in initialization. In this game we change Sim such that when it receives a message (Init, ssid, C, $\mathsf{ID_C}$) from a corrupted $\mathsf{ID_C}$ to an honest S, in addition to decrypting (pw, K, ID'_C, ssid′), checking $\mathsf{ID_C} = \mathsf{ID}'_\mathsf{C}$, ssid = ssid′ and recording ⟨File, $\mathsf{ID_C}$, pw, K, 10⟩, it also gives input (InitC, ssid, pw) to $\mathcal{F}$ (see IC.3(a)). Since a corrupted $\mathsf{ID_C}$ may replay ciphertexts C that were computed by Sim as an encryption of $\perp$, Sim sets pw = $\perp$ for its InitC query if C decrypts to $\perp$. This cannot modify the view of $\mathcal{Z}$ as Sim outputs Fail to the corrupt client if such a replay happens

Initially, compute $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and store record $\langle \mathsf{sid}, \mathsf{sk_{Enc}}, \mathsf{pk_{Enc}} \rangle$. *Wait for* X means that $\textsc{Sim}^{\mathsf{encPw}}$ does not proceed to the next instruction before receiving X and meanwhile gives back activation to $\mathcal{A}$. Once it receives X, it first proceeds with the instructions on input X and then continues at the instruction, where it waited for X. If a record cannot be retrieved, the query is ignored. For brevity we omit session identifier sid from all inputs, outputs, and records.

On $(\textsc{InitC}, \mathsf{ssid}, \mathsf{ID_C})$ from $\mathcal{F}_{\mathsf{PPKR}}$:

```
I.1  Retrieve record $\langle \mathsf{sid}, *, [\mathsf{pk_{Enc}}] \rangle$ ($\mathbf{G}_1$)
I.2  Compute $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, \perp)$ and record $\langle \textsc{Init}, \mathsf{ssid}, \mathsf{ID_C}, C \rangle$ ($\mathbf{G}_2$). Send $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ to S on behalf of $\mathsf{ID_C}$ ($\mathbf{G}_1$).
```

On $(\textsc{InitS}, \mathsf{ssid}, \mathsf{ID_C})$ from $\mathcal{F}_{\mathsf{PPKR}}$:

```
I.3  If S is honest, wait for $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathsf{ID_C}$ to S. ($\mathbf{G}_1$)
```

On $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathcal{A}$ to S on behalf of $\mathsf{ID_C}$:

```
IC.1  Wait for $(\textsc{InitS}, \mathsf{ssid}, \mathsf{ID_C})$ from $\mathcal{F}_{\mathsf{PPKR}}$. ($\mathbf{G}_1$)
IC.2  If $\mathsf{ID_C}$ is honest, retrieve $\langle \textsc{Init}, \mathsf{ssid}, \mathsf{ID_C}, C \rangle$. ($\mathbf{G}_2$)
IC.3  If $\mathsf{ID_C}$ is corrupt:
      (a) Retrieve $\langle \mathsf{sid}, [\mathsf{sk_{Enc}}], * \rangle$, compute $(\mathsf{pw}, K, \mathsf{ID}'_\mathsf{C}, \mathsf{ssid}') \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{Enc}}, C)$ ($\mathbf{G}_1$) and give input $(\textsc{InitC}, \mathsf{ssid}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{PPKR}}$ on behalf of $\mathsf{ID_C}$, where $\mathsf{pw} = \perp$ if the decryption resulted in $\perp$. On response $(\textsc{InitC}, \mathsf{ssid}, \mathsf{ID_C})$ continue below. ($\mathbf{G}_4$)
      (b) If the decryption resulted in $\perp$ ($\mathbf{G}_2$), $\mathsf{ID_C} \neq \mathsf{ID}'_\mathsf{C}$, or $\mathsf{ssid} \neq \mathsf{ssid}'$, send $(\textsc{InitRes}, \mathsf{ssid}, \textsc{Fail})$ to $\mathsf{ID_C}$ ($\mathbf{G}_1$) and $(\textsc{CompleteInitS}, \mathsf{ssid}, 0)$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_5$) // 0 meaning fail.
      (c) Otherwise, store $\langle \textsc{File}, \mathsf{ID_C}, K \rangle$, overwriting any existing $\langle \textsc{File}, \mathsf{ID_C}, * \rangle$. ($\mathbf{G}_1$) // No need to store pw and ctr, as $\mathcal{F}_{\mathsf{PPKR}}$ takes care of this.
IC.4  If a record $\langle \textsc{Init}, \mathsf{ssid}, \mathsf{ID_C}, C \rangle$ exists, store record $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{ctr} \leftarrow 10 \rangle$. ($\mathbf{G}_1$) // $\textsc{Sim}$ must keep counter for $\textsc{DelRec}$ in honest IDC, honest server case.
IC.5  Send $(\textsc{InitRes}, \mathsf{ssid}, \textsc{Succ})$ to $\mathsf{ID_C}$ ($\mathbf{G}_1$) and give input $(\textsc{CompleteInitS}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_5$) // 1 meaning no fail.
```

On $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ on behalf of corrupt S:

```
IC.6  If there is a record $\langle \textsc{Init}, \mathsf{ssid}, \mathsf{ID_C}, C \rangle$ ($\mathbf{G}_2$), then give input $(\textsc{InitS}, \mathsf{ssid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_5$). // Honest $C$ from $\mathsf{ID_C}$ is used
      On response $(\textsc{InitS}, \mathsf{ssid}, \mathsf{ID_C})$, send $(\textsc{InitRes}, \mathsf{ssid}, \textsc{Succ})$ to S ($\mathbf{G}_2$). Send $(\textsc{CompleteInitS}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_5$). // $b_S = 1$ for successful Init.
IC.7  Else if there is no such record, then retrieve record $\langle \mathsf{sid}, [\mathsf{sk_{Enc}}], * \rangle$ and compute $(\mathsf{pw}^*, K^*, \mathsf{ID}^*_\mathsf{C}, \mathsf{ssid}^*) \leftarrow \mathsf{Dec}(\mathsf{sk_{Enc}}, C)$ ($\mathbf{G}_1$). If the decryption results in $\perp$ ($\mathbf{G}_2$), $\mathsf{ID}^*_\mathsf{C} \neq \mathsf{ID_C}$, or $\mathsf{ssid}^* \neq \mathsf{ssid}$, then return $(\textsc{RecRes}, \mathsf{ssid}, \textsc{Fail})$ to S ($\mathbf{G}_1$). Else give input $(\textsc{MaliciousInit}, \mathsf{ID_C}, \mathsf{pw}^*, K^*)$ to $\mathcal{F}_{\mathsf{HSM}}$ ($\mathbf{G}_3$), delete any existing record $\langle \textsc{File}, \mathsf{ID_C}, * \rangle$ ($\mathbf{G}_{12}$), and send $(\textsc{InitRes}, \mathsf{ssid}, \textsc{Succ})$ to S ($\mathbf{G}_1$). // Maliciously chosen key and pwd.
```

On $(\textsc{InitRes}, \mathsf{ssid}, out)$ from $\mathcal{A}$ to $\mathsf{ID_C}$ on behalf of S:

```
IO.1  If $(\textsc{InitRes}, \mathsf{ssid}, out)$ was never output by $\textsc{Sim}^{\mathsf{encPw}}$ on behalf of S or $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ ($\mathbf{G}_{10}$), send $(\textsc{CompleteInitC}, \mathsf{ssid}, 0)$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_{11}$).
IO.2  Else, give input $(\textsc{CompleteInitC}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_{11}$) // 1 meaning no success.
```

Figure 11: The initialization part of the simulator $\textsc{Sim}^{\mathsf{encPw}}$ for the protocol π^{encPw}.

(cf. IC.3(b)). Note that the $\textsc{InitC}$ interface of $\mathcal{F}$ is still a dummy interface, so this change does not alter the view of $\mathcal{Z}$, so

$$\Pr[\mathbf{G}_4] = \Pr[\mathbf{G}_3].$$

Game $\mathbf{G}_5$: Let $\mathcal{F}$ generate server output in initialization. In this game, we change the initialization interfaces $\textsc{InitC}$ and $\textsc{InitS}$, and add the interface $\textsc{CompleteInitS}$ to $\mathcal{F}$. We change $\textsc{InitC}$ and $\textsc{InitS}$ to be exactly as in $\mathcal{F}_{\mathsf{PPKR}}$ except that $\textsc{InitC}$ still provides $\textsc{Sim}$ with the input pw of $\mathsf{ID_C}$. Further, we change $\textsc{Sim}$ such that it uses $\textsc{CompleteInitS}$ to produce output for honest S in the initialization phase. Now, when $\textsc{Sim}$ receives $(\textsc{InitS}, \mathsf{ssid}, \mathsf{ID_C})$ from $\mathcal{F}$ and a message $(\textsc{InitC}, \mathsf{ssid}, C, \mathsf{ID_C})$ to the server, where either C was produced by $\textsc{Sim}$ itself or $(\mathsf{pw}, K, \mathsf{ID}'_\mathsf{C}, \mathsf{ssid}') \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{Enc}}, C)$ and $\mathsf{ID}'_\mathsf{C} = \mathsf{ID_C}$ and $\mathsf{ssid}' = \mathsf{ssid}$, then $\textsc{Sim}$ does not directly make the server output $(\textsc{InitRes}, \mathsf{ssid}, \mathsf{ID_C}, \textsc{Succ})$ but instead gives $(\textsc{CompleteInitS}, \mathsf{ssid}, \mathsf{ID_C}, 1)$ to $\mathcal{F}$ (IC.5). If the decrypted ID'_C is different from $\mathsf{ID_C}$ or ssid' is different from ssid, then $\textsc{Sim}$ gives input $(\textsc{CompleteInitS}, \mathsf{ssid}, \mathsf{ID_C}, 0)$ to $\mathcal{F}$ (IC.3(b)).

Furthermore, $\textsc{Sim}$ uses the $\textsc{CompleteInitS}$ interface, whenever a corrupt S honestly delivers a message from some honest $\mathsf{ID_C}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$. Even though in that case the interface gives its output to $\textsc{Sim}$, this query is necessary to ensure that $\mathcal{F}$ creates a $\textsc{File}$ record. Note that $\mathcal{F}$ only creates a $\textsc{File}$ record, if it received the $\textsc{InitC}$ and $\textsc{InitS}$ input. Thus, since S is corrupt, $\textsc{Sim}$ first gives the input $(\textsc{InitS}, \mathsf{ssid}, \mathsf{ID_C})$ to $\mathcal{F}$ (see IC.6).

We can see that the output of the honest server did not change. In $\mathbf{G}_4$ the honest server outputs $(\textsc{InitRes}, \mathsf{ssid}, \textsc{Succ})$ if it received a ciphertext that decrypted to the correct $\mathsf{ID_C}$, ssid. If C contains a different ID'_C or ssid', the server aborts. The simulator makes $\mathcal{F}$ provide the corresponding output by choosing the bit b_S

```
On $(\textsc{RecC}, \mathsf{ssid}, \mathsf{ID_C}, \mathit{match})$ from $\mathcal{F}_{\mathsf{PPKR}}$:
  R.1  Retrieve record $\langle \mathsf{sid}, [\mathsf{sk_{Enc}}], [\mathsf{pk_{Enc}}] \rangle$. ($\mathbf{G}_1$)
  R.2  Choose $k_{sym} \xleftarrow{\$} \{0,1\}^{\lambda}$ ($\mathbf{G}_1$). Compute $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, \perp)$ and record $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, C, k_{sym} \rangle$ ($\mathbf{G}_6$). Send $(\textsc{Rec}, \mathsf{ssid}, C, \mathsf{ID_C})$
       to S on behalf of $\mathsf{ID_C}$. ($\mathbf{G}_1$)
On $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID'_C}, \mathit{match})$ from $\mathcal{F}_{\mathsf{PPKR}}$:
  R.3  If S is honest, wait for $(\textsc{Rec}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathsf{ID_C}$ to S. ($\mathbf{G}_1$)
On $(\textsc{Rec}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathcal{A}$ to S on behalf of $\mathsf{ID_C}$:
  RC.1  Wait for $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C}, \mathit{match})$ from $\mathcal{F}_{\mathsf{PPKR}}$. ($\mathbf{G}_1$)
  RC.2  If $\mathsf{ID_C}$ is honest:
        (a) Retrieve $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, C, [k_{sym}] \rangle$. ($\mathbf{G}_6$)
        (b) Retrieve record $\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{ctr}] \rangle$. If $\mathsf{ctr} = 0$ set $C' \leftarrow \textsc{DelRec}$. Else if $\mathit{match} = 1$ ($\mathbf{G}_{14}$), compute $C' \xleftarrow{\$} \mathsf{SE.Enc}(k_{sym}, \perp)$ ($\mathbf{G}_9$)
            and set ctr in the record to 10. Else set $C' \leftarrow \textsc{Fail}$ ($\mathbf{G}_1$) and ctr in the record to $\mathsf{ctr} - 1$.
  RC.3  If $\mathsf{ID_C}$ is corrupt:
        (a) Retrieve $\langle \mathsf{sid}, [\mathsf{sk_{Enc}}], * \rangle$, compute $(k_{sym}, \mathsf{pw}', \mathsf{ID'_C}, \mathsf{ssid}') \leftarrow \mathsf{PKE.Dec}(\mathsf{sk_{Enc}}, C)$ ($\mathbf{G}_1$), and give input $(\textsc{RecC}, \mathsf{ssid}, \mathsf{pw}')$ to
            $\mathcal{F}_{\mathsf{PPKR}}$ on behalf of $\mathsf{ID_C}$, where $\mathsf{pw}' = \perp$ if the decryption resulted in $\perp$. On response $(\textsc{RecC}, \mathsf{ssid}, \mathsf{ID_C}, \mathit{match})$ continue below.
            ($\mathbf{G}_7$)
        (b) If the decryption resulted in $\perp$ ($\mathbf{G}_6$), $\mathsf{ID_C} \neq \mathsf{ID'_C}$, or $\mathsf{ssid} \neq \mathsf{ssid}'$, send $(\textsc{RecRes}, \mathsf{ssid}, \textsc{Fail})$ to $\mathsf{ID_C}$. ($\mathbf{G}_1$)
        (c) Send $(\textsc{CompleteRecC}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\mathsf{PPKR}}$. On response $(\textsc{RecRes}, \mathsf{ssid}, K)$, if $K \in \{\textsc{Fail}, \textsc{DelRec}\}$, set $C' \leftarrow K$. ($\mathbf{G}_{12}$)
        (d) If $K \notin \{\textsc{Fail}, \textsc{DelRec}\}$ and a record $\langle \textsc{File}, \mathsf{ID_C}, [K'] \rangle$ exists, compute $C' \xleftarrow{\$} \mathsf{SE.Enc}(k'_{sym}, K')$ ($\mathbf{G}_{12}$). // Ignore, $K$, it is different
            from the extracted $K'$
        (e) If no such record exists, compute $C' \xleftarrow{\$} \mathsf{SE.Enc}(k'_{sym}, K)$. ($\mathbf{G}_1$) // $\mathsf{ID_C}$ was honest at Init
  RC.4  Send $(\textsc{CompleteRecS}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_8$) and $(\textsc{RecRes}, \mathsf{ssid}, C')$ to $\mathsf{ID_C}$. ($\mathbf{G}_1$)
On $(\textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, C)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ on behalf of corrupt S:
  RC.5  If a record $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, C, [k_{sym}] \rangle$ exists, then give input $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C}, \perp, \perp, \perp)$ to $\mathcal{F}_{\mathsf{PPKR}}$. On response
        $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C}, \mathit{match})$, give input $(\textsc{CompleteRecS}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_8$) and on the subsequent response $(\textsc{RecRes}, \mathsf{ssid}, K')$
        execute the step RC.2(b). ($\mathbf{G}_{14}$)
  RC.6  If no record $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, C, * \rangle$ exists, then retrieve $\langle \mathsf{sid}, \mathsf{sk_{Enc}}, \mathsf{pk_{Enc}} \rangle$ and compute $(k_{sym}, \mathsf{pw}', \mathsf{ID'_C}, \mathsf{ssid}') \leftarrow \mathsf{Dec}(\mathsf{sk_{Enc}}, C)$.
        ($\mathbf{G}_1$)
  RC.7  If the decryption results in $\perp$ ($\mathbf{G}_6$), $\mathsf{ID'_C} \neq \mathsf{ID_C}$, or $\mathsf{ssid}' \neq \mathsf{ssid}$, then send $(\textsc{RecRes}, \mathsf{ssid}, \textsc{Fail})$ to S. ($\mathbf{G}_1$)
  RC.8  Give input $(\textsc{MaliciousRec}, \mathsf{ssid}, \mathsf{ID_C}, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{PPKR}}$ and on response $K'$ determine the output as follows ($\mathbf{G}_{12}$).
        (a) If $K' \in \{\textsc{DelRec}, \textsc{Fail}\}$, then send $(\textsc{RecRes}, \mathsf{ssid}, K')$ to S. ($\mathbf{G}_{12}$)
        (b) If $K' \notin \{\textsc{DelRec}, \textsc{Fail}\}$, then compute $C' \leftarrow \mathsf{SE.Enc}(k_{sym}, K)$ if a record $\langle \textsc{File}, \mathsf{ID_C}, [K] \rangle$ exists and $C' \leftarrow \mathsf{SE.Enc}(k_{sym}, K')$
            otherwise. Send $(\textsc{RecRes}, \mathsf{ssid}, C')$ to S ($\mathbf{G}_{12}$)
On $(\textsc{RecRes}, \mathsf{ssid}, \mathit{out})$ from $\mathcal{A}$ to $\mathsf{ID_C}$ on behalf of S:
  RO.1  If $(\textsc{RecRes}, \mathsf{ssid}, \mathit{out})$ was never output by $\textsc{Sim}^{\mathsf{encPw}}$ on behalf of S or $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ ($\mathbf{G}_{10}$), send $(\textsc{CompleteInitC}, \mathsf{ssid}, 0)$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_{11}$).
  RO.2  Else, give input $(\textsc{CompleteRecC}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\mathsf{PPKR}}$. ($\mathbf{G}_{11}$) // $\mathit{out} = C'$ is attestated by the HSM and $\mathcal{A}$ didn't tamper with it.
```

Figure 12: The recovery part of the simulator $\textsc{Sim}^{\mathsf{encPw}}$ for the protocol π^{encPw}.

accordingly for the CompleteInitS message. Hence,

$$\Pr[\mathbf{G}_5] = \Pr[\mathbf{G}_4].$$

Note that due to the changes introduced in Games $\mathbf{G}_3$-$\mathbf{G}_5$, $\mathcal{F}$ now stores File records in all initializations, although $\mathcal{F}$ does not use them in recoveries, yet.

Game $\mathbf{G}_6$: Switch ciphertext in recovery. In this game, we change how Sim computes the ciphertext C produced by honest clients during recovery. Instead of computing the ciphertext as $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, (k_{sym}, \mathsf{pw}', \mathsf{ID_C}, \mathsf{ssid}))$ the simulator computes it as $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, \perp)$. Still, it chooses a symmetric key $k_{sym} \xleftarrow{\$} \{0,1\}^{\lambda}$ and stores a record $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, C, k_{sym}, \mathsf{pw}' \rangle$ to remember the symmetric key (see R.2). Note that Sim still gets the client's input pw' from $\mathcal{F}$, which we change in $\mathbf{G}_{14}$).

We further change how Sim reacts on a message $(\textsc{Rec}, \mathsf{ssid}, C, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$, where C was computed by Sim itself for the recovery of an honest client. Then, Sim does not retrieve $\mathsf{sk_{Enc}}$ to decrypt C (that would yield $\perp$ anyways). Instead, Sim retrieves the record $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, C, [k_{sym}], [\mathsf{pw}'] \rangle$ (RC.2(a)) and proceeds as in $\mathbf{G}_5$ by executing the code of $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$. Similarly to $\mathbf{G}_2$, the changes introduced here may lead to Sim decrypting C to $\perp$, e.g., if $\mathcal{A}$ replays some C that was computed by Sim on behalf of some honest $\mathsf{ID_C}$. Again, we let Sim output Fail to S in that case (cf. RC.3(b) and RC.7).

Note that the output behavior of Sim did not change. That is because Sim essentially behaves like $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ except that it does not encrypt and decrypt pw', k_{sym} but it stores these values and uses the stored values later. However, the

distribution of C did change. Now, if the environment could distinguish $\mathbf{G}_6$ from $\mathbf{G}_5$, we can construct an adversary $\mathcal{B}_2$ against the IND-CCA security of PKE as follows:
Let $q_{\text{Rec}} \in \mathbb{N}$ be the number of recovery phases executed. First, $\mathcal{B}_2$ does not compute $(\mathsf{sk}_{\mathsf{Enc}}, \mathsf{pk}_{\mathsf{Enc}})$ itself anymore but uses the pk^* provided by its challenger. We construct a sequence of games $\mathbf{G}_5^{(0)}, \dots, \mathbf{G}_5^{(q_{\text{Rec}})}$, where in $\mathbf{G}_5^{(i)}$ the first i ciphertexts that honest clients produce in recovery are replaced by encryptions of $\perp$ and the remaining ciphertexts stay as in $\mathbf{G}_5$ (except that pk^* is used). If $\mathcal{Z}$ can distinguish $\mathbf{G}_6$ from $\mathbf{G}_5$ then there is an index $i^* \in [q_{\text{Rec}}]$ such that $\mathcal{Z}$ distinguishes $\mathbf{G}_5^{(i^*)}$ and $\mathbf{G}_5^{(i^*-1)}$ with non-negligible advantage. $\mathcal{B}_2$ internally runs $\mathcal{Z}$ and simulates $\mathbf{G}_5^{(i^*-1)}$ except for the i^*-th recovery. In this recovery, $\mathcal{B}_2$ gives the messages $m_0 := (k_{sym}, \mathsf{pw}', \mathsf{ID_C}, \mathsf{ssid})$ and $m_1 := \perp$ to its challenger. When the challenger responds with a ciphertext C^* then $\mathcal{B}_2$ uses C^* as the ciphertext in the message $(\text{Rec}, \mathsf{ssid}, C^*, \mathsf{ID_C})$ that the honest client sends to S. Further, if $\mathcal{B}_2$ receives at some point a message $(\text{Rec}, \mathsf{ssid}, C, \mathsf{ID_C})$ from a corrupted server to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$, where there is no record $\langle \text{Rec}, [\mathsf{ssid}'], [\mathsf{ID}'_{\mathsf{C}}], C, [k_{sym}] \rangle$, i.e., the ciphertext towards the HSM is tampered with, then $\mathcal{B}_2$ gives C to the decryption oracle provided by the IND-CCA challenger. On the oracle's answer $(k^*_{sym}, \mathsf{pw}^*, \mathsf{ID_C}^*, \mathsf{ssid}^*)$ the reduction checks if $\mathsf{ID_C}^* = \mathsf{ID_C}$ and $\mathsf{ssid}^* = \mathsf{ssid}$. If these checks, fail the reduction gives output $(\mathsf{ssid}, \text{Fail})$ to the server. Else, the reduction retrieves the record $\langle \text{File}, \mathsf{sid}, \mathsf{ID_C}, [\mathsf{pw}], [K], [\mathsf{ctr}] \rangle$. If $\mathsf{ctr} = 0$, then the reduction deletes the record and sends $(\text{DelRec}, \mathsf{ID_C})$ to S. Else, if $\mathsf{pw} = \mathsf{pw}^*$, the reduction sets $\mathsf{ctr} \leftarrow 10$, computes $C' \xleftarrow{\$} \mathsf{SE.Enc}(k^*_{sym}, K)$, and sends (ssid, C') to S. Else, the reduction decrements ctr and sends $(\mathsf{ssid}, \text{Fail})$ to S. Finally, $\mathcal{B}_2$ outputs whatever $\mathcal{Z}$ outputs.
Note that if C^* encrypts m_0, then the view of $\mathcal{Z}$ is distributed exactly as in $\mathbf{G}_5^{(i^*-1)}$, and if C^* encrypts m_1, then the view of $\mathcal{Z}$ is distributed exactly as in $\mathbf{G}_5^{(i^*)}$. We get

$$|\Pr[\mathbf{G}_6] - \Pr[\mathbf{G}_5]| \leq q_{\text{Rec}} \mathbf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE}, \mathcal{B}_2}(\lambda).$$

Game $\mathbf{G}_7$: Provide $\mathcal{F}$ with input of corrupted clients in recovery. In this game we change Sim such that when it receives a message $(\text{Rec}, \mathsf{ssid}, C, \mathsf{ID_C})$ from a corrupted $\mathsf{ID_C}$ the simulator now, in addition to decrypting $(k'_{sym}, \mathsf{pw}', \mathsf{ID}'_{\mathsf{C}}, \mathsf{ssid}') \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{Enc}}, C)$ and checking $\mathsf{ID_C} = \mathsf{ID}'_{\mathsf{C}}, \mathsf{ssid} = \mathsf{ssid}'$, also gives input $(\text{RecC}, \mathsf{ssid}, \mathsf{ID_C}, \mathsf{pw}')$ to $\mathcal{F}$. Similarly to $\mathbf{G}_4$, we use $\mathsf{pw}' = \perp$ if C is decrypted to $\perp$ (RC.3(a)). Note that the RecC interface of $\mathcal{F}$ is still a dummy interface, so this change does not alter the view of $\mathcal{Z}$, so

$$\Pr[\mathbf{G}_7] = \Pr[\mathbf{G}_6].$$

Game $\mathbf{G}_8$: Let $\mathcal{F}$ generate server output in recovery. In this game, we change the RecC and RecS interfaces of $\mathcal{F}$ and we add the CompleteRecS interface to $\mathcal{F}$.
We change RecC and RecS such that they are as in $\mathcal{F}_{\mathsf{PPKR}}$ except that RecC still forwards the secret client input pw' to the simulator. Further, we change Sim such that it uses the interfaces to produce outputs for honest servers in the recovery phase. That means, when Sim receives a message $(\text{Rec}, \mathsf{ssid}, C, \mathsf{ID_C})$ from an honest or corrupt $\mathsf{ID_C}$, it first proceeds as in $\mathbf{G}_7$, and before sending its output to $\mathsf{ID_C}$, it sends the message $(\text{CompleteRecS}, \mathsf{ssid}, \mathsf{ID_C}, 1)$ to $\mathcal{F}$ (RC.4).
Additionally, Sim uses the CompleteRecS interface whenever the corrupt server honestly delivers a message from some honest $\mathsf{ID_C}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$. Similarly to $\mathbf{G}_5$, this query gives its output back to Sim, but this query is again necessary to appropiately update the internal state of $\mathcal{F}$, in particular this interface resets the counter of $\mathsf{ID_C}$ to 10 if the recovery is successful. Again, note that this requires the simulator to first give the input $(\text{RecS}, \mathsf{ssid}, \mathsf{ID_C}, \perp, \perp, \perp)$ to $\mathcal{F}$ (RC.5).
First, note that by definition, the interface RecC, resp. RecS, is called whenever an honest client, resp. honest server, starts a recovery. Next, note that Sim also ensures that the RecC input is given to $\mathcal{F}$ when a corrupted client performs a recovery. As Sim controls $\mathsf{sk}_{\mathsf{Enc}}$, it is able to extract pw' by decrypting C and can give this as input to $\mathcal{F}$. Thus, $\mathcal{F}$ always retrieves a record $\langle \text{RecC}, \mathsf{ssid}, \mathsf{ID_C}, \mathsf{pw}', \mathsf{pw}, K \rangle$ with $\mathsf{pw}' \neq \perp$ when Sim provides it with the CompleteRecS input. Consequently, if the provided password was correct, $\mathcal{F}$ outputs Succ to S and else it outputs Fail to S. Finally, note that $\mathcal{F}$ maintains the counter exactly as $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ did, i.e., the counter is set to 10 when a new File record is created, the counter is decremented when a password guess was wrong, and the counter is reset to 10 when a password guess was correct. Thus, from now on, Sim no longer keeps a counter in its File records. Overall, the output of an honest server did not change in this game and we get

$$\Pr[\mathbf{G}_8] = \Pr[\mathbf{G}_7].$$

Game $\mathbf{G}_9$: Switch symmetric ciphertext in recovery. In this game, we change how Sim computes the ciphertext C' produced by the HSM during recoveries of honest clients. Instead of computing $C' \xleftarrow{\$} \mathsf{SE.Enc}(k_{sym}, K)$, where K is the recovered key, the simulator now computes $C' \xleftarrow{\$} \mathsf{SE.Enc}(k_{sym}, \perp)$ (RC.2(b)). Note that k_{sym} is still chosen by Sim uniformly at random for every honest client that starts a recovery.
Thus, if $\mathcal{Z}$ can distinguish $\mathbf{G}_8$ and $\mathbf{G}_9$, then we can construct an adversary $\mathcal{B}_3$ against the IND-CPA security of SE. We construct a sequence of games $\mathbf{G}_8^{(0)}, \dots, \mathbf{G}_8^{(q_{\text{Rec}})}$, where in $\mathbf{G}_8^{(i)}$ the first i ciphertexts C' are replaced by encryptions of $\perp$ as described above. If $\mathcal{Z}$ can distinguish $\mathbf{G}_8$ from $\mathbf{G}_9$, then there is an index i^* such that $\mathcal{Z}$ has non-negligible advantage in distinguishing $\mathbf{G}_8^{(i^*)}$ from $\mathbf{G}_8^{(i^*-1)}$. The reduction $\mathcal{B}_3$ internally runs $\mathcal{Z}$ and plays the role of $\mathcal{F}$ and Sim in $\mathbf{G}_8^{(i^*)}$ except for the i^*-th recovery. There, $\mathcal{B}_3$ does not choose a symmetric key k_{sym} but sends the messages $m_0 := K$ and $m_1 := \perp$ to its challenger, where K is the backup key that the HSM would use to compute C'. When the challenger returns a ciphertext C^*, $\mathcal{B}_3$ outputs the message $(\text{RecRes}, \mathsf{ssid}, C^*)$.

Note that the view of $\mathcal{Z}$ in the case that C^* encrypts m_0 is distributed exactly as in $\mathbf{G}_8^{(i^*-1)}$ and in the case that C^* encrypts m_1 the view is distributed exactly as in $\mathbf{G}_8^{(i^*)}$. Therefore we get

$$|\Pr[\mathbf{G}_9] - \Pr[\mathbf{G}_8]| \leq q_{\text{Rec}}\mathbf{Adv}^{\text{IND-CPA}}_{\text{SE},\mathcal{B}_3}(\lambda).$$

Game $\mathbf{G}_{10}$: Output Fail on tampered message from HSM. In this game, we change how the simulator reacts when an honest client receives a message from the HSM that was tampered with. Whenever an honest client receives a message (InitRes, ssid, Succ) where Sim never produced this message on behalf of S or $\mathcal{F}^{\text{encPw}}_{\text{HSM}}$, then Sim makes the client output Fail (see IO.1). Similarly, when an honest client receives a message (RecRes, ssid, *out*) that was never produced by Sim on behalf of the HSM, then Sim makes the client output Fail (see RO.1).

This game is identical to $\mathbf{G}_9$ unless $\mathcal{A}$ delivers a message to ID_C that was never output by Sim but the attestation signature is still valid. If an environment $\mathcal{Z}^*$ exists that can distinguish between $\mathbf{G}_{10}$ and $\mathbf{G}_9$, we can construct an adversary $\mathcal{B}_4$ against the sEUF-CMA security of the signature scheme Sig used for attestation. $\mathcal{B}_4$ internally runs the whole experiment including $\mathcal{Z}^*$ and outputs the above described message with the signature to its challenger. We get

$$|\Pr[\mathbf{G}_{10}] - \Pr[\mathbf{G}_9]| \leq \mathbf{Adv}^{\text{sEUF-CMA}}_{\text{Sig},\mathcal{B}_4}(\lambda).$$

Game $\mathbf{G}_{11}$: Generate output for honest clients by $\mathcal{F}$. In this game, we change $\mathcal{F}$ and Sim such that Sim lets $\mathcal{F}$ produce the output for an honest client in the initialization and recovery phase. To this end, Sim will make use of the interfaces CompleteInitC and CompleteRecC that we add to $\mathcal{F}$ exactly as in $\mathcal{F}_{\text{PPKR}}$. We change the following things:

- Whenever the simulator produces a message (Init, ssid, C, ID_C) on behalf of some honest client ID_C, the simulator no longer draws a random backup key K. Further, when it then receives a message (InitRes, ssid, *out*) to ID_C, it uses the CompleteInitC interface with the bit b_C set accordingly (IO.2 and IO.1).
- Similarly, when Sim receives a message (RecRes, ssid, C') towards an honest client ID_C, the simulator now uses the CompleteRecC interface with the bit b_C set depending on the value of *out* (RO.1 and RO.2).

As a consequence, Sim no longer needs to store the key K in the File records for honest initializations and the values pw and K in the Init records.

Note that as of $\mathbf{G}_5$, $\mathcal{F}$ already chooses a uniformly random key K when it receives an InitC message, although that key is not used up to this game. Now, the CompleteInitC interface ensures that it is given as output to ID_C in the initialization phase. Similarly, in the recovery phase of $\mathbf{G}_{10}$ the simulator retrieved its own stored backup key from the initialization phase to give it as output to ID_C in the recovery phase. Now, $\mathcal{F}$, and not Sim, stores the backup key K in its File record, retrieves it when a recovery is successful, and gives it as output to ID_C or Sim when Sim sends a CompleteRecC message for a successful subsession.

Overall, the distribution of $\mathcal{Z}$'s view did not change and we get

$$\Pr[\mathbf{G}_{11}] = \Pr[\mathbf{G}_{10}].$$

Game $\mathbf{G}_{12}$: Retrieve maliciously initialized records from $\mathcal{F}$. In this game we change how Sim responds in recoveries by a corrupt party. When it receives a message (RecC, ssid, C^*, ID_C) from $\mathcal{A}$ on behalf of a corrupted ID_C, after checking $\text{ID}'_\text{C} = \text{ID}_\text{C}$ and ssid$'$ = ssid, Sim now additionally sends (CompleteRecC, ssid, 1) to $\mathcal{F}$. If $\mathcal{F}$ answers with (RecRes, ssid, K'), where $K \in \{\text{Fail}, \text{DelRec}\}$, then Sim forwards this to ID_C on behalf of S (RC.3(c)). If $K' \notin \{\text{Fail}, \text{DelRec}\}$, then Sim needs to produce the ciphertext C'. However, the key K' received from $\mathcal{F}$ may be different from the key K that the corrupt ID_C chose in its last initialization, as $\mathcal{F}$ always chooses a random key when receiving the input InitC even for a corrupted ID_C. Therefore, if a record $\langle \text{File}, \text{ID}_\text{C}, [K]\rangle$ exists, which implies that ID_C executed an initialization phase after being corrupted, then Sim encrypts K instead of K' to obtain C' (RC.3(d) and RC.3(e)). Note that this requires the simulator to delete any potentially existing File record of some ID_C whenever the corrupted S maliciously initializes for ID_C to ensure that it always returns the key of the most recent initialization (cf. IC.7).

Furthermore, we add the MaliciousRecover interface as in $\mathcal{F}_{\text{PPKR}}$ to $\mathcal{F}$, which Sim now uses in recoveries by a corrupt S. When $\mathcal{A}$ instructs the corrupted server to send a message (RecC, ssid, C^*, ID_C) to $\mathcal{F}^{\text{encPw}}_{\text{HSM}}$, where C^* was never sent by an honest client, which is again captured by checking if C^* decrypts to $\perp$ (RC.7), Sim gives the input (MaliciousRecover, ssid, ID_C, pw*) to $\mathcal{F}$, where pw* is the password it obtained from decrypting C^* (RC.8). If the response of $\mathcal{F}$ is (DelRec, ID_C) or Fail, then Sim forwards the response to the server (RC.8(a)). Else, $\mathcal{F}$ responds with K'. As before, we need to check whether the key returned by $\mathcal{F}$ is the correct one by checking whether a File record exists for ID_C (RC.8(b)).

Note that the added interface does not output anything to the protocol parties but only to Sim. Further, note that $\mathcal{F}$ now behaves exactly as $\mathcal{F}^{\text{encPw}}_{\text{HSM}}$ did in $\mathbf{G}_{11}$ when a corrupted server interacts with $\mathcal{F}^{\text{encPw}}_{\text{HSM}}$. More precisely, on a (MaliciousInit, ssid, ID_C, pw*, K^*) the functionality creates a record containing the password pw* and the backup key K^* together with a counter, initialized to 10. Then on a (MaliciousRecover, ssid, ID_C, pw*) the functionality tries to retrieve this record, checks the counter, and compares the provided password with the stored password.

$$\Pr[\mathbf{G}_{12}] = \Pr[\mathbf{G}_{11}].$$

Game $\mathbf{G}_{13}$: Add unused attack interfaces. In this game, we add the interfaces LeakFile, Corrupt, FullyCorrupt, and OfflineAttack to $\mathcal{F}$. Note that our simulator never uses these interfaces. That is because $\mathcal{F}^{\text{encPw}}_{\text{HSM}}$ does not leak any key files and cannot be corrupted. Thus, the protocol even realizes a version of $\mathcal{F}_{\mathsf{PPKR}}$ that does not allow offline attacks. We get

$$\Pr[\mathbf{G}_{13}] = \Pr[\mathbf{G}_{12}].$$

Game $\mathbf{G}_{14}$: Ideal world. We change the ideal functionality such that no more private input pw, pw′ is given to Sim. Note that Sim only used these inputs to determine if the password pw′ used in a recovery is correct. Instead it can now use the output *match* from $\mathcal{F}$ (RC.2(b) and RC.5). Hence, we finally remove pw′ from the Rec and File records. Also, we take away the ability of Sim to give output to parties. This is also not used anymore. Thus we get

$$\Pr[\mathbf{G}_{14}] = \Pr[\mathbf{G}_{13}].$$

After this change, we reached the ideal-world execution of the protocol encPw with the simulator $\textsc{Sim} = \textsc{Sim}^{\mathsf{encPw}}$ as described in Figures 11 and 12 and the functionality $\mathcal{F} = \mathcal{F}_{\mathsf{PPKR}}$ as described in Figures 1 and 2. In particular, we never had to change how Sim reacts on receiving a GetPK message when acting as $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$.

□

D.2 Proof of Theorem 2

Proof. As in the proof of Theorem 1 we construct a sequence of hybrid games starting from the real world and ending in the ideal world. As the majority of game hops are identical to the previous proof, we only provide the games that are new. We write $\mathbf{G}_x$ for $x \in (i, i+1) \subset \mathbb{R}$ to denote that the game is added between game $\mathbf{G}_i$ and $\mathbf{G}_{i+1}$ in the sequence of games from the proof of Theorem 1.

Game $\mathbf{G}_{1.1}$: Abort on salt collision. In this game, the simulator aborts if it randomly samples a salt value that is already used for some other $\mathsf{ID_C}$. More precisely, when Sim simulates $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$ and draws s_1 and s_2 uniformly at random during initialization, Sim checks if it already sampled either value in some previous initialization. If that is the case, Sim aborts the execution (see LF.3(b)).

Let $q_{\textsc{Init}} \in \mathbb{N}$ be the number of initializations. We get

$$|\Pr[\mathbf{G}_{1.1}] - \Pr[\mathbf{G}_1]| \le \frac{q_{\textsc{Init}}(2q_{\textsc{Init}} - 1)}{2^\lambda}.$$

Game $\mathbf{G}_{5.1}$: Add OfflineAttack and LeakFile interfaces to $\mathcal{F}$. In this game, we add the interfaces OfflineAttack and LeakFile to $\mathcal{F}$ exactly as they are in $\mathcal{F}_{\mathsf{PPKR}}$. Because Sim does currently not use them, we get

$$\Pr[\mathbf{G}_{5.1}] = \Pr[\mathbf{G}_5].$$

Game $\mathbf{G}_{5.2}$: Use LeakFile interface of $\mathcal{F}$. In this game, we change how Sim responds to a (LeakFile) message from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$. Sim gives input (LeakFile) to $\mathcal{F}$ and receives a list $\mathbf{ID_C}$ from $\mathcal{F}$ (see LF.1). The simulator initializes $L \leftarrow \emptyset$. Now, for all $(\mathsf{ID_C}, \mathsf{ctr}) \in \mathbf{ID_C}$ the simulator tries to retrieve a record $\langle \textsc{leaked}, \mathsf{ID_C}, [c], [h], [s_1], [s_2], *, * \rangle$ that is not marked old.

If no such record exists then Sim chooses $s_1, s_2 \leftarrow \{0,1\}^\lambda$ uniformly at random. Then, Sim computes $h \leftarrow H(s_1, \mathsf{pw})$ and $c \leftarrow K \oplus H(s_2, \mathsf{pw})$ and creates a record $\langle \textsc{leaked}, \mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr}, 1 \rangle$. Sim appends $(\mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr})$ to L.

Otherwise, Sim adds $(\mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr})$ to L and records $\langle \textsc{leaked}, \mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr}, i+1 \rangle$, where i is the biggest number such that a record $\langle \textsc{leaked}, \mathsf{ID_C}, c, h, s_1, s_2, *, i \rangle$ exists.

Sim also does an additional step when an initialization was completed, i.e., in IC.6 or IC.8, depending on whether the server is corrupted or not. Sim goes through all so far leaked records and marks them as old. That means that they still can be offline attacked, but the next time when a record is leaked, Sim will simulate the creation of a new file. In particular, Sim keeps leaked records of already overwritten initializations. This will be important for programming H later. However, the records that are added to L are always the "current" records, i.e., records that correspond to File records of $\mathcal{F}$.

Also, note that Sim only appends leaked data for $\mathsf{ID_C}$ if $\mathsf{ID_C} \in \mathbf{ID_C}$. However, in the previous games, we ensured that $\mathcal{F}$ creates a File record whenever $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$ would have done so. More precisely, when Sim receives a message $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathcal{A}$ to S on behalf of a corrupted $\mathsf{ID_C}$ and all the checks pass then Sim gives input $(\textsc{CompleteInitC}, \mathsf{ssid}, \mathsf{ID_C}, 1)$ to $\mathcal{F}$ so $\mathcal{F}$ creates a File record. Similarly, when Sim receives a message $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$ on behalf of S and all the checks pass then Sim sends either a MalInit message or $(\textsc{CompleteInitS}, \mathsf{ssid}, \mathsf{ID_C}, 1)$ to $\mathcal{F}$ so $\mathcal{F}$ creates a File record.

Overall, the view of $\mathcal{Z}$ did not change.

$$\Pr[\mathbf{G}_{5.2}] = \Pr[\mathbf{G}_{5.1}].$$

Game $\mathbf{G}_{5.3}$: Simulate the password hash h. In this game, we let Sim simulate $h = H(s_1, \mathsf{pw})$ without using pw and we let Sim program the random oracle accordingly.

First, when Sim creates a record $\langle \textsc{leaked}, \mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr}, i \rangle$ it no longer computes h like in the real protocol but now chooses it uniformly at random, i.e., $h \xleftarrow{\$} \{0,1\}^\lambda$ (see LF.3(b))

Second, Sim observes the random oracle queries. If there is a query $H(s_1^*, \mathsf{pw}^*)$ such that a record $\langle \textsc{leaked}, [\mathsf{ID_C}], *, [h], s_1^*, *, *, [i] \rangle$ exists, then Sim gives input $(\textsc{OfflineAttack}, \mathsf{ID_C}, \mathsf{pw}^*, i)$ to $\mathcal{F}$. If $\mathcal{F}$ answers with Fail, then Sim responds with a uniformly random value. If $\mathcal{F}$ answers with $K \ne \textsc{Fail}$, then Sim programs $H(s_1^*, \mathsf{pw}^*)$ to h (see H.2). This ensures that if $\mathcal{A}$ guesses the password of some client whose file was leaked previously, then the output of the random oracle query is consistent with the values that Sim output in the leaked file.

Since H is modeled as a random oracle, choosing h as a uniformly random value does not modify the distribution of h. However, observe that $\mathcal{Z}$ could distinguish this game from the previous one if $\mathcal{A}$ guesses the salt s_1^* that some $\mathsf{ID_C}$ uses together with the password pw^* before leaking the file of $\mathsf{ID_C}$. If that happens, Sim would output a random value h^* on the oracle query (s_1^*, pw^*). Then, upon $\mathcal{A}$ leaking the file of $\mathsf{ID_C}$, the simulator would not output h^* in the leaked file and instead choose a random value h since it cannot know that $\mathsf{ID_C}$ chose the password pw^*. As s_1 is chosen

On $(\textsc{InitC}, \mathsf{ssid}, \mathsf{ID_C})$ from $\mathcal{F}_{\mathsf{PPKR}}$:

I.1 Wait for $(\textsc{Init}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ from S to $\mathsf{ID_C}$. ($\mathbf{G}_1$)

On $(\textsc{InitS}, \mathsf{ssid}, \mathsf{ID_C})$ from $\mathcal{F}_{\mathsf{PPKR}}$:

I.2 Compute $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \xleftarrow{\$} \mathsf{PKE.Gen}(1^\lambda)$ and store $\langle \mathsf{ssid}, \mathsf{sk_{Enc}}, \mathsf{pk_{Enc}} \rangle$. Send $(\textsc{Init}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ to $\mathsf{ID_C}$ on behalf of S. ($\mathbf{G}_1$)

On $(\textsc{Init}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ from $\mathcal{A}$ to $\mathsf{ID_C}$ on behalf of S:

IPK.1 Compute $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, \perp)$ and store $\langle \textsc{Init}, \mathsf{ssid}, \mathsf{ID_C}, C \rangle$ ($\mathbf{G}_2$). Send $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ to S on behalf of $\mathsf{ID_C}$. ($\mathbf{G}_1$) // $\mathsf{pk_{Enc}}$ cannot be tampered with.

On $(\textsc{RecC}, \mathsf{ssid}, \mathsf{ID_C}, \mathit{match})$ from $\mathcal{F}_{\mathsf{PPKR}}$:

R.1 Wait for $(\textsc{Rec}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ from S to $\mathsf{ID_C}$. ($\mathbf{G}_1$)

On $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C}', \mathit{match})$ from $\mathcal{F}_{\mathsf{PPKR}}$:

R.2 Compute $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \xleftarrow{\$} \mathsf{PKE.Gen}(1^\lambda)$ and store $\langle \mathsf{ssid}, \mathsf{sk_{Enc}}, \mathsf{pk_{Enc}} \rangle$. Send $(\textsc{Rec}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ to $\mathsf{ID_C}$ on behalf of S. ($\mathbf{G}_1$)

On $(\textsc{Rec}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ from $\mathcal{A}$ to $\mathsf{ID_C}$ on behalf of S:

RPK.1 Compute $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, \perp)$, choose $k_{sym} \xleftarrow{\$} \{0,1\}^\lambda$, and store $\langle \textsc{Rec}, \mathsf{ssid}, \mathsf{ID_C}, C, k_{sym} \rangle$ ($\mathbf{G}_2$). Send $(\textsc{Rec}, \mathsf{ssid}, C, \mathsf{ID_C})$ to S on behalf of $\mathsf{ID_C}$ ($\mathbf{G}_1$). // $\textsc{RecC}$ came before $\mathsf{pk_{Enc}}$.

On $(\textsc{Init}, \mathsf{ssid}, \textsc{GetPK})$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ on behalf of S:

G.1 Compute $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \xleftarrow{\$} \mathsf{PKE.Gen}(1^\lambda)$ and store $\langle \mathsf{ssid}, \mathsf{sk_{Enc}}, \mathsf{pk_{Enc}} \rangle$. Send $(\textsc{Init}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ to S on behalf of $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ ($\mathbf{G}_1$). // attestated

On $(\textsc{Rec}, \mathsf{ssid}, \textsc{GetPK})$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ on behalf of S:

G.2 Compute $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}}) \xleftarrow{\$} \mathsf{PKE.Gen}(1^\lambda)$ and store $\langle \mathsf{ssid}, \mathsf{sk_{Enc}}, \mathsf{pk_{Enc}} \rangle$. Send $(\textsc{Rec}, \mathsf{ssid}, \mathsf{pk_{Enc}})$ to S on behalf of $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ ($\mathbf{G}_1$). // attestated

On $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathcal{A}$ to S on behalf of $\mathsf{ID_C}$:

IC.1 through IC.5 as in Figure 11.

IC.6 Try to retrieve all records $\langle \textsc{leaked}, \mathsf{ID_C}, [c], [h], [s_1], [s_2], *, * \rangle$ not marked old. If such records exist, then mark all of them as old. ($\mathbf{G}_{5.2}$) // After init, the file contains fresh values.

On $(\textsc{Init}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$ on behalf of S:

IC.6 and IC.7 as in Figure 11.

IC.8 Try to retrieve all records $\langle \textsc{leaked}, \mathsf{ID_C}, [c], [h], [s_1], [s_2], *, * \rangle$ not marked old. If such records exist, then mark all of them as old. ($\mathbf{G}_{5.2}$)// After init, the file contains fresh values.

On a query $(\textsc{LeakFile})$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$:

LF.1 Give input $(\textsc{LeakFile})$ to $\mathcal{F}_{\mathsf{PPKR}}$. $\mathcal{F}_{\mathsf{PPKR}}$ returns $\mathbf{ID_C}$. ($\mathbf{G}_{5.2}$)

LF.2 $L \leftarrow \emptyset$ ($\mathbf{G}_{5.2}$)

LF.3 For each $(\mathsf{ID_C}, \mathsf{ctr}) \in \mathbf{ID_C}$:

- (a) Try to retrieve a record $\langle \textsc{leaked}, \mathsf{ID_C}, [c], [h], [s_1], [s_2], *, [i] \rangle$ not marked old. ($\mathbf{G}_{5.2}$)
- (b) If no such record exists, choose $c, h, s_1, s_2 \xleftarrow{\$} \{0,1\}^\lambda$ ($\mathbf{G}_{5.3}$ and $\mathbf{G}_{5.4}$). If s_1 or s_2 was used before as salt by $\textsc{Sim}$, then abort ($\mathbf{G}_{1.1}$). Else store $\langle \textsc{leaked}, \mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr}, 1 \rangle$. Append $(\mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr})$ to L. ($\mathbf{G}_{5.2}$)
- (c) Else store a record $\langle \textsc{leaked}, \mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr}, i+1 \rangle$, where i is the biggest number such that a record $\langle \textsc{leaked}, \mathsf{ID_C}, [c], [h], [s_1], [s_2], *, i \rangle$ exists. Append $(\mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr})$ to L. ($\mathbf{G}_{5.2}$)

LF.4 Send L to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw+}}$. ($\mathbf{G}_1$)

On a query $H(s^*, \mathsf{pw}^*)$:

H.1 If a record $\langle H, s^*, \mathsf{pw}^*, [y] \rangle$ exists, output y. ($\mathbf{G}_1$)

H.2 Try to retrieve the record $\langle \textsc{leaked}, [\mathsf{ID_C}], *, [h], s^*, *, *, [i] \rangle$. If such a record exists, give input $(\textsc{OfflineAttack}, \mathsf{ID_C}, \mathsf{pw}^*, i)$ to $\mathcal{F}_{\mathsf{PPKR}}$. If $\mathcal{F}_{\mathsf{PPKR}}$ responds with $K \neq \textsc{Fail}$, record $\langle H, s^*, \mathsf{pw}^*, h \rangle$ and output h. If $K = \textsc{Fail}$, choose $y \xleftarrow{\$} \{0,1\}^\lambda$, record $\langle H, s^*, \mathsf{pw}^*, y \rangle$ and output y. ($\mathbf{G}_{5.3}$)

H.3 If no such $\textsc{leaked}$ record exists, try to retrieve the record $\langle \textsc{leaked}, [\mathsf{ID_C}], [c], *, *, s^*, *, [i] \rangle$. If such a record exists, give input $(\textsc{OfflineAttack}, \mathsf{ID_C}, \mathsf{pw}^*, i)$ to $\mathcal{F}_{\mathsf{PPKR}}$. If $\mathcal{F}_{\mathsf{PPKR}}$ responds with $K \neq \textsc{Fail}$, record $\langle H, s^*, \mathsf{pw}^*, c \oplus K \rangle$ and output $c \oplus K$. If $K = \textsc{Fail}$, choose $y \xleftarrow{\$} \{0,1\}^\lambda$, record $\langle H, s^*, \mathsf{pw}^*, y \rangle$ and output y. ($\mathbf{G}_{5.4}$)

H.4 If both such $\textsc{leaked}$ do not exist ($\mathbf{G}_{5.4}$), choose $y \xleftarrow{\$} \{0,1\}^\lambda$, record $\langle H, s^*, \mathsf{pw}^*, y \rangle$ and output y ($\mathbf{G}_1$).

Figure 13: Differences from the simulator $\textsc{Sim}^{\mathsf{encPw}}$ (Figures 11 to 12) to the simulator $\textsc{Sim}^{\mathsf{encPw+}}$.

uniformly at random from $\{0,1\}^\lambda$, the probability of this happening can be bounded by $q_H 2^{-\lambda}$, where $q_H \in \mathbb{N}$ is the number of queries to H. Note that this even holds if other clients use the same pw* because due to $\mathbf{G}_{1.1}$ they will use a different salt s_1' and $H(s_1', \mathsf{pw}^*)$ is independent of $H(s_1^*, \mathsf{pw}^*)$. Thus, we have

$$|\Pr[\mathbf{G}_{5.3}] - \Pr[\mathbf{G}_{5.2}]| \leq \frac{q_{\text{INIT}} q_H}{2^\lambda}.$$

Game $\mathbf{G}_{5.4}$: Simulate the password hash c. In this game, we let SIM simulate the leaked records without using K and pw and we let SIM program the random oracle accordingly.
First, when SIM creates a record $\langle \text{LEAKED}, \mathsf{ID_C}, c, h, s_1, s_2, \mathsf{ctr}, i\rangle$ it no longer computes c like in the real protocol but now chooses it uniformly at random, i.e., $c \leftarrow \{0,1\}^\lambda$ (see LF.3(b)).
Second, SIM observes the random oracle queries. If there is a query $H(s_2^*, \mathsf{pw}^*)$ such that a record $\langle \text{LEAKED}, [\mathsf{ID_C}], [c], *, *, s_2^*, *, [i]\rangle$ exists, then SIM gives input $(\text{OFFLINEATTACK}, \mathsf{ID_C}, \mathsf{pw}^*, i)$ to $\mathcal{F}$. If $\mathcal{F}$ answers with FAIL, then SIM responds with a uniformly random value. If $\mathcal{F}$ answers with $K \neq$ FAIL, then, SIM programs $H(s_2^*, \mathsf{pw}^*) \leftarrow c \oplus K$. Note that for honest $\mathsf{ID_C}$, SIM uses the key K it chose on behalf of $\mathsf{ID_C}$ in the initialization instead of the one returned by $\mathcal{F}$. Once we let $\mathcal{F}$ generate the output of honest clients in $\mathbf{G}_{11}$, we change this to use the key returned from $\mathcal{F}$. This again ensures that the leaked files are consistent with the random oracle outputs (H.3).
Again, it holds that the distribution of c does not change by choosing it as uniformly random value since H is a random oracle. However, similarly to $\mathbf{G}_{5.3}$, the environment could distinguish this game from the previous one if $\mathcal{A}$ guesses the salt s_2^* of some $\mathsf{ID_C^*}$. Following the same arguments as in $\mathbf{G}_{5.3}$, we have

$$|\Pr[\mathbf{G}_{5.4}] - \Pr[\mathbf{G}_{5.3}]| \leq \frac{q_{\text{INIT}} q_H}{2^\lambda}.$$

Changed Reductions. Also, we have to change the reductions on IND-CCA security in $\mathbf{G}_2$ and $\mathbf{G}_6$ because of the difference in computing $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}})$:

Game $\mathbf{G}_2$: Let $q_{\text{INIT}} \in \mathbb{N}$ be the number of initializations. We construct a sequence of games $\mathbf{G}_1^{(0)}, \dots, \mathbf{G}_1^{(q_{\text{INIT}})}$, where in $\mathbf{G}_1^{(i)}$ the first i ciphertexts are computed as encryptions of $\perp$ and the remaining ciphertexts are encrypted as in $\mathbf{G}_1$. We have $\mathbf{G}_1 = \mathbf{G}_1^{(0)}$ and $\mathbf{G}_2 = \mathbf{G}_1^{(q_{\text{INIT}})}$. Because $\mathcal{Z}$ can distinguish $\mathbf{G}_1$ from $\mathbf{G}_2$, there must be an index $i^* \in [q_{\text{INIT}}]$ such that $\mathcal{Z}$ has a non-negligible advantage in distinguishing $\mathbf{G}_1^{(i^*)}$ and $\mathbf{G}_1^{(i^*-1)}$. Now, in the i^*-th initialization the reduction $\mathcal{B}_1$ does not choose a new key-pair (sk, pk) when it receives a GETPK or INITS message, but uses the public key that it is provided by the challenger. When it simulates $\mathsf{ID_C}$ in the i^*-th simulation, it gives $m_0 := (k_{sym}, \mathsf{pw}', \mathsf{ID_C}, \mathsf{ssid})$ and $m_1 := \perp$ to the challenger and uses the returned C^* as the ciphertext for the initialization. Further, if $\mathcal{B}_1$ receives a message $(\text{INIT}, \mathsf{ssid}, C, \mathsf{ID_C})$ from $\mathcal{A}$ to S or to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ such that $C \neq C^*$, then $\mathcal{B}_1$ uses its decryption oracle to decrypt C^* and proceeds as in $\mathbf{G}_1$.

Now, if C^* encrypts m_0 the game is distributed as in $\mathbf{G}_1^{(i^*-1)}$ and if it encrypts m_1 then the game is distributed as in $\mathbf{G}_1^{(i^*)}$.

Game $\mathbf{G}_6$: Let $q_{\text{REC}} \in \mathbb{N}$ be the number of recovery phases executed. We construct a sequence of games $\mathbf{G}_5^{(0)}, \dots, \mathbf{G}_5^{(q_{\text{REC}})}$, where in $\mathbf{G}_5^{(i)}$ the first i ciphertexts that honest clients produce in recovery are replaced by encryptions of $\perp$ and the remaining ciphertexts stay as in $\mathbf{G}_5$. If $\mathcal{Z}$ can distinguish $\mathbf{G}_6$ from $\mathbf{G}_5$, then there is an index $i^* \in [q_{\text{REC}}]$ such that $\mathcal{Z}$ distinguishes $\mathbf{G}_5^{(i^*)}$ and $\mathbf{G}_5^{(i^*-1)}$ with non-negligible advantage. $\mathcal{B}_2$ internally runs $\mathcal{Z}$ and simulates $\mathbf{G}_5^{(i^*-1)}$ except for the i^*-th recovery. In this recovery, $\mathcal{B}_2$ does not choose a fresh pair $(\mathsf{pk_{Enc}}, \mathsf{sk_{Enc}})$ when it receives a GETPK message, but uses the public key $\mathsf{pk_{Enc}^*}$ provided by the challenger. $\mathcal{B}_2$ gives the messages $m_0 := (\mathsf{pw}, K, \mathsf{ID_C}, \mathsf{ssid})$ and $m_1 := \perp$ to its challenger. When the challenger responds with a ciphertext C^*, then $\mathcal{B}_2$ uses C^* as the ciphertext in the message $(\text{REC}, \mathsf{ssid}, C^*, \mathsf{ID_C})$ that the honest client sends to S. Further, if $\mathcal{B}_2$ receives at some point a message $(\text{REC}, \mathsf{ssid}, C, \mathsf{ID_C})$ to S or $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$, where $C \neq C^*$, then $\mathcal{B}_2$ gives C to the decryption oracle provided by the IND-CCA challenger and proceeds as in $\mathbf{G}_5$. Finally, $\mathcal{B}_2$ outputs whatever $\mathcal{Z}$ outputs.

Note that if C^* encrypts m_0 then the view of $\mathcal{Z}$ is distributed exactly as in $\mathbf{G}_5^{(i^*-1)}$ and if C^* encrypts m_1 then the view of $\mathcal{Z}$ is distributed exactly as in game $\mathbf{G}_5^{(i^*)}$. □

D.3 Proof of Theorem 3

We depict our simulator in Figures 14 to 17. In the below, when referring to blue-colored boxes such as I.1, we always mean the ones from these figures. The gray boxes CIS.4 refer to $\mathcal{F}_{\mathsf{PPKR}}$ instructions from Figures 1 and 2. The simulator works with session state records

$$\langle \text{INIT}, \mathsf{ID_C}, \mathsf{kid}, \mathsf{ssid}, a, \mathsf{ID_C^*}, \mathsf{kid}, a^*, \mathsf{sk_{Enc}}, \mathsf{sk}_C, b, C\rangle$$

for initialization. All values are initialized to $\perp$ and potentially updated throught an initialization. At the end of an initialization by $\mathsf{ID_C}$, the record has the following semantics.

- INIT_1 $\in \{\mathsf{ID_C}, \perp\}$ is either the $\mathsf{ID_C}$ of an honest client running initialization or it is set to $\perp$ if the client is corrupt or if the corrupt S impersonates $\mathsf{ID_C}$;
- INIT_2 $\in \{\mathsf{kid}, \perp\}$ is either the number of initializations started by $\mathsf{ID_C}$, as seen by the HSM, or it is set to $\perp$ if the client is corrupt or if the corrupt S impersonates $\mathsf{ID_C}$;
- INIT_3 $=$ ssid indicates the sub-session id used by $\mathsf{ID_C}$;
- INIT_4 $\in \{a, \perp\}$ is either the a value sent by a honest $\mathsf{ID_C}$, or $\perp$ if $\mathsf{ID_C}$ is corrupt or if the corrupt S impersonates $\mathsf{ID_C}$.
- INIT_5 $\in \{\mathsf{ID_C^*}, \perp\}$ is either the client name, delivered to the HSM, or set to $\perp$ if the server is fully corrupt;
- INIT_6 $\in \{\mathsf{kid}, \perp\}$ is the number of initializations started by $\mathsf{ID_C^*}$, as seen by the HSM, or set to $\perp$ if the server is fully corrupt;
- INIT_7 $\in \{a^*, \perp\}$ indicates the a value, delivered to the HSM, or set to $\perp$ if the server is fully corrupt;
- INIT_8 $\in \{\mathsf{sk_{Enc}}, \perp\}$ is the encryption secret key, generated by the HSM for the current sub-session, or set to $\perp$ if the server is fully corrupt;
- INIT_9 $\in \{\mathsf{sk_C}, \perp\}$ is the client's signature secret key either decrypted from c or simulated for an honest client. It is set to $\perp$ if it cannot be extracted from a corrupted initialization;

Simulator $\textsc{Sim}^{\text{OPRF-PPKR}}$, part 1

$\textsc{Sim}^{\text{OPRF-PPKR}}$ stores a list $\mathsf{kid}[\cdot]$. Initially $\textsc{Sim}^{\text{OPRF-PPKR}}$ executes $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Sig.KeyGen}(1^\lambda)$ and stores $\langle PK, \mathsf{pk}, \mathsf{sk}\rangle$. Whenever it outputs a message towards some $\mathsf{ID_C}$ or from $\mathcal{F}_{\text{HSM}}^{\text{OPRF-PPKR}}$ to a corrupt S, it attestates the message with sk.

Wait for X means that $\textsc{Sim}^{\text{OPRF-PPKR}}$ does not proceed to the next instruction before receiving X and meanwhile gives back activation to $\mathcal{A}$. Once it receives X, it first proceeds with the instructions on input X and then continues at the instruction, where it waited for X. If a record cannot be retrieved, the query is ignored. For brevity we omit session identifier sid from all inputs, outputs, and records. ($\mathbf{G}_1$)

On ($\textsc{InitC}$, ssid, $\mathsf{ID_C}$) from $\mathcal{F}_{\text{PPKR}}$: // honest $\mathsf{ID_C}$

- I.1 If $\mathsf{kid}[\mathsf{ID_C}]$ is undefined, set $\mathsf{kid}[\mathsf{ID_C}] \leftarrow 0$. Otherwise, set $\mathsf{kid}[\mathsf{ID_C}] \leftarrow \mathsf{kid}[\mathsf{ID_C}]+1$. ($\mathbf{G}_3$) // new OPRF key for each init
- I.2 Give input ($\textsc{Eval}$, sid, $\mathsf{ID_C} \parallel \mathsf{kid}[\mathsf{ID_C}]$, ssid, S, 0) to $\mathcal{F}_{\text{OPRF}}$ from $\mathsf{ID_C}$. This triggers Step 5 of $\textsc{Sim}_{\text{OPRF}}$, which outputs (sid, $\mathsf{ID_C}$, ssid, a). ($\mathbf{G}_3$)
  Record $\langle\textsc{Init}, \mathsf{ID_C}, \mathsf{kid}[\mathsf{ID_C}], \mathsf{ssid}, a, \bot, \bot, \bot, \bot, \bot, \bot, \bot\rangle$ and send (ssid, a, $\mathsf{ID_C}$) as message from $\mathsf{ID_C}$ to S. ($\mathbf{G}_1$) // Output of $\mathcal{F}_{\text{OPRF}}$ (eval. of 0) ignored

On ($\textsc{InitS}$, ssid, $\mathsf{ID_C}$) from $\mathcal{F}_{\text{PPKR}}$: // honest S

- I.3 If S honest, wait for (ssid, a, $\mathsf{ID_C}$) from $\mathsf{ID_C}$ to S. ($\mathbf{G}_1$) // See (ssid, a, $\mathsf{ID_C}$) interface

On (`Init`, ssid, a, $\mathsf{ID_C}$) from $\mathcal{A}$ to S on behalf of $\mathsf{ID_C}$: // any $\mathsf{ID_C}$, honest S

- Ia.1 If S is honest, wait for ($\textsc{InitS}$, ssid, $\mathsf{ID_C}$) from $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_1$)
- Ia.2 If $\mathsf{ID_C}$ is corrupt:
  - If $\mathsf{kid}[\mathsf{ID_C}]$ is undefined, set $\mathsf{kid}[\mathsf{ID_C}] \leftarrow 0$, otherwise set $\mathsf{kid}[\mathsf{ID_C}] \leftarrow \mathsf{kid}[\mathsf{ID_C}] + 1$. Set $\mathsf{kid} \leftarrow \mathsf{kid}[\mathsf{ID_C}]$. ($\mathbf{G}_3$) // Ensure new oprf key. If $\mathsf{ID_C}$ is honest this is ensured in $\textsc{InitC}$
  - Record $\langle\textsc{Init}, \bot, \bot, \mathsf{ssid}, \bot, \bot, \bot, \bot, \bot, \bot, \bot, \bot\rangle$ ($\mathbf{G}_1$)
- Ia.3 If $\mathsf{ID_C}$ is honest, retrieve $\langle\textsc{Init}, \mathsf{ID_C}, [\mathsf{kid}], \mathsf{ssid}, *, \bot, \bot, \bot, \bot, \bot, \bot, \bot\rangle$. ($\mathbf{G}_3$)
- Ia.4 Give input ($\textsc{Init}$, sid, $\mathsf{ID_C} \parallel \mathsf{kid}$) to the simulated $\mathcal{F}_{\text{OPRF}}$ from the simulated S. This triggers Step 2 of $\textsc{Sim}_{\text{OPRF}}$. ($\mathbf{G}_3$) // Create new OPRF key
- Ia.5 Give input ($\textsc{SndrComplete}$, sid, $\mathsf{ID_C} \parallel \mathsf{kid}$, ssid) to the simulated $\mathcal{F}_{\text{OPRF}}$ from the simulated S. This triggers Step 6 of $\textsc{Sim}_{\text{OPRF}}$, which outputs (sid, ssid, b). ($\mathbf{G}_3$)
- Ia.6 Compute $(\mathsf{pk_{Enc}}, \mathsf{sk_{Enc}}) \xleftarrow{\$} \mathsf{PKE.Gen}(1^\lambda)$ and overwrite the entries 6-11 in $\langle\textsc{Init}, *, *, \mathsf{ssid}, *, *, *, *, *, *, *, *\rangle$ with $\mathsf{ID_C}, \mathsf{kid}, a, \mathsf{sk_{Enc}}, \bot, b$. Send (ssid, b, $\mathsf{ID_C}$, $\mathsf{pk_{Enc}}$) as HSM-signed message from S to $\mathsf{ID_C}$. ($\mathbf{G}_1$) // Can later access OPRF keys via F records in $\textsc{Sim}_{\text{OPRF}}$

On (`Init`, ssid, a, $\mathsf{ID_C}$) from $\mathcal{A}$ to $\mathcal{F}_{\text{HSM}}^{\text{OPRF-PPKR}}$ on behalf of corrupt S: // any $\mathsf{ID_C}$

- Ia.7 If no record $\langle\textsc{Init}, *, [\mathsf{kid}], \mathsf{ssid}, *, *, *, *, *, *, *, *\rangle$ exists, create $\langle\textsc{Init}, \bot, \bot, \mathsf{ssid}, \bot, \bot, \bot, \bot, \bot, \bot, \bot, \bot\rangle$. ($\mathbf{G}_1$)
  Additionally, if $\mathsf{kid}[\mathsf{ID_C}]$ is undefined set $\mathsf{kid}[\mathsf{ID_C}] \leftarrow 0$ and otherwise $\mathsf{kid}[\mathsf{ID_C}] \leftarrow \mathsf{kid}[\mathsf{ID_C}] + 1$ and set $\mathsf{kid} \leftarrow \mathsf{kid}[\mathsf{ID_C}]$. ($\mathbf{G}_3$) // $\mathsf{ID_C}$ corrupt and S just forwards or S impersonates $\mathsf{ID_C}$
- Ia.8 Execute Steps Ia.4-Ia.6, except that the output is returned to S instead of sending it from S to $\mathsf{ID_C}$. ($\mathbf{G}_3$)

On (ssid, b, ID^*_C, $\mathsf{pk_{Enc}}$) from $\mathcal{A}$ to $\mathsf{ID_C}$: // honest $\mathsf{ID_C}$, any S

- Ib.1 Retrieve $\langle\textsc{Init}, \mathsf{ID_C}, [\mathsf{kid}], \mathsf{ssid}, *, *, *, *, *, *, *, *\rangle$. ($\mathbf{G}_1$)
- Ib.2 If S is not honest, give input ($\textsc{InitS}$, ssid, $\mathsf{ID_C}$) to $\mathcal{F}_{\text{PPKR}}$. On response ($\textsc{InitS}$, ssid, $\mathsf{ID_C}$) from $\mathcal{F}_{\text{PPKR}}$ or if $\mathcal{F}_{\text{PPKR}}$ ignores the input, continue below. ($\mathbf{G}_{10}$) // Ensures that we can produce output for $\mathsf{ID_C}$
- Ib.3 If S is not fully corrupt and (ssid, b, ID^*_C, $\mathsf{pk_{Enc}}$) was never output by $\textsc{Sim}^{\text{OPRF-PPKR}}$ on behalf of S or $\mathcal{F}_{\text{HSM}}^{\text{OPRF-PPKR}}$, or if S is fully corrupt and the attestation signature does not verify, send ($\textsc{CompleteInitC}$, ssid, 0) to $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_{10}$)
- Ib.4 If $\mathsf{ID_C} \neq \mathsf{ID}^*_\mathsf{C}$, send ($\textsc{CompleteInitC}$, ssid, 0) to $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_{10}$) // $\mathsf{ID_C}$'s a rerouted
- Ib.5 Compute $(\mathsf{sk_C}, \mathsf{pk_C}) \leftarrow \mathsf{Sig.Gen}(1^\lambda)$. ($\mathbf{G}_1$)
- Ib.6 If S is fully corrupt:
  - (a) Send ($\textsc{CompleteInitS}$, ssid, 1) to $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_{10}$)
  - (b) For each record $\langle H_2, [\mathsf{pw}] \parallel \mathsf{ID_C}, *, [y]\rangle$ marked $\textsc{Consistent}$ ($\mathbf{G}_{14}$) send ($\textsc{OfflineAttack}$, $\mathsf{ID_C}$, pw, $i+1$) to $\mathcal{F}_{\text{PPKR}}$, where $i \in \mathbb{N}$ is the largest number such that a record $\langle\textsc{leaked}, \mathsf{ID_C}, *, *, *, i\rangle$ exists. ($\mathbf{G}_{10}$)
  - (c) If for any query the output is $K \neq \textsc{Fail}$ ($\mathbf{G}_{10}$), compute $c \xleftarrow{\$} \mathsf{AE.Enc}(y, (K, \mathsf{sk_C}))$. Else set $c \xleftarrow{\$} \textsc{Sim}_{\text{EQV}}(\lambda + |\mathsf{sk_C}|)$. ($\mathbf{G}_6$)
  - (d) Record $\langle\textsc{File}, \mathsf{ID_C}, \mathsf{kid}, \mathsf{pk_C}, \mathsf{sk_C}, c\rangle$ and $\langle\textsc{leaked}, \mathsf{ID_C}, \mathsf{kid}, \mathsf{sk_C}, c, i+1\rangle$, where $i \in \mathbb{N}$ is the largest number such that a record $\langle\textsc{leaked}, \mathsf{ID_C}, *, *, *, i\rangle$ exists or $i = 0$ if no such record exists. ($\mathbf{G}_1$)
  - (e) Compute $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, (\mathsf{ssid}, \mathsf{pk_C}, c))$. ($\mathbf{G}_1$)
  - (f) Overwrite the three last entries of the $\textsc{Init}$ record retrieved above with $\mathsf{sk_C}, b, C$. ($\mathbf{G}_1$)
- Ib.7 If S is not fully corrupt:
  - (a) Compute $c \xleftarrow{\$} \textsc{Sim}_{\text{EQV}}(\lambda + |\mathsf{sk_C}|)$. ($\mathbf{G}_6$)
  - (b) Compute $C \leftarrow \mathsf{PKE.Enc}(\mathsf{sk_{Enc}}, \bot)$, ($\mathbf{G}_{13}$)
  - (c) overwrite the tenth entry of the record retrieved above with $\mathsf{sk_C}$ and the last entry with C. ($\mathbf{G}_1$)
- Ib.8 Send (ssid, C) to S ($\mathbf{G}_1$)
  and ($\textsc{CompleteInitC}$, ssid, 1) to $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_{10}$). Store $\langle C, \mathsf{ssid}, \mathsf{pk_C}, c\rangle$ ($\mathbf{G}_{13}$) .

Figure 14: Simulator $\textsc{Sim}^{\text{OPRF-PPKR}}$ for $\pi^{\text{OPRF-PPKR}}$, part 1.

$\textsc{Init}_{10}$ $= b$ is either the b value computed by the honest HSM or the value received by $\mathsf{ID_C}$ if S is fully corrupt;

$\textsc{Init}_{11}$ $\in \{C, \bot\}$ is the ciphertext C computed by $\mathsf{ID_C}$ or $\bot$ if $\mathsf{ID_C}$ is corrupt.

Similarly, the simulator works with the sollowing session state records

$$\langle\textsc{Rec}, \mathsf{ID_C}, \mathsf{kid}, \mathsf{ssid}, a, \mathit{match}, \mathsf{ID}^*_\mathsf{C}, \mathsf{kid}^*, a^*, \mathsf{pk_C}, b, c\rangle$$

in the recovery. Again, all values are initialized to $\bot$ and potentially updated throught a recovery. At the end of the recovery by $\mathsf{ID_C}$, the record has the following semantics.

$\textsc{Rec}_1$ $\in \{\mathsf{ID_C}, \bot\}$ is either the $\mathsf{ID_C}$ of an honest client running recovery or it is set to $\bot$ if the client is corrupt or if the corrupt S impersonates $\mathsf{ID_C}$;

$\textsc{Rec}_2$ $\in \{\mathsf{kid}, \bot\}$ is either the number of initializations started by $\mathsf{ID_C}$ when $\mathsf{ID_C}$ starts this recovery, as seen by the HSM, or it is set to $\bot$ if the client is corrupt or if the corrupt S impersonates $\mathsf{ID_C}$;

$\textsc{Rec}_3$ $=$ ssid indicates the sub-session id used by $\mathsf{ID_C}$;

$\textsc{Rec}_4$ $\in \{a, \bot\}$ is either the a' value sent by a honest $\mathsf{ID_C}$, or $\bot$ if $\mathsf{ID_C}$ is corrupt or if the corrupt S impersonates $\mathsf{ID_C}$.

$\textsc{Rec}_5$ $\in \{0, 1\}$ indicates whether the password in the recovery is correct

$\textsc{Rec}_6$ $\in \{\mathsf{ID}^*_\mathsf{C}, \bot\}$ is either the client name, delivered to the HSM, or set to $\bot$ if the server is fully corrupt;

$\textsc{Rec}_7$ $\in \{\mathsf{kid}^*, \bot\}$ is the number of initializations started by ID^*_C when S receives the first message in this recovery, as seen by the HSM, or set to $\bot$ if the server is fully corrupt;

$\textsc{Rec}_8$ $\in \{a^*, \bot\}$ indicates the a' value, delivered to the HSM, or set to $\bot$ if the server is fully corrupt;

$\textsc{Rec}_9$ $\in \{\mathsf{pk_C}, \bot\}$ is the signature public key used by the HSM in this recovery, or set to $\bot$ if the server is fully corrupt;

$\textsc{Rec}_{10}$ $= b'$ is either the b value computed by the honest HSM or the value received by $\mathsf{ID_C}$ if S is fully corrupt;

$\textsc{Rec}_{11}$ $\in \{c\}$ is the ciphertext c sent by the honest HSM or $\bot$ if S is fully corrupt.

We construct a sequence of hybrid games $\mathbf{G}_0$ to $\mathbf{G}_{14}$ where we gradually change the real-world execution of the protocol $\pi^{\text{OPRF-PPKR}}$ (interacting with the hybrid functionality $\mathcal{F}_{\text{HSM}}^{\text{OPRF-PPKR}}$) to reach the ideal-world execution, where the environment interacts with the simulator from Figures 14 to 17 and the ideal functionality $\mathcal{F}_{\text{PPKR}}$. We write $\Pr[\mathbf{G}_i]$ to denote the probability that the environment outputs 1 in the hybrid game $\mathbf{G}_i$.

Game $\mathbf{G}_0$: **Real world.** This is the real world.

Game $\mathbf{G}_1$: **Create simulator.** In this game we create two new entities called the ideal functionality $\mathcal{F}$ and the simulator $\textsc{Sim}$. Initially, $\mathcal{F}$ just forwards the input of the dummy parties to $\textsc{Sim}$ and outputs what $\textsc{Sim}$ instructs it to output. In particular, $\mathcal{F}$ has interfaces ($\textsc{InitC}$, ssid, $\mathsf{ID_C}$, pw), ($\textsc{InitS}$, ssid, $\mathsf{ID_C}$),($\textsc{RecC}$, ssid, $\mathsf{ID_C}$, pw$'$), and ($\textsc{RecS}$, ssid, $\mathsf{ID_C}$) that just forward the input to $\textsc{Sim}$. Additionally, $\mathcal{F}$ has the corruption and attack interfaces $\textsc{LeakFile}$, $\textsc{FullyCorrupt}$, $\textsc{MaliciousInit}$, $\textsc{MaliciousRec}$, and $\textsc{OfflineAttack}$ with the exact same code as in $\mathcal{F}_{\text{PPKR}}$. Note, however, that $\textsc{Sim}$ does not use these interfaces yet.

The simulator executes the code of all honest parties of the protocol internally on the input that it is provided by $\mathcal{F}$ and it internally runs the code of the hybrid functionality $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{OPRF\text{-}PPKR}}$. Additionally, it creates records exactly as $\textsc{Sim}^{\mathsf{OPRF\text{-}PPKR}}$ does, but at this point never uses the values from any of these records. Note that these are just syntactical changes and the protocol is still executed as in the real world. We have

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_0].$$

Game G_2: Output $\textsc{Fail}$ on tampered message from HSM. In this game, we change how the simulator reacts when an honest client receives a message from the HSM that was tampered with. Whenever an honest client receives a message ($\textsc{InitRes}$, ssid, $\textsc{Succ}$) while S is not fully corrupt, where $\textsc{Sim}$ never produced this message on behalf of S or $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{OPRF\text{-}PPKR}}$, then $\textsc{Sim}$ makes the client output $\textsc{Fail}$ (see Ib.3). Similarly, when an honest client receives a message ($\textsc{RecRes}$, ssid, out) that was never produced by $\textsc{Sim}$ on behalf of the HSM, then $\textsc{Sim}$ makes the client output $\textsc{Fail}$ (see Rb.5).

This game is identical to $\mathbf{G}_1$ unless $\mathcal{A}$ delivers a message to $\mathsf{ID_C}$ that was never output by $\textsc{Sim}$ but the attestation signature is still valid. If an environment $\mathcal{Z}^*$ exists that can distinguish between $\mathbf{G}_2$ and $\mathbf{G}_1$, we can construct an adversary $\mathcal{B}_1$ against the sEUF-CMA security of the signature scheme Sig used for attestation. $\mathcal{B}_1$ internally runs the whole experiment including $\mathcal{Z}^*$ and outputs the above described message with the signature to its challenger. We get

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq \mathbf{Adv}_{\mathsf{Sig},\mathcal{B}_1}^{\mathsf{sEUF\text{-}CMA}}(\lambda).$$

Game G_3: Simulate the OPRF execution. In this game, we replace the 2HashDH protocol with its ideal execution, namely functionality $\mathcal{F}_{\mathsf{OPRF}}^{\ell}$ of Figure 7 and simulator $\textsc{Sim}_{\mathsf{OPRF}}(\mathsf{sid}, H_1, H_2, N)$ of Figure 8. We consequently let the simulated parties use the interfaces of $\mathcal{F}_{\mathsf{OPRF}}^{\ell}$, e.g., instead of computing $a \leftarrow H_1(\mathsf{pw} \parallel \mathsf{ID_C})^r$, a party sends ($\textsc{Eval}$, sid, $\mathsf{ID_C} \parallel$ kid, ssid, S, pw $\parallel \mathsf{ID_C}$) to $\mathcal{F}_{\mathsf{OPRF}}^{\ell}$, where for the first initialization by $\mathsf{ID_C}$ we have kid $= 0$ and with each subsequent initialization by $\mathsf{ID_C}$ it is incremented by 1. Whenever a party is supposed to obtain a PRF value, the simulator calls the $\textsc{RcvComplete}$ interface of $\mathcal{F}_{\mathsf{OPRF}}^{\ell}$. Since in this ideal OPRF execution, OPRF keys are chosen by $\textsc{Sim}_{\mathsf{OPRF}}(\mathsf{sid}, H_1, H_2, N)$ (see Step 2 of Figure 8), the simulator takes these keys whenever it has to store a file. Finally, the 2HashDH protocol messages a, b sent by honest parties are now simulated by $\textsc{Sim}_{\mathsf{OPRF}}(\mathsf{sid}, H_1, H_2, N)$. The formal changes can be read from Figures 14 to 17, marked with ($\mathbf{G}_3$).

This and the previous game are indistinguishable up to the advantage of distinguishing the real 2HashDH execution from the ideal execution of $\mathcal{F}_{\mathsf{OPRF}}^{\ell}$ and $\textsc{Sim}_{\mathsf{OPRF}}(\mathsf{sid}, H_1, H_2, N)$, which by Theorem 4 (originally Theorem 2 of [DFG+23a]) is bounded by

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq q_{\textsc{Init}}\mathbf{Adv}_{\mathcal{B}_2,\mathbb{G}}^{(q_E+q_H,q_E)-\mathsf{OMDH}}(\lambda) + (q_E+q_H)^2/q,$$

where $q_{\textsc{Init}}$ is the number of $\textsc{Init}$ inputs, q_E the number of overall $\textsc{Init}$ and $\textsc{Rec}$ inputs, q_H the number of H_1 queries, q is the order of $\mathbb{G}$ and $\mathbf{Adv}_{\mathcal{B}_2,\mathbb{G}}^{(x,y)-\mathsf{OMDH}}(\lambda)$ denotes the advantage of breaking the one-more Diffie-Hellman problem in $\mathbb{G}$ (see Definition 4).

Game G_4: Abort upon ambiguous ciphertexts. We let the simulator abort if it obtains an adversarially-generated AE ciphertext c such that for two values k, k' from records $\langle F, \mathsf{S}, [\mathsf{kid}], k, *\rangle$, $\langle F, \mathsf{S}, [\mathsf{kid}'], k', *\rangle$ stored by the OPRF simulator $\textsc{Sim}_{\mathsf{OPRF}}$, c successfully decrypts under k and k', i.e., the output of AE.Dec is not $\perp$ (cf. IC.2(c), IC.7(c)(ii), and Rb.3(a)).

This and the previous game are indistinguishable by the random-key robustness of AE. The reduction is straightforward, running on two challenge keys k, k': it randomly chooses two occasions where $\mathcal{F}_{\mathsf{OPRF}}$ samples $F_{\mathsf{sid,S,kid}}(x) \overset{\$}{\leftarrow} \{0,1\}^{\ell}$ and instead sets $F_{\mathsf{sid,S,kid}}(x) \leftarrow k$ for the first one and $\leftarrow k'$ for the second. Note that this does not change the distribution as k, k' are uniformly random. Ciphertexts c passing the winning condition of the random-key robustness game can be detected by trial-decrypting with k and k'. We hence have

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_3]| \leq q_E^2\mathbf{Adv}_{\mathcal{B}_3,\mathsf{AE}}^{\mathsf{rkr}}(\lambda).$$

Game G_5: Honest $\mathsf{ID_C}$ fails upon malicious AE ciphertext. In this game, we modify how $\textsc{Sim}$ proceeds when receiving an adversarial message or honestly delivered message (ssid, b', c, $\mathsf{ID_C}$) from $\mathcal{A}$ to some honest $\mathsf{ID_C}$. $\textsc{Sim}$ checks whether all of the following conditions hold:

- c was not produced by $\textsc{Sim}$,
- There is no value $F_{\mathsf{sid,S},*}(\mathsf{pw}' \parallel \mathsf{ID_C})$ defined in $\mathcal{F}_{\mathsf{OPRF}}$ that successfully decrypts c,
- There is a value $F_{\mathsf{sid,S,[kid]}}(\mathsf{pw}' \| \mathsf{ID_C})$ defined in $\mathcal{F}_{\mathsf{OPRF}}$ that successfully decrypts c, but in this recovery a malicious S did not use the OPRF key with kid kid.

Formally, $\textsc{Sim}$ executes these checks via the code in Rb.3(a) and Rb.3(b). If all conditions hold, then $\textsc{Sim}$ lets $\mathsf{ID_C}$ immediately output ($\textsc{RecRes}$, ssid, $\textsc{Fail}$) instead of following the protocol as in $\mathbf{G}_4$. If any condition does not hold, it proceeds as in $\mathbf{G}_4$. Note that $\mathbf{G}_5$ only differs if $\mathsf{ID_C}$ outputs ($\textsc{RecRes}$, ssid, $\textsc{Succ}$) in $\mathbf{G}_4$ and ($\textsc{RecRes}$, ssid, $\textsc{Fail}$) in $\mathbf{G}_5$, which we denote by the event E_{AE}. We now construct an adversary $\mathcal{B}_4$ that breaks the INT-CTXT-security of AE given any environment $\mathcal{Z}$ that causes E_{AE}.

$\mathcal{B}_4$ acts like $\textsc{Sim}$ and chooses $i^* \overset{\$}{\leftarrow} \{1, \ldots, q_{\textsc{Init}}\}$. Let ID_C^* denote the $\mathsf{ID_C}$ that executes the i^*-th initialization. If ID_C^* is corrupt, $\mathcal{B}_4$ aborts. Otherwise, it does not compute c by encrypting $(K, \mathsf{sk_C})$ under ρ^* but instead submits $(K, \mathsf{sk_C})$ to its encryption oracle. Note that this implicitly programs ρ^* to the key k^* chosen by the INT-CTXT experiment. Since k^* is chosen uniformly at random and $\mathcal{F}_{\mathsf{OPRF}}$ chooses its outputs uniformly at random, this means that there is no change in the distribution of c.

In any subsequent recovery by ID_C^* before ID_C^* executes another initialization, $\mathcal{B}_4$ checks the conditions listed above, and if they hold, it outputs the c' received in that recovery to the INT-CTXT experiment. Further, if the file from the

```
Simulator $\textsc{Sim}^{\text{OPRF-PPKR}}$, part 2

On $(\text{ssid}, C)$ from $\mathcal{A}$ to S on behalf of $\text{ID}_C$: // any $\text{ID}_C$, honest S
  IC.1  Retrieve $\langle \textsc{Init}, *, *, \text{ssid}, *, *, [\text{kid}], *, [\text{sk}_{\text{Enc}}], [\text{sk}_C], *, [C'] \rangle$. $(\mathbf{G}_1)$
  IC.2  If $\text{ID}_C$ is corrupt: $(\mathbf{G}_8)$
        (a) Execute HSM code, $(\mathbf{G}_1)$
            where the entry $k_{\text{OPRF}}$ of the stored $\textsc{File}$ record is set to $\perp$, $(\mathbf{G}_3)$
            up to determining $out$ and denote the decryption by $(\text{ssid}', \text{pk}_C, c)$. $(\mathbf{G}_1)$
            If $out = \textsc{Fail}$, send $(\textsc{CompleteInitS}, \text{ssid}, 0)$ to $\mathcal{F}_{\text{PPKR}}$ and otherwise continue.
            $(\mathbf{G}_{11})$ // $\textsc{Sim}_{\text{OPRF}}$ has the OPRF key
        (b) Find a record $\langle H_2, [\text{pw}] \parallel *, *, [y] \rangle$ that is marked $\textsc{Consistent}$ such that
            $\perp \neq (K, \text{sk}_C) \leftarrow \text{AE.Dec}(y, c)$. Otherwise set $\text{pw} \leftarrow \perp$. $(\mathbf{G}_8)$
        (c) If more than one consistent record is found, abort the simulation. $(\mathbf{G}_4)$
        (d) Send input $(\textsc{InitC}, \text{ssid}, \text{pw})$ to $\mathcal{F}_{\text{PPKR}}$ on behalf of $\text{ID}_C$. $(\mathbf{G}_8)$
            On response $(\textsc{InitC}, \text{ssid}, \text{ID}_C)$ from $\mathcal{F}_{\text{PPKR}}$ send message
            $(\textsc{CompleteInitS}, \text{ssid}, 1)$ to $\mathcal{F}_{\text{PPKR}}$ $(\mathbf{G}_{11})$.
  IC.3  If $\text{ID}_C$ is honest, retrieve $\langle C, \text{ssid}, [\text{pk}_C], [c] \rangle$. $(\mathbf{G}_{13})$
        Store $\langle \textsc{File}, \text{ID}_C, \text{pk}_C, c, \perp, 10 \rangle$ overwriting any $\langle \textsc{File}, \text{ID}_C, *, *, *, * \rangle$ $(\mathbf{G}_1)$, and send
        message $(\textsc{CompleteInitS}, \text{ssid}, 1)$ to $\mathcal{F}_{\text{PPKR}}$. $(\mathbf{G}_{10})$

On $(\text{ssid}, C)$ from $\mathcal{A}$ to $\mathcal{F}_{\text{HSM}}^{\text{OPRF-PPKR}}$ on behalf of corrupt S: // any $\text{ID}_C$, corrupt S
  IC.4  Retrieve $\langle \textsc{Init}, [\text{ID}_C], *, \text{ssid}, [a], [\text{ID}_C^*], [\text{kid}], [a^*], [\text{sk}_{\text{Enc}}], [\text{sk}_C], *, [C'] \rangle$. $(\mathbf{G}_1)$
  IC.5  If $C' = \perp$, give input $(\textsc{InitS}, \text{ssid}, \text{ID}_C)$ to $\mathcal{F}_{\text{PPKR}}$. On response $(\textsc{InitS}, \text{ssid}, \text{ID}_C)$
        from $\mathcal{F}_{\text{PPKR}}$ continue below. $(\mathbf{G}_{10})$ // $C' \neq \perp$ implies that $b$ was deliverd to $\text{ID}_C$,
        where the simulator gave input $\textsc{InitS}$ to $\mathcal{F}_{\text{PPKR}}$
  IC.6  If $C = C'$, retrieve record $\langle C, [\text{ssid}], [\text{pk}_C], [c] \rangle$ $(\mathbf{G}_{13})$. Give input $(\textsc{CompleteInitS},$
        $\text{ssid}, 1)$ to $\mathcal{F}_{\text{PPKR}}$ $(\mathbf{G}_{10})$,
        record $\langle \textsc{File}, \text{ID}_C^*, \text{kid}, \text{pk}_C, \text{sk}_C, c \rangle$ overwriting any $\langle \textsc{File}, \text{ID}_C^*, *, *, *, * \rangle$, and send
        $(\textsc{InitRes}, \text{ssid}, \textsc{Succ})$ to S as output of the HSM. $(\mathbf{G}_1)$ // honest delivery of $C$, which
        also implies honest delivery of $a$ and $\text{ID}_C = \text{ID}_C^*$ as otherwise $C' = \perp$
  IC.7  If $C \neq C'$:
        (a) Search for a record $\langle C, [\text{ssid}'], [\text{pk}_C], [c] \rangle$. $(\mathbf{G}_{13})$
            If $\text{ssid}' \neq \text{ssid}$, send $(\textsc{InitRes}, \text{ssid}, \textsc{Fail})$ to S. Otherwise, record
            $\langle \textsc{File}, \text{ID}_C^*, \text{kid}, \text{pk}_C, \text{sk}_C, c \rangle$ overwriting any $\langle \textsc{File}, \text{ID}_C^*, *, *, *, * \rangle$. $(\mathbf{G}_1)$
        (b) If no such record $\langle C, *, *, * \rangle$ exists $(\mathbf{G}_{13})$, execute the HSM code on input
            $(\text{ssid}, C)$, $(\mathbf{G}_1)$
            where the entry $k_{\text{OPRF}}$ of the stored $\textsc{File}$ record is set to $\perp$, $(\mathbf{G}_3)$
            up to determining $out$ and denote the result of the decryption by $(\text{ssid}', \text{pk}_C, c)$.
            If $out = \textsc{Fail}$, send $(\textsc{InitRes}, \text{ssid}, \textsc{Fail})$ to S. If $out \neq \textsc{Fail}$, record
            $\langle \textsc{File}, \text{ID}_C^*, \text{kid}, \text{pk}_C, \text{sk}_C, c \rangle$ overwriting any $\langle \textsc{File}, \text{ID}_C^*, *, *, *, * \rangle$. $(\mathbf{G}_1)$
        (c) Search a record $\langle H_2, [\text{pw}] \parallel *, *, [y] \rangle$ that is marked $\textsc{Consistent}$ such that
            $\perp \neq (K, \text{sk}_C) \leftarrow \text{AE.Dec}(y, c)$. Overwrite the third to last entry of the $\textsc{Rec}$
            record retrieved above with $\text{sk}_C$. $(\mathbf{G}_8)$
            (i)   If no such record exists, send $(\textsc{MaliciousInit}, \text{ID}_C^*, \perp, \perp)$ to $\mathcal{F}_{\text{PPKR}}$. $(\mathbf{G}_8)$
                  // $\mathcal{Z}$ has not yet computed the PRF value for $a$
            (ii)  If more than one consistent record is found, abort the simulation. $(\mathbf{G}_4)$
            (iii) In all other cases, send $(\textsc{MaliciousInit}, \text{ssid}, \text{ID}_C^*, \text{pw}, K)$ to $\mathcal{F}_{\text{PPKR}}$. $(\mathbf{G}_8)$
                  // $\mathcal{Z}$ can decrypt $c$
        (d) Send $(\textsc{InitRes}, \text{ssid}, \textsc{Succ})$ to S as output of the HSM. $(\mathbf{G}_1)$

On random oracle query $H_1(\text{pw} \parallel \text{ID}_C)$ from $\mathcal{A}$:
  H1.1  Query $H_1(\text{pw} \parallel \text{ID}_C)$ to $\textsc{Sim}_{\text{OPRF}}$ ($\rightarrow$ Step 4 of $\textsc{Sim}_{\text{OPRF}}$). Return output. $(\mathbf{G}_3)$

On random oracle query $H_2(\text{pw} \parallel \text{ID}_C, x)$ from $\mathcal{A}$:
  H2.1  If a record $\langle H_2, \text{pw} \parallel \text{ID}_C, x, y \rangle$ exists, output $y$. $(\mathbf{G}_1)$
  H2.2  If there exists a record $\langle F, \text{S}, \text{ID}_C \parallel [\text{kid}], [k_{\text{OPRF}}], * \rangle$ in $\textsc{Sim}_{\text{OPRF}}$ such that $x =$
        $H_1(\text{pw} \parallel \text{ID}_C)^{k_{\text{OPRF}}}$: $(\mathbf{G}_{14})$
        (a) Retrieve $\langle \textsc{leaked}, \text{ID}_C, \text{kid}, [\text{sk}_C], [c], [i] \rangle$. If multiple exist, choose smallest
            $i$. $(\mathbf{G}_{10})$
        (b) Send $(\textsc{OfflineAttack}, \text{ID}_C, \text{pw}, i)$ to $\mathcal{F}_{\text{PPKR}}$. If it outputs $K \neq \textsc{Fail}$, $(\mathbf{G}_{10})$
            run $y \xleftarrow{\$} \textsc{Sim}_{\text{EQV}}(c, (K, \text{sk}_C))$, record $\langle H_2, \text{pw} \parallel \text{ID}_C, x, y \rangle$ and output $y$. $(\mathbf{G}_6)$
            Otherwise continue.
  H2.3  Query $H_2(\text{pw} \parallel \text{ID}_C, x)$ to $\textsc{Sim}_{\text{OPRF}}$, which triggers Step 8 of $\textsc{Sim}_{\text{OPRF}}$. Let $y$ denote
        its output. Record $\langle H_2, \text{pw} \parallel \text{ID}_C, x, y \rangle$. If $y$ was produced by $\mathcal{F}_{\text{OPRF}}$ mark the record
        as $\textsc{Consistent}$. Output $y$. $(\mathbf{G}_3)$

On $(\textsc{RecC}, \text{ssid}, \text{ID}_C, match)$ from $\mathcal{F}_{\text{PPKR}}$ // Any $\text{ID}_C$
  R.1  If no record $\langle \textsc{File}, \text{ID}_C, [\text{kid}], *, *, * \rangle$ exists, set $\text{kid} \leftarrow \perp$. $(\mathbf{G}_3)$
  R.2  Give input $(\textsc{Eval}, \text{sid}, \text{ID}_C \parallel \text{kid}, \text{ssid}, \text{S}, 0)$ to the simulated $\mathcal{F}_{\text{OPRF}}$ from the simulated
       $\text{ID}_C$. This triggers Step 5 of $\textsc{Sim}_{\text{OPRF}}$, which outputs $(\text{sid}, \text{ID}_C, \text{ssid}, a')$. $(\mathbf{G}_3)$
  R.3  If a record $\langle \textsc{Rec}, \perp, \perp, \text{ssid}, \perp, \perp, *, *, *, *, *, * \rangle$ exists, overwrite the second, third,
       fifth, and sixth entry with $(\text{ID}_C, \text{kid}, a', match)$. $(\mathbf{G}_1)$ // This record exists if $\textsc{RecS}$
       was executed previously
  R.4  Otherwise, record $\langle \textsc{Rec}, \text{ID}_C, \text{kid}, \text{ssid}, a', match, \perp, \perp, \perp, \perp, \perp, \perp \rangle$. $(\mathbf{G}_1)$
  R.5  Send $(\text{Rec}, \text{ssid}, a', \text{ID}_C)$ as message from $\text{ID}_C$ to S. $(\mathbf{G}_1)$

On $(\textsc{RecS}, \text{ssid}, \text{ID}_C, match)$ from $\mathcal{F}_{\text{PPKR}}$: // honest, corrupt, or fully corrupt S
  R.6  If a record $\langle \textsc{Rec}, *, *, \text{ssid}, *, \perp, *, *, *, *, *, * \rangle$ exists, overwrite $\perp$ with $match$. $(\mathbf{G}_1)$ //
       This record exists if $\textsc{RecC}$ was executed previously
  R.7  Otherwise, record $\langle \textsc{Rec}, \perp, \perp, \text{ssid}, \perp, match, \perp, \perp, \perp, \perp, \perp, \perp \rangle$. $(\mathbf{G}_1)$
  R.8  Wait for $(\text{Rec}, \text{ssid}, a', \text{ID}_C)$. $(\mathbf{G}_1)$ // See the $(\text{Rec}, \text{ssid}, a', \text{ID}_C)$ interface below.
```

Figure 15: Simulator $\textsc{Sim}^{\text{OPRF-PPKR}}$ for $\pi^{\text{OPRF-PPKR}}$, part 2.

i^*-th initialization was leaked, in any subsequent recovery by ID_C^*, $\mathcal{B}_4$ checks the same conditions and outputs the c' received in that recovery to the INT-CTXT experiment if they hold.

If such a recovery by ID_C^* is the first that causes the event E_{AE}, which happens with probability $1/q_{\textsc{Init}}$, $\mathcal{B}_4$ wins the INT-CTXT experiment due to the following. Let c^* be the ciphertext that causes E_{AE}, which means that in $\mathbf{G}_4$, ID_C^* would successfully decrypt c^* using ρ^*. Since $\mathcal{B}_4$ implicitly set $\rho^* = k^*$, c^* successfully decrypts under k^* and constitutes a forgery in the INT-CTXT experiment. Therefore, we have

$$|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| \leq q_{\textsc{Init}} \mathbf{Adv}_{\mathcal{B}_4,\text{AE}}^{\text{INT-CTXT}}(\lambda).$$

Simulator $\textsc{Sim}^{\text{OPRF-PPKR}}$, part 3.

On $(\mathsf{Rec}, \mathsf{ssid}, a', \mathsf{ID_C})$ from $\mathcal{A}$ to S on behalf of $\mathsf{ID_C}$: // any $\mathsf{ID_C}$, any (if jumping here from the $(\mathsf{Rec}, \mathsf{ssid}, a', \mathsf{ID_C})$ from $\mathcal{A}$ to $\mathcal{F}_{\text{HSM}}^{\text{OPRF-PPKR}}$ interface) S

Ra.1 If S is honest, wait for $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C}, \mathit{match})$ message from $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_1$) // See $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C}, \mathit{match})$ interface.

Ra.2 Retrieve $\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{kid}], [\mathsf{pk_C}], *, [c'] \rangle$. If no such record exists and S is honest, send $(\textsc{CompleteRecS}, \mathsf{ssid}, 0)$ to $\mathcal{F}_{\text{PPKR}}$ ($\mathbf{G}_{12}$)

Ra.3 Execute HSM code on input $(\mathsf{Rec}, \mathsf{ssid}, a', \mathsf{ID_C})$ up to determining the output ($\mathbf{G}_1$) and with the computation of b' substituted as follows. Give input $(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{ID_C} \parallel \mathsf{kid}, \mathsf{ssid})$ to the simulated $\mathcal{F}_{\text{OPRF}}$ from the simulated S. This triggers Step 6 of $\textsc{Sim}_{\text{OPRF}}$, which outputs $(\mathsf{sid}, \mathsf{ssid}, b')$. ($\mathbf{G}_3$)

Ra.4 If the HSM output is $\mathsf{ssid}, \textsc{DelRec}$, record $\langle \textsc{DelRec}, \mathsf{ssid} \rangle$ and send $(\mathsf{ssid}, \textsc{DelRec})$ as HSM-signed message from S to $\mathsf{ID_C}$. If the output is $\mathsf{ssid}, b', c, \mathsf{ID_C}$, overwrite the six last entries of $\langle \textsc{Rec}, *, *, \mathsf{ssid}, *, *, *, *, *, *, *, * \rangle$ with $\mathsf{ID_C}, \mathsf{kid}, a', \mathsf{pk_C}, b', c'$ and send $(\mathsf{ssid}, b', c', \mathsf{ID_C})$ as HSM-signed message to $\mathsf{ID_C}$. ($\mathbf{G}_1$)

On $(\mathsf{Rec}, \mathsf{ssid}, a', \mathsf{ID_C})$ from $\mathcal{A}$ to $\mathcal{F}_{\text{HSM}}^{\text{OPRF-PPKR}}$ on behalf of corrupt S:

Ra.5 If no record $\langle \textsc{Rec}, *, *, \mathsf{ssid}, *, *, *, *, *, *, *, * \rangle$ exists, create $\langle \textsc{Rec}, \perp, \perp, \mathsf{ssid}, \perp, \perp, \perp, \perp, \perp, \perp, \perp, \perp \rangle$. ($\mathbf{G}_1$)

Ra.6 Execute Steps Ra.2-Ra.4 except that the output is returned to S instead of sending it from S to $\mathsf{ID_C}$ ($\mathbf{G}_3$)

On $(\mathsf{ssid}, \textsc{DelRec})$ from $\mathcal{A}$ to $\mathsf{ID_C}$ on behalf of S: // honest $\mathsf{ID_C}$, any S

Del.1 If no record $\langle \textsc{Rec}, \mathsf{ID_C}, *, \mathsf{ssid}, *, *, *, *, *, *, *, * \rangle$ exists or if there already was a message $(\mathsf{ssid}, b', c, \mathsf{ID}_\mathsf{C}^*)$ to $\mathsf{ID_C}$, ignore. ($\mathbf{G}_1$)

Del.2 If S is not honest, send $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C}, 0, \textsc{DelRec}, 0)$ to $\mathcal{F}_{\text{PPKR}}$. On response $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C}, \mathit{match})$ from $\mathcal{F}_{\text{PPKR}}$ continue below. ($\mathbf{G}_{10}$)

Del.3 If HSM signature verifies, send $(\textsc{CompleteRecC}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\text{PPKR}}$ and otherwise $(\textsc{CompleteRecC}, \mathsf{ssid}, \mathsf{ID_C}, 0)$. ($\mathbf{G}_{10}$)

On $(\mathsf{ssid}, b', c, \mathsf{ID}_\mathsf{C}^*)$ from $\mathcal{A}$ to $\mathsf{ID_C}$ on behalf of S: // honest $\mathsf{ID_C}$, any S

Rb.1 Retrieve $\langle \textsc{Rec}, \mathsf{ID_C}, *, \mathsf{ssid}, [a'], [\mathit{match}], *, [\mathsf{kid}], *, *, *, * \rangle$. ($\mathbf{G}_1$)
If no such record exists or if there already was a message $(\mathsf{ssid}, \textsc{DelRec})$ to $\mathsf{ID_C}$, ignore. ($\mathbf{G}_1$) // $\mathsf{ID_C}$ did not start recovery or record deleted.

Rb.2 If S is corrupt, set $\mathsf{pw} \leftarrow \perp, K \leftarrow \perp, i \leftarrow \perp$. ($\mathbf{G}_{12}$)

Rb.3 If S is fully corrupt:

(a) If record $\langle H_2, [\mathsf{pw}] \parallel \mathsf{ID_C}, [u], [y] \rangle$ marked $\textsc{Consistent}$ exists ($\mathbf{G}_{14}$) and records $\langle \mathsf{ID_C} \parallel \mathsf{kid}, \mathsf{ssid}, \mathsf{ID_C}, [r'] \rangle$ and $\langle H_1, \mathsf{pw} \parallel \mathsf{ID_C}, [r] \rangle$ exist in $\textsc{Sim}_{\text{OPRF}}$ s.t. $\perp \neq (K, \mathsf{sk_C}) \leftarrow \mathsf{AE.Dec}(y, c)$, and $b'^{1/r'} = u^{1/r}$, ($\mathbf{G}_5$)
set $i \leftarrow 0$. ($\mathbf{G}_{10}$)
If more than one consistent record is found, abort the simulation. ($\mathbf{G}_4$) // Adversarial OPRF key

(b) Otherwise, if records $\langle \textsc{leaked}, \mathsf{ID_C}, [\mathsf{kid}'], [\mathsf{sk_C}], c, [i] \rangle$ and $\langle \textsc{Init}, \mathsf{ID_C}, \mathsf{kid}', *, [a], *, *, *, *, *, [b], * \rangle$ exist and a record $\langle F, \mathsf{S}, *, [k], * \rangle$ exists in $\textsc{Sim}_{\text{OPRF}}$ such that $a^k = b$ and $a'^k = b'$, ($\mathbf{G}_5$)
set $\mathsf{pw} \leftarrow \perp, K \leftarrow \perp$. ($\mathbf{G}_{10}$) // Impersonation with old file

(c) Otherwise, set $\mathsf{pw} \leftarrow 0, K \leftarrow \textsc{Fail}, i \leftarrow 0, \mathsf{sk_C} \leftarrow \perp$. ($\mathbf{G}_{10}$)

Rb.4 If S is not honest, give input $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID}_\mathsf{C}^*, \mathsf{pw}, K, i)$ to $\mathcal{F}_{\text{PPKR}}$. On response $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID}_\mathsf{C}^*, \mathit{match})$ overwrite the sixth entry in the record retrieved in Step 1 with match. ($\mathbf{G}_{10}$) // Ensures that we can produce output for $\mathsf{ID_C}$

Rb.5 If S is not fully corrupt and $(\mathsf{ssid}, b', c, \mathsf{ID}_\mathsf{C}^*)$ was never output by $\textsc{Sim}^{\text{OPRF-PPKR}}$ on behalf of S or $\mathcal{F}_{\text{HSM}}^{\text{OPRF-PPKR}}$, or if S is fully corrupt and the attestation signature does not verify, send $(\textsc{CompleteInitC}, \mathsf{ssid}, 0)$ to $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_{10}$)

Rb.6 If $\mathsf{ID_C} \neq \mathsf{ID}_\mathsf{C}^*$, send $(\textsc{CompleteRecC}, \mathsf{ssid}, 0)$ to $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_{10}$) // $\mathsf{ID_C}$'s a rerouted

Rb.7 If S is not fully corrupt, retrieve $\langle \textsc{File}, \mathsf{ID_C}, \mathsf{kid}, *, [\mathsf{sk_C}], * \rangle$. ($\mathbf{G}_{14}$)

Rb.8 If $\mathit{match} = 1$ and $\mathsf{sk_C} \neq \perp$, compute $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sk_C}, (a', \mathsf{ID_C}, \mathsf{ssid}, b', c))$, send (ssid, σ) as message from $\mathsf{ID_C}$ to S ($\mathbf{G}_{14}$)
and send $(\textsc{CompleteRecC}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_{10}$)

Rb.9 If $\mathit{match} = 0$ or $\mathsf{sk_C} = \perp$, ($\mathbf{G}_{14}$)
send $(\textsc{CompleteRecC}, \mathsf{ssid}, 0)$ to $\mathcal{F}_{\text{PPKR}}$. ($\mathbf{G}_{10}$)

Figure 16: Simulator $\textsc{Sim}^{\text{OPRF-PPKR}}$ for $\pi^{\text{OPRF-PPKR}}$, part 3.

Game $\mathbf{G}_6$: Simulate ciphertexts of honest initializations without passwords. In this game, we change how $\textsc{Sim}$ computes the AE ciphertext c in initializations of honest clients. Now, when $\textsc{Sim}$ receives a message $(\mathsf{ssid}, b, \mathsf{ID_C}, \mathsf{pk_{Enc}})$ and $\mathcal{A}$ has not yet queried $(\mathsf{pw} \parallel \mathsf{ID_C}, b^{1/r})$ to the random oracle H_2, $\textsc{Sim}$ does not compute $c \xleftarrow{\$} \mathsf{AE.Enc}(\rho, (K, \mathsf{sk_C}))$. Instead it uses the equivocability simulator of AE and runs $c \xleftarrow{\$} \textsc{Sim}_{\text{EQV}}(\lambda + |\mathsf{sk_C}|)$. Note that this can only happen with non-negligible probability if S is fully corrupt as otherwise $\mathcal{A}$ would have to guess the OPRF key used by $\textsc{Sim}_{\text{OPRF}}$.

Then, $\textsc{Sim}$ records $\langle \mathsf{AE}, c, \mathsf{pw} \parallel \mathsf{ID_C}, b^{1/r}, K, \mathsf{sk_C} \rangle$. Recall that $\textsc{Sim}$ still receives pw as input and can create these records. We emphasize that the creation of these records is only a temporary change that will be removed later in $\mathbf{G}_{14}$ and any step introduced in this game that relies on these records will be changed later to work without them (cf. $\mathbf{G}_{10}$ and $\mathbf{G}_{14}$).

We now use the AE records to ensure that recovery still works correctly. To that end, we change $\textsc{Sim}$ such that when it receives a message $(\mathsf{ssid}, b', c, \mathsf{ID_C})$ in a recovery of an honest $\mathsf{ID_C}$, $\textsc{Sim}$ retrieves the record $\langle \mathsf{AE}, c, \mathsf{pw}' \parallel \mathsf{ID_C}, b'^{1/r'}, [K], [\mathsf{sk_C}] \rangle$. If such a record exists, c was produced $\textsc{Sim}_{\text{EQV}}$, and $\textsc{Sim}$ proceeds using K and $\mathsf{sk_C}$ from that record. If no such record exists, then c was computed by $\mathcal{Z}$, and $\textsc{Sim}$ checks whether $\mathsf{ID_C}$ can decrypt it. For this, it again checks the conditions from $\mathbf{G}_5$. If they hold, $\textsc{Sim}$ let's $\mathsf{ID_C}$ output $\textsc{Fail}$ and otherwise decrypts c using $F_{\mathsf{sid},\mathsf{S},\mathsf{ID_C} \parallel *}(\mathsf{pw}' \parallel \mathsf{ID_C})$ to obtain K and $\mathsf{sk_C}$. Note that the output behavior of $\mathsf{ID_C}$ here is the same as in $\mathbf{G}_5$. Formally, all these changes that do not rely on the AE records are done in Ib.6(c) and Ib.7(a).

To ensure that $\mathcal{Z}$ cannot distinguish $\mathbf{G}_6$ from $\mathbf{G}_5$, $\textsc{Sim}$ additionally has to properly program the random oracle H_2 to account for password guessing. For each query $(\mathsf{pw} \| \mathsf{ID_C}, y)$ to H_2, it retrieves the record $\langle \mathsf{AE}, [c], \mathsf{pw} \| \mathsf{ID_C}, y, [K], [\mathsf{sk_C}] \rangle$, then runs $\rho \leftarrow \textsc{Sim}_{\text{EQV}}(c, (K, \mathsf{sk_C}))$, and sets $H_2(\mathsf{pw}, y) \leftarrow \rho$ (H2.2(b)). That means, whenever $\mathcal{A}$ queries H_2 on the input that would yield the key for c in $\mathbf{G}_5$, $\textsc{Sim}$ now equivocates c to the matching message and programs H_2 accordingly. If no such record exists, $\textsc{Sim}$ forwards the query to the random oracle H_2 of $\textsc{Sim}_{\text{OPRF}}$ as in $\mathbf{G}_5$.

We claim that for any $\mathcal{Z}$ that is able to distinguish $\mathbf{G}_6$ and $\mathbf{G}_5$, we can construct an adversary $\mathcal{B}_5$ that wins the EQV experiment for AE. Since in $\mathbf{G}_5$, ρ is chosen by $\mathcal{F}_{\text{OPRF}}$ uniformly at random and c is encrypted under ρ, the distribution of ρ and c in $\mathbf{G}_5$ is exactly the same as in the real game of the EQV experiment. In $\mathbf{G}_6$, c is output by $\textsc{Sim}_{\text{EQV}}$ and thus its distribution is exactly the same as in the ideal game of the EQV experiment. The distribution of ρ in $\mathbf{G}_6$ remains unchanged unless there is a query to H_2 such that $\textsc{Sim}$ equivocates c. In that case ρ is distributed exactly the same as in the ideal world in the EQV experiment as it is computed by $\textsc{Sim}_{\text{EQV}}$. Thus, we have

$$|\Pr[\mathbf{G}_6] - \Pr[\mathbf{G}_5]| \leq \mathbf{Adv}^{\text{EQV}}_{\mathcal{B}_5,\mathsf{AE}}(\lambda).$$

Game $\mathbf{G}_7$: Abort upon signature forgery. In this game, we let $\textsc{Sim}$ abort upon observing a forged signature. Concretely, if $\textsc{Sim}$ receives a message $(\mathsf{ssid}, \sigma^*)$ from $\mathcal{A}$ to S on behalf of $\mathsf{ID_C}$ then $\textsc{Sim}$ aborts if all of the following conditions hold (cf. σ.1):

- There is a record $\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{pk_C}], *, *, * \rangle$ such that $\mathsf{Sig.Vfy}(\mathsf{pk_C}, (a', \mathsf{ID_C}, \mathsf{ssid}, b', c), \sigma^*) = 1$,
- The signed c was not equivocated using $\textsc{Sim}_{\text{EQV}}$, i.e., $\textsc{Sim}$ never computed $\rho \leftarrow \textsc{Sim}_{\text{EQV}}(c, (K, \mathsf{sk_C}))$,
- σ^* was not computed by $\textsc{Sim}$ on behalf of an honest $\mathsf{ID_C}$.

Similarly, if Sim receives a message $(\mathsf{ssid}, \sigma^*)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{OPRF\text{-}PPKR}}$ on behalf of S then Sim aborts if all of the above conditions hold (cf. $\sigma.2$). We call this event E_{Sig}. If $\mathcal{A}$ can provoke E_{Sig} to happen, then we can use $\mathcal{A}$ to construct an adversary $\mathcal{B}_6$ against the sEUF-CMA security of Sig. $\mathcal{B}_6$ internally runs $\mathcal{Z}$ and plays the role of Sim and $\mathcal{F}$ in the execution of the protocol. $\mathcal{B}_6$ starts by randomly choosing $i^* \in \{1, \ldots, q_{\mathrm{INIT}}\}$, where q_{INIT} is the number of initializations. Let ID_C^* denote the ID_C that exeutes the i^*-th initialization. If ID_C^* is honest, then $\mathcal{B}_6$ does not generate a signing key pair $(\mathsf{sk}_\mathsf{C}, \mathsf{pk}_\mathsf{C})$ but instead uses the public key pk^* that it gets from its challenger. Since it does not know the key sk^*, it stores $\perp$ instead of sk_C in the AE record in the i^*-th initialization.

If ID_C^* is corrupt, then $\mathcal{B}_6$ aborts. Further, if $\mathcal{A}$ makes an H_2 query such that Sim would equivocate c^* by computing $\rho \leftarrow \mathrm{Sim}_{\mathsf{EQV}}(c^*, (K, \mathsf{sk}_\mathsf{C}))$, where c^* is the AE ciphertext produced in the i^*-th initialization, then $\mathcal{B}_6$ also aborts.

Now, whenever ID_C^* executes a recovery, $\mathcal{B}_6$ checks two conditions:

- ID_C^* receives a message $(\mathsf{ssid}, b', c^*, \mathsf{ID}_\mathsf{C}^*)$, where c^* is the ciphertext it produced in the i^*-th initialization.
- The recovery is successful (in this game, $\mathcal{B}_6$ still gets the client input pw' and can check $\mathsf{pw} = \mathsf{pw}'$).

If both conditions are satisfied, then $\mathcal{B}_6$ does not compute $\mathsf{Sig.Sign}(\mathsf{sk}_\mathsf{C}, (a', \mathsf{ID}_\mathsf{C}, \mathsf{ssid}, b', c^*))$ but instead uses its $\mathsf{Sign}(\cdot)$ oracle provided by its challenger to get a signature σ on the message $m = (a', \mathsf{ID}_\mathsf{C}, \mathsf{ssid}, b', c^*)$. If any condition does not hold, $\mathcal{B}_6$ proceeds just like Sim in $\mathbf{G}_6$. Now, $\mathcal{B}_6$ continues the simulation and observes all (ssid, σ) messages that it receives from $\mathcal{A}$. If there is a message such that E_{Sig} happens, then $\mathcal{B}_6$ outputs σ.

Note that the view of $\mathcal{Z}$ in the reduction did not change with respect to $\mathbf{G}_7$. The public key pk_C and the signatures σ are distributed exactly as in $\mathbf{G}_6$. In the i^*-th initialization, $\mathcal{B}_6$ did not use sk_C, as it simulated $c^* \xleftarrow{\$} \mathrm{Sim}_{\mathsf{EQV}}(\lambda + |\mathsf{sk}_\mathsf{C}|)$ without sk_C as in $\mathbf{G}_6$. $\mathcal{B}_6$ also did *not* have to equivocate c^* with $\rho \leftarrow \mathrm{Sim}_{\mathsf{EQV}}(c^*, (K, \mathsf{sk}_\mathsf{C}))$ (which it would not be able to do). If the first signature σ^* that caused the event E_{Sig} is valid under pk^*, which happens with probability at least $1/q_{\mathrm{INIT}}$, $\mathcal{B}_6$ wins. Overall, we get

$$|\Pr[\mathbf{G}_7] - \Pr[\mathbf{G}_6]| \le q_{\mathrm{INIT}} \mathbf{Adv}_{\mathcal{B}_6, \mathsf{Sig}}^{\mathsf{sEUF\text{-}CMA}}(\lambda).$$

Game $\mathbf{G}_8$: Extract from malicious initialization. In this game, we change the behaviour of Sim whenever it receives a message (ssid, C) from a corrupt ID_C or the corrupt S queries (ssid, C) to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{OPRF\text{-}PPKR}}$. For this, we first modify $\mathcal{F}$ in the following way. We change the interface $(\mathrm{InitC}, \mathsf{ssid}, \mathsf{pw})$ such that it acts exactly as $\mathcal{F}_{\mathsf{PPKR}}$ if and only if the ID_C that makes the query is corrupt. Further, we add an interface $(\mathrm{CompleteInitS}, \mathsf{ssid}, \mathsf{ID}_\mathsf{C}, \mathsf{pw}, K)$ that executes the steps CIS.3 and CIS.4 of $\mathcal{F}_{\mathsf{PPKR}}$. Again, the addition of this interface is only a temporary change and it is removed again in $\mathbf{G}_{11}$.

When receiving a message (ssid, C) from a corrupt party, Sim executes the protocol as in $\mathbf{G}_7$ and then searches for a record $\langle H_2, \mathsf{pw} \parallel *, *, y\rangle$ such that c decrypts successfully to some $(K, \mathsf{sk}_\mathsf{C})$ under y. Note that as of game $\mathbf{G}_4$, there is at most one such y. If no such record exists, it sets $K \leftarrow \perp, \mathsf{pw} \leftarrow \perp$.

If the message (ssid, C) came from a corrupt ID_C, Sim gives the input $(\mathrm{InitC}, \mathsf{ssid}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{PPKR}}$ on behalf of ID_C and sends $(\mathrm{CompleteInitS}, \mathsf{ssid}, \mathsf{ID}_\mathsf{C}, \mathsf{pw}, K)$ to $\mathcal{F}_{\mathsf{PPKR}}$. On the other hand, if (ssid, C) came from the corrupt S, Sim queries $(\mathrm{MaliciousInit}, \mathsf{ssid}, \mathsf{ID}_\mathsf{C}, \mathsf{pw}, K)$ to $\mathcal{F}_{\mathsf{PPKR}}$, where ID_C is the value the corrupt S sent in the first message of the subsession ssid. Formally, the changes can be read from IC.2 (Step IC.2(b) and the first part of Step IC.2(d)) and IC.7(c) (Steps IC.7(c)(i) and IC.7(c)(iii)).

The changes introduced in this game are only syntactical and do not affect any output of Sim. Essentially we only let $\mathcal{F}$ store some records for initializations executed by a corrupt party, but $\mathcal{F}$ never uses these records yet to produce an output for any party. Hence, we have

$$\Pr[\mathbf{G}_8] = \Pr[\mathbf{G}_7].$$

Game $\mathbf{G}_9$: Extract from malicious Recoveries. We change the simulator whenever it receives message (ssid, σ) from a corrupt ID_C to S, or from a corrupt S to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{OPRF\text{-}PPKR}}$. As of game $\mathbf{G}_7$, we know that if σ verifies under pk_C stored in the file of ID_C, either the adversary previously initialized (on behalf of corrupt ID_C or a corrupt S) that file record or it guessed the password of ID_C. As in the previous game, we let the simulator extract pw from that malicious initialization or password guess and submit $(\mathrm{RecC}, \mathsf{ssid}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{PPKR}}$ on behalf of corrupt ID_C (see $\sigma.2$(a) and $\sigma.2$(b)), resp. $(\mathrm{MaliciousRec}, \mathsf{ID}_\mathsf{C}^*, \mathsf{pw}')$ for a corrupt S (see $\sigma.8$). Because the changes in the simulation again only affect the state of $\mathcal{F}_{\mathsf{PPKR}}$ which is not yet influencing protocol outputs, the change is only syntactical, and we have

$$Pr[\mathbf{G}_9] = \Pr[\mathbf{G}_8].$$

Game $\mathbf{G}_{10}$: Let $\mathcal{F}$ produce the output of the client. In this game, we change the simulator such that the output for honest clients is generated by $\mathcal{F}$. To this end we have to introduce several changes to $\mathcal{F}$. We modify the interfaces $(\mathrm{InitC}, \mathsf{ssid}, \mathsf{pw})$ and $(\mathrm{RecC}, \mathsf{ssid}, \mathsf{pw})$ to act exactly like $\mathcal{F}_{\mathsf{PPKR}}$, except that they still forward all inputs to Sim. Next, we add the interfaces $(\mathrm{InitS}, \mathsf{ssid}, \mathsf{ID}_\mathsf{C})$, $(\mathrm{RecS}, \mathsf{ssid}, \mathsf{ID}_\mathsf{C})$, $(\mathrm{CompleteInitC}, \mathsf{ssid}, b_C)$, and $(\mathrm{CompleteRecC}, \mathsf{ssid}, b_C)$ and let them act exactly like in $\mathcal{F}_{\mathsf{PPKR}}$. Finally, we introduce the interfaces $(\mathrm{CompleteInitS}, \mathsf{ssid}, b_S)$ and $(\mathrm{CompleteRecS}, \mathsf{ssid}, b_S)$, which both act as in $\mathcal{F}_{\mathsf{PPKR}}$, except that they never give any output to S.

Furthermore, we change Sim to always use the interfaces CompleteInitC and CompleteRecC of $\mathcal{F}$ whenever it wants to produce output of some honest ID_C with b_C set appropriately (Ib.3, Ib.4, Ib.8, Del.3, Rb.5, Rb.6, Rb.8, last step, and Rb.9, last step). Note that these interfaces only produce output if both InitC and InitS, resp. RecC and RecS, were queried previously and thus require Sim to give the inputs to $\mathcal{F}$ on behalf of S if S is not honest (Ib.2, Del.2, Rb.4). Additionally, Sim has to ensure that $\mathcal{F}$ internally creates File records that can be accessed during recoveries.

```
Simulator $\textsc{Sim}^{\text{OPRF-PPKR}}$, part 4.

On $(\mathsf{ssid}, \sigma)$ from $\mathcal{A}$ to S on behalf of $\mathsf{ID_C}$: // any $\mathsf{ID_C}$, honest S
  σ.1 Abort if all of the following conditions hold: ($\mathbf{G}_7$)
      • There is either a record $\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{pk_C}], *, *, * \rangle$ or Sim leaked $\mathsf{pk_C}$ to $\mathcal{A}$ such
        that $\mathsf{Sig.Vfy}(\mathsf{pk_C}, (a', \mathsf{ID_C}, \mathsf{ssid}, b', c), \sigma^*) = 1$, ($\mathbf{G}_7$)
      • The signed $c$ was not equivocated using $\textsc{Sim}_{\mathsf{EQV}}$, i.e., Sim never computed
        $\rho \leftarrow \textsc{Sim}_{\mathsf{EQV}}(c, (K, \mathsf{sk_C}))$, ($\mathbf{G}_7$)
      • Sim never output $\sigma$ on behalf of an honest $\mathsf{ID_C}$. ($\mathbf{G}_7$)
  σ.2 If $\mathsf{ID_C}$ is corrupt, do the following.
      (a) Retrieve $\langle \textsc{Rec}, *, *, \mathsf{ssid}, *, *, \mathsf{ID_C}, *, [a'], [\mathsf{pk_C}], [b'], [c] \rangle$. If a record
          $\langle \textsc{DelRec}, \mathsf{ssid} \rangle$ exists, ignore. ($\mathbf{G}_1$)
          Determine pw′ as follows:
            • If $\mathsf{Sig.Vfy}(\mathsf{pk_C}, (a', \mathsf{ID_C}, \mathsf{ssid}, b', c), \sigma) = 0$, set $\mathsf{pw}' = \perp$. ($\mathbf{G}_9$)
            • Otherwise, search for a record $\langle H_2, [\mathsf{pw}'] \| *, *, [y] \rangle$ marked Consistent
              such that $\perp \neq (K, \mathsf{sk_C}) \leftarrow \mathsf{AE.Dec}(y, c)$. If noch such a record exists, set
              $\mathsf{pw}' = \perp$. ($\mathbf{G}_9$)
      (b) Give input $(\textsc{RecC}, \mathsf{ssid}, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{PPKR}}$ on behalf of $\mathsf{ID_C}$. On response
          $(\textsc{RecC}, \mathsf{ssid}, \mathsf{ID_C}, \mathit{match})$ from $\mathcal{F}_{\mathsf{PPKR}}$ continue below. ($\mathbf{G}_9$)
      (c) Execute HSM code on input $(\mathsf{ssid}, \sigma)$ up to determining $\mathit{out}$. ($\mathbf{G}_1$)
          If $\mathit{out} = \textsc{Fail}$, send message $(\textsc{CompleteRecS}, \mathsf{ssid}, 0)$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_{12}$), and oth-
          erwise $(\textsc{CompleteRecS}, \mathsf{ssid}, 1)$ ($\mathbf{G}_{10}$)
  σ.3 If $\mathsf{ID_C}$ is honest, execute HSM code on input $(\mathsf{ssid}, \sigma)$ and
      send message $(\textsc{CompleteRecS}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\mathsf{PPKR}}$. ($\mathbf{G}_{12}$) // If $\mathsf{ID_C}$ is honest, we
      only simulate $\sigma$ if $\mathit{match} = 1$ and do not need to check again here. Due to $\mathsf{ID_C}$-
      authentication the adversary also cannot inject any messages

On $(\mathsf{ssid}, \sigma)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\text{OPRF-PPKR}}$ on behalf of corrupt S: // any $\mathsf{ID_C}$
  σ.4 Abort if all of the following conditions hold: ($\mathbf{G}_7$)
      (a) There is either a record $\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{pk_C}], *, *, * \rangle$ or Sim leaked $\mathsf{pk_C}$ to $\mathcal{A}$ such
          that $\mathsf{Sig.Vfy}(\mathsf{pk_C}, (a', \mathsf{ID_C}, \mathsf{ssid}, b', c), \sigma^*) = 1$, ($\mathbf{G}_7$)
      (b) The signed $c$ was not equivocated using $\textsc{Sim}_{\mathsf{EQV}}$, i.e., Sim never computed
          $\rho \leftarrow \textsc{Sim}_{\mathsf{EQV}}(c, (K, \mathsf{sk_C}))$, ($\mathbf{G}_7$)
      (c) Sim never output $\sigma$ on behalf of an honest $\mathsf{ID_C}$. ($\mathbf{G}_7$)
  σ.5 Retrieve $\langle \textsc{Rec}, [\mathsf{ID_C}], *, \mathsf{ssid}, *, [\mathit{match}], [\mathsf{ID_C^*}], *, [a'], [\mathsf{pk_C}], [b'], [c] \rangle$. If a record
      $\langle \textsc{DelRec}, \mathsf{ssid} \rangle$ exists, ignore. Execute HSM code on input $(\mathsf{ssid}, \sigma)$ up to determining
      $\mathit{out}$. ($\mathbf{G}_1$)
  σ.6 If $\mathit{match} = \perp$, give input $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C^*}, \perp, \perp, \perp)$ to $\mathcal{F}_{\mathsf{PPKR}}$. On response
      $(\textsc{RecS}, \mathsf{ssid}, \mathsf{ID_C^*}, \mathit{match})$ from $\mathcal{F}_{\mathsf{PPKR}}$ continue below. ($\mathbf{G}_{10}$) // $\mathit{match} = \perp$ indicates
      that RecS input was not given to $\mathcal{F}_{\mathsf{PPKR}}$ yet, which is necessary to decrease counter
  σ.7 If $\mathit{out} = \textsc{Fail}$, send $(\textsc{RecRes}, \mathsf{ssid}, \textsc{Fail})$ to S. ($\mathbf{G}_1$)
  σ.8 If $\mathit{out} = \textsc{Succ}$, search for a record $\langle H_2, [\mathsf{pw}'] \| *, *, [y] \rangle$ marked Consistent such
      that $\perp \neq (K, \mathsf{sk_C}) \leftarrow \mathsf{AE.Dec}(y, c)$, otherwise set $\mathsf{pw}' \leftarrow \perp$. Then do: ($\mathbf{G}_9$) // need
      to reset ctr in $\mathcal{F}_{\mathsf{PPKR}}$
      (a) // both messages in this recovery come from corrupt S If $\mathsf{ID_C} = \perp$, give input
          $(\textsc{MaliciousRec}, \mathsf{ID_C^*}, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{PPKR}}$. ($\mathbf{G}_9$)
          Independent of the response from $\mathcal{F}_{\mathsf{PPKR}}$, send $(\textsc{RecRes}, \mathsf{ssid}, \textsc{Succ})$ to S. ($\mathbf{G}_1$)
          // We give the same password as in the MaliciousInit query, which ensures
          that the counter is reset
      (b) // $a'$ message came from $\mathsf{ID_C}$, $\sigma$ from corrupt S If $\mathsf{ID_C} \neq \perp$:
            • If $\mathsf{ID_C}$ is corrupt, input $(\textsc{RecC}, \mathsf{ssid}, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{PPKR}}$ ($\mathbf{G}_9$)
            • Give input $(\textsc{CompleteRecS}, \mathsf{ssid}, 1)$ to $\mathcal{F}_{\mathsf{PPKR}}$ and send
              $(\textsc{RecRes}, \mathsf{ssid}, \textsc{Succ})$ to S. ($\mathbf{G}_{10}$)

On GetPK from anyone to $\mathcal{F}_{\mathsf{HSM}}^{\text{OPRF-PPKR}}$:
  pk.1 Retrieve $\langle PK, [\mathsf{pk}], * \rangle$ and return pk. ($\mathbf{G}_1$)

On LeakFile from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\text{OPRF-PPKR}}$ on behalf of S:
  LF.1 Send (LeakFile) to $\mathcal{F}_{\mathsf{PPKR}}$ to obtain $\mathbf{ID_C}$. ($\mathbf{G}_{10}$)
  LF.2 $L \leftarrow \emptyset$. For each $(\mathsf{ID_C}, \mathit{ctr})$ in $\mathbf{ID_C}$: ($\mathbf{G}_{10}$)
       (a) Retrieve $\langle \textsc{File}, \mathsf{ID_C}, [\mathsf{kid}], [\mathsf{pk_C}], [\mathsf{sk_C}], [c] \rangle$. ($\mathbf{G}_{10}$)
       (b) Retrieve $\langle F, \mathsf{S}, \mathsf{ID_C} \| \mathsf{kid}, [k_{\mathsf{OPRF}}], * \rangle$ in OPRF simulator. ($\mathbf{G}_{10}$)
       (c) If no record $\langle \textsc{leaked}, \mathsf{ID_C}, *, *, *, * \rangle$ exists, record $\langle \textsc{leaked}, \mathsf{ID_C}, \mathsf{kid}, \mathsf{sk_C}, c, 1 \rangle$.
           Otherwise, record $\langle \textsc{leaked}, \mathsf{ID_C}, \mathsf{kid}, \mathsf{sk_C}, c, i+1 \rangle$ where $i \in \mathbb{N}$ is the
           biggest number such that a record $\langle \textsc{leaked}, \mathsf{ID_C}, *, *, *, i \rangle$ exists. Append
           $(\mathsf{ID_C}, \mathsf{pk_C}, c, k_{\mathsf{OPRF}}, \mathit{ctr})$ to $L$. ($\mathbf{G}_{10}$)
  LF.3 Output $L$ to $\mathcal{A}$. ($\mathbf{G}_{10}$)

On FullyCorrupt from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\text{OPRF-PPKR}}$ on behalf of S: ($\mathbf{G}_{10}$)
  FC.1 Send (FullyCorrupt, S) to $\mathcal{F}_{\mathsf{PPKR}}$, which outputs $\mathbf{ID_C}$. ($\mathbf{G}_{10}$)
  FC.2 Execute Step 2 of LeakFile using $\mathbf{ID_C}$. ($\mathbf{G}_{10}$)
  FC.3 Retrieve $\langle PK, *, [\mathsf{sk}] \rangle$ and output sk. ($\mathbf{G}_{10}$)
```

Figure 17: Simulator $\textsc{Sim}^{\text{OPRF-PPKR}}$ for $\pi^{\text{OPRF-PPKR}}$, part 4.

For this reason, whenever $\mathcal{A}$ sends some (ssid, C) from an honest $\mathsf{ID_C}$, Sim sends $(\textsc{CompleteInitS}, \mathsf{ssid}, 1)$ to $\mathcal{F}$ (IC.3, IC.6), once again first giving the InitS input to $\mathcal{F}$ if S is not honest to ensure that the CompleteInitS query proceeds (IC.5). Lastly, Sim acts exactly like $\textsc{Sim}^{\text{OPRF-PPKR}}$ on the queries LeakFile and (FullyCorrupt, S) by $\mathcal{A}$ (LF.1-LF.3, FC.1-FC.3).

A difficulty for Sim resulting from these changes is that for honest clients, K is now chosen by $\mathcal{F}$ instead of Sim and is unknown to Sim. In particular, this affects the changes introduced in $\mathbf{G}_6$ as Sim cannot use the records $\langle \mathsf{AE}, \ldots \rangle$ anymore in $\mathbf{G}_{10}$ to store and obtain K. Instead, whenever Sim equivocates some ciphertext c during a query to H_2, it has to extract K from $\mathcal{F}$. Note that $\mathcal{A}$ can only make a query that requires Sim to equivocate if the corresponding file containing c was leaked or if S is fully corrupt as otherwise $\mathcal{A}$ does not know the OPRF key used by $\textsc{Sim}_{\mathsf{OPRF}}$. This means that in $\mathcal{F}$ there is now a leaked record that Sim can address in the OfflineAttack interface (H2.2(a) and the first part of H2.2(b)). This allows Sim to obtain K by using the password from the query by $\mathcal{A}$ and properly equivocate c. Sim proceeds analagously if some honest $\mathsf{ID_C}$ receives a message $(\mathsf{ssid}, b, \mathsf{ID_C^*}, \mathsf{pk_{Enc}})$ and $\mathcal{A}$ has already queried $(\mathsf{pw} \| \mathsf{ID_C}, b^{1/r})$ to H_2, except that it first has to send $(\textsc{CompleteInitS}, \mathsf{ssid}, 1)$ to $\mathcal{F}$ in order to let $\mathcal{F}$ create the leaked record (Ib.6(a)-Ib.6(c)).

Next, we change the handling of messages $(\mathsf{ssid}, b', c, \mathsf{ID_C^*})$ in Sim. Here we need to appropriately choose the values pw^*, K^*, and i for the RecS query if S is fully corrupt. For this, Sim again relies on similar checks as introduced in $\mathbf{G}_5$. If there is a record $F_{\mathsf{sid},\mathsf{S},*}(\mathsf{pw}' \| \mathsf{ID_C^*})$ that successfully decrypts c, Sim sets $\mathsf{pw}^* \leftarrow \mathsf{pw}'$ and $K^* \leftarrow K$, where K is the key obtained from decrypting c (Rb.3(a)). If no such record exists but S used the same OPRF key as in the initalization that produced c, Sim sets $\mathsf{pw}^* \leftarrow \perp, K \leftarrow \perp$ and chooses i such that it indicates the leaked record that contains c (Rb.3(b)). Otherwise, it sets $\mathsf{pw}^* \leftarrow 0$ and $K^* \leftarrow \textsc{Fail}$, which ensures that $\mathsf{ID_C}$ outputs Fail independent of the password (Rb.3(c)). Further, Sim uses the $\mathsf{sk_C}$ obtained from decrypting c, resp. the leaked record, to compute the signature σ.

Finally, we need to ensure that the counter for $\mathsf{ID_C}$ is updated in $\mathcal{F}$. Hence, whenever Sim gives the output $(\textsc{RecRes}, \mathsf{ssid}, \textsc{Succ})$ to S, it additionally sends $(\textsc{CompleteRecS}, \mathsf{ssid}, 1)$ to $\mathcal{F}$ (last part of σ.2(c), σ.3, second step of σ.8(b)).

Let us now argue why the changes introduced in $\mathbf{G}_{10}$ are indistinguishable from $\mathbf{G}_9$. In $\mathbf{G}_9$, K was chosen uniformly at random by Sim, and K is chosen uniformly at random by $\mathcal{F}$ in $\mathbf{G}_{10}$. Therefore the distribution of K obviously does not change. Moreover, the other significant change introduced in $\mathbf{G}_{10}$ to the initialization phase is removing K from the records $\langle \mathsf{AE}, \ldots \rangle$. However, as argued above, Sim is always able to extract K from $\mathcal{F}$ whenever necessary. Thus, all outputs for any $\mathsf{ID_C}$ in the initialization phase remain unchanged.

In the recovery phase, the removal of K from the records $\langle \mathsf{AE}, \ldots \rangle$ has no effect, since in $\mathbf{G}_9$ in the recovery phase the record was only used to obtain K when Sim needed to output it to $\mathsf{ID_C}$ in a successful recovery, where $\mathsf{ID_C}$

received an equivocal c. However, this is not necessary in $\mathbf{G}_{10}$, as K is output by $\mathcal{F}$ to $\mathsf{ID_C}$.
In both phases, the ouputs of S remain unchanged as S still gets its output from Sim and the CompleteInitS and CompleteRecS interfaces introduced here do not produce output to S. Overall, we therefore have

$$\Pr[\mathbf{G}_{10}] = \Pr[\mathbf{G}_9].$$

Game $\mathbf{G}_{11}$: Let $\mathcal{F}$ produce the output of honest servers in initialization. In this game, we change Sim such that it produces the output for honest servers by calling the appropriate interfaces of $\mathcal{F}$. We also change the CompleteInitS interface of $\mathcal{F}$ to provide output to S.
Concretely, in IC.2(a), we still let Sim compute $(\mathrm{ssid}', \mathsf{pk_C}, c) \leftarrow \mathsf{Dec}(\mathsf{sk_{Enc}}, C)$ and check $\mathrm{ssid} = \mathrm{ssid}'$. However, if the check fails, Sim now sends (CompleteInitS, ssid, 0) to $\mathcal{F}$ to let the server output Fail.
Further, we remove the (CompleteInitS, ssid, $\mathsf{ID_C}$, pw, K) interface from $\mathcal{F}$ that we introduced in $\mathbf{G}_8$. In $\mathbf{G}_8$ the interface was used by Sim to let $\mathcal{F}$ create File records. However, as of $\mathbf{G}_{10}$, $\mathcal{F}$ stores all the necessary records because of the added InitC and InitS interfaces. Therefore, Sim can execute IC.2(d) exactly as Sim_{OPRF}, i.e., Sim sends (CompleteInitS, ssid, 1) to $\mathcal{F}$ instead of (CompleteInitS, ssid, $\mathsf{ID_C}$, pw, K).
We argue that the above changes do not alter the view of $\mathcal{Z}$: First, whenever in $\mathbf{G}_{10}$ Sim would have given output (InitRes, ssid, $\mathsf{ID_C}$, Fail) directly to the honest S, we now use the (CompleteInitS, ssid, 0) message to $\mathcal{F}$. The effect is the same and $\mathcal{F}$ outputs (InitRes, ssid, $\mathsf{ID_C}$, Fail) to S. The same holds for the (CompleteInitS, ssid, 1) messages and Succ output. In addition, CompleteInitS makes $\mathcal{F}$ record File records that use $\mathsf{ID_C}$, pw, K from $\mathcal{F}$'s Init records (instead of the values provided to CompleteInitS). But these records were already created in $\mathbf{G}_{10}$, and therefore $\mathsf{ID_C}$, pw, K are the same in both games. We get

$$\Pr[\mathbf{G}_{11}] = \Pr[\mathbf{G}_{10}].$$

Game $\mathbf{G}_{12}$: Let $\mathcal{F}$ produce the output of honest servers in recovery. In this game, we change Sim such that it produces the output for honest servers in recovery by calling the appropriate interfaces of $\mathcal{F}$. In $\mathbf{G}_{11}$, Sim only used the CompleteRecS interface when it wanted to produce the output Succ for S. Hence, here we change Sim to also send the message (CompleteRecS, ssid, 0) to $\mathcal{F}$ when it wants to produce the output Fail for S (Ra.2 and σ.2(c)). By setting $b_S = 0$, S always gets output Fail from $\mathcal{F}$ unless $\mathcal{F}$ deletes the file for $\mathsf{ID_C}$ in this recovery, however in that case Sim ignores the message (ssid, σ) (see σ.2(a), σ.5). Therefore, whenver Sim sends (CompleteRecS, ssid, 0) to $\mathcal{F}$, this produces the output (RecResssid, Fail) to S.
Further, it is easy to verify that any (CompleteRecS, ssid, 1) query from Sim, where S is honest, produces the output (RecRes, ssid, Succ) to S (σ.2(c), σ.3). In σ.2, Sim is either able to extract the correct password due to $\mathbf{G}_9$, which leads to CompleteRecS outputting Succ, or if it cannot extract a password the file must have been created in a malicious initialization and CompleteRecS then outputs Succ as well. If $\mathsf{ID_C}$ is honest (cf. σ.3), the signature σ received in this interface is only valid if it was output by Sim, which only happens if $\mathsf{ID_C}$ used the correct password. Thus, CompleteRecS outputs Succ as well and we have

$$\Pr[\mathbf{G}_{12}] = \Pr[\mathbf{G}_{11}].$$

Game $\mathbf{G}_{13}$: Simulate C during Init. In this game, we change how Sim computes the message (ssid, C) for honest $\mathsf{ID_C}$ as long as S is not fully corrupt. Whenever Sim receives a message (ssid, b, $\mathsf{ID_C}$, $\mathsf{pk_{Enc}}$) to an honest $\mathsf{ID_C}$, the simulator does not compute $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, (\mathrm{ssid}, \mathsf{pk_C}, c))$ but now computes $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, \perp)$ (see Ib.7(b)).
We also change Sim such that it does not decrypt the ciphertext C produced for an honest $\mathsf{ID_C}$ anymore. In IC.3, when C was computed by Sim for an honest $\mathsf{ID_C}$, then Sim retrieves the record containing $\mathsf{pk_C}$ and c that it stored when computing C. Similarly, in IC.6, when C was computed by Sim, then Sim just retrieves the stored values $\mathsf{pk_C}$ and c. Additionally, we need to check if the corrupt S replays some C that was computed as $C \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk_{Enc}}, \perp)$ and if so, retrieve the corresponding values from the record instead of decrypting C (cf. IC.7(a) and IC.7(b)). The only change in the view of $\mathcal{Z}$ is that the distribution of the ciphertexts C. If $\mathcal{Z}$ can detect this difference, we can construct an adversary $\mathcal{B}_7$ against the IND-CCA security of PKE as follows:
Let $q_{\text{Init}} \in \mathbb{N}$ be the number of initializations. We construct a sequence of games $\mathbf{G}_{12}^{(0)}, \ldots, \mathbf{G}_{12}^{(q_{\text{Init}})}$, where in $\mathbf{G}_{12}^{(i)}$ the first i ciphertexts are computed as encryptions of $\perp$ and the remaining ciphertexts are encrypted as in $\mathbf{G}_{12}$. We have $\mathbf{G}_{12} = \mathbf{G}_{12}^{(0)}$ and $\mathbf{G}_{13} = \mathbf{G}_{12}^{(q_{\text{Init}})}$. Because $\mathcal{Z}$ can distinguish $\mathbf{G}_{12}$ from $\mathbf{G}_{13}$ there must be an index $i^* \in [q_{\text{Init}}]$ such that $\mathcal{Z}$ has a non-negligible advantage in distinguishing $\mathbf{G}_{12}^{(i^*-1)}$ and $\mathbf{G}_{12}^{(i^*)}$. its challenger. Now, let ssid* denote the ssid of the i^*-th initialization and ID_C^* denote the $\mathsf{ID_C}$ that executes that initialization. In that initialization the reduction $\mathcal{B}_7$ does not compute $(\mathsf{sk_{Enc}}, \mathsf{pk_{Enc}})$ itself in Ia.6 but uses the pk^* provided by its challenger. Then, in Ib.7(b), $\mathcal{B}_7$ gives $m_0 := (\mathrm{ssid}^*, \mathsf{pk_C}, c)$ and $m_1 := \perp$ to the challenger and uses the returned C^* as ciphertext for the i^*-th initialization. In any subsequent recovery by ID_C^* In IC.7(b), when $\mathcal{B}_7$ receives a message (ssid, C) from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{HSM}}^{\mathsf{encPw}}$ on behalf of a corrupted server such that there is no record $\langle \text{Init}, *, *, \mathrm{ssid}^*, *, *, *, *, *, *, C\rangle$ then $\mathcal{B}_7$ uses the challenger's decryption oracle to decrypt C. Similarly, in IC.2(a), if there is no record $\langle \text{Init}, *, *, \mathrm{ssid}^*, *, *, *, *, *, *, C\rangle$ then Sim again uses the challenger's decryption oracle to decrypt C.
Then we have that if C^* encrypts m_0 the game is distributed as in $\mathbf{G}_{12}^{(i^*-1)}$ and if it encrypts m_1 then the game is distributed as in $\mathbf{G}_{12}^{(i^*)}$. We get

$$|\Pr[\mathbf{G}_{13}] - \Pr[\mathbf{G}_{12}]| \leq q_{\text{Init}} \mathbf{Adv}_{\mathsf{PKE},\mathcal{B}_7}^{\mathsf{IND\text{-}CCA}}(\lambda).$$

Game $\mathbf{G}_{14}$: Remove password forwarding from $\mathcal{F}$. In this game, we do not give Sim any private input of the parties anymore. Up to $\mathbf{G}_{13}$, Sim still stored the records $\langle \mathsf{AE}, c, \mathrm{pw} \,\|$

$\mathsf{ID}_\mathsf{C}, b^{1/r}, \mathsf{sk}_\mathsf{C}\rangle$ and used them for three purposes: (1) to obtain the sk_C that is supposed to be contained in an equivocable c, (2) to check whether a client used the correct password in a recovery, and (3) to determine critical queries $(\mathsf{pw} \parallel \mathsf{ID}_\mathsf{C}, b^{1/r})$ to H_2 that require Sim to equivocate c. As we now finally remove the AE records, we have to simulate these steps in another way.

Issue (1) can be solved trivially, and we instead use the File records, which also store c and sk_C (see Rb.7). For issue (2), we now rely on the bit *match* that Sim receives from $\mathcal{F}$ and indicates whether the password used in the recovery is correct (see Rb.8 and Rb.9).

To solve issue (3), we can instead rely on the records $\langle H_2, \mathsf{pw}\| \mathsf{ID}_\mathsf{C}, u, y\rangle$ and the Consistent marking. An H_2 record if marked Consistent if and only if $u = H_1(\mathsf{pw} \parallel \mathsf{ID}_\mathsf{C})^k$ for some OPRF key k, where k can be either created by Sim_OPRF (cf. H2.2), i.e., when S is not fully corrupt, or adversarially chosen (cf. Ib.6(b)), i.e., when S is fully corrupt. Then, whenever Sim would query the OfflineAttack interface in $\mathbf{G}_{13}$ with the password pw obtained from the AE record, in this game we instead query the interface with all passwords pw from Consistent H_2 records (Ib.6(b)).

With this change, we finally reach the point where $\text{Sim} = \text{Sim}^\text{OPRF-PPKR}$ and $\mathcal{F} = \mathcal{F}_\text{PPKR}$. We clearly added all interfaces of $\mathcal{F}_\text{PPKR}$ to $\mathcal{F}$. One can also verify that Sim is indeed $\text{Sim}^\text{OPRF-PPKR}$.