Security Analysis of the WhatsApp End-to-End Encrypted Backup Protocol

Gareth T. Davies[1] Sebastian Faller[2,3] Kai Gellert[1]
Tobias Handirk[1] Julia Hesse[2] Máté Horváth[1] Tibor Jager[1]

[1]Bergische Universität Wuppertal, Wuppertal, Germany
[2]IBM Research Europe – Zurich, Switzerland
[3]ETH Zurich, Switzerland

June 6, 2023

Abstract

WhatsApp is an end-to-end encrypted (E2EE) messaging service used by billions of people. In late 2021, WhatsApp rolled out a new protocol for backing up chat histories. The E2EE WhatsApp backup protocol (WBP) allows users to recover their chat history from passwords, leaving WhatsApp oblivious of the actual encryption keys. The WBP builds upon the OPAQUE framework for password-based key exchange, which is currently undergoing standardization.

While considerable efforts have gone into the design and auditing of the WBP, the complexity of the protocol's design and shortcomings in the existing security analyses of its building blocks make it hard to understand the actual security guarantees that the WBP provides.

In this work, we provide the first formal security analysis of the WBP. Our analysis in the universal composability (UC) framework confirms that the WBP provides strong protection of users' chat history and passwords. It also shows that a corrupted server can under certain conditions make more password guesses than what previous analysis suggests.

1 Introduction

WhatsApp is the most popular instant messaging app in the world with over 100 billion messages sent per day, containing personal and business communications. WhatsApp provides *end-to-end encrypted* (E2EE) communications [34], where no party but sender and receiver should be able to read (or modify) messages. This specifically prevents the WhatsApp service provider from breaking security guarantees such as confidentiality if the service gets compromised. E2EE is considered a default standard for modern secure messaging protocols, and several formal analyses of the E2EE messaging protocol used by WhatsApp and other messaging apps such as Signal exist [14, 4, 31, 2, 25, 13, 32, 5, 10].

Gareth T. Davies, Tobias Handirk, and Tibor Jager have 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.

Bypassing E2EE via Backups. Protecting the *transmission* of confidential data is necessary to enable secure messaging, however, it is not sufficient. WhatsApp clients can back up the user's chat history so that they can recover it if the device is lost, for example by theft or switching to a new phone. Naturally, the backup mechanism must offer strong protection as well, so as not to undermine the security of the E2EE messaging protocol.

Before the end of 2021, whenever a user[1] initiated the procedure for backing up their messages (to be stored on iCloud or Google Drive), they would encrypt these messages using a key that was known to WhatsApp. While this simple approach allowed WhatsApp to return the backup encryption key to the user if the original device were to be lost, it allowed access to backed up messages beyond the control of the user. For example, by US law, federal governments could have forced WhatsApp to reveal the backup key (and the storage provider to reveal the encrypted contents) through a court order, and the previously well protected private communication suddenly becomes evidence in a lawsuit [29]. More generally, the fact that the E2EE security can be circumvented by accessing the backups harbors a great potential for abuse, for instance by malfeasant governments, malicious employees, or in case of a compromise of both the storage provider's servers and WhatsApp's servers.

WhatsApp E2EE Backups. In late 2021, WhatsApp rolled out an improved protocol for protecting backups [33], with the aim to extend the E2EE security guarantees in a user-friendly way that enables users to restore their backup keys from a password in case a device is lost. By early 2023, over 100 million WhatsApp users have already switched to this option [12]. The underlying protocol, which we call the *WhatsApp backup protocol* (WBP) in this work, makes use of hardware security modules (HSMs). Intuitively, an HSM can be programmed once with code and then "locked" in such a way that it is infeasible to change its code afterwards. This enables even the protection of a protocol against corruption of the party running the HSM (here, the WhatsApp servers). The challenge now lies in designing the code run by the HSM in such a way that (1) users can retrieve their backup keys from a password, but (2) WhatsApp servers not knowing the user password cannot retrieve the backup key.

The core idea of the WBP is to outsource all cryptographic computations to the client and the HSM, while the WhatsApp "main server" essentially only relays messages (with some minor modifications) between client and HSM. The protocol is designed such that during the initialization phase[2] both client and HSM enter a secret value (a password pw for the client, and a "per-client-secret"[3] for the HSM). Furthermore, the client chooses a symmetric backup key K and the HSM receives an encrypted version of the key, without either of them learning the secret input of the counterparty. The client uses the backup key K to encrypt the backup data. If a user loses their device, they can initiate a recovery protocol from a new client device. To this end, the new client and the WhatsApp "main server" (which, again, relays messages to the HSM) execute a protocol where the client recovers the backup key, with the password pw used during initialization and the HSM contributing the same per-client-secret as during initialization.

The password-based nature of the WBP introduces several technical challenges. Firstly, being password-based while aiming at keeping the stored key from a potentially compromised WhatsApp server implies that the protocol must not leak user passwords to the server. To this end, the WBP deploys OPAQUE [24], an asymmetric password-based key exchange protocol (aPAKE) [19] that allows a key exchange from a

[1]In this paper we will refer to the people using a device that runs the WhatsApp client software as *users*, to the device as a *client*, and to the servers that provide the WhatsApp chat and backup service as *servers*.

[2]Note that WhatsApp refer to this phase as *registration*.

[3]It is actually a "per-backup-secret", which is determined during initialization. If a client were to re-register, a new "per-backup-secret" would be chosen.

password without disclosing the password itself to the server. Another essential feature the WBP aims to provide is security against *password guessing attacks*, where a malicious client repeatedly executes the recovery protocol with password guesses pw'. If the password guess pw' equals the password pw used during initialization, an adversary would gain access to the secret backup key. Note that this attack is especially effective if the user password only has low entropy, which is often the case for human-memorable passwords in practice. The deployed protocol limits the number of admissible incorrect guesses to ten [33, 17], after which the HSM destroys the encrypted version of the backup key (and thus makes the backup irrecoverable). This guarantee should even hold if the WhatsApp server were to be compromised.

Contributions. The WBP protocol is a widely-used real-world cryptographic protocol that addresses the fundamental problem of recovering data from encrypted backups based on human-memorizable secrets. It aims to provide strong security properties that match the E2EE-security of the messaging protocol, even against a corrupted service provider. This work presents the first rigorous security analysis of the WBP. Concretely, our results can be summarized as follows.

- We formalize the security properties expected from the WBP protocol in terms of a *password-protected key retrieval* (PPKR) scheme, where users store cryptographic keys on an untrusted server, retrievable with a password. This formalization serves as a foundation for our work, and may also support future analyses of alternative (potentially non-HSM-based) PPKR protocols and their comparison.
- We provide a *full description* of the cryptographic core of the WBP protocol. This description is based on a whitepaper published by WhatsApp [33], a public security assessment of the backup system conducted by the NCC Group [17], and personal correspondence with the WBP designers [1] to fill subtle but essential technical gaps left in the protocol descriptions of [33, 17].
- We present the *first formal security analysis* of the WBP protocol. Our analysis is conducted in the universal composibility (UC) framework [9], which is simulation-based and therefore facilitates the consideration of low-entropy passwords. We formally confirm several prior statements about the security guarantees of the WBP.
- We describe how a corrupted server could get more than ten password guesses per encrypted backup, even though prior security analysis [17] claimed that after ten incorrect tries the account is irremediably locked by the HSM software, and the backup data cannot be retrieved in plaintext. Concretely, we show that a corrupted server can get ten password guesses *per backup initialization*. For example, a corrupted server could suppress protocol messages to simulate a failed initialization, such that either the WhatsApp client app retries sending of initialization messages automatically, or the user re-initializes a backup manually, in order to increase the number of password guesses against the HSM.
- We give a formal analysis of the 2HashDH oblivious pseudorandom function [22] (that is used in OPAQUE) in the multi-key setting, where the domains of the two hash functions used as a building block are not assumed to be separated for different keys. For our work, this result is required since the WBP does not apply hash domain separation. Beyond that, our findings provide the basis for analyzing the *about-to-be-standardized* version of OPAQUE [6][4] and the 2HashDH protocol currently in last call at the IRTF [16] even under the usage of *multiple* OPRF keys.

[4]The existing formal analysis of the OPAQUE protocol [24] assumes hash domain separation in 2HashDH and hence does not apply to the version of OPAQUE in the most recent Internet Draft [6].

Paper Organization. The remainder of this work is structured as follows. Section 2 contains technical preliminaries and Section 3 provides a full protocol description of the WBP. We then describe our model for PPKR in Section 4 and give a proof intuition in Section 5 (the full proof can be found in Appendix E). Our work concludes with a discussion in Section 6.

Responsible Disclosure. The research conducted for this work did not impact the entire WhatsApp system or the privacy of WhatsApp users. In particular, there was no interaction with the WhatsApp servers, HSMs, or any WhatsApp users. The protocol description was written with the help of WhatsApp employees [1], and no reverse-engineering of any implemented code took place. The scenario in which a *corrupt* WhatsApp server can increase the number of password guesses against a user was never demonstrated in practice, but it was acknowledged by WhatsApp that this would indeed be possible. WhatsApp does not object to the publication of this paper.

1.1 Related Work

Password-protected secret sharing (PPSS) [3, 22] allows a user to share a secret value among a number of servers and later retrieve it using a partial set of the servers (in the event that one or more servers become compromised or unavailable) if and only if the password used during retrieval is the same as the one used during the sharing step. This primitive has been analyzed in the UC framework [8, 22], and several constructions based on oblivious pseudorandom functions (OPRF) exist (an overview can be found in [11]).

The WhatsApp approach can be viewed as a one-out-of-one version of PPSS, where WhatsApp's HSM is the only server. This makes comparisons with work on PPSS difficult: we need to model corruption of the WhatsApp communication server (but not the HSM) and assess security in this context, something that prior models that do not split the server's role cannot capture. Nonetheless, our formalization of PPKR in the UC model takes great inspiration from existing functionalities for PPSS [22].

Beyond PPSS, there are several works that aim at bootstrapping encryption keys (or symmetric encryptions directly) from user passwords with the assistance of a server. Updatable oblivious key management [23] relies on server assistance to let a user derive file-specific encryption keys from a password, while requiring strong user authentication. The distributed password-authenticated symmetric encryption scheme DPaSE [15] aims for the same, while relying on the assistance of several servers but not requiring user authentication. Like the WBP, all the above schemes rely on OPRFs to shield passwords from curious servers, but none of them aims to provide a restriction in the number of guessing attempts after the compromise of the server, which the WBP aims for.

Password-hardened encryption services [28, 7] let users outsource the encryption to a fully trusted frontend server. The protocols do not require OPRFs and can hence provide better throughput, at the cost of revealing the user's password to the frontend server.

2 Preliminaries

Notation. We denote the security parameter as λ. For any $\lambda \in \mathbb{N}$ let 1^λ be the unary representation of λ and let $[\ell] = \{1, \ldots, \ell\}$. We write $x \xleftarrow{\$} \mathcal{S}$ to indicate that we choose an element x uniformly at random from set $\mathcal{S}$. For a probabilistic polynomial-time algorithm $\mathcal{A}$ we define $y \xleftarrow{\$} \mathcal{A}(x_1, \ldots, x_\ell)$ as the execution of $\mathcal{A}$ (with fresh random coins) on input $x_1, \ldots, x_\ell$ and assigning the output to y.

We use records of form $\langle x_1, x_2, x_3 \rangle$ for bookkeeping in our formal arguments. For convenience, we introduce a notation that combines retrieval and assignment of such records, i.e., when retrieving $\langle \mathsf{value}, *, * \rangle$, we retrieve a record that contains the value value in the first field and arbitrary values in the second and third field (denoted by a wildcard symbol $*$). Additionally, we use brackets to indicate variable assignment after retrieval, i.e., when retrieving $\langle \mathsf{value}, [x_2], [x_3] \rangle$, we retrieve the record holding the value value in its first entry and assign the second and third entry to the variables x_2 and x_3, respectively.

Cryptographic Building Blocks and Their Security. We defer standard definitions such as collision resistance for hash functions, strong EUF-CMA security of digital signatures and message authentication codes, IND-CPA security of public key encryption, and standard definitions for authenticated encryption to Appendix A.

3 E2EE Backups in WhatsApp

In this section, we give a detailed description of the WBP. Our presentation is based on a whitepaper published by WhatsApp [33], a public security assessment of the backup system conducted by NCC Group [17], and personal correspondence with WhatsApp (Meta) staff [1].

We will start with a simplified explanation of the overall protocol layout in Section 3.1 to give a high-level overview of its main idea. Then, to prepare the detailed protocol description, we will discuss the creation of a communication channel between clients and the backup server via the WhatsApp client registration protocol in Section 3.2. In Section 3.3, we elaborate on how WhatsApp uses HSMs; in Section 3.4, we outline how these HSMs outsource storage to servers that are considered untrusted. In Section 3.5, we then provide a detailed description of the actual WBP. We conclude with Section 3.6, where we describe how a malicious server can increase the admissible number of password guesses in certain settings.

3.1 High-level Protocol Overview

There are four entities in the system: the user, a WhatsApp client running on the user's device, the WhatsApp server, and the HSM that only the WhatsApp server can communicate with. We will now focus on the latter three. In this high-level overview, we simplify by describing the WhatsApp server as merely relaying messages between the HSM and the client. In the actual WBP protocol it additionally authenticates clients and stores files encrypted by the HSM. As the encryption and outsourced storage of the data at a cloud provider is done using symmetric encryption, we focus on the initialization and recovery of the encryption key from a password.

Initialization. When activating WhatsApp's E2EE backups for the first time, the client chooses a backup key K to encrypt the chat history and the WBP key initialization phase is executed (see Figure 1). To this end, the client first runs the OPAQUE protocol with the HSM, which takes a password pw from the client and a key K_{OPRF} from the HSM as inputs. It then outputs a key K^{export} to the client[5] and the "envelope" env to the HSM. This envelope is encrypted under the key K^{export} (which is derived using both pw and K_{OPRF}) and contains freshly chosen key material of the client that is used during recovery to perform, e.g., a Diffie–Hellman key exchange, among other things.

To conclude initialization, the client encrypts the backup key K first under the symmetric key K^{export} and then under the HSM's public encryption key $\mathsf{pk}_{\mathsf{HSM}}$, and sends the result E to the HSM. The HSM

[5]The option to derive this additional key was originally not part of OPAQUE [24]. However, it exists in the OPAQUE Internet Draft version 03 [26], which is deployed by the WBP.

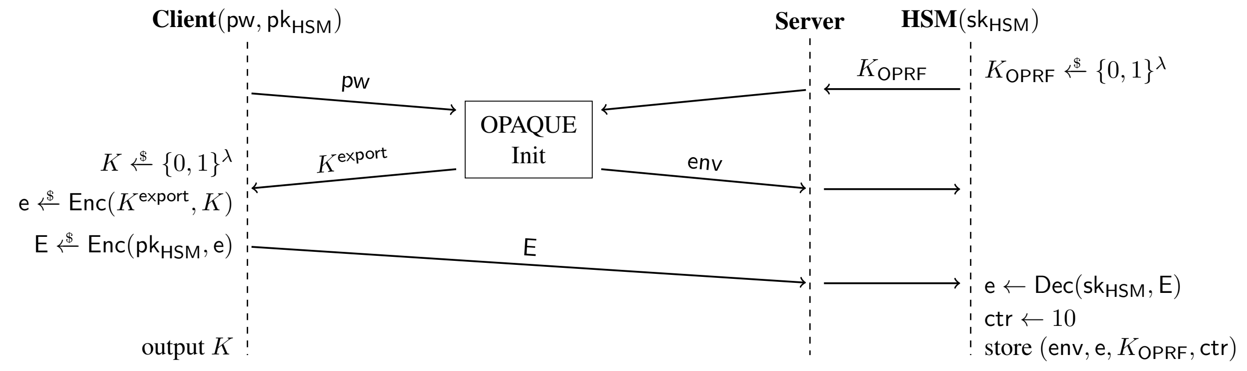


Figure 1: The WBP key initialization, high-level layout. The value K_{OPRF} is freshly chosen by the HSM for each initialization request.

removes the outer encryption layer and stores the encrypted backup key e, the OPAQUE envelope env, the key K_{OPRF}, and a counter ctr initialized with 10 that tracks password guessing attempts.

Key Recovery. If the client has lost their client device (and thus lost their backup key K), they can re-authenticate their new client device with WhatsApp (via a challenge-response protocol that takes place after re-installing the WhatsApp application), and subsequently start the recovery part of the WBP depicted in Figure 2.

The first step of the recovery phase is that client and HSM engage in the key exchange phase of the OPAQUE protocol. To this end, the client uses a value pw' as input and the HSM contributes the values K_{OPRF} and env as established during the initialization phase. If the password pw' entered by the client is equal to the password pw during initialization, the OPAQUE protocol guarantees that (1) the client recovers the former export key $\bar{K}^{\mathsf{export}} = K^{\mathsf{export}}$ and (2) both client and HSM derive the same value $\mathsf{shk}' = \mathsf{shk}$. However, if $\mathsf{pw}' \neq \mathsf{pw}$, the client will have to abort instead.

The HSM decrements its counter for attempted password guesses each time a recovery procedure is initiated. If the client can convince the HSM that it has computed the same value $\mathsf{shk}' = \mathsf{shk}$ (via a key confirmation), the HSM will learn that the entered password pw' was indeed correct and hence reset its counter to ten and send the stored ciphertext c to the client. Lastly, the client can use the derived $\bar{K}^{\mathsf{export}}$ to decrypt the ciphertext c and recover their backup key K. This concludes the high-level overview of the cryptographic core of the WBP.

3.2 Client Registration

The WBP essentially relies on a communication channel between clients and the backup server, which is realized in an indirect way. Upon installing the WhatsApp application, the main WhatsApp server sets up a mutually authenticated channel with each new client. In the WhatsApp ecosystem this is done by a server called ChatD, which is physically distinct form the WhatsApp server handling the WBP. We decided to view all WhatsApp servers as a single WhatsApp server entity, since a distinction would make the already complex protocol description and security analysis unnecessarily more complex without providing additional insight.[6]

[6]For example, we want a corruption of the WhatsApp server to model a malicious WhatsApp service provider, and therefore we want to consider the entire service as corrupted in this case, without a need to distinguish between the ChatD and the backup server.

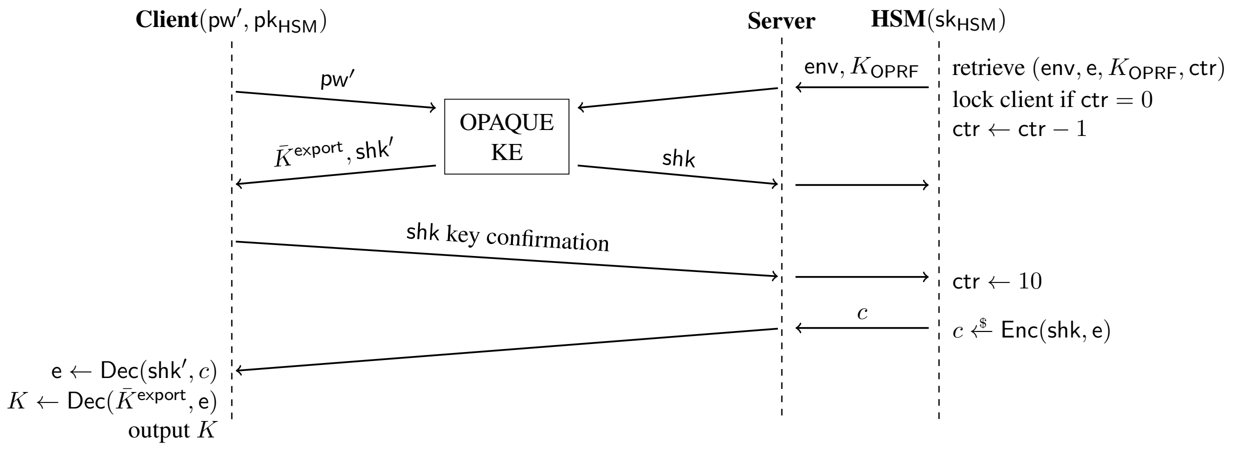


Figure 2: The WBP key recovery, high-level layout.

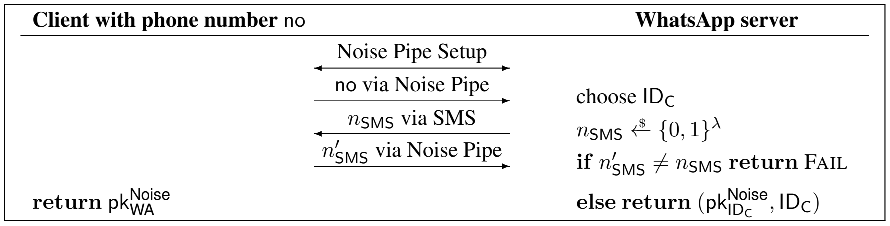


Figure 3: The WhatsApp client registration that is independent from the WBP. Upon conclusion, the client can be identified via a unique identifier $\mathsf{ID_C}$, and both client and server are mutually authenticated.

That is, at first, a secure channel is set up between the client and WhatsApp using the Noise framework [30]. Then WhatsApp uses SMS authentication to verify that the phone number it received via the freshly set up Noise channel belongs to the client. Upon conclusion, the client stores WhatsApp's public key $\mathsf{pk}_{\mathsf{WA}}^{\mathsf{Noise}}$ and WhatsApp stores the freshly generated client's public key $\mathsf{pk}_{\mathsf{ID_C}}^{\mathsf{Noise}}$ together with a unique client identifier $\mathsf{ID_C}$. Subsequently, the WhatsApp server handles all incoming client requests via the Noise channel and also mediates the WBP messages between the client and the backup service.

3.3 Hardware Security Modules

The WBP deploys HSMs as a core component of their PPKR protocol. Intuitively, an HSM is a hardware device that can be programmed once with code and then "locked" in such a way that it is infeasible to change its code afterwards. After the HSMs are set up they hence serve as an incorruptible entity in the backup ecosystem. In the WBP, the HSM performs most cryptographic relevant computations on the "WhatsApp side" (with minor computations performed by a different, non-HSM WhatsApp "main server") and is responsible for coordinating the secure storage of backup keys.

Trusted Setup Ceremony. For the HSM to serve as an incorruptible entity in the system, it must be ensured that (1) its secret key material is not leaked and that (2) its code cannot be modified after setup. This is usually ensured via a process called *trusted setup ceremony* or *key ceremony*. During such a ceremony, a ceremony leader essentially unpacks new modules, sets them up with program code, and generates fresh key material for the HSM (where public keys are copied and secret keys remain secret). After setup, the HSM's "programming key" is destroyed, ensuring that it cannot be modified after the ceremony has taken place. The public key material of the HSM is hard-coded into the WhatsApp application [17]. Naturally, this procedure can only be trusted if it was executed faithfully. For the remainder of this work we assume that the setup ceremony was conducted such that

- the following key material has been generated
  - a symmetric encryption key $K^{\mathsf{Enc}}_{\mathsf{HSM}} \xleftarrow{\$} \{0,1\}^\lambda$,
  - a signature key pair $(\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{Sig}}_{\mathsf{HSM}}) \xleftarrow{\$} \Sigma.\mathsf{KeyGen}(1^\lambda)$,
  - a public-key encryption key pair $(\mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{Enc}}_{\mathsf{HSM}}) \xleftarrow{\$} \mathsf{PKE.KeyGen}(1^\lambda)$,
  - a static Diffie–Hellman key pair $(\mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}) \xleftarrow{\$} \mathsf{DH.KeyGen}(1^\lambda)$;
- the HSM uses the secret key material to execute the WBP's computations via predefined interfaces for each protocol step;
- the HSM can only be queried by a WhatsApp server, and only via the specified interfaces. In particular, the HSM does not leak any of its secret key material.

We remark that the HSM solution deployed by WhatsApp consists of multiple HSMs, which ensures that user cannot get locked out of their backup if an HSM breaks down. All HSMs are set up such that they coordinate their state changes via a consensus protocol, ensuring that each HSM behaves in the same manner towards a user [1]. Without loss of generality, we treat this set of HSMs as a single HSM entity. The analysis of the HSM consensus protocol is beyond the scope of this work.

3.4 Secure Outsourced Storage

One might be tempted to store sensitive data along with key material in an HSM. However, storing data in the internal memory of an HSM is very expensive and limited in capacity. Thus, the internal storage of an HSM is not viable to store large quantities of sensitive data for millions or billions of users. Therefore WhatsApp uses storage fully controlled by the WhatsApp server. The HSM uses a dedicated symmetric encryption key $K^{\mathsf{Enc}}_{\mathsf{HSM}} \xleftarrow{\$} \{0,1\}^\lambda$ which is used for authenticated encryption, and this essentially ensures confidentiality and integrity of stored records. Whenever the HSM requests a stored record, it decrypts the record and verifies its integrity before processing it. In addition, a Merkle tree based protocol is deployed to "tie" the encrypted records together and to prevent a replay of old state records that were previously deleted by the HSM. Whenever the WhatsApp server provides an encrypted record to the HSM, it also has to provide a proof (using the Merkle tree) that this ciphertext is consistent with the current state of the encrypted database. The HSM verifies this against a locally stored root of the Merkle tree. If this verification fails, the HSM rejects the record. A formal analysis of this mechanism is beyond the scope of our work. Therefore we model the interaction of the HSM with the outsourced storage mechanism via function calls, which are defined as follows.

- $\mathsf{sec\_store}(\mathsf{id}, \mathsf{data})$ takes as input a unique identifier id and to-be-stored data value data. It encrypts the data using the approach described above. The record can be identified via id.

- $\mathsf{sec\_retr}(\mathsf{id})$ takes as input an identifier id and retrieves the record associated with identifier id. The returned record is processed as described above.

- $\mathsf{sec\_delete}(\mathsf{id})$ takes as input an identifier id and "tombstones" the record associated with identifier id, in that its existing data is overwritten with the information that is has been deleted.

- For brevity, we define one additional function call that allows the HSM to change counter values. $\mathsf{sec\_set\_ctr}(\mathsf{id}, \mathsf{ctr})$ takes as input an identifier id and an integer ctr with $0 \leq \mathsf{ctr} \leq 10$. It takes the record associated with identifier id and sets the corresponding counter to the value ctr. Integrity and authenticity is ensured via the process described above.

Should any of the function calls fail (e.g., due to a non-existing record or a failed integrity check), the respective function call would return an error symbol $\perp$. We assume that this mechanism provides secure outsourced storage for the HSM, which is immutable for any entity but the HSM.

3.5 WhatsApp Backup Protocol (WBP) Description

The detailed descriptions of the WBP's key initialization and recovery phases are depicted in Figure 4 and Figure 5, respectively. As already discussed in Section 3.1, the WBP builds on the OPAQUE Internet Draft v3 [26] the steps of which are highlighted in the figures. For a comparison of these OPAQUE steps with [26], we refer to Appendix H. Note that the figures include events like $\mathbf{return}(\textsc{Interface}, \mathsf{value})$. These can bee seen as messages delivered to higher-level application processes, which, e.g., output a successfully established symmetric backup key to be used for backing up the actual data. We make these calls explicit since they will also appear in parts of our security model.

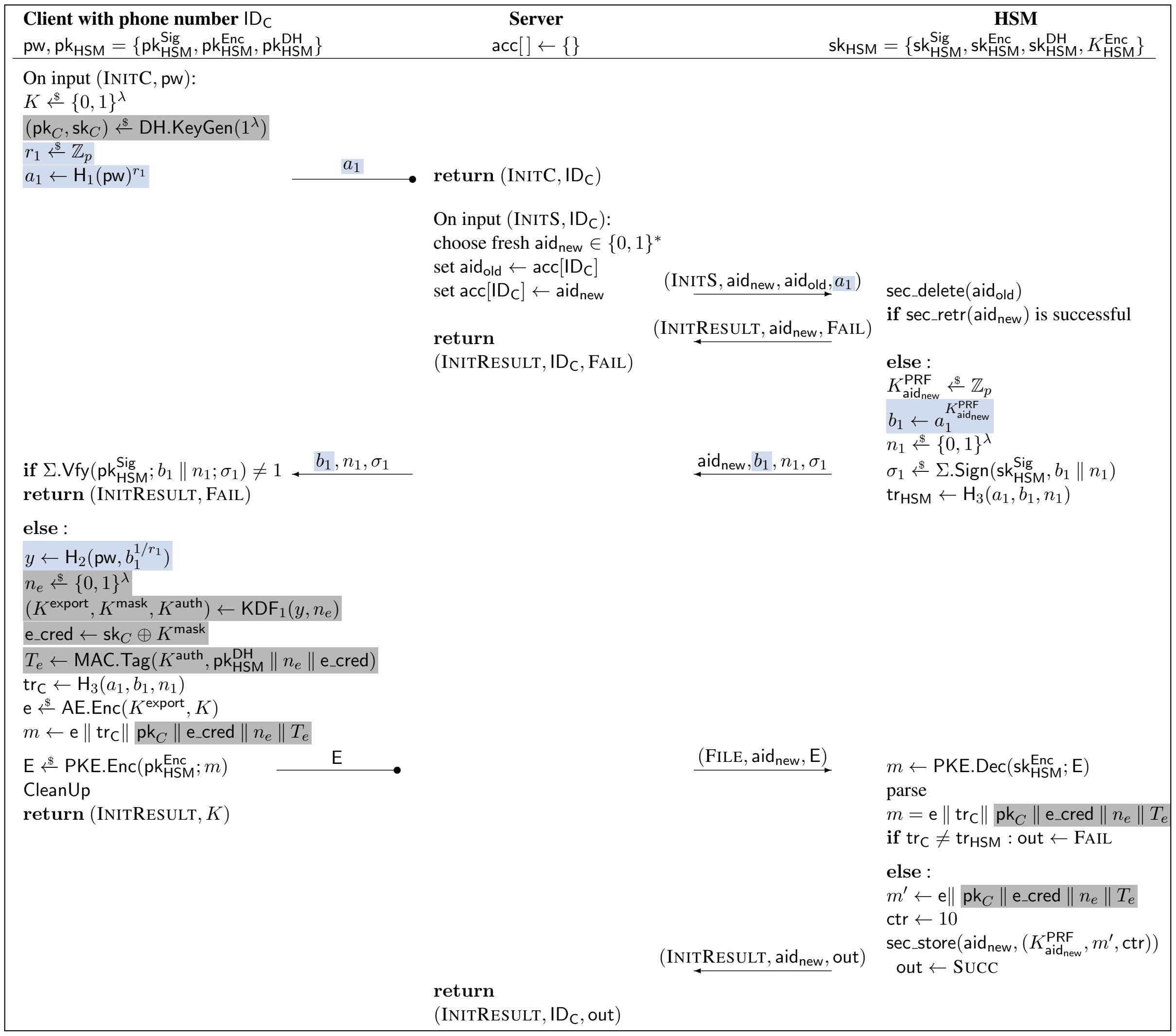
Participants. There are three participants in the protocol. Client refers to the WhatsApp client application of a user with unique identifier $\mathsf{ID_C}$. The client is in possession of the HSM's public key, which is composed of a public key for a signature scheme $\mathsf{pk}^{\mathsf{Sig}}_{\mathsf{HSM}}$, an encryption public key $\mathsf{pk}^{\mathsf{Enc}}_{\mathsf{HSM}}$, and a static Diffie–Hellman public key $\mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}$. Those keys are hard-coded into the WhatsApp client and thus authenticated.

The server is run by WhatsApp and it mostly relays messages between clients and the HSM. For this it communicates with the client via the previously established $\mathsf{ID_C}$-authenticated channel (see Figure 3) and with the HSM directly through a TLS channel. The server also maintains an array $\mathsf{acc}[\cdot]$ of identifier pairs $(\mathsf{ID_C}, \mathsf{aid})$, which "tie" a so-called *account identifier* aid (described below) to the client identifier $\mathsf{ID_C}$. If some $\mathsf{ID_C}$ is not contained in acc, we let $\mathsf{acc}[\mathsf{ID_C}] = \perp$. As already described in Section 3.4, the server also acts as an external (untrusted) storage medium for the HSM.

Finally the HSM is (a trusted entity that is) in possession of the secret keys $\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}$, $\mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}$, $\mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}$ corresponding to the respective public keys, as well as a key $K^{\mathsf{Enc}}_{\mathsf{HSM}}$ that is used to encrypt records stored on the server (see Section 3.3).

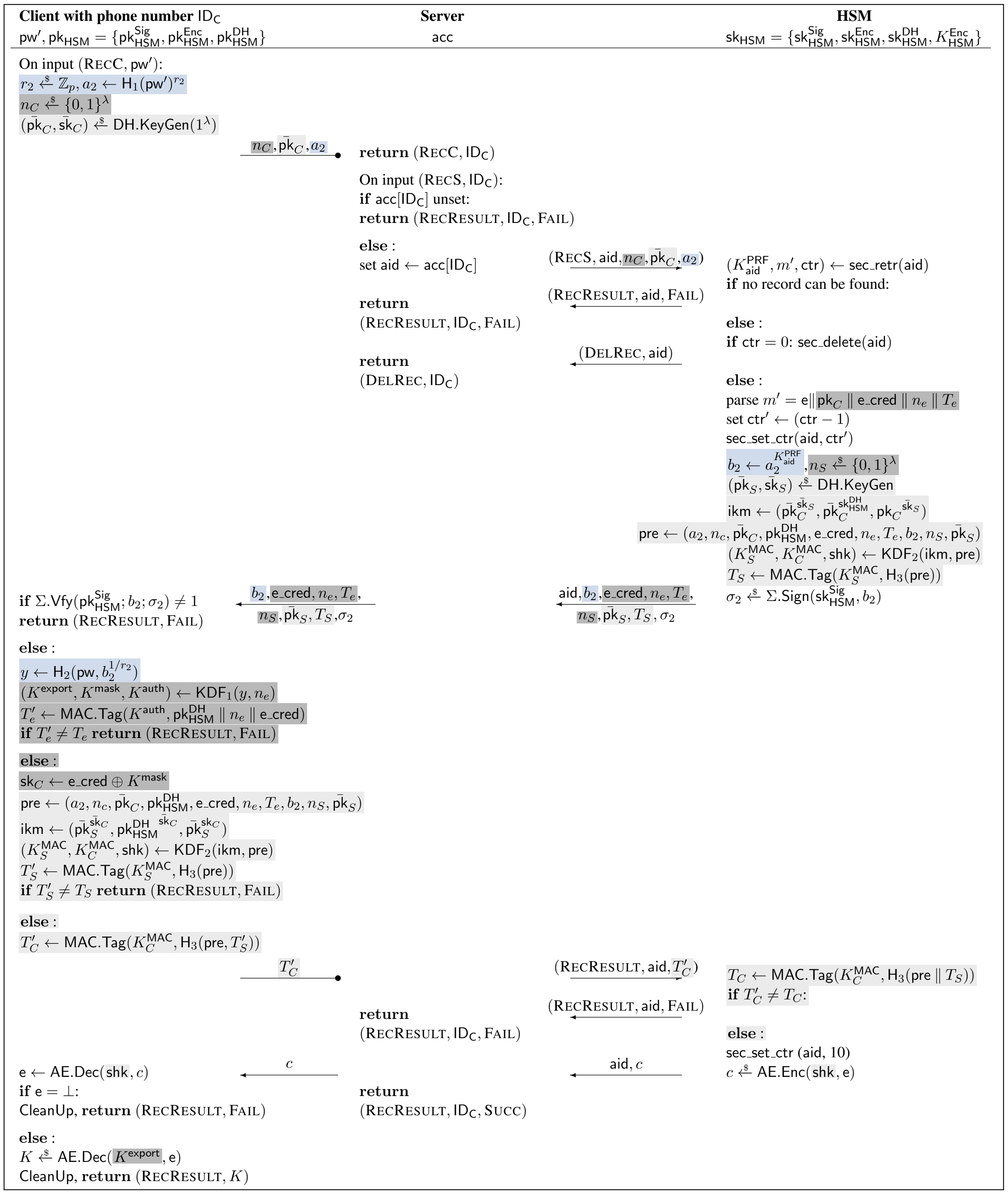
Dealing with unexpected protocol messages. We assume that the client, the server, and the HSM ignore messages that are sent out-of-order, i.e., messages that the party expects at a different point in the protocol execution. The first messages of an initialization (i.e., a_1) and recovery (i.e., $(n_C, \bar{\mathsf{pk}}_C, a_2)$) can be sent anytime, leading to the parties deleting all temporary data of the non-finished initialization/recovery and starting with a new initialization/recovery. This implies that there is never more than one initialization/recovery running at a time.

Key Initialization. A user with password pw and $\mathsf{ID_C}$ initializes the backup as follows. On input of pw, the WhatsApp client app first chooses a uniformly random backup key K that can be used for encrypting the backups and which is going to be preserved via WBP. Next, it samples a Diffie–Hellman key-pair $(\mathsf{pk}_C, \mathsf{sk}_C)$ that will be used later in the OPAQUE key exchange step. Executing the 2HashDH OPRF protocol [21] with the HSM, the client samples $r_1 \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random. This is then used to blind the password pw by



```
Client with phone number $\mathsf{ID_C}$
$\mathsf{pw}, \mathsf{pk_{HSM}} = \{\mathsf{pk^{Sig}_{HSM}}, \mathsf{pk^{Enc}_{HSM}}, \mathsf{pk^{DH}_{HSM}}\}$

On input $(\textsc{InitC}, \mathsf{pw})$:
    $K \xleftarrow{\$} \{0,1\}^\lambda$
    $(\mathsf{pk}_C, \mathsf{sk}_C) \xleftarrow{\$} \mathsf{DH.KeyGen}(1^\lambda)$
    $r_1 \xleftarrow{\$} \mathbb{Z}_p$
    $a_1 \leftarrow \mathsf{H}_1(\mathsf{pw})^{r_1}$
    send $a_1$ to Server
if $\Sigma.\mathsf{Vfy}(\mathsf{pk^{Sig}_{HSM}}; b_1 \| n_1; \sigma_1) \neq 1$
    return $(\textsc{InitResult}, \textsc{Fail})$
else:
    $y \leftarrow \mathsf{H}_2(\mathsf{pw}, b_1^{1/r_1})$
    $n_e \xleftarrow{\$} \{0,1\}^\lambda$
    $(K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}}) \leftarrow \mathsf{KDF}_1(y, n_e)$
    $\mathsf{e\_cred} \leftarrow \mathsf{sk}_C \oplus K^{\mathsf{mask}}$
    $T_e \leftarrow \mathsf{MAC.Tag}(K^{\mathsf{auth}}, \mathsf{pk^{DH}_{HSM}} \| n_e \| \mathsf{e\_cred})$
    $\mathsf{tr_C} \leftarrow \mathsf{H}_3(a_1, b_1, n_1)$
    $\mathsf{e} \xleftarrow{\$} \mathsf{AE.Enc}(K^{\mathsf{export}}, K)$
    $m \leftarrow \mathsf{e} \| \mathsf{tr_C} \| \mathsf{pk}_C \| \mathsf{e\_cred} \| n_e \| T_e$
    $\mathsf{E} \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk^{Enc}_{HSM}}; m)$
    send $\mathsf{E}$ to Server
    CleanUp
    return $(\textsc{InitResult}, K)$

Server
$\mathsf{acc}[\,] \leftarrow \{\}$

return $(\textsc{InitC}, \mathsf{ID_C})$

On input $(\textsc{InitS}, \mathsf{ID_C})$:
    choose fresh $\mathsf{aid_{new}} \in \{0,1\}^*$
    set $\mathsf{aid_{old}} \leftarrow \mathsf{acc}[\mathsf{ID_C}]$
    set $\mathsf{acc}[\mathsf{ID_C}] \leftarrow \mathsf{aid_{new}}$
    send $(\textsc{InitS}, \mathsf{aid_{new}}, \mathsf{aid_{old}}, a_1)$ to HSM
    on $(\textsc{InitResult}, \mathsf{aid_{new}}, \textsc{Fail})$ from HSM:
        return $(\textsc{InitResult}, \mathsf{ID_C}, \textsc{Fail})$
    on $\mathsf{aid_{new}}, b_1, n_1, \sigma_1$ from HSM: send $b_1, n_1, \sigma_1$ to Client
    on $\mathsf{E}$ from Client: send $(\textsc{File}, \mathsf{aid_{new}}, \mathsf{E})$ to HSM
    on $(\textsc{InitResult}, \mathsf{aid_{new}}, \mathsf{out})$ from HSM:
        return $(\textsc{InitResult}, \mathsf{ID_C}, \mathsf{out})$

HSM
$\mathsf{sk_{HSM}} = \{\mathsf{sk^{Sig}_{HSM}}, \mathsf{sk^{Enc}_{HSM}}, \mathsf{sk^{DH}_{HSM}}, K^{\mathsf{Enc}}_{\mathsf{HSM}}\}$

$\mathsf{sec\_delete}(\mathsf{aid_{old}})$
if $\mathsf{sec\_retr}(\mathsf{aid_{new}})$ is successful
    send $(\textsc{InitResult}, \mathsf{aid_{new}}, \textsc{Fail})$
else:
    $K^{\mathsf{PRF}}_{\mathsf{aid_{new}}} \xleftarrow{\$} \mathbb{Z}_p$
    $b_1 \leftarrow a_1^{K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}}$
    $n_1 \xleftarrow{\$} \{0,1\}^\lambda$
    $\sigma_1 \xleftarrow{\$} \Sigma.\mathsf{Sign}(\mathsf{sk^{Sig}_{HSM}}, b_1 \| n_1)$
    $\mathsf{tr_{HSM}} \leftarrow \mathsf{H}_3(a_1, b_1, n_1)$
    send $\mathsf{aid_{new}}, b_1, n_1, \sigma_1$
on $(\textsc{File}, \mathsf{aid_{new}}, \mathsf{E})$:
    $m \leftarrow \mathsf{PKE.Dec}(\mathsf{sk^{Enc}_{HSM}}; \mathsf{E})$
    parse
    $m = \mathsf{e} \| \mathsf{tr_C} \| \mathsf{pk}_C \| \mathsf{e\_cred} \| n_e \| T_e$
    if $\mathsf{tr_C} \neq \mathsf{tr_{HSM}}$ : $\mathsf{out} \leftarrow \textsc{Fail}$
    else:
        $m' \leftarrow \mathsf{e} \| \mathsf{pk}_C \| \mathsf{e\_cred} \| n_e \| T_e$
        $\mathsf{ctr} \leftarrow 10$
        $\mathsf{sec\_store}(\mathsf{aid_{new}}, (K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}, m', \mathsf{ctr}))$
        $\mathsf{out} \leftarrow \textsc{Succ}$
    send $(\textsc{InitResult}, \mathsf{aid_{new}}, \mathsf{out})$
```

Figure 4: The WBP initialization. Light blue boxes indicate 2HashDH instructions of OPAQUE; dark gray boxes denote other OPAQUE instructions; non-colored parts were added by WhatsApp. $\xrightarrow{\;a\;}\bullet$ is the $\mathsf{ID_C}$-authenticated transmission of a.



```
Client with phone number $\mathsf{ID_C}$              Server              HSM
$\mathsf{pw}', \mathsf{pk_{HSM}} = \{\mathsf{pk^{Sig}_{HSM}}, \mathsf{pk^{Enc}_{HSM}}, \mathsf{pk^{DH}_{HSM}}\}$    $\mathsf{acc}$    $\mathsf{sk_{HSM}} = \{\mathsf{sk^{Sig}_{HSM}}, \mathsf{sk^{Enc}_{HSM}}, \mathsf{sk^{DH}_{HSM}}, K^{\mathsf{Enc}}_{\mathsf{HSM}}\}$

Client:
On input $(\textsc{RecC}, \mathsf{pw}')$:
$r_2 \xleftarrow{\$} \mathbb{Z}_p, a_2 \leftarrow \mathsf{H}_1(\mathsf{pw}')^{r_2}$
$n_C \xleftarrow{\$} \{0,1\}^\lambda$
$(\bar{\mathsf{pk}}_C, \bar{\mathsf{sk}}_C) \xleftarrow{\$} \mathsf{DH.KeyGen}(1^\lambda)$

Client → Server: $n_C, \bar{\mathsf{pk}}_C, a_2$

Server:
return $(\textsc{RecC}, \mathsf{ID_C})$
On input $(\textsc{RecS}, \mathsf{ID_C})$:
if $\mathsf{acc}[\mathsf{ID_C}]$ unset:
return $(\textsc{RecResult}, \mathsf{ID_C}, \textsc{Fail})$
else :
set $\mathsf{aid} \leftarrow \mathsf{acc}[\mathsf{ID_C}]$

Server → HSM: $(\textsc{RecS}, \mathsf{aid}, n_C, \bar{\mathsf{pk}}_C, a_2)$

HSM:
$(K^{\mathsf{PRF}}_{\mathsf{aid}}, m', \mathsf{ctr}) \leftarrow \mathsf{sec\_retr}(\mathsf{aid})$
if no record can be found:
    HSM → Server: $(\textsc{RecResult}, \mathsf{aid}, \textsc{Fail})$
    Server: return $(\textsc{RecResult}, \mathsf{ID_C}, \textsc{Fail})$
else :
if $\mathsf{ctr} = 0$: $\mathsf{sec\_delete}(\mathsf{aid})$
    HSM → Server: $(\textsc{DelRec}, \mathsf{aid})$
    Server: return $(\textsc{DelRec}, \mathsf{ID_C})$
else :
parse $m' = \mathsf{e} \| \mathsf{pk}_C \| \mathsf{e\_cred} \| n_e \| T_e$
set $\mathsf{ctr}' \leftarrow (\mathsf{ctr} - 1)$
$\mathsf{sec\_set\_ctr}(\mathsf{aid}, \mathsf{ctr}')$
$b_2 \leftarrow a_2^{K^{\mathsf{PRF}}_{\mathsf{aid}}}, n_S \xleftarrow{\$} \{0,1\}^\lambda$
$(\bar{\mathsf{pk}}_S, \bar{\mathsf{sk}}_S) \xleftarrow{\$} \mathsf{DH.KeyGen}$
$\mathsf{ikm} \leftarrow (\bar{\mathsf{pk}}_C^{\bar{\mathsf{sk}}_S}, \bar{\mathsf{pk}}_C^{\mathsf{sk^{DH}_{HSM}}}, \mathsf{pk}_C^{\bar{\mathsf{sk}}_S})$
$\mathsf{pre} \leftarrow (a_2, n_c, \bar{\mathsf{pk}}_C, \mathsf{pk^{DH}_{HSM}}, \mathsf{e\_cred}, n_e, T_e, b_2, n_S, \bar{\mathsf{pk}}_S)$
$(K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk}) \leftarrow \mathsf{KDF}_2(\mathsf{ikm}, \mathsf{pre})$
$T_S \leftarrow \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_S, \mathsf{H}_3(\mathsf{pre}))$
$\sigma_2 \xleftarrow{\$} \Sigma.\mathsf{Sign}(\mathsf{sk^{Sig}_{HSM}}, b_2)$

HSM → Server: $\mathsf{aid}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$
Server → Client: $b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$

Client:
if $\Sigma.\mathsf{Vfy}(\mathsf{pk^{Sig}_{HSM}}; b_2; \sigma_2) \neq 1$
return $(\textsc{RecResult}, \textsc{Fail})$
else :
$y \leftarrow \mathsf{H}_2(\mathsf{pw}, b_2^{1/r_2})$
$(K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}}) \leftarrow \mathsf{KDF}_1(y, n_e)$
$T'_e \leftarrow \mathsf{MAC.Tag}(K^{\mathsf{auth}}, \mathsf{pk^{DH}_{HSM}} \| n_e \| \mathsf{e\_cred})$
if $T'_e \neq T_e$ return $(\textsc{RecResult}, \textsc{Fail})$
else :
$\mathsf{sk}_C \leftarrow \mathsf{e\_cred} \oplus K^{\mathsf{mask}}$
$\mathsf{pre} \leftarrow (a_2, n_c, \bar{\mathsf{pk}}_C, \mathsf{pk^{DH}_{HSM}}, \mathsf{e\_cred}, n_e, T_e, b_2, n_S, \bar{\mathsf{pk}}_S)$
$\mathsf{ikm} \leftarrow (\bar{\mathsf{pk}}_S^{\bar{\mathsf{sk}}_C}, \mathsf{pk^{DH}_{HSM}}^{\mathsf{sk}_C}, \bar{\mathsf{pk}}_S^{\mathsf{sk}_C})$
$(K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk}) \leftarrow \mathsf{KDF}_2(\mathsf{ikm}, \mathsf{pre})$
$T'_S \leftarrow \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_S, \mathsf{H}_3(\mathsf{pre}))$
if $T'_S \neq T_S$ return $(\textsc{RecResult}, \textsc{Fail})$
else :
$T'_C \leftarrow \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_C, \mathsf{H}_3(\mathsf{pre}, T'_S))$

Client → Server (authenticated): $T'_C$
Server → HSM: $(\textsc{RecResult}, \mathsf{aid}, T'_C)$

HSM:
$T_C \leftarrow \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_C, \mathsf{H}_3(\mathsf{pre} \| T_S))$
if $T'_C \neq T_C$:
    HSM → Server: $(\textsc{RecResult}, \mathsf{aid}, \textsc{Fail})$
    Server: return $(\textsc{RecResult}, \mathsf{ID_C}, \textsc{Fail})$
else :
$\mathsf{sec\_set\_ctr}$ (aid, 10)
$c \xleftarrow{\$} \mathsf{AE.Enc}(\mathsf{shk}, \mathsf{e})$

HSM → Server: $\mathsf{aid}, c$
Server: return $(\textsc{RecResult}, \mathsf{ID_C}, \textsc{Succ})$
Server → Client: $c$

Client:
$\mathsf{e} \leftarrow \mathsf{AE.Dec}(\mathsf{shk}, c)$
if $\mathsf{e} = \bot$:
CleanUp, return $(\textsc{RecResult}, \textsc{Fail})$
else :
$K \xleftarrow{\$} \mathsf{AE.Dec}(K^{\mathsf{export}}, \mathsf{e})$
CleanUp, return $(\textsc{RecResult}, K)$
```

Figure 5: The WBP key recovery. Light blue boxes indicate 2HashDH instructions of OPAQUE; light gray boxes mark 3DH of OPAQUE; dark gray boxes denote other OPAQUE instructions; non-colored parts were added by WhatsApp. $\xrightarrow{a}\!\bullet$ is the $\mathsf{ID_C}$-authenticated transmission of a.

computing $a_1 \leftarrow \mathsf{H}_1(\mathsf{pw})^{r_1}$ using a hash function $\mathsf{H}_1 \colon \{0,1\}^* \rightarrow \mathbb{G}$. The client sends a_1 to the server over the $\mathsf{ID_C}$-authenticated channel.

Upon receiving a_1 from $\mathsf{ID_C}$, the server chooses a fresh $\mathsf{aid_{new}} \in \{0,1\}^*$ that is called "account identifier" by WhatsApp[7] and checks if the client with $\mathsf{ID_C}$ has ever initiated the protocol and thus has already an aid in its array acc. If so, it sets $\mathsf{aid_{old}} \leftarrow \mathsf{acc}[\mathsf{ID_C}]$ and $\mathsf{acc}[\mathsf{ID_C}] \leftarrow \mathsf{aid_{new}}$, otherwise it sets $\mathsf{aid_{old}} = \perp$. Finally, the server sends $\mathsf{aid_{new}}, \mathsf{aid_{old}}, a_1$ to the HSM. Observe that the HSM *never* receives any identifying information (e.g., $\mathsf{ID_C}$) about the clients other than the value aid. That is, the HSM is *not* aware of the concept of a client $\mathsf{ID_C}$.

Upon receiving the server's message, the HSM "tombstones" all information associated with $\mathsf{aid_{old}}$ from its outsourced storage (with the instruction $\mathsf{sec\_delete}(\mathsf{aid_{old}})$). Then it checks whether $\mathsf{aid_{new}}$ has ever been used before.[8] If it was, then the HSM aborts and outputs $\mathsf{aid_{new}}, \textsc{Fail}$ to the server. If the HSM sees $\mathsf{aid_{new}}$ for the first time, it picks a random key $K^{\mathsf{PRF}}_{\mathsf{aid_{new}}} \xleftarrow{\$} \mathbb{Z}_p$ for that specific $\mathsf{aid_{new}}$ to be used in the 2HashDH OPRF. The HSM then uses the client's blinded password a_1 to compute $b_1 \leftarrow a_1^{K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}}$. Furthermore, the HSM samples a nonce $n_1 \xleftarrow{\$} \{0,1\}^\lambda$ uniformly at random and signs $b_1 \parallel n_1$ under its secret key $\mathsf{sk^{Sig}_{HSM}}$. Finally, it computes a transcript hash $\mathsf{tr_{HSM}}$ of the values a_1, b_1, n_1 with a hash function $\mathsf{H}_3 \colon \{0,1\}^* \rightarrow \{0,1\}^\lambda$, and sends back $\mathsf{aid_{new}}, b_1, n_1$ together with the resulting signature σ_1 to the server, which relays b_1, n_1, σ_1 to the client.

After receiving the HSM's message from the server, the client first verifies σ_1 using $\mathsf{pk^{Sig}_{HSM}}$ and aborts if the verification fails. Otherwise, it unblinds the server's response b_1 using the randomness r_1 and derives the OPRF output $y \leftarrow \mathsf{H}_2(\mathsf{pw}, b_1^{1/r_1})$ with hash function $\mathsf{H}_2 \colon \{0,1\}^* \rightarrow \{0,1\}^\lambda$. Next, further keys $K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}}$ are derived from the OPRF output y with the help of a key derivation function KDF_1. The obtained keys are used as follows. K^{mask} is used as an XOR mask to obtain e_cred, hiding the client's Diffie-Hellman secret sk_C. K^{auth} is used to compute a MAC tag T_e over $\mathsf{pk^{DH}_{HSM}} \parallel n_e \parallel \mathsf{e\_cred}$, where n_e is a randomly sampled nonce of length λ. Finally, K^{export} is used to encrypt K to produce the envelope[9] $\mathsf{e} \leftarrow \mathsf{AE.Enc}(K^{\mathsf{export}}, K)$. Similarly to the HSM, the client also computes a transcript hash $\mathsf{tr_C} \leftarrow \mathsf{H}_3(a_1, b_1, n_1)$ and compose a message $m \leftarrow \mathsf{e} \parallel \mathsf{tr_C} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e$, which is then encrypted under the HSM's public key ($\mathsf{E} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk^{Enc}_{HSM}}, m)$) to hide its content from the intermediary server. The encrypted envelope E is sent to the server over the $\mathsf{ID_C}$-authenticated channel. The client runs CleanUp to delete all assigned variables and received messages[10] (including r_1, sk_C) and outputs the backup key K.

Upon receiving E from a client with $\mathsf{ID_C}$, the server looks up $\mathsf{acc}[\mathsf{ID_C}] \leftarrow \mathsf{aid_{new}}$ and forwards $\mathsf{aid_{new}}, \mathsf{E}$ to the HSM. After receiving the message, the HSM decrypts E and checks whether the received transcript hash $\mathsf{tr_C}$ matches its own view of the transcript ($\mathsf{tr_{HSM}}$). If the transcripts do not match, it aborts sending $\mathsf{aid_{new}}, \textsc{Fail}$ to the server that outputs $\mathsf{ID_C}, \textsc{Fail}$. In case of matching transcripts, the HSM initializes a counter ctr that aims to track the unsuccessful key recovery attempts and stores in the secure storage the tuple $(\mathsf{aid_{new}}, K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}, \mathsf{e} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e, \mathsf{ctr})$. Finally, it informs the server about the successful completion of the initialization phase by sending $\mathsf{aid_{new}}, \textsc{Succ}$ that outputs $\mathsf{ID_C}, \textsc{Succ}$ concluding the key initialization.

[7]We remark that this terminology is slightly misleading, as aid does not identify a client's account but is rather a "backup identifier". If the same client initializes many backups, possibly with different passwords, then each backup will be assigned a new aid and only the most recent backup is kept.

[8]To this end, the HSM tries to retrieve a backup associated with $\mathsf{aid_{new}}$ from the secure storage. If $\mathsf{aid_{new}}$ is currently in use, this will succeed. If $\mathsf{aid_{new}}$ was previously used but corresponds to an already deleted backup, an empty "tombstoned" backup is returned to the HSM, showing that $\mathsf{aid_{new}}$ is not fresh.

[9]Note that the WBP's envelope is not equivalent to an OPAQUE envelope.

[10]We note that the abstract CleanUp instruction might be implemented without any explicit deletion, e.g., by keeping these ephemeral values only in volatile memory and never storing them persistently.

Key Recovery. The goal of the WBP recovery phase is to enable users, who do not have access anymore to their backup key K, to recover it using the password they entered during key initialization. On input of pw', the WhatsApp client app first prepares its input r_2, a_2 to the 2HashDH OPRF the same way as during initialization, samples an ephemeral Diffie–Hellman key-pair $(\bar{\mathsf{pk}}_C, \bar{\mathsf{sk}}_C)$ for the 3DH protocol, and also samples a uniformly random nonce n_C. Then $n_C, \bar{\mathsf{pk}}_C, a_2$ are sent to the server over the $\mathsf{ID_C}$-authenticated channel.

Upon receiving the client's message, the server checks the array acc and if it does not contain $\mathsf{ID_C}$, then it aborts because no user with the identifier $\mathsf{ID_C}$ initialized any backup keys. Otherwise, it attaches the account identifier $\mathsf{aid} \leftarrow \mathsf{acc}[\mathsf{ID_C}]$ to the client's message and sends these to the HSM.

When receiving the server's message, the HSM retrieves the record from the secure storage that is indexed by aid. If no such record is found, it returns failure to the server. Otherwise the HSM retrieves the record containing the per-client OPRF-key $K^{\mathsf{PRF}}_{\mathsf{aid}}$, the current counter value ctr, and m' that is parsed as $m' = \mathsf{e} \,\|\, \mathsf{pk}_C \,\|\, \mathsf{e\_cred} \,\|\, n_e \,\|\, T_e$. If $\mathsf{ctr} = 0$, then the HSM deletes the record indexed with aid from the storage and informs the server of this. If $\mathsf{ctr} > 0$, then its value is decreased by one and the stored record for aid is updated with the new ctr value. Next, as in the key initialization phase, the HSM computes $b_2 \leftarrow a_2^{K^{\mathsf{PRF}}_{\mathsf{aid}}}$ as a step of 2HsahDH. After sampling a uniformly random nonce n_S, the execution of the 3DH protocol steps follows. The HSM samples an ephemeral Diffie–Hellman key pair $(\bar{\mathsf{pk}}_S, \bar{\mathsf{sk}}_S)$ and computes three shared Diffie–Hellman secrets: $\bar{\mathsf{pk}}_C^{\bar{\mathsf{sk}}_S}, \bar{\mathsf{pk}}_C^{\mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}}, \mathsf{pk}_C^{\bar{\mathsf{sk}}_S}$. Using these shared secrets and the preamble pre, which is essentially a concatenation of the full protocol transcript $(a_2, n_c, \bar{\mathsf{pk}}_C, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}, \mathsf{e\_cred}, n_e, T_e, b_2, n_S, \bar{\mathsf{pk}}_S)$, it derives the keys $K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C$, and shk from a key derivation function KDF_2. Finally, with K^{MAC}_S it computes a MAC tag T_S over the hashed preamble, signs b_2 with $\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}$ to get signature σ_2 and sends its response composed of $\mathsf{aid}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$ to the server, who removes aid from the message and forwards the rest to the client.

After receiving the HSM's response from the server, the client first verifies the signature σ_2 and aborts if the verification fails. It then again derives the keys $K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}}$ and verifies the MAC T_e that was created during the initialization. If the verification failed, it aborts. Otherwise it continues to reconstruct sk_C by unmasking $\mathsf{e\_cred}$ with K^{mask} and then to derive the keys $K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk}$ from the three shared Diffie–Hellman secrets $\bar{\mathsf{pk}}_S^{\bar{\mathsf{sk}}_C}, {\mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}}^{\bar{\mathsf{sk}}_C}, \bar{\mathsf{pk}}_S^{\mathsf{sk}_C}$ and the preamble pre. Using K^{MAC}_S it verifies the MAC T_S and aborts if this is not successful. Otherwise the client computes MAC T'_C over a hash of $\mathsf{pre} \,\|\, T_S$ with the key K^{MAC}_C, which it then sends to the server through the $\mathsf{ID_C}$-authenticated channel.

After attaching aid to T'_C, the server forwards these values to the HSM. Since the HSM knows all values for computing the MAC tag that it has just received, it can verify the MAC. Note that with the MAC verification it essentially checks whether $\mathsf{pw} = \mathsf{pw}'$. If the MAC verification fails, it aborts, otherwise it resets the counter ctr to 10. Finally, the HSM encrypts e using shk and sends the resulting ciphertext c and aid to the server who forwards c to the client.

As the final steps of the recovery phase, the client first decrypts c to obtain e (it aborts if the AE decryption fails) and then decrypts e using the key K^{export} to obtain the backup key K. As in case of the initialization, before returning any output, the client always deletes all assigned variables and received messages (including $\mathsf{shk}, \mathsf{sk}_C\bar{\mathsf{sk}}_C, r_2$).

3.6 Extending the Number of Password Guesses

As we already noted in the protocol description in Section 3.5, the WBP only authenticates the client towards the server but not towards the HSM. The usage of the so-called "account identifier" aid aims to bridge this gap. The way it is used ensures that the HSM always associates every recovery request from the same $\mathsf{ID_C}$

with the same unique aid that was assigned for this $\mathsf{ID_C}$ during its last successful key initialization request. Furthermore, the HSM only keeps records of the last key initialization of a user under the aid that was assigned to the corresponding $\mathsf{ID_C}$ during this last initialization. Recall that in order to limit the number of password guesses against some account, each password guess has to be associated with the targeted account. It turns out that this cannot be guaranteed in case of the WBP when the server is malicious. The reason for this is that the server is in charge of assigning aids for $\mathsf{ID_C}$s, and neither the client nor the HSM can check this because the former never sees their assigned aid, and the latter never learns the $\mathsf{ID_C}$ of clients. This allows the server to increase the number of password guesses in certain cases.

We demonstrate the attack with an example. Let us assume that some client with identity $\mathsf{ID_C}$ has already initialized a key and the HSM stored the corresponding record under aid. Now if the same client runs key initialization again with the same password,[11] the server is assumed to instruct the HSM to delete the previous record by setting $\mathsf{aid_{old}} \leftarrow \mathsf{aid}$. However, the execution of this step completely depends on the server acting honestly. A malicious server might however proceed as if it has never seen $\mathsf{ID_C}$ before and make the HSM store q_I records for $\mathsf{ID_C}$, if the client with $\mathsf{ID_C}$ runs the key initialisation q_I times. If the client used the same password pw all q_I times, the malicious server will have $10q_I$ password guessing opportunities, since it knows all the q_I aids that are associated with $\mathsf{ID_C}$'s records.

Mitigating the attack. If the transcripts $\mathsf{tr_{HSM}}$ and $\mathsf{tr_C}$ contained information about the client identity $\mathsf{ID_C}$, in a way that both the client and the HSM can verify this, then they would be able to notice if the server is dishonest about the client identity. However, note that this countermeasure is very difficult to deploy retroactively, since any changes in the programming of the HSM would require the setup ceremony to be performed again.

4 Password-Protected Key Retrieval

In this section we give a formal definition of password-protected key retrieval (PPKR) in terms of an ideal functionality. For the unfamiliar reader, we provide a short introduction to the basic UC framework in Appendix B.

The Cryptographic Abstraction of the WBP. We introduce the concept of a *password-protected key retrieval* (PPKR) protocol, which is a 2-party protocol executed by a client and a server. (Note that in the WBP protocol, this client is the user's WhatsApp client and this server is the combination of the WhatsApp server and the HSM.) A PPKR protocol consists of two phases: (1) an initialization phase, where the client generates a symmetric encryption key and, using a password, securely stores it with a server, and (2) a recovery phase, where the client can recover their symmetric encryption key using a password.

The server may neither learn any information about the client's password, nor their key from these interactions, but only whether a recovery was successful. To provide a high-level intuition, we depict the input-output behavior of the PPKR functionality in Figure 6. Besides this, we demand several properties from a PPKR scheme that seem relevant for such a primitive in practice. These include protection against online and offline dictionary attacks, and that it provides cryptographically strong encryption keys.

Remark 1. We note here that alternative modelings are possible. For example, one could case-tailor the definition to the WhatsApp setting and formulate a variant of PPKR with three parties: the client, the

[11] WhatsApp is for mobile devices, connection loss may happen leading to a failure. After an unsuccessful attempt, the user would most probably re-run initialization, likely with the same password.

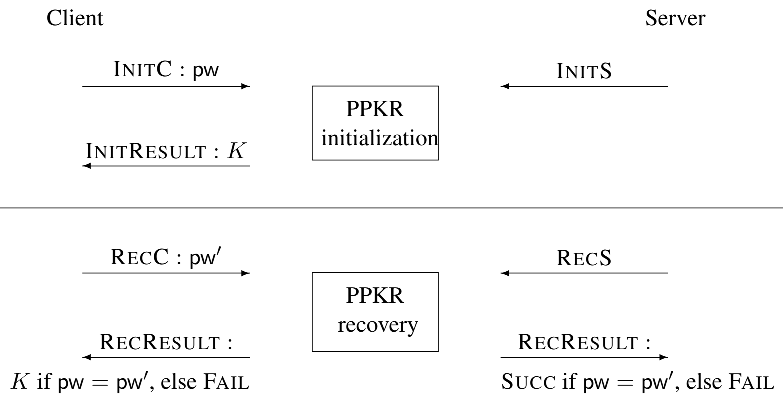


Figure 6: Schematic overview of password-protected key retrieval with initialization on top and recovery at the bottom, already using the interface names of our functionality $\mathcal{F}_{\mathsf{PPKR}}$. In both phases, the server does not provide any particular input but still has to participate in the protocol for the client to successfully initialize and recover a key K, which is modeled in the ideal functionality by letting it provide the INITS and RECS messages.

server, and the HSM. However, our notion with only two parties is more versatile: it can be used to analyze protocols where the server relies on an HSM, as well as protocols where the server acts on its own, or relies on arbitrarily many other entities, such as a cloud provider, offline storage, etc. The reason why this is possible is that usage of such "helpers" is well integrated into the UC framework [9] through the notion of so-called *hybrid functionalities*, which can be modularly "plugged" into protocol descriptions without "spilling" into the definition of the underlying primitive.

Expected Security Properties. We now describe which security properties we intuitively expect from a PPKR scheme, and thus want to formalize in an ideal PPKR functionality $\mathcal{F}_{\mathsf{PPKR}}$. The key K that is generated in a PPKR scheme should be usable without restrictions in any other application, thus we require that K is indistinguishable from random. Note that in particular this implies that neither the server nor the adversary can influence the generation of K in any way. Moreover, we expect K to remain secret from everyone but the client that computed it throughout the lifetime of the PPKR protocol. Hence, $\mathcal{F}_{\mathsf{PPKR}}$ should leak no information on the key, and any output given by $\mathcal{F}_{\mathsf{PPKR}}$ to the server or the adversary must be independent from K, unless the server gets corrupted and correctly guesses the client's password. In order to limit the ability to guess passwords via dictionary attacks, we expect $\mathcal{F}_{\mathsf{PPKR}}$ to (1) not leak any information about a password used by some client to any other party beyond whether a password used in a recovery is the same as in the most recent initialization by that client, and (2) allow only a small number of failed recovery attempts before K is deleted. We further expect that clients are authenticated towards the server[12] and cannot be impersonated by other clients or the adversary, to ensure that any client's key remains secret from all other clients even if they knew the correct password. However, note that a corrupt server might be able to skip client authentication[13] and execute the protocol on behalf of any client by itself,

[12]We leave the concrete means of authentication to the application. In the case of the WBP, SMS-based authentication is used, creating a one-to-one correspondence between $\mathsf{ID_C}$ and phone numbers of WhatsApp users. Other authentication methods such as biometrics (where $\mathsf{ID_C}$ would correspond to, e.g., a fingerprint) or even device-bound strong authentication using signatures are possible as well.

[13]We opted for a general treatment here, i.e., allowing client impersonation by the server. In fact, we could strengthen this (see Section 4.2 for more details) but this depends on which mechanisms on the server side are corruptible.

i.e., without interacting with any client. Further, we expect that the server is authenticated towards clients and cannot be impersonated by the adversary.

We summarize the list of expected security properties below.

- **Pseudorandomness of K**: Honest clients compute pseudorandom keys K, even if the server acts maliciously.
- **Secrecy of K**: Initialized and recovered keys of any honest client remain hidden from even a malicious server as long as the server does not correctly guess the honest client's password.
- **Uniqueness of K**: If the server is honest, initialization of a key K by user C buries any key that C previously initialized.
- **Oblivious passwords**: The initialization phase does not leak any information about the password to even a malicious server.
- **No online dictionary attacks**: The recovery phase does not leak any information (1) about the password used by the client to even a malicious server, and (2) about the initialized password to even a malicious client, beyond whether the password used during recovery matches the one used to create the backup.
- **Limited number of recovery attempts with wrong passwords**: Let K denote the key initialized by honest client C, and assume that C later runs the recovery phase 10 times consecutively with a wrong password. Then the server erases all K-dependent information, i.e., K cannot be recovered anymore. This must hold even if the client becomes maliciously corrupted after initializing K.
- **Limited number of offline guesses**: The above guarantee extends to maliciously corrupted servers, i.e., after 10 wrong password guesses to recover any honestly initialized key K, K is buried and cannot be retrieved by anyone anymore.[14]
- **Client authentication**: Only the client who initialized K or a malicious server can attempt to recover K. This must hold even if the password used during initialization becomes publicly known.
- **Server authentication**: There is only one server in the system and it cannot be impersonated by the adversary, unless the server gets corrupted.

4.1 A PPKR Functionality

In Figures 7 and 8 we describe the ideal functionality $\mathcal{F}_{\mathsf{PPKR}}$ for password-protected key retrieval. On a high level, $\mathcal{F}_{\mathsf{PPKR}}$ implements a password protected lookup table that contains clients' keys. When some client executes the initialization phase, an entry for that client is created in the lookup table or updated if an entry already existed. By executing the recovery phase, clients can access their entry in the table and recover their key, but only if they pass password authentication. If they fail password authentication 10 times in a row, $\mathcal{F}_{\mathsf{PPKR}}$ "buries the key" by erasing the corresponding table entry of that user. Note that while $\mathcal{F}_{\mathsf{PPKR}}$ maintains the table entries using client *identifiers*, $\mathcal{F}_{\mathsf{PPKR}}$ does not enforce the initialization and recovery processes to run on the same physical client machine. In our model, we understand the client's

[14]Note that the phrasing "any initialized" here reflects that the adversary can extend the number of admissible password guesses, as described in Section 3.6. This is necessary to model the security achieved by WhatsApp's protocol. We will discuss in Section 4.2 how the functionality can be strengthened.

party identifier as the identity under which the client device can authenticate. This way, if multiple devices can authenticate under the same identity (as is possible, e.g., for the SMS-based authentication in the WBP), InitC and RecC can be called from different machines.

Next we will explain the interfaces and record keeping of $\mathcal{F}_{\mathsf{PPKR}}$. In Figures 7 and 8 we labeled all instructions for easy referencing. $\mathcal{F}_{\mathsf{PPKR}}$ interacts with arbitrary clients and a single server S, where S is encoded in the session identifier sid (**server authentication**). The functionality internally maintains different types of records to keep track of ongoing and finished initialization and recovery phases. If $\mathcal{F}_{\mathsf{PPKR}}$ ever tries to retrieve a record that does not exist, it ignores the query causing this.

Initialization Phase. Whenever a client $\mathsf{ID_C}$ starts a new initialization, it calls the InitC interface with the password pw the user has chosen. $\mathcal{F}_{\mathsf{PPKR}}$ then (IC.1) generates a fresh key $K \xleftarrow{\$} \{0,1\}^\lambda$ (ensuring **pseudorandomness** of K) for $\mathsf{ID_C}$ and records that $\mathsf{ID_C}$ has started a new initialization by creating a record $\langle \text{InitC}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}, K\rangle$ (IC.2). This newly created record overwrites any existing record of type InitC for $\mathsf{ID_C}$, which ensures that a client can only have one ongoing intialization session. Finally, the functionality informs the server S and the adversary $\mathcal{A}$ that the client $\mathsf{ID_C}$ has started a new initialization (IC.3).

If the server agrees to participate in the initialization with $\mathsf{ID_C}$, it calls the InitS interface, which takes as input two (not necessarily distinct) client identities $\mathsf{ID_C}$ and $\mathsf{ID_C}^*$. The additional identity $\mathsf{ID_C}^*$ is only effective if the server is corrupt, and reflects that a malicious server can simply ignore client authentication and claim a different identity has authenticated to him. $\mathcal{F}_{\mathsf{PPKR}}$ now retrieves the InitC record of $\mathsf{ID_C}$, which ensures that the query only proceeds if $\mathsf{ID_C}$ has started a new initialization (IS.1). Then it creates a File record containing the password pw and key K from the retrieved InitC record and, depending on whether the server is honest or corrupt, either the identity $\mathsf{ID_C}$ or $\mathsf{ID_C}^*$ (IS.2 and IS.3). Thus, a corrupt server can freely choose for which identity the File record is created, while File records created for an honest server always contain the same identity as the corresponding InitC record. A newly created record overwrites any existing record of the same identity to ensure that any key K' that may have been generated in a previous initialization by $\mathsf{ID_C}$ cannot be recovered anymore (**Uniqueness of the key**). However, since a malicious server can make $\mathcal{F}_{\mathsf{PPKR}}$ store files under different identities $\mathsf{ID_C}^*$, this guarantee holds only as long as the server is honest. Indeed, $\mathcal{F}_{\mathsf{PPKR}}$ allows a malicious server to make $\mathcal{F}_{\mathsf{PPKR}}$ store *all the password-protected key records* that any honest client ever initialized (see Section 4.2 for a discussion of this weakness).

After the server agreed to participate in the initialization with $\mathsf{ID_C}$, the adversary may let $\mathcal{F}_{\mathsf{PPKR}}$ compute the output of the initialization phase for the client with either the interface CompleteInitC or CompleteInitC-DoS and for the server with either CompleteInitS or CompleteInitS-DoS. The functionality does not enforce an order in which the parties receive their output and leaves this decision to the adversary $\mathcal{A}$. All these interfaces ensure that only one output can be generated towards client and server for every ongoing initialization session, by retrieving (CIC.1,CICD.1) and deleting (CIS.1,CISD.1) the corresponding session records (in the case of CIS.1, the record is not deleted but assigned a special marking - see below for details). The adversary's choices are as follows:

- CompleteInitC outputs K from the session record to the client (CIC.2)
- CompleteInitC-DoS outputs Fail to the client (CICD.2)
- CompleteInitS outputs Succ to the server (CIS.3)
- CompleteInitS-DoS outputs Fail to the server (CISD.2)

Additionally to these outputs, the CompleteInitS interface ensures that $\mathcal{F}_{\mathsf{PPKR}}$ installs a password-protected backup key file for $\mathsf{ID_C}$ that contains K. This works as follows: instead of deleting it, $\mathcal{F}_{\mathsf{PPKR}}$ marks the File record of $\mathsf{ID_C}$ as Stored (CIS.1). Looking ahead, in the recovery phase clients can only recover keys from File records that are marked Stored. $\mathcal{F}_{\mathsf{PPKR}}$ then initializes a counter $\mathsf{tx_{sid}}[\mathsf{ID_C}]$ to 10

$\mathcal{F}_{\mathsf{PPKR}}$ 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 $(\textsc{MaliciousInit}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}^*, K^*)$ 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 $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}^*, K^* \rangle$, overwriting any existing record $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}, *, * \rangle$. Set $\mathsf{tx_{sid}}[\mathsf{ID_C}] \leftarrow 10$

On input $(\textsc{MaliciousRec}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}^*)$ 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 $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}, [\mathsf{pw}], [K] \rangle$ marked $\textsc{Stored}$.

MR.3 If $\mathsf{tx_{sid}}[\mathsf{ID_C}] = 0$, delete record $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}, K \rangle$ and output $(\textsc{DelRec}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{A}$ // The key is buried if zero guesses remain.

MR.4 If $\mathsf{pw}^* = \mathsf{pw}$, 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 $(\mathsf{sid}, \textsc{Fail})$ to $\mathcal{A}$

Initialization phase

On input $(\textsc{InitC}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw})$ 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 $\langle \textsc{InitC}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}, K \rangle$, overwriting any existing record $\langle \textsc{InitC}, \mathsf{sid}, \mathsf{ID_C}, *, * \rangle$ // Storing $\mathsf{ID_C}$'s current init state; a client can only be in one initialization session

IC.3 Send $(\textsc{InitC}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{A}$ and to S

On input $(\textsc{InitS}, \mathsf{sid}, \mathsf{ID_C}, \boxed{\mathsf{ID_C}^*})$ 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 $\langle \textsc{InitC}, \mathsf{sid}, \mathsf{ID_C}, [\mathsf{pw}], [K] \rangle$ // Continue only if $\mathsf{ID_C}$ started initialization already

IS.2 $\boxed{\text{If S is honest,}}$ record $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}, K \rangle$, overwriting any existing record $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}, *, * \rangle$, and send $(\textsc{InitS}, \mathsf{sid}, \mathsf{ID_C})$ 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 $\boxed{\text{Otherwise, record } \langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}^*, \mathsf{pw}, K \rangle \text{, overwriting any existing record } \langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}^*, *, * \rangle.}$

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

CIC.1 Retrieve record $\langle \textsc{InitC}, \mathsf{sid}, \mathsf{ID_C}, *, [K] \rangle$ and delete it

CIC.2 Output $(\textsc{InitResult}, \mathsf{sid}, K)$ to $\mathsf{ID_C}$ (or $\mathcal{A}$ if $\mathsf{ID_C}$ is corrupt)

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

CIS.1 Retrieve record $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}, *, * \rangle$ not marked $\textsc{Stored}$ and mark it $\textsc{Stored}$ // Note: there is only one such record thanks to overwriting in $\textsc{InitS}$ interface. This becomes the stored key now!

CIS.2 Set $\mathsf{tx_{sid}}[\mathsf{ID_C}] \leftarrow 10$

CIS.3 Send $(\textsc{InitResult}, \mathsf{sid}, \mathsf{ID_C}, \textsc{Succ})$ to S (or $\mathcal{A}$ if S is corrupt)

Attacks on Initialization Phase

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

CICD.1 Retrieve record $\langle \textsc{InitC}, \mathsf{sid}, \mathsf{ID_C}, *, * \rangle$ and delete it

CICD.2 Output $(\textsc{InitResult}, \mathsf{sid}, \textsc{Fail})$ to $\mathsf{ID_C}$ (or $\mathcal{A}$ if $\mathsf{ID_C}$ is corrupt)

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

CISD.1 Delete any record $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}, *, * \rangle$ // Server's state in current initialization session no longer needed

CISD.2 Send $(\textsc{InitResult}, \mathsf{sid}, \mathsf{ID_C}, \textsc{Fail})$ to S (or $\mathcal{A}$ if S is corrupt)

Figure 7: Ideal functionality $\mathcal{F}_{\mathsf{PPKR}}$ for password-protected key retrieval, offline attacks and initialization interfaces. For a stronger version of $\mathcal{F}_{\mathsf{PPKR}}$, boxed code can be dropped (see Section 4.2).

Recovery Phase

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

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

On input $(\textsc{RecS}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{ID_C}^*)$ 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 $\langle \textsc{RecC}, \mathsf{sid}, \mathsf{ID_C}, [\mathsf{pw}'] \rangle$ // Continue only if $\mathsf{ID_C}$ started recovery already
- RS.3 If there exists no record $\langle \textsc{File}, \mathsf{ID_C}', \mathsf{sid}, [\mathsf{pw}], [K] \rangle$ marked $\textsc{Stored}$, send $(\textsc{RecResult}, \mathsf{aid}, \textsc{Fail})$ 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 $\langle \textsc{File}, \mathsf{sid}, \mathsf{ID_C}', \mathsf{pw}, K \rangle$ marked $\textsc{Stored}$ and send $(\textsc{DelRec}, \mathsf{sid}, \mathsf{ID_C}')$ 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 $\langle \textsc{RecC}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}' \rangle$, overwriting any existing record $\langle \textsc{RecC}, \mathsf{sid}, \mathsf{ID_C}, *, *, * \rangle$ // The recovery session of $\mathsf{ID_C}$ is used
- RS.7 Send $(\textsc{RecS}, \mathsf{sid}, \mathsf{ID_C}', \mathsf{pw} \overset{?}{=} \mathsf{pw}')$ to $\mathcal{A}$ // A ppKR protocol may not hide whether recovery was successful or not

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

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

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

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

Attacks on Recovery Phase

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

- CRCD.1 Retrieve record $\langle \textsc{RecC}, \mathsf{sid}, \mathsf{ID_C}, *, *, * \rangle$ and delete it.
- CRCD.2 Send $(\textsc{RecResult}, \mathsf{sid}, \textsc{Fail})$ to $\mathsf{ID_C}$ (or $\mathcal{A}$ if $\mathsf{ID_C}$ is corrupt).

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

- CRSC.1 Retrieve record $\langle \textsc{RecC}, \mathsf{sid}, \mathsf{ID_C}, *, *, * \rangle$ and delete it.
- CRSD.2 Output $(\textsc{RecResult}, \mathsf{sid}, \mathsf{ID_C}, \textsc{Fail})$ to S (or $\mathcal{A}$ if S is corrupt).

Figure 8: Ideal functionality $\mathcal{F}_{\mathsf{PPKR}}$, recovery interfaces.

(CIS.2), which indicates the remaining recovery attempts for the newly created FILE record, and sends the output $(\textsc{InitResult}, \mathsf{sid}, \mathsf{ID_C}, \textsc{Succ})$ to the server (CIS.3).

This concludes the description of $\mathcal{F}_{\mathsf{PPKR}}$'s initialization phase. The absence of any K- or pw-dependent information in the outputs towards the server (IC.3,CIS.3,CISD.2) and the adversary (IS.2) ensures **secrecy of** K and **password obliviousness** during initialization.

Recovery Phase. The general structure of record keeping and interfaces of the recovery phase are very similar to the initialization phase. First, a client starts a recovery session with the RECC interface, then the server has to agree in participating in the recovery with the RECS interface, and finally there are again for each party two interfaces to let $\mathcal{F}_{\mathsf{PPKR}}$ output either success or failure to the parties. For the RECC interface, $\mathsf{ID_C}$ provides as input the password pw' it chose for this recovery attempt. $\mathcal{F}_{\mathsf{PPKR}}$ then records this in a RECC record (RC.1), again overwriting any existing RECC record, and outputs to S and $\mathcal{A}$ that $\mathsf{ID_C}$ started a recovery session (RC.2).

If the server agrees to participate in the recovery session with $\mathsf{ID_C}$ it calls the RECS interface with two (not necessarily distinct) client identities $\mathsf{ID_C}$ and $\mathsf{ID_C}^*$, where again the second identity $\mathsf{ID_C}^*$ being given by the server allows a malicious server to ignore client authentication and claim a different identity is recovering. $\mathcal{F}_{\mathsf{PPKR}}$ now does three things:

- $\mathcal{F}_{\mathsf{PPKR}}$ checks whether $\mathsf{ID_C}$ has started recovery in the first place by looking for a corresponding client recovery session (RS.2) and drops the query otherwise.
- Then, $\mathcal{F}_{\mathsf{PPKR}}$ grabs password and key from the stored key record with identity $\mathsf{ID_C}'$ (RS.3), and writes them into the client's recovery session record. With this, $\mathcal{F}_{\mathsf{PPKR}}$ lets a *corrupt* server, who can let $\mathsf{ID_C}'$ be any $\mathsf{ID_C}^*$ (RS.1), re-route the recovery attempt of any honest client to the password-protected key record of *any other* honest client, which is again motivated by the fact that a malicious server can simply skip client authentication. Conversely, if the server is honest, $\mathcal{F}_{\mathsf{PPKR}}$ ensures $\mathsf{ID_C}' = \mathsf{ID_C}$ and hence gives access to some FILE record containing an identity $\mathsf{ID_C}$ only to a client with that identity (**Client authentication** during recovery).
- After finding a file for a recovery attempt in the previous step, $\mathcal{F}_{\mathsf{PPKR}}$ checks if the recovery attempt counter for $\mathsf{ID_C}'$ has reached zero (RS.4). In that case, it deletes the FILE record of $\mathsf{ID_C}'$ marked STORED to ensure that the key contained in that record cannot be recovered anymore (**limited number of recovery attempts with wrong passwords**) and outputs DELREC to S and $\mathcal{A}$. Otherwise, the recovery attempt counter for $\mathsf{ID_C}'$ is decremented (RS.5) and the password pw and key K obtained from the FILE record are appended to the RECC record (RS.6).

Extending the record again serves the purpose of recording that S agreed to participate in the recovery with $\mathsf{ID_C}$. Note that this extended record is not deleted if $\mathsf{ID_C}$ starts a new recovery session with another call to the RECC interface. This reflects the fact that the server should still be able to finish a recovery session until it agrees to participate in another recovery session with $\mathsf{ID_C}$. Finally, the functionality gives the output $(\textsc{RecS}, \mathsf{sid}, \mathsf{ID_C}', \mathsf{pw} \overset{?}{=} \mathsf{pw}')$ to the adversary $\mathcal{A}$ (RS.7). We let $\mathcal{F}_{\mathsf{PPKR}}$ leak the latter bit to the adversary because many protocols, including the WBP, leak via their communication pattern whether the client used the correct password or not. For example, a server might only send its last message to the client if it has previously learned that the client's password was correct.

To complete the recovery session, the adversary calls either COMPLETERECC or COMPLETERECC-DOS and either COMPLETERECS or COMPLETERECS-DOS, again with $\mathcal{F}_{\mathsf{PPKR}}$ enforcing no order in which the interfaces are called. In COMPLETERECC the functionality retrieves the extended RECC record (CRC.1) and compares the two password pw and pw' contained in it (CRC.2). If they are the same, it outputs the key K contained in the record to the client and otherwise outputs FAIL to the client. To ensure that both the client and the server can finish the recovery in any order, the RECC record is marked as RECOVERED or deleted if it already is marked as RECOVERED (CRC.1). Similarly, in the COMPLETERECS interface $\mathcal{F}_{\mathsf{PPKR}}$ retrieves the extended RECC record (CRS.1) and outputs either SUCC or FAIL to the server depending on whether the passwords pw and pw' in the RECC record match (CRS.2 and CRS.3). Additionally, if the passwords match, the recovery attempt counter $\mathsf{tx}_{\mathsf{sid}}[\mathsf{ID}_{\mathsf{C}}]$ is reset to 10 (CRS.2). To again enforce no order on which party receives its outputs first, the RECC record is marked as RECOVERED or deleted if it already is marked as RECOVERED. Using this mechanism both parties receive their output once, and the record is deleted after both parties received their output.

The two interfaces COMPLETERECC-DOS and COMPLETERECS-DOS behave exactly the same. The RECC record of ID_{C} is deleted (CRCD.1 and CRSD.1) and $\mathcal{F}_{\mathsf{PPKR}}$ outputs FAIL to the corresponding party (CRCD.2 and CRSD.2). In these interface we do not use the mechanism of marking the record RECOVERED, since we assume that if either party produces the output FAIL the other party cannot successfully finish the recovery anymore.

It can be seen from the outputs towards the server (RC.2,CRS.2-3,CRSD.2) and the adversary (RS.7) that only one bit of information about the password used by an honest client during recovery (i.e., match or no match) is leaked, protecting the client from **online dictionary attacks**. Similarly, adversary and client learn only the match bit about the password contained in the file that the server uses during a recovery session (RS.7,CRC.3,CRCD.2).

Offline Attacks. The adversary has access to two interfaces MALICIOUSREC and MALICIOUSINIT to mount offline attacks. In both interfaces the adversary impersonates some client ID_{C}, however, as described in the discussion of the expected security properties, a client can only be impersonated if the server is corrupt. Therefore, both queries are ignored by $\mathcal{F}_{\mathsf{PPKR}}$ if S is honest (MR.1 and MI.1).

With the MALICIOUSINIT interface the adversary impersonates a client ID_{C} and executes a new initialization for ID_{C}. For this, $\mathcal{A}$ can choose a new password pw^* and a new key K^*, which are then stored in a new FILE record marked STORED that overwrites any existing FILE record for ID_{C} (MI.2). $\mathcal{F}_{\mathsf{PPKR}}$ resets the counter $\mathsf{tx}_{\mathsf{sid}}$ to 10 since with any new initialization the client gets 10 new recovery attempts.

With the interface MALICIOUSREC the adversary impersonates a client and tries to recover a client's key from a guessed password. To this end, $\mathcal{A}$ inputs the client identity ID_{C} and a password guess pw^*. $\mathcal{F}_{\mathsf{PPKR}}$ retrieves the FILE record of ID_{C} marked STORED (MR.2) and checks if ID_{C} has any recovery attempts left by checking if $\mathsf{tx}_{\mathsf{sid}}[\mathsf{ID}_{\mathsf{C}}] = 0$. If ID_{C} has no recovery attempts left, it deletes the FILE record of ID_{C} (**limited number of offline guesses**) and outputs $(\text{DELREC}, \mathsf{sid}, \mathsf{ID}_{\mathsf{C}})$ to $\mathcal{A}$ to notify $\mathcal{A}$ that the record was deleted (MR.3). Otherwise, the functionality proceeds to check whether the guessed password pw^* matches the password pw from the FILE record. If the passwords match, the adversary gets access to the key stored in the FILE record, and otherwise $\mathcal{F}_{\mathsf{PPKR}}$ returns FAIL to the adversary (MR.4).

Differences between PPKR and 1-PPSS A password-protected secret sharing scheme [3] allows a user to retrieve a password-protected secret from a set of servers. The servers cannot derive or offline-attack the user's data unless a certain subset of them colludes. A PPKR scheme could be interpreted as a 1-PPSS scheme, i.e., where only one server is involved in storing and retrieving the password-protected secret. While both primitives are very similar considering only honest parties, it is actually the server corruption model that greatly differs for PPKR and 1-PPSS. Intuitively, the one server of a 1-PPSS scheme holds the only share of a secret (or key, in the terminology of PPKR), i.e., the full secret. If such a server is compromised, unlimited offline guesses on the shared user secret are unavoidable. PPKR is stronger: upon server compromise, only a *limited* number of password guesses are allowed on user secrets. Hence, PPKR never falls back to 1-PPSS, due to the stronger guarantees upon server compromise.

4.2 On Strengthening $\mathcal{F}_{\mathsf{PPKR}}$

We discuss a potential strengthening of $\mathcal{F}_{\mathsf{PPKR}}$ regarding the limitation of offline guesses by a malicious server. While $\mathcal{F}_{\mathsf{PPKR}}$ buries keys of honest users whenever a user re-initializes (e.g., when refreshing the key, or when changing the password) as long as the server is honest, it does not guarantee uniqueness of clients' backup keys if the server is corrupted. Consequently, the limitation of offline guessing attempts holds only *per initialized key* of a user, and not *per user*. The reason why we go with the weaker $\mathcal{F}_{\mathsf{PPKR}}$ is that the WBP cannot guarantee the stronger version, and thus we have to reflect this weakness for the security analysis of the actual protocol. However, for completeness, we state here the properties that we would ideally like to demand from $\mathcal{F}_{\mathsf{PPKR}}$, and the corresponding functionality can be read from Figure 7 by dropping the $\boxed{\text{boxed code}}$. Figure 8 does not change. A bit more detailed, we could strengthen $\mathcal{F}_{\mathsf{PPKR}}$ by disallowing the adversary to reroute initializations of honest clients $\mathsf{ID_C}$ to new identities $\mathsf{ID_C}^*$, such that upon honest re-initialization, $\mathcal{F}_{\mathsf{PPKR}}$ always buries the former key of that client. The security guarantees are then strengthened as follows.

- **Uniqueness of** K: ~~If the server is honest,~~ initialization of a key K by user C buries any key that C previously initialized, *even if the server is malicious.*
- **Limited number of offline guesses**: The above guarantee extends to maliciously corrupted servers, i.e., after 10 wrong password guesses to recover ~~any honestly initialized~~ *the latest honestly initialized key K of any client*, K is buried and cannot be retrieved by anyone anymore.

5 Security Analysis

The Difficulty of a Security Analysis. One might hope that the security of the WBP directly follows from the security of its OPAQUE component. There are two main reasons why that does not hold true. First, OPAQUE is a *key exchange* protocol that results in two parties sharing a key, while the goal of the WBP is to *hide the key from the server*. Second, the WBP deploys version 03 of the OPAQUE Internet Draft [26] to which the security analysis of the OPAQUE framework [24] does not apply, for a multitude of reasons that we describe in Appendix G.

The main challenge when performing a security analysis of the WBP is to tame its complexity (cf. Figure 4 and Figure 5). Since we want to focus on the actual cryptographic protocol, we do not to include the way in which the HSM outsources data storage in the security analysis. WhatsApp deploys a Merkle tree based outsourced storage routine that lets the HSM put encrypted data onto the WhatsApp server's storage, which should guarantee integrity of the data and the ability of the HSM to detect malicious erasures by the WhatsApp server. We leave the analysis of this scheme as a future task, and in this work simply make the exact same assumptions about it that are claimed in the WBP whitepaper [33]. To further tame the complexity of the proof of the WBP, we modularize the security proofs of the underlying OPRF and the authenticated key exchange (AKE) scheme 3DH. Since previous security analyses did not apply to the protocol versions deployed by the WBP, we first show their security separately, which may be of independent interest. Then we use the resulting simulators as subcomponents of the WBP simulator. This proof technique, which was already used for AKE in [24], avoids formulating the WBP in the AKE- and OPRF-hybrid model. The latter is not even possible, due to the non-black-box use that the WBP makes of these components. Altogether, our analysis (1) shows a lower bound on the security of the WBP, (2) shows the security of the multi-key

2HashDH OPRF and AKE building blocks as deployed by the OPAQUE internet draft version 3 [26], as well as malicious client security of that OPAQUE version.[15]

Modeling the HSM. We model the HSM as a hybrid functionality $\mathcal{F}_{\mathsf{HSM}}$ that can be queried by the server as in Figure 4 and Figure 5. That is, $\mathcal{F}_{\mathsf{HSM}}$ contains exactly the code that the HSM contains in these figures. For completeness and clarity, $\mathcal{F}_{\mathsf{HSM}}$ is depicted in Figure 9. In addition, $\mathcal{F}_{\mathsf{HSM}}$ provides an interface for clients to retrieve $\mathsf{pk}_{\mathsf{HSM}}$, which models the setup process that ensures that clients have the "right" WhatsApp public key hard-coded into their smartphones.[16]

In the UC framework, messages sent between some party and an ideal functionality are perfectly secure, meaning no network adversary can intercept them or tamper with them. So modeling the HSM key distribution as idealized communication expresses our assumptions

- that the user installs the correct WhatsApp client on her phone,
- that WhatsApp's setup ceremony of the HSM leads to honestly generated keys being distributed to the clients,
- and that only the HSM knows the secret key of the HSM.

However, analyzing the mechanisms to ensure the above assumptions is not in the scope of this work.

Treatment of Out-of-Order Messages. We assume that the protocol parties ignore messages that are received out-of-order. That means that during an initialization or a recovery, the parties do nothing upon receiving a message that is not formatted as expected. The only exception to this is the respective first message of an initialization or recovery, i.e., a_1 for initialization or $(n_C, \bar{\mathsf{pk}}_C, a_2)$ for recovery. If such a message is received, the currently running initialization (or recovery, resp.) is discarded and the new initialization (or recovery, resp.) is continued. This way, there is never more than one initialization (or recovery, resp.) running per user.

Corruption Model. All corruptions are malicious, meaning that the adversary can fully control not only the communication but also the behavior of a corrupted party. We consider adaptive corruptions of clients and the server. However, we add the restriction that clients cannot be corrupted during an ongoing initialization or recovery session. More precisely, the environment is not allowed to corrupt some client $\mathsf{ID_C}$ if, following the most recent $(\textsc{InitC}, *)$ input from the environment to $\mathsf{ID_C}$, $\mathsf{ID_C}$ did not produce a corresponding output $(\textsc{InitResult}, *)$ yet. Analogously, if following the most recent $(\textsc{RecC}, *)$ input from the environment to $\mathsf{ID_C}$, $\mathsf{ID_C}$ did not produce a corresponding output $(\textsc{RecResult}, *)$ yet, the environment is not allowed to corrupt $\mathsf{ID_C}$ as well. We deem this reasonable, since the time it takes to execute the initialization or recovery protocol is relatively short. Formally, this means that the effect of adaptive client corruptions is that the adversary (1) learns all values that the client stores after completion of an initialization or recovery phase, namely key K, and (2) controls the client behavior from that point on.

Let AE' be the AE scheme that is implicitly used in Figures 4 and 5 to encrypt sk_C, that is:

- $\mathsf{KeyGen}'(1^\lambda) \to (K^{\mathsf{mask}}, K^{\mathsf{auth}})$, where $K^{\mathsf{mask}}, K^{\mathsf{auth}} \overset{\$}{\leftarrow} \{0,1\}^\lambda$

[15]Our proof considers the security of OPAQUE only against a malicious client, since the OPAQUE server is run on the incorruptible HSM.

[16]One might be tempted to model this by giving the HSM's public key as input to the client instead. However, that would mean that the UC environment machine can give public keys to clients for which the environment knows the corresponding secret key. For WBP the clients have a hard-coded public key for which only the HSM knows the secret key, so this would not adequately model WBP and make the already complex security analysis unreasonably more complex.

Ideal functionality $\mathcal{F}_{\mathsf{HSM}}$

The functionality talks to a server S hardcoded in sid and to arbitrary other parties P.

```
Initially:
$K^{\mathsf{Enc}}_{\mathsf{HSM}} \xleftarrow{\$} \{0,1\}^{\lambda}$
$(\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{Sig}}_{\mathsf{HSM}}) \xleftarrow{\$} \Sigma.\mathsf{KeyGen}(1^{\lambda})$
$(\mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{Enc}}_{\mathsf{HSM}}) \xleftarrow{\$} \mathsf{PKE.KeyGen}(1^{\lambda})$
$(\mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}) \xleftarrow{\$} \mathsf{DH.KeyGen}(1^{\lambda})$
$\mathsf{pk}_{\mathsf{HSM}} = \{\mathsf{pk}^{\mathsf{Sig}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{Enc}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}\}$
$\mathsf{sk}_{\mathsf{HSM}} = \{\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}, \mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}, \mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}, K^{\mathsf{Enc}}_{\mathsf{HSM}}\}$

On input $(\textsc{GetPK})$ from $\mathsf{P} \in \{\mathsf{ID_C}, \mathsf{S}, \mathcal{A}\}$
return $\mathsf{pk}_{\mathsf{HSM}}$ to $\mathsf{P}$

On input $(\textsc{InitS}, \mathsf{aid_{new}}, \mathsf{aid_{old}}, a_1)$ from $\mathsf{S}$ :
$\mathsf{sec\_delete}(\mathsf{aid_{old}})$
if $\mathsf{sec\_retr}(\mathsf{aid_{new}})$ is successful:
    return $(\textsc{InitResult}, \mathsf{aid_{new}}, \textsc{Fail})$
else :
$K^{\mathsf{PRF}}_{\mathsf{aid_{new}}} \xleftarrow{\$} \mathbb{Z}_p$
$b_1 \leftarrow a_1^{K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}}$
$n_1 \xleftarrow{\$} \{0,1\}^{\lambda}$
$\sigma_1 \xleftarrow{\$} \Sigma.\mathsf{Sign}(\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}; b_1 \parallel n_1)$
$\mathsf{tr_{HSM}} \leftarrow \mathsf{H}_3(a_1, b_1, n_1)$
return $(\mathsf{aid_{new}}, b_1, n_1, \sigma_1)$ to $\mathsf{S}$

On input $(\textsc{File}, \mathsf{aid_{new}}, \mathsf{E})$ :
$m \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}; \mathsf{E})$
parse $m = (\mathsf{e} \parallel \mathsf{tr_C} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e)$
if $\mathsf{tr_C} \neq \mathsf{tr_{HSM}}$ :
    return $(\textsc{InitResult}, \mathsf{aid_{new}}, \textsc{Fail})$ to $\mathsf{S}$
else :
$\mathsf{sec\_store}(\mathsf{aid_{new}}, (K^{\mathsf{PRF}}_{\mathsf{aid_{new}}},$
    $\mathsf{e} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e, \mathsf{ctr} \leftarrow 10))$
return $(\textsc{InitResult}, \mathsf{aid_{new}}, \textsc{Succ})$ to $\mathsf{S}$

On input $(\textsc{RecS}, \mathsf{aid}, n_c, \bar{\mathsf{pk}}_C, a_2)$ :
$(K^{\mathsf{PRF}}_{\mathsf{aid}}, m, \mathsf{ctr}) \leftarrow \mathsf{sec\_retr}(\mathsf{aid})$
if no record can be found:
    return $(\textsc{RecResult}, \mathsf{aid}, \textsc{Fail})$
parse $m = \mathsf{e} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e$
if $\mathsf{ctr} = 0$ :
    return $(\textsc{DelRec}, \mathsf{aid})$
else :
set $\mathsf{ctr}' \leftarrow \mathsf{ctr} - 1$
$\mathsf{sec\_set\_ctr}(\mathsf{aid}, \mathsf{ctr}')$
$b_2 \leftarrow a_2^{K^{\mathsf{PRF}}_{\mathsf{aid}}}$
$n_S \xleftarrow{\$} \{0,1\}^{\lambda}$
$(\bar{\mathsf{pk}}_S, \bar{\mathsf{sk}}_S) \xleftarrow{\$} \mathsf{DH.KeyGen}$
$\mathsf{pre} \leftarrow (a_2, n_c, \bar{\mathsf{pk}}_C, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}, \mathsf{e\_cred},$
    $n_e, T_e, b_2, n_S, \bar{\mathsf{pk}}_S)$
$\mathsf{ikm} \leftarrow (\bar{\mathsf{pk}}_C^{\bar{\mathsf{sk}}_S}, \bar{\mathsf{pk}}_C^{\mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}}, \mathsf{pk}_C^{\bar{\mathsf{sk}}_S})$
$(K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk}) \leftarrow \mathsf{KDF}_2(\mathsf{ikm}, \mathsf{pre})$
$T_S \leftarrow \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_S, \mathsf{H}_3(\mathsf{pre}))$
$\sigma_2 \xleftarrow{\$} \Sigma.\mathsf{Sign}(\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}; b_2)$
return $(\mathsf{aid}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2)$

On input $(\textsc{RecResult}, \mathsf{aid}, T'_C)$ :
$T_C \leftarrow \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_C, \mathsf{H}_3(\mathsf{pre} \parallel T_S))$
if $T'_C \neq T_C$ :
    return $(\textsc{RecResult}, \mathsf{aid}, \textsc{Fail})$ to $\mathsf{S}$
else :
$c \xleftarrow{\$} \mathsf{AE.Enc}(\mathsf{shk}; \mathsf{e})$
$\mathsf{sec\_set\_ctr}(\mathsf{aid}, 10)$
return $(\mathsf{aid}, c)$ to $\mathsf{S}$
```

Figure 9: The ideal functionality $\mathcal{F}_{\mathsf{HSM}}$.

- $\mathsf{Enc}'((K^{\mathsf{mask}}, K^{\mathsf{auth}}), m) \rightarrow (m \oplus K^{\mathsf{mask}}, \mathsf{MAC.Tag}(K^{\mathsf{auth}}, m \oplus K^{\mathsf{mask}}))$
- $\mathsf{Dec}'((K^{\mathsf{mask}}, K^{\mathsf{auth}}), (\mathsf{e\_cred}, T_e)) \rightarrow \begin{cases} \perp \text{ if } \mathsf{MAC.Vfy}(K^{\mathsf{auth}}, m) = 0 \\ \mathsf{e\_cred} \oplus K^{\mathsf{mask}} \text{ else} \end{cases}$

Theorem 1. *Let* $\mathsf{H}_1, \mathsf{H}_2, \mathsf{KDF}_1, \mathsf{KDF}_2$ *be random oracles such that 2HashDH UC-realizes the "multi-key" functionality* $\mathcal{F}_{\mathsf{OPRF}}$ *of Figure 10 and 3DH UC-realizes the authenticated key exchange functionality* $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ *of Figure 15. Let* $\Sigma = (\Sigma.\mathsf{KeyGen}, \Sigma.\mathsf{Sign}, \Sigma.\mathsf{Vfy})$ *be an* sEUF-CMA*-secure signature scheme. Let* $\mathsf{MAC} = (\mathsf{MAC.Tag}, \mathsf{MAC.Vfy})$ *be an* sEUF-CMA*-secure MAC. Let* $\mathsf{PKE} = (\mathsf{PKE.KeyGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *be an IND-CPA-secure public key encryption scheme. Let* $\mathsf{AE} = (\mathsf{AE.KeyGen}, \mathsf{AE.Enc}, \mathsf{AE.Dec})$ *be an authenticated encryption scheme. Let* $\mathsf{AE}' = (\mathsf{KeyGen}', \mathsf{Enc}', \mathsf{Dec}')$ *have random-key robustness. Let* $\mathsf{H}_3 : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ *be a collision resistant hash function.*

Then the WBP as described in Figure 4 and Figure 5 UC-realizes the PPKR functionality of Figure 7 and Figure 8 in the $\mathcal{F}_{\mathsf{HSM}}$*-hybrid model, assuming malicious adaptive corruption of clients as defined above, malicious adaptive corruption of the server, and a client-authenticated channel between clients and the server. Concretely, for any efficient adversary against WBP (interacting with* $\mathcal{F}_{\mathsf{HSM}}$*), there is an efficient simulator* Sim *that interacts with* $\mathcal{F}_{\mathsf{PPKR}}$ *and produces a view such that for every efficient environment* $\mathcal{Z}$*, it holds that*

$$\begin{aligned}
\mathbf{Dist}_{\mathcal{Z}}^{\mathsf{WBP},\{\mathcal{F}_{\mathsf{PPKR}},\textsc{Sim}\}}(\lambda) \leq\ & \mathbf{Dist}_{\mathcal{Z}^*}^{\mathsf{2HashDH},\{\mathcal{F}_{\mathsf{OPRF}},\textsc{Sim}_{\mathsf{OPRF}}\}}(\lambda) \\
&+ \mathbf{Dist}_{\mathcal{Z}^*}^{\mathsf{3DH},\{\mathcal{F}_{\mathsf{AKE\text{-}KCI}},\textsc{Sim}_{\mathsf{3DH}}\}}(\lambda) \\
&+ \mathbf{Adv}_{\mathcal{B},\Sigma}^{\mathsf{sEUF\text{-}CMA}}(\lambda) + 2Q_{\textsc{Rec}}\mathbf{Adv}_{\mathcal{B},\mathsf{MAC}}^{\mathsf{sEUF\text{-}CMA}}(\lambda) \\
&+ 2\mathbf{Adv}_{\mathcal{B},\mathsf{H}_3}^{\mathsf{CR}}(\lambda) + Q_{\textsc{Init}}\mathbf{Adv}_{\mathcal{B},\mathsf{PKE}}^{\mathsf{IND\text{-}CPA}}(\lambda) \\
&+ Q_{\textsc{Rec}}\mathbf{Adv}_{\mathcal{B},\mathsf{AE}}^{\mathsf{IND\text{-}CPA}}(\lambda) + Q_{\textsc{Rec}}\mathbf{Adv}_{\mathcal{B},\mathsf{AE}}^{\mathsf{INT\text{-}CTXT}}(\lambda) \\
&+ \binom{Q_{\textsc{Init}}}{2}2^{-\lambda} + \binom{Q_{\textsc{Rec}}}{2}2^{-\lambda+1} + \binom{Q_{\mathsf{KDF}_1}}{2}\mathbf{Adv}_{\mathcal{B},\mathsf{AE}'}^{\mathsf{RKR}}(\lambda),
\end{aligned}$$

where $Q_{\textsc{Init}} \in \mathbb{N}$ *is an upper bound on the number of initializations,* $Q_{\textsc{Rec}} \in \mathbb{N}$ *is an upper bound on the number of recoveries,* Q_{KDF_1} *is an upper bound on the number of* KDF_1 *queries, and* $\mathcal{B}$*, resp.* $\mathcal{Z}^*$*, is the adversary in the corresponding security experiments, which are detailed in Appendix E.*

Proof. A proof sketch outlining the proof steps can be found in Appendix E, where we also give the full proof and simulator. □

Discussion on the proof of the Theorem. We focus on analyzing an interaction between an honest client and a corrupt WhatsApp server, for which we use H-C (for *Honest-Corrupt*) as shorthand notation. Recall that the protocol is executed between clients who can authenticate themselves to the WhatsApp server, who in turn uses an HSM which is incorruptible and runs the code depicted in Figure 9. The interfaces of $\mathcal{F}_{\mathsf{HSM}}$ can only be accessed by the WhatsApp server, except for the GetPK which makes the HSM's public keys available to all protocol participants.

We argue that we only need to consider an interaction between an honest client and a corrupt server, by explaining why this captures already all the other cases. Note that there are four cases overall, since we have two types of entities (clients, and a server), both of which can be either corrupt or honest.

- Interactions between honest clients and an honest WhatsApp server: security against a semi-honest server is a sub-case of the H-C analysis, since the corrupted server in the H-C case can also honestly follow the protocol.

- Interactions between corrupt clients and a corrupt WhatsApp server: normally, it is not necessary to analyze a setting where all parties are corrupted. However, in this case there is still the incorruptible HSM, and our analysis needs to make sure that clients and the server cannot "team up" to, e.g., extract the HSM's signing key. However, since the corrupt WhatsApp server can already impersonate any honest client (since it is only the channel between the client and the WhatsApp server that is client-authenticated), corrupt clients cannot "add" to the power of a corrupt server, and hence this case is covered by the H-C analysis as well.
- Interactions between (a) statically or (b) adaptively corrupt clients and an honest WhatsApp server: (a) the honest WhatsApp has no secret inputs and hence simulating the honest server is trivial (or, put differently, there is nothing to protect the server from). (b) adaptive client corruptions reveal no values that were not already known to the environment, so there is nothing to simulate upon corruption. The ability to newly authenticate under a previously honest client identity is already captured by the H-C case, since a corrupt WhatsApp server can claim towards the HSM that any honest client sent a specific message.

We can hence focus on simulating an interaction between an honest client and a corrupt WhatsApp server using an (incorruptible) $\mathcal{F}_{\mathsf{HSM}}$.

6 Conclusion and Future Work

We have presented the first formal security analysis of the widely-used WhatsApp backup protocol and confirmed that the WBP indeed provides strong security guarantees such as online protection of the password, and the strength and secrecy of the backup key. However, we also show how a compromised WhatsApp server can increase the number of admissible password guesses from only $\mathtt{ctr}$ on the *most recent* password of the user, to $q_{\mathsf{pw}} * \mathtt{ctr}$ on *any* password pw ever entered by the user, where q_{pw} is the number of initializations performed with pw by the WhatsApp client device. Our analysis and formal modeling further spans a multitude of interesting research questions, that we divide into three categories.

Widening the Scope of the Analysis. In this work we have only focused on the cryptographic core of the WBP. Potential avenues for future research include a formal analysis of the (Merkle tree based) protocol which the HSM uses to securely outsource storage to untrusted servers. Another interesting direction would be to extend the corruption model to *proactive corruptions*, which allows to investigate whether WBP participants can recover faithfully from corruptions, e.g., after having granted compelled access at border control.

Direct Improvements of the WBP. The increased number of password guesses is only possible because the HSM cannot authenticate the client but has to trust the WhatsApp server that it indeed only acts on demand of honest clients. This can, e.g., be achieved if the client directly authenticates towards the HSM. That way, the server cannot impersonate clients towards the HSM anymore. However, this approach also requires WhatsApp to modify its authentication infrastructure, which is currently independent of the WBP. An "easy" modification of WBP is to let the HSM sign *all* messages. This provides direct protection of replay attacks by the adversary, against which the WBP protects implicitly using message authentication codes (MACs). Such "full" signing would greatly simplify the analysis of the protocol. This raises also the interesting question whether other potentially more costly protection mechanisms of the integrity of messages could be dropped if signing ensures HSM-authenticated channels, ending up with a more efficient protocol overall. Lastly, regarding efficiency improvements, one could look into whether a "less secure" (and potentially more efficient) version of OPAQUE could be plugged into the WBP, where protection against server compromise is dropped. Server compromise seems not a realistic attack scenario in this particular application of OPAQUE, because the OPAQUE server part is run on an HSM, which needs to be trusted anyway.

We note here that the above discussed efficiency improvements might not be of immediate interest to WhatsApp, where the WBP is only seldomly executed per user. Improvements regarding the security must be carefully analyzed and justified, since updating the HSM code of WBP requires WhatsApp to replace all HSMs with new ones and to perform the setup ceremony again.

Constructing PPKR Differently. The WBP is built around the OPAQUE protocol, which in turn deploys an oblivious pseudorandom function (OPRF) and an authenticated key exchange (AKE) protocol. The authenticated key exchange part serves as a tool for convincing the HSM of correctness of the client's password, which is necessary to ensure that the guess attempt counter can be reset to 10 guesses. It is an interesting open question whether this could be achieved from simpler primitives, such as symmetric primitives, which are more efficient than the public-key-based AKE.

7 Acknowledgments

We wish to extend our gratitude to Kevin Lewi and other members of the Meta security team for their patience in answering our questions about the WBP. We would like to thank Kristina Hostáková for helpful discussions of the UC modeling aspects of HSMs.

This paper would not have been possible without the support of Anna, Camilla, Judit, and Nikolas.

References

[1] Direct correspondences with Kevin Lewi and other members of the WhatsApp engineering team, 2022-2023

[2] Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Heidelberg, 2019

[3] Bagherzandi, A., Jarecki, S., Saxena, N., Lu, Y.: Password-protected secret sharing. In: ACM CCS 2011. pp. 433–444. ACM Press, 2011

[4] Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 619–650. Springer, Heidelberg, 2017

[5] Bienstock, A., Fairoze, J., Garg, S., Mukherjee, P., Raghuraman, S.: A more complete analysis of the Signal double ratchet algorithm. In: CRYPTO 2022, Part I. LNCS, vol. 13507, pp. 784–813. Springer, Heidelberg, 2022

[6] Bourdrez, D., Krawczyk, D.H., Lewi, K., Wood, C.A.: The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-irtf-cfrg-opaque-09, Internet Engineering Task Force, 2022, https://datatracker.ietf.org/doc/draft-irtf-cfrg-opaque/09/, work in Progress

[7] Brost, J., Egger, C., Lai, R.W.F., Schmid, F., Schröder, D., Zoppelt, M.: Threshold password-hardened encryption services. In: ACM CCS 2020. pp. 409–424. ACM Press, 2020

[8] Camenisch, J., Lysyanskaya, A., Neven, G.: Practical yet universally composable two-server password-authenticated secret sharing. In: ACM CCS 2012. pp. 525–536. ACM Press, 2012

[9] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press, 2001

[10] Canetti, R., Jain, P., Swanberg, M., Varia, M.: Universally composable end-to-end secure messaging. In: CRYPTO 2022, Part II. LNCS, vol. 13508, pp. 3–33. Springer, Heidelberg, 2022

[11] Casacuberta, S., Hesse, J., Lehmann, A.: SoK: Oblivious pseudorandom functions. In: IEEE EuroS&P 2022. IEEE, 2022

[12] Cathcart, W.: https://twitter.com/wcathcart/status/1600603826477617152, 2022

[13] Chase, M., Perrin, T., Zaverucha, G.: The signal private group system and anonymous credentials supporting efficient verifiable encryption. In: ACM CCS 2020. pp. 1445–1459. ACM Press, 2020

[14] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: EuroS&P. pp. 451–466. IEEE, 2017

[15] Das, P., Hesse, J., Lehmann, A.: DPaSE: Distributed password-authenticated symmetric-key encryption, or how to get many keys from one password. In: ASIACCS 22. pp. 682–696. ACM Press, 2022

[16] Davidson, A., Faz-Hernandez, A., Sullivan, N., Wood, C.A.: Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups. Internet-Draft draft-irtf-cfrg-voprf-17, Internet Engineering Task Force, 2023, https://datatracker.ietf.org/doc/draft-irtf-cfrg-voprf/17/, work in Progress

[17] Doussot, G., Lacharité, M.S., Schorn, E.: End-to-End Encrypted Backups Security Assessment. https://research.nccgroup.com/wp-content/uploads/2021/10/NCC_Group_WhatsApp_E001000M_Report_2021-10-27_v1.2.pdf, 2021

[18] Fischlin, M., Günther, F.: Replay attacks on zero round-trip time: The case of the tls 1.3 handshake candidates. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 60–75. 2017

[19] Gentry, C., MacKenzie, P., Ramzan, Z.: A method for making password-based key exchange resilient to server compromise. In: CRYPTO 2006. LNCS, vol. 4117, pp. 142–159. Springer, Heidelberg, 2006

[20] Gu, Y., Jarecki, S., Krawczyk, H.: KHAPE: Asymmetric PAKE from key-hiding key exchange. In: CRYPTO 2021, Part IV. LNCS, vol. 12828, pp. 701–730. Springer, Heidelberg, Virtual Event, 2021

[21] Jarecki, S., Kiayias, A., Krawczyk, H.: Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In: ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 233–253. Springer, Heidelberg, 2014

[22] Jarecki, S., Kiayias, A., Krawczyk, H., Xu, J.: 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. pp. 276–291. IEEE, 2016

[23] Jarecki, S., Krawczyk, H., Resch, J.K.: Updatable oblivious key management for storage systems. In: ACM CCS 2019. pp. 379–393. ACM Press, 2019

[24] Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In: EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 456–486. Springer, Heidelberg, 2018

[25] Jost, D., Maurer, U., Mularczyk, M.: A unified and composable take on ratcheting. In: TCC 2019, Part II. LNCS, vol. 11892, pp. 180–210. Springer, Heidelberg, 2019

[26] Krawczyk, D.H., Lewi, K., Wood, C.A.: The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-irtf-cfrg-opaque-03, Internet Engineering Task Force, 2021, https://datatracker.ietf.org/doc/draft-irtf-cfrg-opaque/03/, work in Progress

[27] Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg, 2010

[28] Lai, R.W.F., Egger, C., Reinert, M., Chow, S.S.M., Maffei, M., Schröder, D.: Simple password-hardened encryption services. In: USENIX Security 2018. pp. 1405–1421. USENIX Association, 2018

[29] Novak, M.: Paul Manafort Learns That Encrypting Messages Doesn't Matter If the Feds Have a Warrant to Search Your iCloud Account. https://gizmodo.com/paul-manafort-learns-that-encrypting-messages-doesnt-ma-1826561511, 2018

[30] Perrin, T.: The noise protocol framework, http://noiseprotocol.org/noise.html

[31] Rösler, P., Mainka, C., Schwenk, J.: More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In: EuroS&P. pp. 415–429. IEEE, 2018

[32] Vatandas, N., Gennaro, R., Ithurburn, B., Krawczyk, H.: On the cryptographic deniability of the Signal protocol. In: ACNS 20, Part II. LNCS, vol. 12147, pp. 188–209. Springer, Heidelberg, 2020

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

[34] WhatsApp: WhatsApp Encryption Overview. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, 2021

Appendix

A Cryptographic Building Blocks and Their Security

Definition 1 (Collision Resistance of Hash Functions)**.** The advantage of an adversary $\mathcal{A}$ against the *collision resistance* (CR) of a hash function $H : \{0,1\}^* \to \{0,1\}^\lambda$ is defined as

$$\mathbf{Adv}^{\mathsf{CR}}_{\mathcal{A},H}(\lambda) := \Pr[H(m) = H(m') \mid (m, m') \xleftarrow{\$} \mathcal{A}(1^\lambda)],$$

where $m \neq m'$.

Definition 2 (Digital Signatures)**.** A *digital signature scheme* is a tuple $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vfy})$ consisting of the following three probabilistic polynomial-time algorithms.

- KeyGen takes as input the security parameter 1^λ and outputs a key pair $(\mathsf{pk}, \mathsf{sk})$.
- Sign takes as input a secret key sk and a message m, and outputs a signature σ.
- Vfy takes as input a public key pk, a message m, and a signature σ. It outputs 1 ("accept") or 0 ("reject").

We call a signature scheme correct if $\mathsf{Vfy}(\mathsf{pk}, m, \mathsf{Sign}(sk, m)) = 1$ holds for all $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$ and all messages m.

Definition 3 (sEUF-CMA Security for Digital Signatures)**.** The advantage of an adversary $\mathcal{A}$ against the *strong existential unforgeability under chosen message attacks* (sEUF-CMA) of a signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vfy})$ is defined as

$$\mathbf{Adv}^{\mathsf{sEUF\text{-}CMA}}_{\mathcal{A},\Sigma}(\lambda) := \Pr[\mathsf{Vfy}(\mathsf{pk}, m^*, \sigma^*) = 1 \mid (m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathsf{Sign}(\mathsf{sk},\cdot)}(\mathsf{pk})],$$

where $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$, and (m^*, σ^*) is fresh in the sense that m^* was never queried to $\mathsf{Sign}(\mathsf{sk}, \cdot)$ resulting in σ^* as output.

Definition 4 (Message Authentication Codes)**.** A *message authentication code* (MAC) is a tuple $\mathsf{MAC} = (\mathsf{KeyGen}, \mathsf{Tag}, \mathsf{Vfy})$ consisting of the following three probabilistic polynomial-time algorithms.

- KeyGen takes as input the security parameter 1^λ and outputs a symmetric key K.
- Tag takes as input a key K and a message m, and outputs a tag T.
- Vfy takes as input a key K, a message m, and a tag T. It outputs 1 ("accept") or 0 ("reject").

We call a MAC correct if $\mathsf{Vfy}(K, m, \mathsf{Tag}(K, m)) = 1$ holds for all $K \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$ and all messages m.

Definition 5 (sEUF-CMA Security for MACs)**.** The advantage of an adversary $\mathcal{A}$ against the *strong existential unforgeability under chosen message attacks* (sEUF-CMA) of a MAC $\mathsf{MAC} = (\mathsf{KeyGen}, \mathsf{Tag}, \mathsf{Vfy})$ is defined as

$$\mathbf{Adv}^{\mathsf{sEUF\text{-}CMA}}_{\mathcal{A},\mathsf{MAC}}(\lambda) := \Pr[\mathsf{Vfy}(K, m^*, T^*) = 1 \mid (m^*, T^*) \xleftarrow{\$} \mathcal{A}^{\mathsf{Tag}(K,\cdot),\mathsf{Vfy}(K,\cdot,\cdot)}(1^\lambda)],$$

where $K \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$, and (m^*, T^*) is fresh in the sense that m^* was never queried to $\mathsf{MAC}(K, \cdot)$ resulting in σ^* as output.

Definition 6 (Public Key Encryption)**.** A *public key encryption* (PKE) scheme is a tuple $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ consisting of the following three probabilistic polynomial-time algorithms.

- KeyGen takes as input the security parameter 1^λ and outputs a key pair $(\mathsf{pk}, \mathsf{sk})$.
- Enc takes as input a public key pk and a message m, and outputs a ciphertext c.
- Dec takes as input a secret key sk and a ciphertext c, and outputs a message m.

We call a PKE scheme correct if $\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, m)) = m$ holds for all $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$ and all messages m.

Definition 7 (IND-CPA Security for PKE)**.** The advantage of an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ against the *indistinguishability of ciphertexts under chosen plaintext attacks* (IND-CPA) of a PKE scheme $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is defined as

$$\mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{A},\mathsf{PKE}}(\lambda) := \left| \Pr\left[b = b^* \;\middle|\; \begin{array}{c} (m_0, m_1, st) \xleftarrow{\$} \mathcal{A}_0(\mathsf{pk}), \\ b \xleftarrow{\$} \{0,1\}, \\ c^* = \mathsf{Enc}(\mathsf{pk}, m_b), \\ b^* \xleftarrow{\$} \mathcal{A}_1(st, c^*) \end{array} \right] - \frac{1}{2} \right|,$$

where $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$.

Definition 8 (Authenticated Encryption)**.** An *authenticated encryption* (AE) scheme is a tuple $\mathsf{AE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ consisting of the following three probabilistic polynomial-time algorithms.

- KeyGen takes as input the security parameter 1^λ and outputs a key K.
- Enc takes as input a key K and a message m, and outputs a ciphertext c.
- Dec takes as input a key K and a ciphertext c, and outputs a message m or an error symbol $\perp$.

We call an AE scheme correct if $\mathsf{Dec}(K, \mathsf{Enc}(K, m)) = m$ holds for all $K \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$ and all messages m.

Definition 9 (IND-CPA Security for AE)**.** The advantage of an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ against the *indistinguishability of ciphertexts under chosen plaintext attacks* (IND-CPA) of an AE scheme $\mathsf{AE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is defined as

$$\mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{A},\mathsf{AE}}(\lambda) := \left| \Pr\left[b = b^* \;\middle|\; \begin{array}{c} (m_0, m_1, st) \xleftarrow{\$} \mathcal{A}_0^{\mathsf{Enc}(K,\cdot)}(1^\lambda), \\ b \xleftarrow{\$} \{0,1\}, \\ c^* = \mathsf{Enc}(\mathsf{pk}, m_b), \\ b^* \xleftarrow{\$} \mathcal{A}_1(st, c^*) \end{array} \right] - \frac{1}{2} \right|,$$

where $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$.

Definition 10 (INT-CTXT Security for AE)**.** The advantage of an adversary $\mathcal{A}$ against the *integrity of ciphertexts* (INT-CTXT) of an AE scheme $\mathsf{AE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is defined as

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

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

Definition 11 (Random Key Robustness)**.** The advantage of an adversary $\mathcal{A}$ against the *random-key robustness* (RKR) of an AE scheme $\mathsf{AE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ with key space $\mathcal{K}$ is defined as

$$\mathbf{Adv}^{\mathsf{RKR}}_{\mathcal{A},\mathsf{AE}}(\lambda) := \Pr[\mathsf{Dec}(k_1, c) \neq \perp, \mathsf{Dec}(k_2, c) \neq \perp \mid k_1, k_2 \xleftarrow{\$} \mathcal{K}, c \leftarrow \mathcal{A}(k_1, k_2)].$$

This property can be achieved, e.g., by using encrypt-then-MAC with a MAC that is collision resistant with respect to the message and key [24]. In particular AE' as defined in Section 5 follows this construction principle as it can be seen as a one-time-pad combined with an HMAC.

B Modeling WBP in the Universal Composability Framework

The UC framework uses a simulation-based approach to state security guarantees of protocols π. That is, proofs usually consist of a description of an efficient *simulator* that has access to an ideal functionality $\mathcal{F}$ and "mimics" the behavior of the "real" protocol π to some distinguishing entity called the *environment*. If the environment cannot efficiently distinguish a simulator "mimicking" the protocol behavior from the behavior of the "real" protocol π, then the protocol is as secure as the ideal functionality. The functionality $\mathcal{F}$ essentially defines the intended behavior of the protocol. Note that this means that the security of a protocol π can only be related to the "security" provided by the ideal functionality $\mathcal{F}$. That is, the protocol can only be proven as secure as the ideal functionality, yielding a lower bound on the security guarantees of π. In UC terminology we say that π *UC-realizes* $\mathcal{F}$.

Let us illustrate this via a toy example. A toy functionality $\mathcal{F}$ for the WBP could work as follows: It takes two inputs, a password pw from the client and a secret value s from the server, then it internally executes the protocol $(k, E) \leftarrow \mathsf{WBP}(pw, s)$, and returns the symmetric backup key k to the client and an encrypted version of the key E to the server. This ideal functionality essentially behaves like a trusted third party, which executes the protocol π on behalf of the client and server, where all inputs and outputs are distributed via secure channels. If we proved that the WBP "realizes" this toy functionality, the we would prove that the WBP is as secure as if it would be when executed via a trusted third party. However, this is not possible for the WBP since an adversary can, for example, re-arrange messages or try to guess passwords. If this "adversarial influence" is not reflected in the description of the functionality $\mathcal{F}$, it could be used by to trivially distinguish the "real" protocol from the simulation: one would allow this influence, while the other would not.

This means that our toy functionality is *too strong* and that we need to *weaken* it by introducing additional interfaces that "leak" the necessary information. A ridiculous option would be that the functionality leaks all internal secret values to the simulator. Note that this intuitively corresponds to the behavior of a trusted third party that leaks all of its secrets. While this makes it easy to simulate the behavior of the "real" protocol, the security guarantees are not meaningful at all. This variant of our toy functionality is *too weak*.

The challenge of designing a meaningful ideal functionality is to find the sweet spot where the functionality provides the least amount of informational leakage, but such that the simulator can efficiently "mimic" the behavior of the real protocol, computationally indistinguishable for any efficient distinguishing environment. Note that if a protocol π UC-realizes $\mathcal{F}$, then there cannot exist substantially more leakage than what is formalized in the functionality. For the WBP this would mean that there are no further attacks possible but the ones that require the adversarial influence prescribed by the functionality.

C Preliminaries on Oblivious Pseudorandom Functions

In this Section we give the preliminaries on oblivious pseudorandom functions (OPRFs) in the UC model. We first state a "multi-key" version of the OPRF functionality from [24] that allows evaluation of PRFs with respect to many different keys, and where we drop the ability of the functionality to export a transcript prefix to the application. The reasons for these changes are as follows:

- **Multi-key setting:** WhatsApp's PPKR scheme in Figures 4 and 5 uses an OPRF called "2HashDH" [22] (see Figure 11 for the protocol description) where hash functions H_1 and KDF_2 do not have domains that are separated for different PRF keys. More concretely, if two users initialize or recover using the same password pw, both compute the same value $H_1(\mathsf{pw})$ as a first step of the 2HashDH protocol. Hence, the security analysis of the OPRF part of WhatsApp's PPKR scheme cannot rely on any domain separation occurring.[17]
- **Dropping prefixes:** As we do not use $\mathcal{F}_{\mathsf{OPRF}}$ to formulate WhatsApp's PPKR scheme (i.e., we do not formulate it in the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid model) but only rely on the existence of a simulator for 2HashDH in the proof, we do not require the export of parts of the transcript in order to, e.g., sign or compare them. The OPRF functionality used in the analysis of OPAQUE [24] had to introduce such exportation in order to be able to state the OPAQUE protocol in the OPRF-hybrid setting.

We note that [24] proves the security of 2HashDH without reliance on authenticated channels between the user and the server (previous works [22] still relied on such channels). This fits our setting, where the OPRF is run between the user and the HSM holding all PRF keys. Neither is the user authenticated to the HSM (it is only authenticated toward the server, but a malicious server can lie to the HSM about this authentication), nor can the user determine which key was used by the HSM (the malicious server can let the HSM use any of its PRF keys).

While WhatsApp's implementation of 2HashDH is in a strong setting where servers/PRF key holders are all played by the HSM and hence can be assumed incorruptible and uncompromisable (i.e., they will always follow the protocol, and they will never leak their PRF keys), we opt for a general treatment of OPRFs including server corruption and compromise in this section. While we do not require the analysis of these settings within this work, we believe that a general treatment and analysis of 2HashDH *without* domain separation has great relevance for other works [24, 6, 16].

Having summarized where we reuse results from, how we change them, and why, we now describe the technical contents of this section. In Figure 10 we state a multi-key OPRF functionality adopted from [24]. In Figure 11 we give the multi-key version of the 2HashDH OPRF of [24]. In Figure 12 we give the simulator that demonstrates that 2HashDH UC-realizes $\mathcal{F}_{\mathsf{OPRF}}$.

C.1 Multi-Key OPRF Model

Our functionality $\mathcal{F}_{\mathsf{OPRF}}$ closely follows the design from [24], but we extend the functionality to be able to handle multiple servers and even multiple PRF keys per server. $\mathcal{F}_{\mathsf{OPRF}}$ is depicted in Figure 10 and we

[17] We note that the security analysis of OPAQUE [24] is carried out with respect to a single-key OPRF functionality and hence proves the security of OPAQUE only when hash domains of the two hash functions of 2HashDH are separated, e.g., by hashing unique session/key identifiers alongside the other inputs. However, OPAQUE in practice (e.g., [6]) does not deploy such domain separation, since the negotiation and memorization of session identifiers is expensive and partly contradicts with the purpose of the scheme being password-only. Hence, for a meaningful analysis of these deployed versions of OPAQUE, our multi-key version of $\mathcal{F}_{\mathsf{OPRF}}$ should be used.

mark in gray the changes over [24] that enable our $\mathcal{F}_{\mathsf{OPRF}}$ to handle multiple PRF keys and servers. We now explain the functionality's interfaces and parameters in detail.

The functionality $\mathcal{F}_{\mathsf{OPRF}}$ implements oblivious access to a truly random function family $F_{\mathsf{sid},\mathsf{S},\mathsf{kid}}(\cdot) : \{0,1\}^* \to \{0,1\}^\ell$. The functions are parameterized by a global session identifier sid and parameters S, kid that can take arbitrary values and can be interpreted as taking the role of the PRF key. For each pair S, kid, a truly random function table is maintained.

The INIT *interface.* Any server S can call this interface to initialize a new PRF key with identifier kid. We let $\mathcal{F}_{\mathsf{OPRF}}$ ignore subsequent inputs of same key identifiers per server, which models that we expect key identifiers to be unique per server (e.g., they correspond to account names of users where each user evaluates their "own" PRF, or they describe the purpose of the PRF that is evaluated with this key). Key identifiers are not kept secret (i.e., they are leaked to the adversary). For each newly initialized PRF key by server S with identifier kid, $\mathcal{F}_{\mathsf{OPRF}}$ stores a record $\langle \mathsf{S}, \mathsf{kid} \rangle$ and sets a counter $\mathsf{tx}[\mathsf{S}, \mathsf{kid}]$ to 0.

The COMPROMISE *interface.* The effect of the compromise interface is that a record $\langle \mathsf{S}, \mathsf{kid} \rangle$ is marked COMPROMISED. This corresponds to compromise of a PRF key, e.g., due to a breach at the server. We model key-wise compromise by letting the adversary specify which kid it wants to compromise at what server S, as it is possible that, e.g., a server only leaks keys that he recently has touched, while others remain securely stored. If a server gets corrupted (i.e., fully controlled by the adversary), all its keys are considered COMPROMISED. Looking ahead, a key being marked compromised allows the adversary to evaluate the corresponding PRF an unbounded number of times (see the OFFLINEEVAL interface explanation below).

The OFFLINEEVAL *interface.* This interface can be used by the adversary to evaluate any of the random functions $F_{\mathsf{sid},\mathsf{S},\mathsf{kid}}(\cdot)$ (identified by the "PRF key" S, kid) on any input x. However, $\mathcal{F}_{\mathsf{OPRF}}$ will only return the corresponding PRF value if the corresponding key is considered to be in the hands of the adversary. This is the case if S is corrupt or S, kid was already compromised, or if S, kid was never honestly initialized. If any of these checks pass, $\mathcal{F}_{\mathsf{OPRF}}$ returns $F_{\mathsf{sid},\mathsf{S},\mathsf{kid}}(x)$ to the adversary.

The EVAL *interface.* This interface is called by any user P who wants to evaluate a specific PRF identified by S, kid on a secret input x. In order to allow for parallel evaluation sessions, the interface takes a subsession identifier ssid. $\mathcal{F}_{\mathsf{OPRF}}$ stores the request and informs the adversary, keeping input x private.

The SNDRCOMPLETE *interface.* This interface allows a server to signal that it wants to assist in a specific evaluation identified by ssid, using the PRF key of that server identified by kid'. Note that $\mathcal{F}_{\mathsf{OPRF}}$ does not enforce the intended key identifier (specified in EVAL input by the user) and the used key identifier (specified in SNDRCOMPLETE input by the server) to be the same. This allows for the analysis of OPRF protocols that do not assume users to be authenticated, and hence messages by users can be modified. In particular the method of users telling the server which key to use might not be tamper-proof. To proceed with the interface explanation, $\mathcal{F}_{\mathsf{OPRF}}$ allows server participation only if the corresponding key exists at a server (or the server is using a malicious key). $\mathcal{F}_{\mathsf{OPRF}}$ then increases the "evaluation ticket" counter $\mathsf{tx}[\mathsf{S}, \mathsf{kid}']$ by 1. Looking ahead, this counter will reflect the number of PRF evaluations that a server agreed to assist with, on a per-key basis.

The RCVCOMPLETE *interface.* Finally, the RCVCOMPLETE interface can be called by the adversary at any time to let a user P finalize an open evaluation session identified by ssid. $\mathcal{F}_{\mathsf{OPRF}}$ only continues the request if P is expecting to receive an output for ssid. The adversary has the freedom to specify with respect to which key $\mathsf{S}^*, \mathsf{kid}^*$ the user receives the evaluation for, with only one constraint: there needs to be an evaluation ticket $\mathsf{tx}[\mathsf{S}^*, \mathsf{kid}^*]$. This ensures that evaluation of the PRF with respect to honest keys held by servers cannot happen more times than the corresponding server has agreed to assist in the evaluation. If

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 10: A multi-key version of the ideal functionality $\mathcal{F}_{\mathsf{OPRF}}$ from [24] (without prefixes). The ability to maintain multiple PRF keys is reflected in the addition of "key identifiers" kid, and we highlight the changes using gray boxes.

an evaluation ticket for the specified key is found, it is taken away by decreasing the counter, and the PRF output is sent to P. We note that the flexibility of the adversary in $\mathcal{F}_{\mathsf{OPRF}}$ that lets it decide at the very latest point how to spend evaluation tickets is what has allowed to prove universal composability for efficient protocols such as 2HashDH [22] in the past, as it allows for "late extraction" of adversarial PRF keys.

C.2 Security of Multi-Key 2HashDH

The multi-key version of 2HashDH can be found in Figure 11. The changes over single-key 2HashDH are quite minimal: essentially all interfaces receive a key identifier kid as additional input. The user who wants to evaluate a PRF with key identifier kid informs the server about it by sending kid alongside the first message. The server who receives the first message takes the kid and looks up the PRF key according to this identifier. There is no authenticity of key identifiers if channels are not user-authenticated, as in the setting of the WhatsApp backup protocol.

We stress that the addition of key identifiers results in a subtle but crucial difference: in multi-key 2HashDH (Figure 11), hash function domains are not separated for each key identifier kid. That is, Alice's computation to evaluate her PRF identified by $\mathsf{kid}_{\mathsf{Alice}}$ at input x involves computation of $H_1(x)$, which is the same value that Bob computes if he wants to evaluate his PRF identified by $\mathsf{kid}_{\mathsf{Bob}}$ at the same input x. Without user-authenticated channels, the adversary can hence maliciously "reroute" PRF evaluation to different keys. More specifically, Alice computes $H_1(x)^r$ and sends $(\mathsf{kid}_{\mathsf{Alice}}, H_1(x)^r)$ to her server. The adversary rewrites this message to $(\mathsf{kid}_{\mathsf{Bob}}, H_1(x)^r)$ such that the server applies Bob's key and sends back to Alice the value $(H_1(x)^r)^{k_{\mathsf{Bob}}}$. Alice removes the blinding factor r and outputs $H_2(x, H_1(x)^{k_{\mathsf{Bob}}})$, which is an evaluation of x of Bob's PRF. Note that Alice cannot notice that she computed somebody else's PRF.[18]

We prove the security of the multi-key 2HashDH OPRF in Figure 11 under the following assumption.

Definition 12 ((N, Q) one-more DH assumption [22])**.** We say that 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 $\mathcal{A}$,

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

is negligible, where $S = \{(g_{j_s}, g_{j_s}^k) \mid s = 1, \ldots, Q+1\}$ 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 2. *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 of Figure 11 UC-realizes the "multi-key" functionality $\mathcal{F}_{\mathsf{OPRF}}$ of Figure 10, 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

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

[18]We note that the analysis of OPAQUE [24], which is carried out using multiple instances of single-key 2HashDH, does not examine the effect of this attack since the parallel execution of many single-key 2HashDH instances introduces domain separation into all random oracles.

2HashDH(H_1, H_2)

Components:
Hash functions $H_1 : \{0,1\}^* \to \mathbb{G}, H_2 : \{0,1\}^* \times \mathbb{G} \to \{0,1\}^\ell$ with $\ell \in \mathbb{N}$ and $\mathbb{G} = \langle g \rangle$ a group of order q. The two hash functions are specified for one particular sid, which has to be folded into the input.

Initialization:
On $(\text{INIT}, \mathsf{sid}, \boxed{\mathsf{kid}})$, the server S picks $k \xleftarrow{\$} \mathbb{Z}_q$ and stores $\langle \mathsf{sid}, \boxed{\mathsf{kid}}, k\rangle$.

Server Compromise:
On $(\text{COMPROMISE}, \mathsf{sid}, \boxed{\mathsf{kid}})$, if there is a record $\langle \mathsf{sid}, \boxed{\mathsf{kid}}, k\rangle$, the server S reveals k to the adversary.

Offline Evaluation:
On $(\text{OFFLINEEVAL}, \mathsf{sid}, \boxed{\mathsf{kid}}, x)$, the server retrieves record $\langle \mathsf{sid}, \boxed{\mathsf{kid}}, k\rangle$ and outputs $(\text{OFFLINEEVAL}, \mathsf{sid}, \boxed{\mathsf{kid}}, H_2(x, H_1(x)^k))$.

Online Evaluation:

- On $(\text{EVAL}, \mathsf{sid}, \boxed{\mathsf{kid}}, \mathsf{ssid}, \mathsf{S}', x)$ the user U picks $r \xleftarrow{\$} \mathbb{Z}_q$, records $\langle \mathsf{sid}, \mathsf{ssid}, r\rangle$ sends $(\mathsf{sid}, \boxed{\mathsf{kid}}, \mathsf{ssid}, H_1(x)^r)$ to the server S′.
- On $(\text{SNDRCOMPLETE}, \mathsf{sid}, \boxed{\mathsf{kid}}, \mathsf{ssid})$ and message $(\mathsf{sid}, \boxed{\mathsf{kid}}, \mathsf{ssid}, a)$ from U, s.t. $a \in \mathbb{G}$, the server S retrieves record $\langle \mathsf{sid}, \boxed{\mathsf{kid}}, k\rangle$ and sends $(\mathsf{sid}, \mathsf{ssid}, a^k)$ to U.
- On $(\mathsf{sid}, \mathsf{ssid}, b)$ s.t. $b \in \mathbb{G}$, the user U retrieves record $\langle \mathsf{sid}, \mathsf{ssid}, r\rangle$, aborts if the tuple is not found and else outputs $(\text{EVAL}, \mathsf{sid}, \mathsf{ssid}, H_2(x, b^{1/r}))$.

Figure 11: The multi-key version of protocol 2HashDH that realizes $\mathcal{F}_{\mathsf{OPRF}}$. The changes introduced over [24] due to the handling of multiple PRF keys are marked with gray boxes, and the exportation of prefixes is dropped.

where q_I is the number of honestly initialized keys in the system, and $\mathbf{Adv}_{\mathcal{A},\mathbb{G}}^{(x,y)\text{-OMDH}}$ denotes the probability of violating the (x, y) one-more DH assumption.

Proof intuition. There is already a proof of universal composability of single-key 2HashDH in [24], by giving an algorithm that simulates the protocol run if there is only one honest PRF key. If there are several honest PRF keys, the question to resolve in our proof is whether the individual simulators from [24] can be orchestrated to run in parallel, without any clashes in programming the random oracles. The idea to avoid clashes is the following: first, due to the high entropy in PRF keys, a clash in H_2 programming is unlikely to occur, since k is part of the H_2 inputs. Inputs to H_1 are the values at which a PRF should be evaluated, and hence they can coincide (e.g., different users have the same passwords that they need to feed into their PRF). However, the single-key simulator of [24] does not rely on programming H_1 outputs to values specific to a certain PRF key, but rather relies on *knowledge of a trapdoor* of the hash output. Our multi-key simulator can thus apply the following strategy: it first chooses trapdoors itself and plants them into H_1 outputs, and then it runs the individual simulators on these joint trapdoors. The precise code of our simulator is given in Figure 12 and the detailed proof follows below.

PROOF. We argue that $\text{SIM}_{\mathsf{OPRF}}$ of Figure 12 generates a view to an arbitrary environment $\mathcal{Z}$ that is indistinguishable from $\mathcal{Z}$'s interaction with the real world where parties run protocol 2HashDH of Figure 11. Without loss of generality, suppose $\mathcal{A}$ is the dummy adversary [9] who merely passes through all its messages to and from $\mathcal{Z}$. The interfaces and view of $\mathcal{Z}$ are as follows:

- Client: $\mathcal{Z}$ sends $(\text{EVAL}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{S}, x)$ to a client and eventually receives back a PRF value $(\text{EVALOUT}, \mathsf{sid}, \mathsf{ssid}, y)$ under the key identified by kid, held by S.
- Server: $\mathcal{Z}$ sends $(\text{SNDRCOMPLETE}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid})$ to a server in order to let that server finish session ssid using key identifier kid. $\mathcal{Z}$ does not expect to see any output from the server upon sending this input.
- Adversary: $\mathcal{Z}$ expects to receive both protocol messages from $\mathcal{A}$ in case client and server are honest. $\mathcal{Z}$ sends $(\mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, a)$ to $\mathcal{A}$ as any client's message, and $(\mathsf{sid}, \mathsf{ssid}, b)$ as any server's message. Observe that, since 2HashDH is run over unauthenticated channels, such messages can be introduced by $\mathcal{Z}$ without corrupting anybody.
- Random oracles: $\mathcal{Z}$ can query both $H_1(x)$ and $H_2(y, z)$ to $\mathcal{A}$ for any values x, y, z.

We now describe the above view of $\mathcal{Z}$ in more detail and for both worlds. For any chosen kid, x, $\mathcal{Z}$ receives transcript values a, b from $\mathcal{A}$ for corresponding honest clients and servers, and a PRF value $(\text{EVALOUT}, \mathsf{sid}, \mathsf{ssid}, y)$ in case the client is honest. In the real execution, we have $a = H(x)^r$ for a randomly chosen $r \in \mathbb{Z}_q$, $b = a^k$ for a randomly chosen $k \in \mathbb{Z}_q$, $y = H_2(x, H_1(x)^k)$, and queries to H_1 and H_2 answered consistently and uniformly at random. In the ideal execution, we have $a = g_J$ a randomly chosen group element, $b = g_J^{k'}$ for a randomly chosen $k' \xleftarrow{\$} \mathbb{Z}_q$, $y \xleftarrow{\$} \{0,1\}^\ell$ as chosen by $\mathcal{F}_{\mathsf{OPRF}}$, and H_1 and H_2 values answered with uniform values from the appropriate ranges. We now argue indistinguishability of both worlds in detail.

- Message a ($H(x)^r$ vs. g_J): since the uniformly chosen $r \in \mathbb{Z}_q$ is never revealed by an honest client, $H(x)^r$ is uniformly random to $\mathcal{Z}$ and hence indistinguishable from g_J.
- Message b (a^k vs. $a^{k'}$): k is chosen at random by the honest server and not revealed to $\mathcal{Z}$, while k' is chosen at random by $\text{SIM}_{\mathsf{OPRF}}$, and not revealed to $\mathcal{Z}$. Hence both a^k and $a^{k'}$ are indistinguishable for $\mathcal{Z}$.

- Output $(\textsc{EvalOut}, \mathsf{sid}, \mathsf{ssid}, y)$ $(H_2(x, H_1(x)^k)$ vs. $F_{\mathsf{sid},i}(x))$: the only way to distinguish the real world y from the ideal world is to query $(x, H_1(x)^k)$ to the H_2 oracle. However, $\textsc{Sim}_{\mathsf{OPRF}}$ is able to detect this: if k is the key used by an honest server, then there is a record $\langle F, *, \mathsf{kid}, k, u^{1/r}\rangle$ for the corresponding r from the H_1 record of x (cf. first bullet of step 8 in Figure 12). Hence in this case $\textsc{Sim}_{\mathsf{OPRF}}$ learns the key identifier kid which this H_2 query of $\mathcal{Z}$ is consistent with. If on the other hand k is a key already used by a corrupt server/the network adversary, then $\textsc{Sim}_{\mathsf{OPRF}}$ has a record $\langle M, \mathcal{A}, i, \perp, u^{1/r}\rangle$ for the corresponding r from the H_1 record of x (cf. second bullet of step 8 in Figure 12). Hence also in this case $\textsc{Sim}_{\mathsf{OPRF}}$ learns the key identifier i. If any key identifier is found, $\textsc{Sim}_{\mathsf{OPRF}}$ obtains the correct PRF value $F_{\mathsf{sid},\mathsf{S},\mathsf{kid}}(x)$ (or $F_{\mathsf{sid},\mathcal{A},i}(x)$) from $\mathcal{F}_{\mathsf{OPRF}}$ via the $\textsc{Eval}$ (or $\textsc{OfflineEval}$) interface (depending on whether the server holding the key identifier is compromised/corrupt or honest), and sets it to be equal to $H_2(x, u)$. Hence, if $\mathcal{F}_{\mathsf{OPRF}}$ replies with a value, the outputs are equal. If not, $\textsc{Sim}_{\mathsf{OPRF}}$ aborts and we analyze the probability for that happening below. Note that this also ensures that $H_2(x, u)$ equals an $\textsc{OfflineEval}$ query of an honest server for one of its own kid.
- Random oracle H_1: since g_J in step 4 of $\textsc{Sim}_{\mathsf{OPRF}}$ is chosen at random, the simulated responses are indistinguishable from the ones chosen by the random oracle in the real protocol.
- Random oracle H_2: $\textsc{Sim}_{\mathsf{OPRF}}$ programs H_2 to either a uniform value (cf. third bullet of step 8 in Figure 12) or to an output of $\mathcal{F}_{\mathsf{OPRF}}$, which is itself chosen by $\mathcal{F}_{\mathsf{OPRF}}$ uniformly at random. Hence, the H_2 outputs of $\textsc{Sim}_{\mathsf{OPRF}}$ are equally distributed to the outputs of the random oracle H_2 in the real world.

It is left to analyze the probability that $\textsc{Sim}_{\mathsf{OPRF}}$ queries $\mathcal{F}_{\mathsf{OPRF}}$ with either $(\textsc{Eval}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{S}, x)$ and $(\textsc{RcvComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathcal{A}, \mathsf{S})$, or with $(\textsc{OfflineEval}, \mathsf{sid}, i, \mathsf{S}', x)$ and does not receive a PRF value as reply. This can happen for $\textsc{Eval}$ and $\textsc{RcvComplete}$ queries in case their inputs do not correspond to each other, or if no tickets are left (i.e., $\mathsf{tx}[\mathsf{S}, \mathsf{kid}] = 0$). For $\textsc{OfflineEval}$, $\textsc{Sim}_{\mathsf{OPRF}}$ only does not receive a reply if S' is honest and has previously initialized key identifier i.

For $\textsc{OfflineEval}$, $\textsc{Sim}_{\mathsf{OPRF}}$ calls this interface with inputs S, i in three places in step 8, where in the first occurrence S is $\textsc{Compromised}$, and in the second and third occurrence $\mathsf{S} = \mathcal{A}$. Thus, $\textsc{OfflineEval}$ always outputs a PRF value y to $\textsc{Sim}_{\mathsf{OPRF}}$.

For $\textsc{Eval}$ and $\textsc{RcvComplete}$, $\textsc{Sim}$ calls these interfaces in step 8, first bullet, second dash. Since $\textsc{Sim}_{\mathsf{OPRF}}$ uses corresponding inputs, $\mathcal{F}_{\mathsf{OPRF}}$ not replying is not due to mismatching inputs but due to $\mathsf{tx}[\mathsf{S}, \mathsf{kid}] = 0$ as checked by $\mathcal{F}_{\mathsf{OPRF}}$ in $\textsc{RcvComplete}$. Let $\textsc{Fail}(\mathsf{S}, \mathsf{kid})$ denote the event that a $(\textsc{RcvComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathcal{A}, \mathsf{S})$ message is ignored. We have $\Pr[\textsc{Sim}_{\mathsf{OPRF}}\text{ aborts}] \leq \sum_{\mathsf{S},\mathsf{kid}} \Pr[\textsc{Fail}(\mathsf{S}, \mathsf{kid})]$.

We now upper bound $\Pr[\textsc{Fail}(\mathsf{S}, \mathsf{kid})$ by reducing to the one-more DH problem using the reduction in Figure 13. The overall strategy of the reduction is the following: the challenge key $\bar{k}$ is only implicitly known as $g^{\bar{k}}$ and the reduction records the tuple $(g, g^{\bar{k}})$ as key of $\tilde{\mathsf{S}}, \tilde{\mathsf{kid}}$. The reduction puts the challenge generators $g_1, \ldots, g_N$ as H_1 replies and first messages of $\textsc{Eval}$ queries for $\tilde{\mathsf{S}}, \tilde{\mathsf{kid}}$. It is now left to run the rest of the execution without knowledge of $\bar{k}$ and the exponents (trapdoors) of the generators. The strategy is as follows:

- The reduction uses its $(\cdot)^{\bar{k}}$ exponentiation oracle to produce messages on behalf of $\tilde{\mathsf{S}}$ for key $\tilde{\mathsf{kid}}$.
- The reduction uses its DDH oracle $\mathsf{DDH}(g, g^{\bar{k}}, X, Y)$ to recognize an adversarially-given tuple (X, Y) that lets it win the one-more DH game.
- The reduction uses its DDH oracle $\mathsf{DDH}(g_j, Y, g_{j'}, B)$ to recognize re-usage of adversarial keys k in two evaluation transcripts (g_j, Y), $(g_{j'}, B)$.

From the reduction code in Figure 13 we can see the following.

- Every time the exponentiation oracle is used (step 6), a $(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{kid}, *)$ query was issued by S and hence $\mathsf{tx}[\mathsf{S}, \mathsf{kid}]$ was increased by 1.
- The counter $\mathsf{tx}[\mathsf{S}, \mathsf{kid}]$ is decreased whenever $(\textsc{RcvComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, *, \mathsf{S})$ is sent to $\mathcal{F}_{\mathsf{OPRF}}$, which happens in the first bullets of steps 7 and 8. It can be seen from the DDH oracle inputs that in both cases the adversary gave a tuple $g_j, g_j^{\bar{k}}$ (with $\bar{k}$ being the challenge key) and g_j is from $g_1, \dots, g_N$.

Thus, if $\textsc{Fail}(\mathsf{S}, \mathsf{kid})$ occurs and the reduction has guessed the correct initialization query, the number of such tuples is one more than the number of oracle queries made by the reduction (assuming there is no collision in $g_1, \dots, g_N$). That is, $\Pr[\textsc{Fail}(\mathsf{S}, \mathsf{kid}) \mid \text{no collision in } g_1, \dots, g_N] \leq q_I \mathbf{Adv}_{\mathcal{A},\mathbb{G}}^{(N,q_E)\text{-OMDH}}$ with $N := q_E + q_H$, where q_I is the number of $\textsc{Init}$ queries (i.e., honestly initialized keys in the system), q_E is the maximum number of $\textsc{Eval}$ queries over all PRF keys, and q_H is the number of H_1 queries made by $\mathcal{Z}$.

On the other hand, the probability that there is a collision in $g_1, \dots, g_N$ is upper bounded by N^2/q. Thus we have

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

$\square$

D Preliminaries on Triple Diffie–Hellman Key Exchange

In this section we analyze the security of the 3DH protocol as used in the WBP and the OPAQUE draft [26]. In previous work [20] it was shown that a slightly different version of 3DH UC-realizes key-hiding AKE (KH-AKE). Even though the two versions differ only in small details, the variant of 3DH used in WBP is unfortunately not covered by the analysis of [20]. To close this gap, we reenact the previous analysis of [20] for the 3DH variant as used in WBP and show that it UC-realizes AKE with security against key compromise impersonation (KCI), which is a security notion related to KH-AKE but which is not known to be implied by KH-AKE. AKE-KCI is used in [24] to prove security of OPAQUE and we use it in Appendix E to prove that the WBP UC-realizes PPKR. The result shown in this section is of independent interest, as it further justifies the decision to use 3DH as the AKE in the OPAQUE draft [26] instead of HMQV, as suggested by [24].

We consider the 3DH variant described in Figure 14, as matches the use in WBP. The difference from the variant considered in [20] lies in the computation of the key k. While in [20] it is computed as $H(\mathsf{sid}, \mathsf{P}, \mathsf{P}', X, Y, \sigma)$, in Figure 14 it is computed as $H(\mathsf{aux}, X, Y, \sigma)$, where aux is an arbitrary auxiliary input string. This reflects real-world scenarios, where often the session key is computed dependent on additional context information. This is also the case in WBP, where shk depends on the transcript pre. Note that we set aux to $\perp$ when creating a new session, since in many applications the full context information may not be available yet. This is also the case in WBP, where pre contains many values that are computed by the server and therefore unknown to the client at the start of the recovery phase.

We introduce two small changes to the functionality $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ presented in [24] and display the modified version of $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ in Figure 15. First, we add an auxiliary input aux from the adversary to the $\textsc{NewKey}$ interface and ensure that $\textsc{NewKey}$ only outputs the same session key for two sessions if they are provided with the same auxiliary input. This reflects that in real-world scenarios, two parties only compute the same key if they agree on the context. Second, we introduce the interface $\textsc{Init}$, which reflects that in 3DH parties generate a long-term secret key when they are initialized.

We prove the security of 3DH under the following assumption.

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 $r_1, ..., r_N \in \mathbb{Z}_q$ and set $g_1 \leftarrow g^{r_1}, ..., g_N \leftarrow g^{r_N}$. 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 tx[S, 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 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 12: The simulator that demonstrates that "multi-key" 2HashDH UC-realizes our "multi-key" $\mathcal{F}_{\mathsf{OPRF}}$. The simulator is adopted from [24], Figure 14. We add key identifiers and run one instance of their simulator (without prefix simulation) per kid.

Reduction $\mathcal{R}(\tilde{\mathsf{S}}, \tilde{\mathsf{kid}}, g, y = g^{\bar{k}}, g_1, ..., g_N)$

The reduction runs simulator $\textsc{Sim}_{\mathsf{OPRF}}(\mathsf{sid}, H_1, H_2, \perp, N)$ with the following modifications:

1. No change.
2. [Place the challenge key:] On $(\textsc{Init}, \tilde{\mathsf{S}}, \tilde{\mathsf{kid}}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{OPRF}}$, record $\langle F, \tilde{\mathsf{S}}, \tilde{\mathsf{kid}}, \perp, y\rangle$ and $\langle \mathsf{S}, \tilde{\mathsf{kid}}\rangle$. From now on, use $\tilde{\mathsf{kid}}$ to denote the guessed key identifier, and $\tilde{\mathsf{S}}$ to denote the server who holds it. For all other Init queries, change the recorded tuple to format $\langle F, \mathsf{S}, \mathsf{kid}, k, (g, g^k)\rangle$.
3. No change.
4. [Put challenge generators in H_1:] Every time when there is a fresh query x to $H_1(\cdot)$, answer it with g_J and record $\langle H_1, x, g_J\rangle$. Set $J \leftarrow J + 1$.
5. [Put challenge generators in $\textsc{Eval}$ queries for m-th key kid_i:] Upon receiving $(\textsc{Eval}, \mathsf{sid}, \tilde{\mathsf{kid}}, \mathsf{ssid}, \mathsf{C}, \tilde{\mathsf{S}})$ from $\mathcal{F}_{\mathsf{OPRF}}$, send $(\mathsf{sid}, \tilde{\mathsf{kid}}, \mathsf{ssid}, g_J)$ to $\mathcal{A}$ as C's message to $\tilde{\mathsf{S}}$ and record $\langle \tilde{\mathsf{kid}}, \mathsf{ssid}, \mathsf{C}, \perp\rangle$. Set $J \leftarrow J + 1$. (NB: for key identifiers other than $\tilde{\mathsf{kid}}$, there is no change in code here.)
6. Upon receiving $(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{S})$ 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 $(\mathsf{S}, \mathsf{kid}) \neq (\tilde{\mathsf{S}}, \tilde{\mathsf{kid}})$ and 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 $\mathcal{A}$.
   - [Use exponentiation oracle to compute b for m-th key:] If $(\mathsf{S}, \mathsf{kid}) = (\tilde{\mathsf{S}}, \tilde{\mathsf{kid}})$ then send b to the exponentiation oracle to receive back $b^{\bar{k}}$, and send $(\mathsf{sid}, \mathsf{ssid}, b^{\bar{k}})$ as the response of S for client C to $\mathcal{A}$. Record $(\mathsf{reftuple}, a, b^{\bar{k}})$ if this was the first usage of the oracle.
7. [Use DDH oracle to compensate for not knowing H_1 exponent r:] 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 $\langle *, \mathsf{ssid}, \mathsf{C}, r\rangle$, g_j sent in step 5 for ssid, C and $(\mathsf{reftuple}, A, B)$ from step 6, and do:
   - [$\mathcal{A}$ delivers b for challenge $\tilde{\mathsf{kid}}$ honestly.] If $\mathsf{DDH}(g_j, b, A, B) = 1$, send $(\textsc{RcvComplete}, \mathsf{sid}, \tilde{\mathsf{kid}}, \mathsf{ssid}, \mathsf{C}, \tilde{\mathsf{S}})$ to $\mathcal{F}_{\mathsf{OPRF}}$.
   - [$\mathcal{A}$ delivers b for non-challenge kid 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 $\langle M, \mathcal{A}, i, \perp, (G, H)\rangle$ with $\mathsf{DDH}(G, H, g_j, b) = 1$, 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 $\langle F, \mathsf{S}, \mathsf{kid}, k, (g, g^k)\rangle$ with $b = g_j^k$ 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 none of the above applies, set $i \leftarrow I$, record $\langle M, \mathcal{A}, i, \perp, (g_j, b)\rangle$, set $I++$ and send $(\textsc{SndrComplete}, \mathsf{sid}, i, \mathsf{ssid})$ and $(\textsc{RcvComplete}, \mathsf{sid}, i, \mathsf{ssid}, \mathsf{C}, \mathcal{A})$ to $\mathcal{F}_{\mathsf{OPRF}}$.
8. Every time when there is a fresh query (x, u) to $H_2(\cdot, \cdot)$, retrieve record $\langle H_1, x, g_j\rangle$. If there is no such record, then pick $H_2(x, u) \xleftarrow{\$} \{0,1\}^l$. Otherwise, retrieve $\langle \mathsf{reftuple}, A, B\rangle$ and do the following:
   - [$u = H(x)^k$ for the challenge key.] If $\mathsf{DDH}(g_j, u, A, B)$ then pick a fresh identifier ssid^* and send $(\textsc{Eval}, \mathsf{sid}, \tilde{\mathsf{kid}}, \mathsf{ssid}^*, \perp, x)$ and $(\textsc{RcvComplete}, \mathsf{sid}, \tilde{\mathsf{kid}}, \mathsf{ssid}^*, \mathcal{A}, \tilde{\mathsf{S}})$ 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 any other server's key.] If some record $\langle F, \mathsf{S}, \mathsf{kid}, k, (g, g^k)\rangle$ satisfies $u = g_j^k$ then:
     - [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 $\langle M, \mathcal{A}, i, \perp, (G, H)\rangle$ with $\mathsf{DDH}(G, H, g_j, u) = 1$ 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 $\langle M, \mathcal{A}, i, \perp, (g_j, u)\rangle$ for $i = I$, send 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 13: Reduction to the one-more DH problem.

```
3DH

P on INIT
    $a \xleftarrow{\$} \mathbb{Z}_p$, $A \leftarrow g^a$
    store $(\mathsf{sk}, \mathsf{pk}) = (\mathsf{sid}, a, A)$ for P

P on $(\textsc{NewSession}, \mathsf{P}')$
    $x \xleftarrow{\$} \mathbb{Z}_p$, $X \leftarrow g^x$
    retrieve $\mathsf{pk} = A$ for P and sid
    send $X, A, \perp$ to $\mathsf{P}'$

P on $Y, B$, aux from $\mathsf{P}'$
    retrieve $(\mathsf{sk}, \mathsf{pk}) = (a, A)$ for P
    if $\mathsf{P} <_{\text{lex}} \mathsf{P}'$
        $\sigma \leftarrow B^x \parallel Y^a \parallel Y^x$
        $k \leftarrow H(\mathsf{aux}, X, Y, \sigma)$
    else
        $\sigma \leftarrow Y^a \parallel B^x \parallel Y^x$
        $k \leftarrow H(\mathsf{aux}, Y, X, \sigma)$
```

Figure 14: Triple Diffie–Hellman Key Exchange 3DH as used in the WBP.

In the description below, we assume $\mathsf{P}, \mathsf{P}' \in \{\mathsf{U}, \mathsf{S}\}$.

- On $(\textsc{Init}, \mathsf{sid})$ from P, send $(\textsc{Init}, \mathsf{sid}, \mathsf{P})$ to $\mathcal{A}$.
- On $(\textsc{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}')$ from P, send $(\textsc{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{P}')$ to $\mathcal{A}$. If ssid was not used before by P, record $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}' \rangle$ and mark it FRESH.
- On $(\textsc{Compromise}, \mathsf{sid}, \mathsf{P})$ from $\mathcal{A}$, mark P COMPROMISED.
- On $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P})$ from $\mathcal{A}$, if P is marked COMPROMISED and there is a record $\langle \mathsf{ssid}, \mathsf{P}, * \rangle$ marked FRESH, mark this record COMPROMISED.
- On $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{aux}, \mathsf{shk}^*)$ from $\mathcal{A}$, where $|\mathsf{shk}^*| = \lambda$, if there is a record $\langle \mathsf{ssid}, \mathsf{P}, [\mathsf{P}'] \rangle$ not marked COMPLETED, do:
  - If the record is marked COMPROMISED, or P or P' is corrupted, set $\mathsf{shk} \leftarrow \mathsf{shk}^*$.
  - If the record is marked FRESH, an output $(\mathsf{sid}, \mathsf{ssid}, \mathsf{aux'}, \mathsf{shk}')$ was sent to P' from $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ while record $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P} \rangle$ was marked FRESH, and $\mathsf{aux} = \mathsf{aux}'$, set $\mathsf{shk} \leftarrow \mathsf{shk}'$.
  - Else pick $\mathsf{shk} \xleftarrow{\$} \{0,1\}^\lambda$.

  Finally, mark $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}' \rangle$ COMPLETED and send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{aux}, \mathsf{shk})$ to P.

Figure 15: Ideal functionality $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$. The changes over [24] are marked with gray boxes.

Simulator $\textsc{Sim}_{\mathsf{3DH}}$

1. On $(\textsc{Init}, \mathsf{sid}, \mathsf{P})$ from $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$: choose $a \xleftarrow{\$} \mathbb{Z}_p$, $A \xleftarrow{\$} g^a$. Store $\langle \mathsf{P}, \mathsf{sid}, a, A \rangle$
2. On $(\textsc{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{P}')$ from $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$:
   - if $\mathsf{P} <_{\text{lex}} \mathsf{P}'$, set $r \leftarrow 0$, else set $r \leftarrow 1$.
   - choose $x \xleftarrow{\$} \mathbb{Z}_p$, set $X = g^x$
   - retrieve $\langle \mathsf{P}, \mathsf{sid}, *, [A] \rangle$
   - store $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{P}', r, x, X \rangle$ and send $X, A, \perp$ to P'.
3. On adversarial message Y, B, aux to P on behalf of P': if there is a record $\langle [\mathsf{sid}], [\mathsf{ssid}], \mathsf{P}, \mathsf{P}', [r], [x], X \rangle$:
   - if there is no record $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{P}', \mathsf{P}, *, [y], Y \rangle$ and P' is compromised in session sid, send $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}')$ to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ and do:
     - if $r = 0$, set $h \leftarrow H(\mathsf{aux}, B^x \parallel Y^a \parallel Y^x)$
     - else, set $h \leftarrow H(\mathsf{aux}, Y^a \parallel B^x \parallel Y^x)$
   - Otherwise, set $h \leftarrow \perp$
   - send $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, h)$ to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$.
4. On $(\textsc{Compromise}, \mathsf{sid}, \mathsf{P})$ from $\mathcal{Z}$: Send $(\textsc{Compromise}, \mathsf{sid}, \mathsf{P})$ to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$, retrieve record $\langle \mathsf{P}, \mathsf{sid}, [a], * \rangle$ and output a to $\mathcal{Z}$.
5. On query (aux, σ) to random oracle H from $\mathcal{Z}$:
   - if there exists a record $\langle H, (\mathsf{aux}, \sigma), [k] \rangle$, output k.
   - Else pick $k \xleftarrow{\$} \{0,1\}^\lambda$ and store record $\langle H, (\mathsf{aux}, \sigma), k \rangle$. Output k.

Figure 16: Simulator $\textsc{Sim}_{\mathsf{3DH}}$ showing that 3DH UC-realizes $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$.

Definition 13 (Gap Computational DH Assumption [20])**.** Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order p. The Gap Computational Diffie–Hellman (GapCDH) assumption holds in $\mathbb{G}$ if for any efficient adversary $\mathcal{A}$,

$$\mathbf{Adv}^{\mathsf{GapCDH}}_{\mathcal{A},\mathbb{G}} := \Pr_{x \xleftarrow{\$} \mathbb{Z}_p^*, y \xleftarrow{\$} \mathbb{Z}_p^*} \left[\mathcal{A}^{\mathsf{DDH}(\cdot,\cdot,\cdot)}(g^x, g^y) = g^{xy} \right]$$

is negligible, where $\mathsf{DDH}(\cdot,\cdot,\cdot)$ is a Diffie–Hellman oracle that takes as input (g^x, g^y, g^z) and returns 1 if $z = xy$ and 0 otherwise.

Theorem 3. *Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order p and $H : \{0,1\}^* \times \mathbb{G}^3 \to \{0,1\}^\lambda$ be a hash function. Suppose the GapCDH Assumption holds in $\mathbb{G}$ and let H be a random oracle.*

Then the 3DH protocol of Figure 14 UC-realizes $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ of Figure 15. More precisely, for any efficient adversary against 3DH, there is an efficient simulator $\textsc{Sim}_{\mathsf{3DH}}$ *that interacts with $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ and produces a view such that for any efficient environment $\mathcal{Z}$ it holds that*

$$\mathbf{Dist}^{\mathsf{3DH},\{\mathcal{F}_{\mathsf{AKE\text{-}KCI}},\textsc{Sim}_{\mathsf{3DH}}\}}_{\mathcal{Z}}(\lambda) \le \mathbf{Adv}^{\mathsf{GapCDH}}_{\mathcal{B}_1,\mathbb{G}}(\lambda) + 2q_I \cdot \mathbf{Adv}^{\mathsf{GapCDH}}_{\mathcal{B}_2,\mathbb{G}}(\lambda) + \frac{q_S^2}{p},$$

where q_I denotes the number of $\textsc{Init}$ *queries to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ and q_S the number of* $\textsc{NewSession}$ *queries to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$.*

$\textsc{Proof}$. We describe the simulator $\textsc{Sim}_{\mathsf{3DH}}$ in Figure 16. We now show a sequence of hybrid experiments $\mathbf{G}_0, \dots, \mathbf{G}_6$, where starting from the real-world execution $\mathrm{EXEC}_{\mathsf{3DH},\mathcal{A},\mathcal{Z}}$ and make small incremental changes until we reach the ideal-world execution $\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{AKE\text{-}KCI}}}$ with $\textsc{Sim}_{\mathsf{3DH}}$. While some steps are similar to the proof in [20], we cannot fully adapt their proof due to the different functionalities. We write $\Pr[\mathbf{G}_i]$ as shorthand for the probability that the environment outputs 1 in $\mathbf{G}_i$. Let $q_H \in \mathbb{N}$ denote the number of H queries and $q_I \in \mathbb{N}$ the number of $\textsc{Init}$ queries to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$.

Game $\mathbf{G}_0$: This is the real world execution $\mathrm{EXEC}_{\mathsf{3DH},\mathcal{A},\mathcal{Z}}$.

Game $\mathbf{G}_1$: In this game we move everything the protocol parties do to the simulator who internally executes all parties. We also add an ideal functionality that does nothing but forwarding every input it gets to the simulator. To make the changes oblivious to the environment we also add dummy parties that forward the input they get from $\mathcal{Z}$ to the functionality. Finally, we equip the functionality with dummy interfaces that allow the simulator to let any party produce any output chosen by the simulator. As these are only syntactical changes, we have

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

Game $\mathbf{G}_2$: Whenever an honest party P sends a message $X, A, \perp$ for $X = g^x$ such that X was already sent by another honest party $\mathsf{P}' \neq \mathsf{P}$, i.e., the simulator stored a record $\langle *, *, \mathsf{P}', *, *, x, X \rangle$, the experiment aborts. Since $x \xleftarrow{\$} \mathbb{Z}_p$ was sampled uniformly at random and there are at most q_S queries to $\textsc{NewSession}$, due to the birthday bound we have

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \le \frac{q_S^2}{p}.$$

Game $\mathbf{G}_3$: In this game, we abort if the environment queries H on input aux, σ where $\sigma = L \parallel M \parallel N$ corresponds to the output of two honest parties. More precisely, if the simulator stored records

$\langle \mathsf{P}, \mathsf{sid}, a, A\rangle$, $\langle \mathsf{P}', \mathsf{sid}, b, B\rangle$, $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{P}', 0, x\rangle$, and $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{P}', \mathsf{P}, 1, y\rangle$, s.t. $L = B^x, M = Y^a$, $N = g^{xy}$, we abort the game. Now, we can construct an adversary $\mathcal{B}_1$ that wins the GapCDH game if $\mathcal{Z}$ ever makes a query of this form. The reduction works as follows.

At the start of the game, it receives a CDH challenge $(\bar{X}, \bar{Y})$. Then, for every message $(\textsc{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{P}')$ from $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$, where P is honest, it simulates the message of P by choosing $s \xleftarrow{\$} \mathbb{Z}_p$ and setting $X \leftarrow \bar{X}^s$ if $r = 0$, or $t \xleftarrow{\$} \mathbb{Z}_p, Y \leftarrow \bar{Y}^t$ if $r = 1$. Instead of storing $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', *, x, X\rangle$, as $x = \mathsf{dlog}_g(X)$ is not known to the reduction, it stores $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{P}', *, s, X\rangle$, resp. $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{P}', *, t, Y\rangle$. Now, on a query $\mathsf{aux}, L \parallel M \parallel N$ to H, the reduction can check with its DDH oracle if $N = \mathsf{CDH}(\bar{X}, Y^s)$ and $N = \mathsf{CDH}(\bar{Y}, X^t)$ for any of the recorded pairs s, X and t, Y. If the check is successful, the reduction returns $N^{1/st}$ as result to its challenger. It is easy to see that the reduction wins iff the abort happens. Thus, we have

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq \mathbf{Adv}^{\mathsf{GapCDH}}_{\mathcal{B}_1,\mathbb{G}}(\lambda).$$

Game $\mathbf{G}_4$: In this game, for honest parties that receive a message from an honest party, we let the functionality compute the session key instead of the simulator. That is, on an adversarial message Y, B, aux from P' to P, the simulator checks if there is a record $\langle [\mathsf{sid}], [\mathsf{ssid}], \mathsf{P}', \mathsf{P}, *, *, Y\rangle$ and a record $\langle \mathsf{P}', \mathsf{sid}, *, B\rangle$. If such records exists, this means that P' is honest and that Y and B were generated by the simulator. It then makes the functionality output a key $\mathsf{shk} \xleftarrow{\$} \{0,1\}^\lambda$ or shk', if the functionality already output a key shk' to P' in the same subsession ssid of session sid. In $\mathbf{G}_4$, P therefore always outputs a uniformly random key. Conversely, in $\mathbf{G}_3$ the key for P was computed by the simulator as $H(\mathsf{aux}, B^x \parallel Y^a \parallel Y^x)$. However, due to the abort condition introduced in $\mathbf{G}_3$ the environment cannot query the random oracle for this exact input. Therefore, the output of P in $\mathbf{G}_4$ and $\mathbf{G}_3$ is indistinguishable and we have

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

Game $\mathbf{G}_5$: In this game, we change the way honest parties receive their output if the received message Y, B, aux was sent maliciously, i.e., there is no record $\langle [\mathsf{sid}], [\mathsf{ssid}], \mathsf{P}', \mathsf{P}, *, y, Y\rangle$ or $\langle \mathsf{P}', \mathsf{sid}, *, B\rangle$. We distinguish three cases here:

- P' is corrupt: We continue to let the functionality output a key shk^* to P that was provided by the simulator. In $\mathbf{G}_3$ the key of any such session was computed as $H(\mathsf{aux}, B^x \parallel Y^a \parallel Y^x)$ or $H(\mathsf{aux}, Y^a \parallel B^x \parallel Y^x)$, depending on the role of P. In $\mathbf{G}_5$ the simulator gives exactly that value to the functionality.
- P' is compromised: The simulator first sends $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}')$ to $\mathcal{F}_{\mathsf{PPKR}}$ to mark this session as $\textsc{compromised}$. Afterwards, we continue as in the case above.
- P' is honest and not compromised: In $\mathbf{G}_3$, P outputs a key that was computed as $H(\mathsf{aux}, B^x \parallel Y^a \parallel Y^x)$ or $H(\mathsf{aux}, Y^a \parallel B^x \parallel Y^x)$, depending on its role. In $\mathbf{G}_5$, we let the functionality output a uniformly random key $\mathsf{shk}^* \xleftarrow{\$} \{0,1\}^\lambda$. $\mathcal{Z}$ can only notice the difference by querying $\mathsf{aux}, B^x \parallel Y^a \parallel Y^x$ to H, where B is the public key of P'. If $\mathcal{Z}$ makes such a query, we abort the game. Since P' is honest and not compromised, b s.t. $B = g^b$ is unknown to $\mathcal{Z}$. We can thus create an adversary $\mathcal{B}_2$ that wins the GapCDH game if the environment ever queries $H(\mathsf{aux}, B^x \parallel Y^a \parallel Y^x)$, if P has role $r = 0$.
  The reduction works as follows: On a challenge $(\bar{X}, \bar{B})$ the reduction guesses an index $i \in \{1, \ldots q_I\}$ and on the i-th $(\textsc{Init}, *, *)$ output from $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ it outputs $\bar{B}$. On every $\textsc{NewSession}$ message from $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$, the reduction chooses $s \xleftarrow{\$} \mathbb{Z}_p$ and computes $X \leftarrow \bar{X}^s$. It outputs X

as message for that party and, similarly as the reduction in game $\mathbf{G}_3$, stores s instead of x. When the reduction receives a query $H(\mathsf{aux}, L \parallel M \parallel N)$ it uses its DDH oracle to check if $L = \mathsf{CDH}(\bar{B}, X)$ for any recorded X. If that is the case, it retrieves the s stored alongside X and outputs $L^{1/s}$ as to its challenger. It is easy to see that the reduction succeeds if the guessed index i is correct. If P has role $r = 1$ and $\mathcal{Z}$ queries $H(\mathsf{aux}, Y^a \parallel B^x \parallel X^y)$, then we construct an analogous reduction that solves the GapCDH problem in essentially the same way.

Overall we get

$$|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| \leq 2q_I \cdot \mathbf{Adv}^{\mathsf{GapCDH}}_{\mathcal{B}_2,\mathbb{G}}(\lambda).$$

Game $\mathbf{G}_6$: In this step we replace the simulator and the functionality described in $\mathbf{G}_5$ with the simulator and the functionality from Figure 16 and Figure 15. One can verify that this does not change the distribution of the experiment, that is,

$$\Pr[\mathbf{G}_6] = \Pr[\mathbf{G}_5].$$

Combining all probabilities yields the bound claimed in the theorem.

□

E Proof of Theorem 1

We start by explaining our proof steps. As explained in Section 5, we can focus on simulating an interaction between an honest client and a corrupt server. We start with the real execution of the protocol as depicted in Figures 4 and 5, using $\mathcal{F}_{\mathsf{HSM}}$ for the HSM part. Then, we gradually change this execution until we end up in the ideal execution with $\mathcal{F}_{\mathsf{PPKR}}$ and the simulator described in Appendices E.1 and E.2. The first step is that we pull the whole protocol execution into one box called the simulator SIM, which allows us to modify the protocol code of parties using instructions as "If message X is adversarial" (since the simulator is aware of the actions of the real-world adversary in the UC model). First, we make some "easy" changes, i.e., we change parts where there are no dependencies within the protocol. For example, we can drop signature verification and instead let clients abort upon adversarial messages b_1 or b_2. The environment cannot notice due to the unforgeability of the signature scheme. The reduction is possible because the signing key is used only for signing b_1 and b_2, i.e., there are no other protocol parts depending on it. Below we list all these "easy" changes and their underlying assumptions.

- $\mathbf{G}_2$: Let the client output FAIL upon adversarial σ_1 or σ_2 [sEUF-CMA security of signature scheme]
- $\mathbf{G}_3$: Abort the simulation if one of the nonces n_e, n_S, n_C is the same for two independent protocol executions.
- $\mathbf{G}_4$: Switch E to an encryption of 0 [IND-CPA of PKE]
- $\mathbf{G}_5$: In an initialization between client and $\mathcal{F}_{\mathsf{HSM}}$ (we call this event $[E_{a_1}]$ as it corresponds to honest delivery of the a_1 message from the client to $\mathcal{F}_{\mathsf{HSM}}$), let $\mathcal{F}_{\mathsf{HSM}}$ output FAIL upon receipt of E if any of the values a_1, b_1, n_1 are not delivered honestly [collision resistance of H_3]
- $\mathbf{G}_6$: Abort the whole simulation if, in any recovery session between a client and $\mathcal{F}_{\mathsf{HSM}}$ (we call this event $[E_{a_2}]$, as it corresponds to honest delivery of the a_2 message from the client to $\mathcal{F}_{\mathsf{HSM}}$), pre of $\mathcal{F}_{\mathsf{HSM}}$ is not equal to pre' of the honest client, but $\mathsf{H}_3(\mathsf{pre}) = \mathsf{H}_3(\mathsf{pre}')$ [collision resistance of H_3]
- $\mathbf{G}_7$: Draw $\mathsf{e\_cred}$ at random instead of computing it from sk_C. SIM records pairs $\mathsf{sk}_C, \mathsf{e\_cred}, K^{\mathsf{mask}}$ to let parties "decrypt" sk_C correctly from $\mathsf{e\_cred}$ and K^{mask}. This step ensures that $\mathsf{e\_cred}$ leaks no information about sk_C [information-theoretic security of the one-time-pad]

Next, in game $\mathbf{G}_8$ we replace the 2HashDH part by $\mathcal{F}_{\mathsf{OPRF}}$ and $\textsc{Sim}_{\mathsf{OPRF}}$. We can do this because we show, in Appendix C, that the 2HashDH protocol of the WBP, which we extract in Figure 11, UC-realizes functionality $\mathcal{F}_{\mathsf{OPRF}}$ (Figure 10), with simulator $\textsc{Sim}_{\mathsf{OPRF}}$ of Figure 12 (we prove this result in Theorem 2). We hence can generate all PRF values by $\mathcal{F}_{\mathsf{OPRF}}$ and transcripts a_1, b_1, a_2, b_2 from simulator $\textsc{Sim}_{\mathsf{OPRF}}$. We then relax the simulated $\mathcal{F}_{\mathsf{OPRF}}$ to always answer to the simulator in game $\mathbf{G}_9$.

We next do the same for the 3DH part of the WBP in game $\mathbf{G}_{10}$, which we extract in Figure 14, and show it to UC-emulate functionality $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ of Figure 15 with simulator $\textsc{Sim}_{\mathsf{3DH}}$ (Figure 16), in Theorem 3. Hence, we let $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ generate all 3DH keys, and use the simulator $\textsc{Sim}_{\mathsf{3DH}}$ to generate keys and transcripts. A subtlety here is that this step only works after making $\mathsf{e\_cred}$ independent of sk_C, which is a value kept secret by honest parties in the 3DH protocol.

With PRF values and AKE session keys freshly chosen by the corresponding functionalities, we can now make the following changes to the protocol execution.

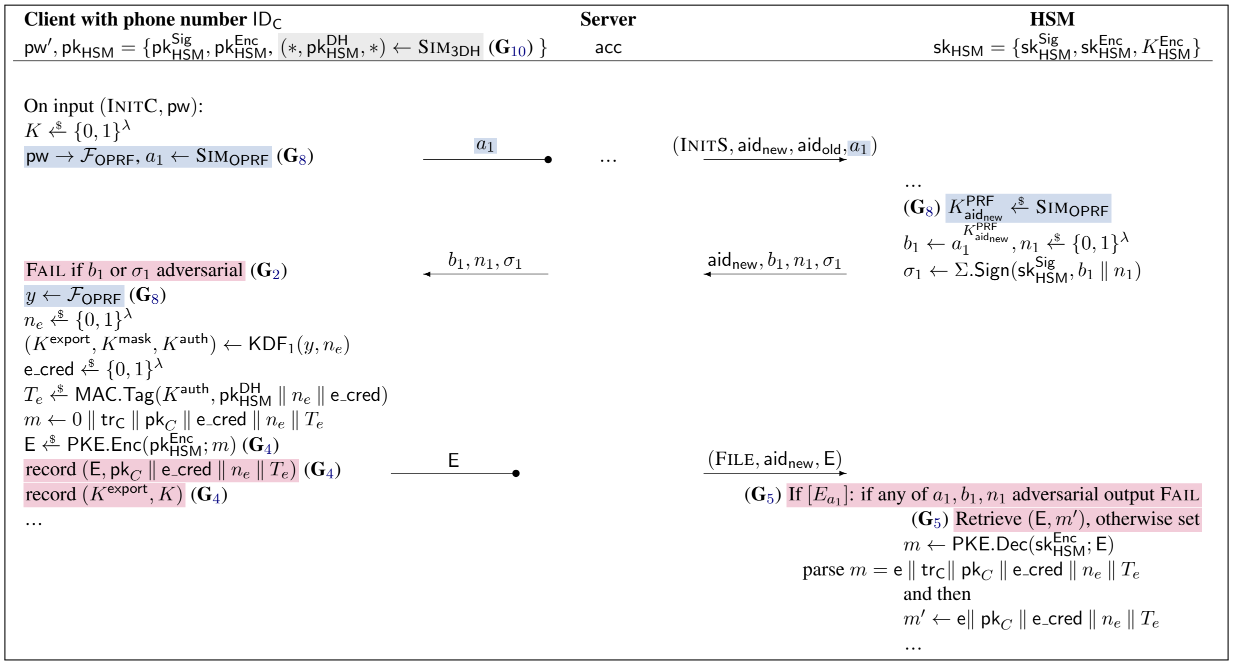
- $\mathbf{G}_{11}$: Abort the whole simulation if an adversarial $\mathsf{e\_cred}, n_e, T_e$ decrypts with $\mathsf{AE}'.\mathsf{Dec}$ under any two K^{auth} known to $\mathcal{A}$ [RKR security of AE' scheme]
- $\mathbf{G}_{12}$: Randomize $K_S^{\mathsf{MAC}}, K_C^{\mathsf{MAC}}, \mathsf{shk}$ following honest AKE sessions [syntactical since $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ chooses them uniformly at random]
- $\mathbf{G}_{13}$: Skip computation of T_S' and instead let the client output $\textsc{Fail}$ upon adversary modifying any elements of pre or T_S during transmission [sEUF-CMA security of MAC scheme]
- $\mathbf{G}_{14}$: In honest recovery sessions ($[E_{a_2}]$), skip computation of T_C and instead let $\mathcal{F}_{\mathsf{HSM}}$ output $\textsc{Fail}$ upon adversary modifying the transmitted T_C', T_S, or any value of pre [sEUF-CMA security of the MAC scheme]
- $\mathbf{G}_{15}$: Client outputs $\textsc{Fail}$ upon adversarial c [Ciphertext integrity of AE]
- $\mathbf{G}_{16}$: Client does not verify the MAC T_e anymore.
- $\mathbf{G}_{17}$: In honest recovery sessions, we let $\mathcal{F}_{\mathsf{HSM}}$ encrypt 0 instead of e to generate ciphertext c [IND-CPA security of AE]

We depict the resulting protocol execution of game $\mathbf{G}_{17}$ in Figures 17 and 18. It can be seen from the figure that the key K chosen by the client upon initialization does not enter any computation and hence the whole protocol transcript is independent of it. This facilitates the introduction of $\mathcal{F}_{\mathsf{PPKR}}$, which can now just sample K on behalf of honest clients. The introduction of the functionality next to the simulated execution of game $\mathbf{G}_{17}$ is detailed in games $\mathbf{G}_{18}$ - $\mathbf{G}_{23}$ below.

In Appendices E.1 to E.2 we describe the simulator for the honest client corrupt server case. In the following we describe a sequence of hybrid experiments and argue for each hop why it is (at least) computationally indistinguishable from the previous. Let π denote the WhatsApp encrypted backups protocol from Figures 4 to 5. We start with $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$, that means the first game is the real-world experiment. The goal is to construct the sequence of games to eventually reach $\mathrm{IDEAL}_{\mathcal{F}_{\mathsf{PPKR}}}$, the ideal-world experiment. We argue step by step why each hop between two experiments is indistinguishable from the previous. We write $\mathbf{Dist}_{\mathcal{Z}}^{\pi,\{\mathcal{F},\textsc{Sim}\}}(\lambda)$ for the distinguishing advantage of environment $\mathcal{Z}$ between the real execution of π and the simulation by $\textsc{Sim}$ interacting with $\mathcal{F}$. Further, we write $\Pr[\mathbf{G}_i]$ as shorthand for the probability that the environment outputs 1 in hybrid $\mathbf{G}_i$.

Game $\mathbf{G}_0$: The real execution.

Game $\mathbf{G}_1$: Create simulator and functionality. In this game we move everything the protocol parties do to the simulator who internally executes all parties. Note that the corrupt server does not actually execute the protocol but does whatever the environment tells it (through $\textsc{Sim}$) to do. We also add an ideal functionality $\mathcal{F}$ that does nothing but forwarding every input it gets to the simulator. To make


Client with phone number $\mathsf{ID_C}$

$\mathsf{pw}', \mathsf{pk_{HSM}} = \{\mathsf{pk}^{\mathsf{Sig}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{Enc}}_{\mathsf{HSM}}, (*, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}, *) \leftarrow \textsc{Sim}_{\mathsf{3DH}}$ ($\mathbf{G}_{10}$) $\}$

On input $(\textsc{InitC}, \mathsf{pw})$:
$K \xleftarrow{\$} \{0,1\}^\lambda$
$\mathsf{pw} \to \mathcal{F}_{\mathsf{OPRF}}, a_1 \leftarrow \textsc{Sim}_{\mathsf{OPRF}}$ ($\mathbf{G}_8$)
$\textsc{Fail}$ if b_1 or σ_1 adversarial ($\mathbf{G}_2$)
$y \leftarrow \mathcal{F}_{\mathsf{OPRF}}$ ($\mathbf{G}_8$)
$n_e \xleftarrow{\$} \{0,1\}^\lambda$
$(K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}}) \leftarrow \mathsf{KDF}_1(y, n_e)$
$\mathsf{e\_cred} \xleftarrow{\$} \{0,1\}^\lambda$
$T_e \xleftarrow{\$} \mathsf{MAC.Tag}(K^{\mathsf{auth}}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}} \parallel n_e \parallel \mathsf{e\_cred})$
$m \leftarrow 0 \parallel \mathsf{tr_C} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e$
$\mathsf{E} \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk}^{\mathsf{Enc}}_{\mathsf{HSM}}; m)$ ($\mathbf{G}_4$)
record $(\mathsf{E}, \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e)$ ($\mathbf{G}_4$)
record (K^{export}, K) ($\mathbf{G}_4$)
…

Server

acc

$\xrightarrow{a_1}$ … $\xrightarrow{(\textsc{InitS}, \mathsf{aid_{new}}, \mathsf{aid_{old}}, a_1)}$

$\xleftarrow{b_1, n_1, \sigma_1}$ $\xleftarrow{\mathsf{aid_{new}}, b_1, n_1, \sigma_1}$

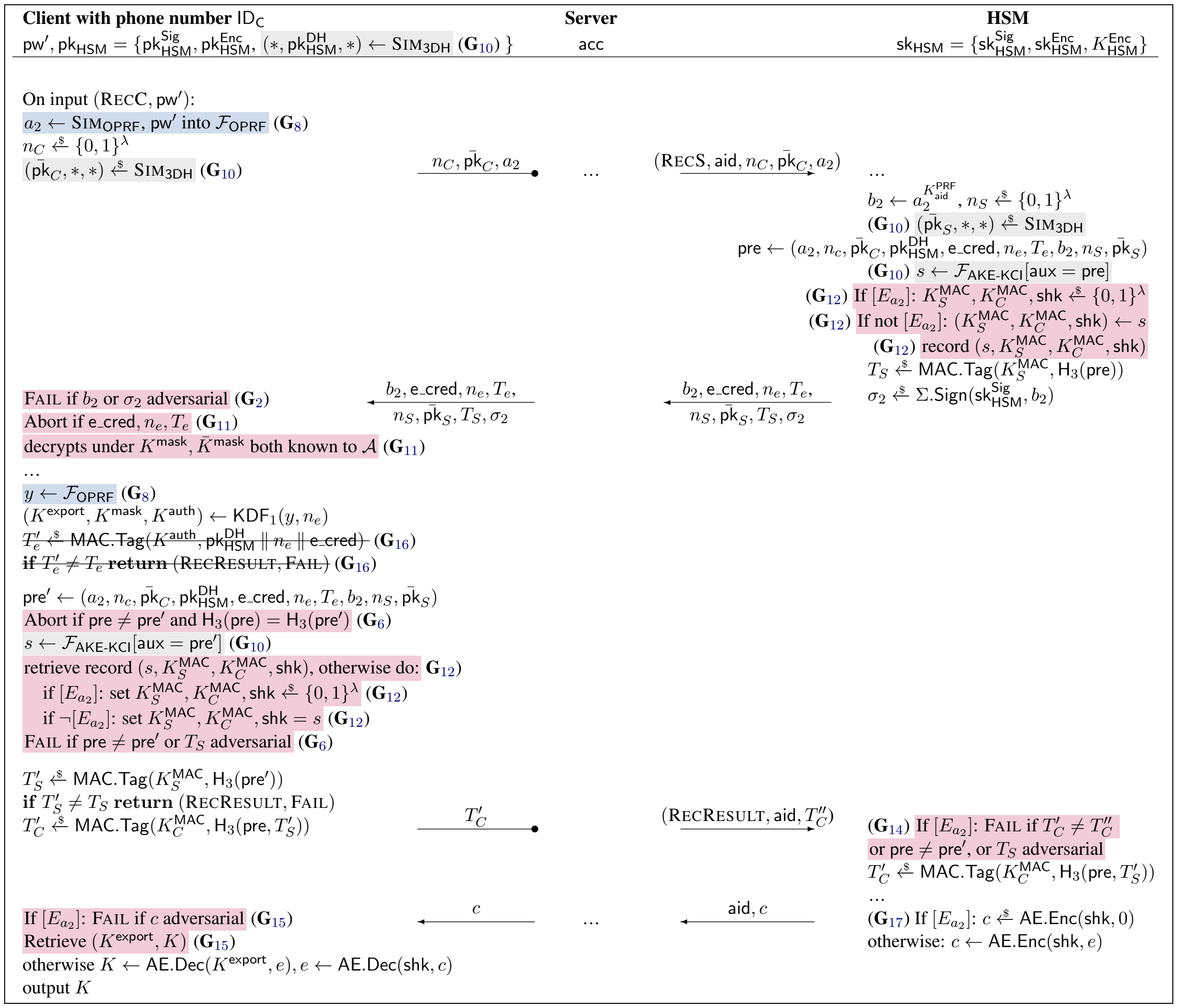
$\xrightarrow{\mathsf{E}}$ $\xrightarrow{(\textsc{File}, \mathsf{aid_{new}}, \mathsf{E})}$

HSM

$\mathsf{sk_{HSM}} = \{\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}, \mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}, K^{\mathsf{Enc}}_{\mathsf{HSM}}\}$

…
($\mathbf{G}_8$) $K^{\mathsf{PRF}}_{\mathsf{aid_{new}}} \xleftarrow{\$} \textsc{Sim}_{\mathsf{OPRF}}$
$b_1 \leftarrow a_1^{K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}}, n_1 \xleftarrow{\$} \{0,1\}^\lambda$
$\sigma_1 \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}, b_1 \parallel n_1)$
($\mathbf{G}_5$) If $[E_{a_1}]$: if any of a_1, b_1, n_1 adversarial output $\textsc{Fail}$
($\mathbf{G}_5$) Retrieve (E, m'), otherwise set
$m \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}; \mathsf{E})$
parse $m = \mathsf{e} \parallel \mathsf{tr_C} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e$
and then
$m' \leftarrow \mathsf{e} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e$
…

Figure 17: The simulated execution of WBP initialization in game $\mathbf{G}_{17}$, particularly marking replaced 2HashDH ($\mathbf{G}_8$) instructions with their idealized versions. We use … for skipping over unchanged parts. Red boxes mark instructions executed by the simulator who can access all values. The event $[E_{a_1}]$ corresponds to the server forwarding a client's a_1 to $\mathcal{F}_{\mathsf{HSM}}$.



```
Client with phone number $\mathsf{ID_C}$                                   Server            HSM
$\mathsf{pw}', \mathsf{pk_{HSM}} = \{\mathsf{pk^{Sig}_{HSM}}, \mathsf{pk^{Enc}_{HSM}}, (*, \mathsf{pk^{DH}_{HSM}}, *) \leftarrow \textsc{Sim}_{\mathsf{3DH}}\ (\mathbf{G}_{10})\ \}$     acc     $\mathsf{sk_{HSM}} = \{\mathsf{sk^{Sig}_{HSM}}, \mathsf{sk^{Enc}_{HSM}}, K^{\mathsf{Enc}}_{\mathsf{HSM}}\}$

Client:
On input $(\textsc{RecC}, \mathsf{pw}')$:
$a_2 \leftarrow \textsc{Sim}_{\mathsf{OPRF}}$, $\mathsf{pw}'$ into $\mathcal{F}_{\mathsf{OPRF}}$ $(\mathbf{G}_8)$
$n_C \xleftarrow{\$} \{0,1\}^\lambda$
$(\bar{\mathsf{pk}}_C, *, *) \xleftarrow{\$} \textsc{Sim}_{\mathsf{3DH}}$ $(\mathbf{G}_{10})$

Client -> Server: $n_C, \bar{\mathsf{pk}}_C, a_2$          …
Server -> HSM: $(\textsc{RecS}, \mathsf{aid}, n_C, \bar{\mathsf{pk}}_C, a_2)$

HSM:
…
$b_2 \leftarrow a_2^{K^{\mathsf{PRF}}_{\mathsf{aid}}}$, $n_S \xleftarrow{\$} \{0,1\}^\lambda$
$(\mathbf{G}_{10})$ $(\bar{\mathsf{pk}}_S, *, *) \xleftarrow{\$} \textsc{Sim}_{\mathsf{3DH}}$
$\mathsf{pre} \leftarrow (a_2, n_c, \bar{\mathsf{pk}}_C, \mathsf{pk^{DH}_{HSM}}, \mathsf{e\_cred}, n_e, T_e, b_2, n_S, \bar{\mathsf{pk}}_S)$
$(\mathbf{G}_{10})$ $s \leftarrow \mathcal{F}_{\mathsf{AKE\text{-}KCI}}[\mathsf{aux} = \mathsf{pre}]$
$(\mathbf{G}_{12})$ If $[E_{a_2}]$: $K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk} \xleftarrow{\$} \{0,1\}^\lambda$
$(\mathbf{G}_{12})$ If not $[E_{a_2}]$: $(K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk}) \leftarrow s$
$(\mathbf{G}_{12})$ record $(s, K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk})$
$T_S \xleftarrow{\$} \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_S, \mathsf{H}_3(\mathsf{pre}))$
$\sigma_2 \xleftarrow{\$} \Sigma.\mathsf{Sign}(\mathsf{sk^{Sig}_{HSM}}, b_2)$

HSM -> Server: $b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$
Server -> Client: $b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$

Client:
$\textsc{Fail}$ if $b_2$ or $\sigma_2$ adversarial $(\mathbf{G}_2)$
Abort if $\mathsf{e\_cred}, n_e, T_e$ $(\mathbf{G}_{11})$
decrypts under $K^{\mathsf{mask}}, \bar{K}^{\mathsf{mask}}$ both known to $\mathcal{A}$ $(\mathbf{G}_{11})$
…
$y \leftarrow \mathcal{F}_{\mathsf{OPRF}}$ $(\mathbf{G}_8)$
$(K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}}) \leftarrow \mathsf{KDF}_1(y, n_e)$
~~$T'_e \xleftarrow{\$} \mathsf{MAC.Tag}(K^{\mathsf{auth}}, \mathsf{pk^{DH}_{HSM}} \| n_e \| \mathsf{e\_cred})$~~ $(\mathbf{G}_{16})$
~~**if** $T'_e \neq T_e$ **return** $(\textsc{RecResult}, \textsc{Fail})$~~ $(\mathbf{G}_{16})$
$\mathsf{pre}' \leftarrow (a_2, n_c, \bar{\mathsf{pk}}_C, \mathsf{pk^{DH}_{HSM}}, \mathsf{e\_cred}, n_e, T_e, b_2, n_S, \bar{\mathsf{pk}}_S)$
Abort if $\mathsf{pre} \neq \mathsf{pre}'$ and $\mathsf{H}_3(\mathsf{pre}) = \mathsf{H}_3(\mathsf{pre}')$ $(\mathbf{G}_6)$
$s \leftarrow \mathcal{F}_{\mathsf{AKE\text{-}KCI}}[\mathsf{aux} = \mathsf{pre}']$ $(\mathbf{G}_{10})$
retrieve record $(s, K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk})$, otherwise do: $\mathbf{G}_{12}$)
    if $[E_{a_2}]$: set $K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk} \xleftarrow{\$} \{0,1\}^\lambda$ $(\mathbf{G}_{12})$
    if $\neg[E_{a_2}]$: set $K^{\mathsf{MAC}}_S, K^{\mathsf{MAC}}_C, \mathsf{shk} = s$ $(\mathbf{G}_{12})$
$\textsc{Fail}$ if $\mathsf{pre} \neq \mathsf{pre}'$ or $T_S$ adversarial $(\mathbf{G}_6)$

$T'_S \xleftarrow{\$} \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_S, \mathsf{H}_3(\mathsf{pre}'))$
**if** $T'_S \neq T_S$ **return** $(\textsc{RecResult}, \textsc{Fail})$
$T'_C \xleftarrow{\$} \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_C, \mathsf{H}_3(\mathsf{pre}, T'_S))$

Client -> Server: $T'_C$
Server -> HSM: $(\textsc{RecResult}, \mathsf{aid}, T''_C)$

HSM:
$(\mathbf{G}_{14})$ If $[E_{a_2}]$: $\textsc{Fail}$ if $T'_C \neq T''_C$
or $\mathsf{pre} \neq \mathsf{pre}'$, or $T_S$ adversarial
$T'_C \xleftarrow{\$} \mathsf{MAC.Tag}(K^{\mathsf{MAC}}_C, \mathsf{H}_3(\mathsf{pre}, T'_S))$
…
$(\mathbf{G}_{17})$ If $[E_{a_2}]$: $c \xleftarrow{\$} \mathsf{AE.Enc}(\mathsf{shk}, 0)$
otherwise: $c \leftarrow \mathsf{AE.Enc}(\mathsf{shk}, e)$

HSM -> Server: $\mathsf{aid}, c$          …
Server -> Client: $c$

Client:
If $[E_{a_2}]$: $\textsc{Fail}$ if $c$ adversarial $(\mathbf{G}_{15})$
Retrieve $(K^{\mathsf{export}}, K)$ $(\mathbf{G}_{15})$
otherwise $K \leftarrow \mathsf{AE.Dec}(K^{\mathsf{export}}, e)$, $e \leftarrow \mathsf{AE.Dec}(\mathsf{shk}, c)$
output $K$
```

Figure 18: The simulated execution of WBP recovery in game $\mathbf{G}_{17}$, particularly marking replaced 2HashDH $(\mathbf{G}_8)$ and 3DH $(\mathbf{G}_{10})$ instructions with their idealized versions. We use … for skipping over unchanged parts. Red boxes mark instructions executed by the simulator who can access all values. The event $[E_{a_2}]$ corresponds to the server forwarding a client's a_2 to $\mathcal{F}_{\mathsf{HSM}}$.

the changes oblivious to the environment we also add dummy parties that forward the input they get from $\mathcal{Z}$ to the functionality. Finally, we equip the functionality with dummy interfaces that allow the simulator to make any party output anything that the simulator wants to.

This is merely a syntactical change as still every message is produced as in the real-world. Thus, both hybrids are identically distributed, i.e.,

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

Game $\mathbf{G}_2$: Abort upon signature forgery. In this game hop we let the simulator track every signature σ_1 or σ_2 that is issued by the HSM. If the simulator receives a signature that is not issued by the HSM, the simulator uses its verification key $\mathsf{pk}_{\mathsf{HSM}}^{\mathsf{Sig}}$ to check if the signature verifies. If it does, the simulator aborts the simulation.

$\mathbf{G}_2$ and $\mathbf{G}_1$ only deviate if a signature is sent to SIM that was not issued by the HSM before. If this happens for signature σ_1 we denote it by E_{Sig}^1 and if it happens for σ_2 we denote it by E_{Sig}^2. If E_{Sig}^1 or E_{Sig}^2 occur with non-negligible probability, we can construct an adversary against the sEUF-CMA security of the signature scheme.

The reduction works as follows: Assume there is an environment $\mathcal{Z}^*$ that leads to event E_{Sig}^1 or E_{Sig}^2 when interacting with the functionality and the simulator. Then the reduction machine $\mathcal{B}$ internally runs the whole experiment including the functionality, the simulator and $\mathcal{Z}^*$. Instead of letting the simulator compute $\mathsf{pk}_{\mathsf{HSM}}^{\mathsf{Sig}}$ itself, $\mathcal{B}$ uses the public key that it gets from the challenger. Whenever the simulator's internal HSM would issue a signature, $\mathcal{B}$ asks the signing oracle to produce the signature. When the simulator detects E_{Sig}^1 or E_{Sig}^2, $\mathcal{B}$ outputs the signature and the message that led to this event as a forgery and halts. It is easy to see that $\mathcal{B}$ wins the sEUF-CMA game whenever E_{Sig}^1 or E_{Sig}^2 occurs. Thus, we have

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

Game $\mathbf{G}_3$: Abort upon nonce collision. In this hybrid the simulator tracks all nonces n_e, n_S, n_C produced by an honest party during execution of the protocol. If it ever occurs that one of the nonces collides with the respective nonce from an independent execution of the protocol, i.e., $n_e = n'_e$, $n_S = n'_S$, or $n_C = n'_C$ for independently chosen $n_e, n'_e, n_S, n'_S, n_C, n'_C \xleftarrow{\$} \{0,1\}^\lambda$, then the simulator aborts the simulation. Note that a nonce n_e is chosen for each initialization and nonces n_C, n_S are chosen for each recovery. Let $Q_{\text{INIT}} \in \mathbb{N}$ an upper bound on the number of initializations, and $Q_{\text{REC}} \in \mathbb{N}$ an upper bound on the number of recoveries. We have

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq \binom{Q_{\text{INIT}}}{2} 2^{-\lambda} + \binom{Q_{\text{REC}}}{2} 2^{-\lambda+1}.$$

Game $\mathbf{G}_4$: Client encrypts 0 instead of e. In this game we change the computation of the ciphertext E. Whenever an honest client computes E, the honest client sets $\mathsf{e} \leftarrow 0$ instead of $\mathsf{e} \leftarrow \mathsf{AE.Enc}(K^{\mathsf{export}}; K)$. The simulator stores K and whenever the client successfully finishes a recovery, the simulator gives K as actual output to the client instead of $\mathsf{AE.Dec}(K^{\mathsf{export}}; K)$.

Assume there is an environment $\mathcal{Z}^*$ that can distinguish $\mathbf{G}_4$ and $\mathbf{G}_3$. We construct an adversary $\mathcal{D}$ against the IND-CPA security of PKE as follows:

We construct Q_{INIT} hybrid games $\mathbf{G}_3 = \mathbf{G}_3^0, \mathbf{G}_3^1, \ldots, \mathbf{G}_3^{Q_{\text{INIT}}} = \mathbf{G}_4$, where we replace in $\mathbf{G}_3^i$ the encryption of the actual e by an encryption of 0 in the i-th initialization. If there is a distinguisher $\mathcal{D}$

between $\mathbf{G}_3$ and $\mathbf{G}_4$ then there is an index i such that $\mathbf{G}_3^i$ and $\mathbf{G}_3^{i+1}$ are distinguishable. A reduction from the IND-CPA security of PKE can now use its public key to produce all ciphertexts except for E in the i-th initialization. We define the challenge message $m_0 \leftarrow \mathsf{e} \parallel \mathsf{tr_C} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e$ as in $\mathbf{G}_3$ and the challenge message $m_1 \leftarrow 0 \parallel \mathsf{tr_C} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e$ as in $\mathbf{G}_4$. The ciphertext E is replaced by the challenge ciphertext. The reduction then wins the IND-CPA game with the same probability that the environment has in telling the hybrids apart.

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_3]| \leq Q_{\text{INIT}} \mathbf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{D},\mathsf{PKE}}(\lambda).$$

Game $\mathbf{G}_5$: Abort upon transcript collision. In this game, the simulator tracks all (honest or adversarial) messages a_1 and b_1, n_1. Whenever there is $(a_1, b_1, n_1) \neq (a'_1, b'_1, n'_1)$ but $\mathsf{H}_3(a_1, b_1, n_1) = \mathsf{H}_3(a'_1, b'_1, n'_1)$ it aborts the simulation.

We denote the event that the simulator sees $(a_1, b_1, n_1) \neq (a'_1, b'_1, n'_1)$ with $\mathsf{H}_3(a_1, b_1, n_1) = \mathsf{H}_3(a'_1, b'_1, n'_1)$ as E_{Coll}. If there is an environment $\mathcal{Z}^*$ such that E_{Coll} happens with non-negligible probability, we can construct an adversary $\mathcal{B}$ against the collision resistance of H_3. This adversary just runs the experiment with $\mathcal{Z}^*$ internally and outputs $(a_1, b_1, n_1), (a'_1, b'_1, n'_1)$ as collision. We therefore have

$$|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| \leq \mathbf{Adv}^{\mathsf{CR}}_{\mathcal{B},\mathsf{H}_3}(\lambda).$$

Game $\mathbf{G}_6$: Abort upon preamble collision. In this game, the simulator tracks all messages that occur during a recovery, i.e., either simulated or sent adversarially. It stores these messages as $\mathsf{pre} = (a_2, n_c, \bar{\mathsf{pk}}_C, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}, \mathsf{e\_cred}, n_e, T_e, b_2, n_S, \bar{\mathsf{pk}}_S)$. If there are $\mathsf{pre} \neq \mathsf{pre}'$ but $\mathsf{H}_3(\mathsf{pre}) = \mathsf{H}_3(\mathsf{pre}')$ it aborts the simulation.

If there is an environment $\mathcal{Z}^*$ such that a collision happens with non-negligible probability, we can construct an adversary $\mathcal{B}$ against the collision resistance of H_3. This adversary just runs the experiment with $\mathcal{Z}^*$ internally and outputs $\mathsf{pre}, \mathsf{pre}'$ as a collision as above. We therefore have

$$|\Pr[\mathbf{G}_6] - \Pr[\mathbf{G}_5]| \leq \mathbf{Adv}^{\mathsf{CR}}_{\mathcal{B},\mathsf{H}_3}(\lambda).$$

Game $\mathbf{G}_7$: Randomize sk_C encryption. In this game we change the computation of the ciphertext $\mathsf{e\_cred}$. Whenever an honest client computes E, the honest client chooses $\mathsf{e\_cred} \xleftarrow{\$} \{0,1\}^\lambda$. The simulator stores sk_C and whenever the client recomputes K^{mask} in recovery, the simulator provides it with sk_C for the key exchange. Note that as of $\mathbf{G}_3$ the nonce n_e is different for every initialization. Therefore, K^{mask} is a fresh uniformly random value in every initialization.

From the information-theoretic security of the one-time-pad it follows that

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

Game $\mathbf{G}_8$: Replace the 2HashDH protocol. In this game we change the simulators behavior concerning the messages of the 2HashDH protocol. Instead of computing the messages of the 2HashDH protocol as in the real-world, the simulator uses its internal ideal OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ and its internal OPRF simulator $\textsc{Sim}_{\mathsf{OPRF}}$ to simulate the messages and outputs. In particular, this affects the messages (or parts of messages) a_1, a_2, b_1, b_2 and queries to H_1 and H_2. In Appendices E.1 and E.2 the respective parts are enclosed by **[OPRFsim:]** and **[endOPRFsim]**.

Assume there is an environment $\mathcal{Z}^*$ that can distinguish between games $\mathbf{G}_7$ and $\mathbf{G}_8$. We can construct an environment $\widehat{\mathcal{Z}}$ that can distinguish between the real 2HashDH protocol and the simulated execution with $\textsc{Sim}_{\mathsf{OPRF}}$ and $\mathcal{F}_{\mathsf{OPRF}}$. The environment $\widehat{\mathcal{Z}}$ internally runs the whole experiment including the functionality and $\textsc{Sim}$ with $\mathcal{Z}^*$, but whenever $\textsc{Sim}$ would execute a operation of 2HashDH it instead generates the corresponding output from the OPRF experiment. If $\widehat{\mathcal{Z}}$ interacts with the real 2HashDH protocol, it perfectly simulates $\mathbf{G}_7$ for $\mathcal{Z}^*$, and if it interacts with $\mathcal{F}_{\mathsf{OPRF}}$ and $\textsc{Sim}_{\mathsf{OPRF}}$, it perfectly simulates $\mathbf{G}_8$ for $\mathcal{Z}^*$.

It follows that we have

$$|\Pr[\mathbf{G}_8] - \Pr[\mathbf{G}_7]| \leq \mathbf{Dist}_{\mathcal{Z}^*}^{\mathsf{2HashDH},\{\mathcal{F}_{\mathsf{OPRF}},\textsc{Sim}_{\mathsf{OPRF}}\}}(\lambda).$$

Game $\mathbf{G}_9$: Weaken the $\mathcal{F}_{\mathsf{OPRF}}$ functionality. In this game, we exchange the OPRF-functionality $\mathcal{F}_{\mathsf{OPRF}}$ that the simulator is executing in its head by a weaker functionality $\mathcal{F}'_{\mathsf{OPRF}}$ that does not check the ticket counter. $\mathcal{F}'_{\mathsf{OPRF}}$ works as Figure 10 but on a $(\textsc{RcvComplete}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{P}, \mathsf{S}^*)$ message it still returns $(\textsc{EvalOut}, \mathsf{sid}, \mathsf{ssid}, F_{\mathsf{sid},\mathsf{S}^*,\mathsf{kid}^*}(x))$ even if $\mathsf{tx}[\mathsf{S}^*, \mathsf{kid}^*] \leq 0$.

UC-security gives a lower bound on the security of a protocol. If a protocol π UC-emulates a functionality $\mathcal{F}$ it means intuitively that every attack that is possible against π is also possible against $\mathcal{F}$. In other words, π is at least as secure as $\mathcal{F}$. If we make $\mathcal{F}$ weaker but keep the same π, then π is still at least as secure as the weaker $\mathcal{F}$. In particular, the same simulator can be used to show UC-emulation (if the interfaces of $\mathcal{F}$ did not change). As we remove in $\mathbf{G}_9$ a condition under which $\mathcal{F}_{\mathsf{OPRF}}$ aborted, we make $\mathcal{F}'_{\mathsf{OPRF}}$ weaker, which means that 2HashDH still UC-realizes $\mathcal{F}'_{\mathsf{OPRF}}$. Therefore, we have

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

Game $\mathbf{G}_{10}$: Replace the 3DH protocol. In this game we change the simulators behavior concerning the messages of the 3DH protocol. Instead of computing the messages of the 3DH protocol as in the real-world, the simulator uses its internal ideal AKE functionality $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ and its internal AKE simulator $\textsc{Sim}_{\mathsf{3DH}}$ to simulate the messages and outputs. In particular, this affects the messages $\bar{\mathsf{pk}}_C, \bar{\mathsf{pk}}_S$, the public and private keys $\mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}, \mathsf{pk}_C, \mathsf{sk}_C$, and queries to KDF_2. In Appendices E.1 and E.2 the respective parts are enclosed by **[AKEsim:]** and **[endAKEsim]**.

Assume there is an environment $\mathcal{Z}^*$ that can distinguish between games $\mathbf{G}_9$ and $\mathbf{G}_{10}$. We can construct an environment $\widehat{\mathcal{Z}}$ that can distinguish between the real 3DH protocol and the simulated execution with $\textsc{Sim}_{\mathsf{3DH}}$ and $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$. The environment $\widehat{\mathcal{Z}}$ internally runs the whole experiment including the functionality and $\textsc{Sim}$ with $\mathcal{Z}^*$, but whenever $\textsc{Sim}$ would execute a operation of 3DH it instead generates the corresponding output from the AKE experiment. If $\widehat{\mathcal{Z}}$ interacts with the real 3DH protocol, it perfectly simulates $\mathbf{G}_9$ for $\mathcal{Z}^*$, and if it interacts with $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ and $\textsc{Sim}_{\mathsf{3DH}}$, it perfectly simulates $\mathbf{G}_{10}$ for $\mathcal{Z}^*$.

It follows that we have

$$|\Pr[\mathbf{G}_{10}] - \Pr[\mathbf{G}_9]| \leq \mathbf{Dist}_{\mathcal{Z}^*}^{\mathsf{3DH},\{\mathcal{F}_{\mathsf{AKE\text{-}KCI}},\textsc{Sim}_{\mathsf{3DH}}\}}(\lambda).$$

Game $\mathbf{G}_{11}$: Rule out ambiguous OPAQUE envelopes $\mathsf{e\_cred}, n_e, T_e$. In this game the simulator tracks all KDF_1 outputs $(K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}})$ that it computes (regardless of them being output to $\mathcal{Z}$ or just used internally by $\textsc{Sim}$ to simulate honest users). Further, $\textsc{Sim}$ checks for every $(\mathsf{e\_cred}, T_e)$ from decrypting E or from message $b_2, \mathsf{e\_cred}, n_e, T_e, \ldots$ if $(\mathsf{e\_cred}, T_e)$ decrypts using $\mathsf{AE}'.\mathsf{Dec}$ for two of the previously recorded $(K^{\mathsf{mask}}, K^{\mathsf{auth}})$ and $(K^{\mathsf{mask}\prime}, K^{\mathsf{auth}\prime})$. If it does then the simulator aborts the simulation.

Suppose $\textsc{Sim}$ sees a tuple $(\mathsf{e\_cred}, T_e)$ that decrypts using $\mathsf{AE}'.\mathsf{Dec}$ for two of the recorded $(K^{\mathsf{mask}}, K^{\mathsf{auth}})$ and $(K^{\mathsf{mask}\prime}, K^{\mathsf{auth}\prime})$. We reduce the random-key robustness of AE' on this event as follows:

Let $Q_{\mathsf{KDF}_1} \in \mathbb{N}$ be an upper bound on the number of outputs requested from KDF_1. The reduction internally executes the experiment. At the beginning it guesses two distinct indices $i, j \in \{1, \ldots, Q_{\mathsf{KDF}_1}\}$. It receives two random keys k_1 and k_2 from the challenger. On the i-th KDF_1 query the reduction programs the output of KDF_1 to be $K^{\mathsf{export}} \| k_1$ for some uniformly random K^{export}, and on the j-th query it programs the output to be $K^{\mathsf{export}\prime} \| k_2$ for some uniformly random $K^{\mathsf{export}\prime}$. When it receives a ciphertext $(\mathsf{e\_cred}, T_e)$ that decrypts under k_1 and under k_2 then it outputs $(\mathsf{e\_cred}, T_e)$ to the challenger. We have

$$|\Pr[\mathbf{G}_{11}] - \Pr[\mathbf{G}_{10}]| \leq \binom{Q_{\mathsf{KDF}_1}}{2} \mathbf{Adv}^{\mathsf{RKR}}_{\mathcal{R},\mathsf{AE}'}(\lambda).$$

Game $\mathbf{G}_{12}$: Randomize MACs T'_S, T_C. In this game we replace the values K_C^{MAC} and K_S^{MAC} that are used after an AKE execution between two honest parties by uniformly random values.

This is only a syntactical change as $(K_C^{\mathsf{MAC}}, K_S^{\mathsf{MAC}}, \mathsf{shk})$ is already output by $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ as uniformly random values. Hence, we have

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

Game $\mathbf{G}_{13}$: Skip MAC verification of T'_S. In this game we do not check the MACs T'_S for validity anymore. Instead the simulator makes the client output $\textsc{Fail}$ if the client receives T_S in a recovery session with preamble $\mathsf{pre}' \neq \mathsf{pre}$, where pre is the preamble that $\textsc{Sim}$ used to produce T_S and pre' is the preamble that an honest client sees of the session. $\textsc{Sim}$ also makes the client fail when the adversary gives a new T_S to the honest client that was not produced by $\textsc{Sim}$, which we call event E^1_{Mac}.

Let Q_{REC} be an upper bound on the number of honest recoveries. The reduction guesses an index $i \in \{1, \ldots, Q_{\mathrm{REC}}\}$ and runs the experiment internally. In the i-th recovery, the reduction does not compute T_S itself but instead lets T_S be computed by the MAC oracle that is provided by the challenger, which implicitly sets K_S^{MAC} to the key of the challenger. This is possible as the client and the HSM are both honest parties and thus K_S^{MAC} is a random value as of $\mathbf{G}_{12}$. If the event E^1_{Mac} occurs in the i-th recovery, the reduction gives the T_S that the client received as output to its challenger. One can see that the reduction wins the $\mathsf{sEUF\text{-}CMA}$ game if it guessed i correctly and T_S verifies in E^1_{Mac}. Therefore, we have

$$|\Pr[\mathbf{G}_{13}] - \Pr[\mathbf{G}_{12}]| \leq Q_{\mathrm{REC}} \mathbf{Adv}^{\mathsf{sEUF\text{-}CMA}}_{\mathcal{A},\mathsf{MAC}}(\lambda).$$

Game $\mathbf{G}_{14}$: Skip MAC verification of T_C in honest recoveries. In this game, the simulator does not verify the MAC T_C anymore in recoveries that are started by the honest client (not for recoveries where the corrupt server sent $n_C, \bar{\mathsf{pk}}_C, a_2$ without the client, i.e., $\textsc{MalRec}$). $\textsc{Sim}$ makes the HSM output $\textsc{Fail}$ if it receives T'_C for an honest recovery session that was not produced by the simulator before, when $\textsc{Sim}$ simulated the honest client. We call this event E^2_{Mac}.

The reduction works similar as above. Let Q_{REC} be an upper bound on the number recoveries (and thus also of honest recovery sessions). The reduction guesses an index $i \in \{1, \ldots, Q_{\mathrm{REC}}\}$ and runs the experiment internally. In the i-th recovery where the reduction simulates the behavior of an honest client, the reduction does not compute T'_C itself but instead lets T'_C be computed by the MAC oracle that is provided by the challenger, which implicitly sets K_C^{MAC} to the key of the challenger. This again is possible as the client and the HSM are both honest parties and thus K_C^{MAC} is a random value as of $\mathbf{G}_{12}$. If the event E^2_{Mac} occurs in the i-th recovery, the reduction gives the T'_C that the HSM received as output to its challenger. One can see that the reduction wins the sEUF-CMA game if it guessed i correctly and T'_C verifies in E^2_{Mac}. Therefore, we have

$$|\Pr[\mathbf{G}_{14}] - \Pr[\mathbf{G}_{13}]| \leq Q_{\text{REC}}\mathbf{Adv}^{\mathsf{sEUF\text{-}CMA}}_{\mathcal{A},\mathsf{MAC}}(\lambda).$$

Game $\mathbf{G}_{15}$: Rule out forged c ciphertexts. In this game the simulator tracks all ciphertexts c and aborts the simulation if the client receives a c that was not computed by the HSM.

If the simulator computed a value c on behalf of the HSM but the corrupt server provides a different c' to the client, where c' decrypts without error we can construct an adversary against the ciphertext integrity of AE. First note that shk is uniformly random as it is chosen by $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ and is not used for anything but the encryption and decryption of c. Furthermore, as the HSM and the client are honest, $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ does not output shk to anyone else. Let Q_{REC} be an upper bound on the number of recoveries. The reduction starts by choosing $i \xleftarrow{\$} \{1, \ldots, Q_{\text{REC}}\}$. Now the reduction executes the whole experiment internally. In the i-th recovery the reduction implicitly lets shk be the key of its challenger instead of using the shk that was output by $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$. To this end, the reduction computes c using its encryption oracle when the reduction needs to simulate the HSM's response to a RECRESULT message. If the reduction gets an adversarial c' for this recovery session, it outputs c' as forgery to its challenger. One can see that we have

$$|\Pr[\mathbf{G}_{15}] - \Pr[\mathbf{G}_{14}]| \leq Q_{\text{REC}}\mathbf{Adv}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{A},\mathsf{AE}}(\lambda).$$

Game $\mathbf{G}_{16}$: Client does not verify T_e anymore. In this game the simulator does not check if the MAC T_e verifies when the client receives a message $b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$.

The distribution of the game would change if there was a situation where T_e is not valid but T_S is valid, as in that case the client would not fail in $\mathbf{G}_{16}$ but it would fail in $\mathbf{G}_{15}$. Assume by way of contradiction that this happens. As T_S verifies, the preamble pre used by the client is not altered by the adversary. Else the game would have aborted, see $\mathbf{G}_6$ and $\mathbf{G}_{13}$. Further, the OPRF response b_2 cannot be altered by the adversary as the game would also abort in that case, see $\mathbf{G}_2$. Hence, the only way in which the adversary can influence the verification of T_e with $\mathsf{MAC.Tag}(K^{\mathsf{auth}}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}} \parallel n_e \parallel \mathsf{e\_cred})$ is by running the client on a password $\mathsf{pw}' \neq \mathsf{pw}$, where pw is the password that was used to produce T_e. If $\mathsf{pw}' \neq \mathsf{pw}$, then we have with overwhelming probability that $K^{\mathsf{auth}} \neq K^{\mathsf{auth}'}$, where K^{auth} is the key that was used to generate T_e and $K^{\mathsf{auth}'}$ is the key that is used to verify T_e. By the correctness of the MAC, T_e verifies under K^{auth}. If T_e also verifies under a second key $K^{\mathsf{auth}'}$ then the simulation already aborts as of $\mathbf{G}_{11}$. But then, the adversary cannot influence the execution in a way that leads to T_e failing while T_S verifies. Hence, the distribution of the game does not change by dropping the check of T_e and we have

$$\Pr[\mathbf{G}_{16}] = \Pr[\mathbf{G}_{15}].$$

Game $\mathbf{G}_{17}$: Server encrypts 0 with AE. In this game the the simulator computes c as $\mathsf{AE.Enc}(\mathsf{shk}, 0)$ when it receives an honest T'_C in a recovery session. If an environment can tell $\mathbf{G}_{17}$ and $\mathbf{G}_{16}$ apart with non-negligible probability, we construct a reduction that breaks the IND-CPA security of AE as follows. We define a sequence of hybrids $\mathbf{G}_{16} = \mathbf{G}^0_{16}, \mathbf{G}^1_{16}, \ldots, \mathbf{G}^{Q_{\text{REC}}}_{16} = \mathbf{G}_{17}$. In the i-th game we replace the encryption of c in an honest recovery by the zero-encryption. If the environment distinguish $\mathbf{G}_{16}$ and $\mathbf{G}_{17}$ then there must be an index i such that the environment can distinguish $\mathbf{G}^i_{16}$ and $\mathbf{G}^{i+1}_{16}$. Now in the i-th recovery the reduction does not produce c itself but uses e and 0 as challenge messages. The challenger will answer with a ciphertext c^* that the reduction uses as c in the experiment. Finally, the reduction outputs whatever the environment outputs. One can see that we get

$$|\Pr[\mathbf{G}_{17}] - \Pr[\mathbf{G}_{16}]| \leq Q_{\text{REC}} \mathbf{Adv}_{\mathcal{R},\mathsf{AE}}^{\mathsf{IND\text{-}CPA}}(\lambda).$$

Game $\mathbf{G}_{18}$: Add adversarial interfaces to $\mathcal{F}$.

In this game we add all adversarial interfaces, i.e. MALRECS, MALINITS, COMPLETEINITC, COMPLETEINITC-DOS, COMPLETERECC, COMPLETERECC-DOS, COMPLETEINITS, COMPLETERECS to the functionality $\mathcal{F}$. But the simulator does not make use of them so far.

As the simulator does not use any of the added interfaces (yet) the distribution of the hybrid does not changed and we get

$$\Pr[\mathbf{G}_{18}] = \Pr[\mathbf{G}_{17}].$$

Game $\mathbf{G}_{19}$: Add INITC and RECC interfaces. In this game we introduce slightly modified interfaces INITC and RECC to the functionality and add the interfaces INITS and RECS. This means that the functionality now does not just forward messages to the simulator but it actually does something when receiving messages. However, in comparison to the functionality from Figures 7 to 8 we still give all information about the input to the simulator. More precisely, we change IC.3 "Send $(\text{INITC}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{A}$ and to S" to "Send $(\text{INITC}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw})$ to $\mathcal{A}$ and $(\text{INITC}, \mathsf{sid}, \mathsf{ID_C})$ to S", and we change RC.2 "Send $(\text{RECC}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{A}$ and S" to "Send $(\text{RECC}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}')$ to $\mathcal{A}$ and $(\text{RECC}, \mathsf{sid}, \mathsf{ID_C})$ to S". We also leave out the counter $\mathsf{tx_{sid}}$ for now. In particular this also means that the functionality now creates FILE records and records INITC and RECC.

We argue separately for each interface that is introduced in the new experiment.

- INITC interface: Instead of just forwarding the password pw to the simulator, the functionality now also chooses a key K and creates a record $\langle \text{INITC}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}, K\rangle$. However, as the output of the user in this experiment is not produced by the functionality but still from the simulator executing the real protocol party on pw, the record is never used and thus, the distribution of the experiment does not change.
- RECC interface: Instead of just forwarding the password guess pw' to the simulator, the functionality also creates a record $\langle \text{RECC}, \mathsf{sid}, \mathsf{ID_C}, \mathsf{pw}'\rangle$. However, as the output of the user in this experiment is not produced by the functionality but still from the simulator executing the real protocol party on pw', the record is never used and thus, the distribution of the experiment does not change.
- INITS and RECS interfaces: As the server is corrupt these interfaces are not used.

Overall we have

$$\Pr[\mathbf{G}_{19}] = \Pr[\mathbf{G}_{18}].$$

Game $\mathbf{G}_{20}$: Output of parties comes from $\mathcal{F}_{\mathsf{PPKR}}$ in initialization. In this game hop we change the behavior of the simulator as follows. Instead of executing the real client and forwarding messages from the environment to the corrupt server for the initialization phase of the protocol, the simulator acts as described in Appendix E.1 on $(\text{INITC}, \mathsf{sid}, \mathsf{ID_C})$ messages from $\mathcal{F}_{\mathsf{PPKR}}$ and on adversarial messages b_1, n_1, σ_1 and on adversarial allowance of delivering E. Additionally, we change the behavior of the simulator on calls to the HSM interfaces $(\textsc{GetPK})$, $(\textsc{InitS}, \mathsf{aid}_{\mathsf{new}}, \mathsf{aid}_{\mathsf{old}}, a_1^*)$, $(\textsc{File}, \mathsf{aid}_{\mathsf{new}}, \mathsf{E})$. Instead of internally executing the code of the real HSM, $\textsc{Sim}$ simulates the messages as described in Appendix E.1. However, the simulator still maintains the counter $\mathsf{tx}_{\mathsf{sid}}$.

- $(\textsc{InitC}, \mathsf{sid}, \mathsf{ID_C})$ from $\mathcal{F}_{\mathsf{PPKR}}$ to $\textsc{Sim}$: Additionally to executing $\textsc{Sim}_{\mathsf{OPRF}}$ to generate a_1, which is already done by $\textsc{Sim}$ in $\mathbf{G}_8$, $\textsc{Sim}$ only creates a record, which does not change the distribution.
- Adversarial b_1, n_1, σ_1: First the simulator checks if there is a $\textsc{Replay}$ record indicating that the values b_1, n_1, σ_1 were computed by the HSM. Note that the simulator stores values for currently running sessions with $\textsc{Init}$ records that get deleted after the session completed, while $\textsc{Replay}$ records are not deleted and are used by $\textsc{Sim}$ to check if some adversarial values are replayed from an old session. If no such record is found, in $\mathbf{G}_{20}$ the simulator gives $(\textsc{CompleteInitC-DoS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$ which leads to $\mathsf{ID_C}$ outputting $(\textsc{InitResult}, \textsc{Fail})$. $\mathbf{G}_{20}$ would deviate here from $\mathbf{G}_{19}$ if the client receives a forged signature (event E^1_{Sig}). But as of $\mathbf{G}_2$, the experiment aborts in this case.
  If there is a $\textsc{Replay}$ record containing b_1, n_1, σ_1, then the values b_1, n_1, σ_1 were computed by the HSM. Either the HSM computed them for this initialization session or the adversary is replaying the values from some old session. In the former case, a client in $\mathbf{G}_{19}$ outputs a key and sends a ciphertext E to the server. Thus, $\textsc{Sim}$ provides input $(\textsc{CompleteInitC}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$ to make the client output a key. Note that the key is generated by $\mathcal{F}_{\mathsf{PPKR}}$ as a uniformly random string in $\{0,1\}^\lambda$, just as in $\mathbf{G}_{19}$ by $\textsc{Sim}$. Further, $\textsc{Sim}$ needs to produce a ciphertext E. This ciphertext is computed the same way independent of whether the received message was benign or replayed. The simulator first obtains the value y that $\mathcal{F}_{\mathsf{OPRF}}$ outputs as the result of 2HashDH. The simulator then uses y to compute T_e. It then computes E but instead of using e the simulator encrypts $\mathsf{e}' = 0$, as it already did in $\mathbf{G}_4$.
  Note that a replay of b_1, n_1, σ_1 from some old session does not make $\mathsf{ID_C}$ abort, as since $\mathbf{G}_2$ the client only checks whether the message is adversarial. Thus, in this case $\textsc{Sim}$ provides input $(\textsc{CompleteInitC}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$ to make the client output a key and computes E just as in the benign case above. However, a replayed b_1, n_1 will lead to the server failing later. As introduced in $\mathbf{G}_5$, the server aborts if any of a_1, b_1, σ_1 were not delivered honestly. Therefore $\textsc{Sim}$ proceeds as with a benign message. $\textsc{Sim}$'s internal HSM will later make the the server fail as the transcript will mismatch.
- $(\textsc{GetPK})$ to $\mathcal{F}_{\mathsf{HSM}}$: The simulator from Appendix E.1 behaves exactly as the functionality $\mathcal{F}_{\mathsf{HSM}}$ on $(\textsc{GetPK})$ queries, i.e., it chooses public and private keys for the signature scheme, for the PKE scheme, and for the key exchange and stores them for later use. Thus, the distribution of the experiment does not change.
- $(\textsc{InitS}, \mathsf{aid}_{\mathsf{new}}, \mathsf{aid}_{\mathsf{old}}, a_1^*)$ to $\mathcal{F}_{\mathsf{HSM}}$: The simulator runs the HSM code on $\mathsf{aid}_{\mathsf{new}}, \mathsf{aid}_{\mathsf{old}}, a_1^*$. If the HSM fails on these inputs, the simulator forwards this to the corrupt server. This corresponds exactly to the behavior of the HSM in $\mathbf{G}_{19}$, which checks in every initialization if the provided $\mathsf{aid}_{\mathsf{new}}$ is really new. Note that we discard the output of the HSM if it does not output $\textsc{Fail}$. Instead, $\textsc{Sim}$ uses its internal OPRF simulator to get the output b_1, which computes $b_1 \leftarrow (a_1^*)^{K^{\mathsf{PRF}}_{\mathsf{aid}}}$, just as the HSM would do. Next, $\textsc{Sim}$ distinguishes whether this initialization was started by an honest $\mathsf{ID_C}$ or whether it was started by the corrupt server without any client interaction. In the first case, a record $\langle \textsc{Init}, \mathsf{ID_C}^*, a_1^* \rangle$ exists, since $\textsc{Sim}$ simulated the message a_1^* previously for the honest $\mathsf{ID_C}$. Then $\textsc{Sim}$ appends $\mathsf{aid}_{\mathsf{new}}$ and the computed output to the record. However, if no record $\langle \textsc{Init}, \mathsf{ID_C}^*, a_1^* \rangle$ exists, it means that the simulator is just communicating with the corrupt server and not with an honest $\mathsf{ID_C}$ that solicited the initialization. In

that case SIM creates a record $\langle \text{MALINIT}, \perp, a_1^*, \mathsf{aid}_{\mathsf{new}}^*, b_1, n_1, \sigma_1 \rangle$.
Note that SIM treats the OPRF functionality slightly different in the two above cases. If the initialization was initiated by an honest client, SIM already chose a key identifier kid for $\mathcal{F}_{\mathsf{OPRF}}$ when simulating the first message a_1. So it must continue using that identifier. If the interaction is started by the corrupt server, SIM can freshly choose a kid and we use $\mathsf{aid}_{\mathsf{new}}$ for the sake of simplicity. However, the kid are only internal to SIM and $\mathcal{Z}$ does not see them.
Next, SIM removes all currently stored records that belong to $\mathsf{aid}_{\mathsf{old}}$. This directly corresponds to the behavior of the HSM in $\mathbf{G}_{19}$ that deletes its record of $\mathsf{aid}_{\mathsf{old}}$ when it receives $(\text{INITS}, \mathsf{aid}_{\mathsf{new}}, \mathsf{aid}_{\mathsf{old}}, a_1^*)$. Deleting the record means that the corresponding initializations cannot be completed anymore.

- $(\text{FILE}, \mathsf{aid}_{\mathsf{new}}, \mathsf{E})$ to $\mathcal{F}_{\mathsf{HSM}}$: If no such message is expected at the moment, SIM ignores the message, just as the HSM in $\mathbf{G}_{19}$. Then the simulator executes the same code as the HSM to check if E contains a valid transcript. SIM makes the server output FAIL when the HSM in the real-world does this as well. Note in particular that if E was simulated for the honest client, it contains a transcript that was computed just as in the real-world. That means, if the corrupt server replayed some old b_1, n_1, σ_1 to an honest client, the transcript does not match (we don't have hash collisions E_{Coll} as of $\mathbf{G}_5$).
The simulator distinguishes whether the current initialization is adversarially started or simulated by SIM for an honest client. In the first case the simulator checks for a record $\langle \text{MALINIT}, \perp, a_1, \mathsf{aid}_{\mathsf{new}}, b_1, n_1, \sigma_1 \rangle$. SIM created such a record if the corrupt server acted towards the HSM as if a client wanted to start an initialization, but there was no corresponding initialization attempt from a client. If the simulator is able to decrypt E it can check if the contained values would be stored in $\mathbf{G}_{19}$ by the HSM. Concretely, if the corrupt server followed the protocol description of an honest client to produce E, it queried H_2, which SIM is able to detect. In that case, the HSM in $\mathbf{G}_{19}$ would store these values as if an honest client initialized. Thus, SIM uses $(\text{MALICIOUSINIT}, \mathsf{sid}, \mathsf{aid}_{\mathsf{new}}, \mathsf{pw}, K)$ to make $\mathcal{F}_{\mathsf{PPKR}}$ store a file for this initialization. Note that SIM uses $\mathsf{aid}_{\mathsf{new}}$ as the client identity towards $\mathcal{F}_{\mathsf{PPKR}}$, since the corrupt server claimed that the key should be stored under this identity. If there is no matching query H_2 however, then SIM just continues this simulation by creating a file record in its internal HSM. This case can happen e.g. when the corrupt server impersonates the client without querying H_2. Instead, the corrupt server can just choose a uniformly random y and store it for later use. However, the probability that the environment later starts a recovery for an honest client with a previously unused password pw, such that $\mathcal{F}_{\mathsf{OPRF}}$ outputs y as PRF value for pw is negligible by the security of the 2HashDH protocol.
It remains the case that E was simulated by SIM for an honest client. Again we distinguish two cases. First, if the whole initialization was performed without interference from the corrupt server, a successful record is stored. SIM gives input COMPLETEINITS to $\mathcal{F}_{\mathsf{PPKR}}$ to ensure that a file is stored that can later be recovered. If the corrupt server sent an adversarial E after acting honestly for the first two messages of the session, the HSM in $\mathbf{G}_{19}$ outputs FAIL due to transcripting mismatching. Thus, in $\mathbf{G}_{20}$ SIM gives COMPLETEINITS-DOS to $\mathcal{F}_{\mathsf{PPKR}}$.

Overall we have
$$\Pr[\mathbf{G}_{20}] = \Pr[\mathbf{G}_{19}].$$

Game $\mathbf{G}_{21}$: Output of parties comes from $\mathcal{F}_{\mathsf{PPKR}}$ in recovery. In this game hop we change the behavior of the simulator as follows. Instead of executing the real client and forwarding messages from the environment to the corrupt server for the recovery phase of the protocol, the simulator acts as in Appendix E.2 on $(\textsc{RecC}\mathsf{sid}, \mathsf{ID_C})$ messages from $\mathcal{F}_{\mathsf{PPKR}}$ and on adversarial messages $b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$ and c and on adversarial allowance of delivering T'_C. Additionally, we change the behavior of the simulator on calls to the HSM interfaces $(\textsc{RecS}, \mathsf{aid}, n^*_C, \bar{\mathsf{pk}}^*_C, a^*_2)$ and $(\textsc{RecResult}, \mathsf{ID_C}, T'_C)$. Instead of internally executing the code of the real HSM, $\textsc{Sim}$ simulates the messages as described in Appendix E.2. However, the simulator still maintains a ticket counter.

- $(\textsc{RecC}, \mathsf{sid}, \mathsf{ID_C})$ from $\mathcal{F}_{\mathsf{HSM}}$ to $\textsc{Sim}$: First, the simulator runs $\textsc{Sim}_{\mathsf{OPRF}}$ to get a message a_2 as in $\mathbf{G}_8$. Note that it retrieves the kid used in the initialization phase. If no kid was recorded for $\mathsf{ID_C}$ then there was no initialization for that client before and the message can be ignored. Then it chooses a nonce n_C at random. It then uses its internal AKE simulator and its internal $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ to produce an ephemeral public key for the key exchange as in $\mathbf{G}_{10}$. Therefore the distribution does not change.
- Adversarial $b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$: First, the simulator checks if there is a record $\langle \textsc{Rec}, \mathsf{ID_C}, [n_C], [\bar{\mathsf{pk}}_C], [a_2], [b], b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2 \rangle$. The existence of such a record indicates that the simulator itself created this message when simulating the HSM. That means it can behave like an honest client on a benign message. The behavior of the client depends on whether the provided password matches or not. Note that in this game the simulator still knows the password pw that $\mathcal{Z}$ provided as input but in later games that won't be the case anymore. Therefore, $\textsc{Sim}$ uses the password bit b that was provided to $\textsc{Sim}$ by the functionality and that $\textsc{Sim}$ stored in the $\textsc{Rec}$ record. If the bit b indicates matching passwords, i.e., $b = 1$, a client in $\mathbf{G}_{20}$ outputs a MAC T'_C. $\textsc{Sim}$ runs its internal AKE simulator to get the MAC key K^{MAC}_C and uses it to compute T'_C, which is the same behaviour as introduced in $\mathbf{G}_{10}$. So the distribution doesn't change in this case.
  If the password does not match, i.e., $b = 0$, a client would output $\textsc{Fail}$. Therefore, $\textsc{Sim}$ gives input $(\textsc{CompleteRecC-DoS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$ to reach the same behavior. This means $\mathbf{G}_{20}$ and $\mathbf{G}_{21}$ deviate here if the passwords do not match but AE' decrypts without an error. In that case the simulation aborts as of $\mathbf{G}_{11}$, so the distribution does not change.
  If no such record exists, the simulator next checks whether a record $\langle \textsc{Rec}, \mathsf{ID_C}, *, *, * \rangle$ exists. If neither of those record exists, this means that $\mathsf{ID_C}$ did not start a recovery and thus does not expect the message $b_2, \ldots, \sigma_2$. Hence, the simulator ignores the message and the distribution does not change.
  If a record $\langle \textsc{Rec}, \mathsf{ID_C}, *, *, * \rangle$ exists, but no record $\langle \textsc{Rec}, \mathsf{ID_C}, [n_C], [\bar{\mathsf{pk}}_C], [a_2], [b], b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2 \rangle$ exists, then either the message $n_C, \bar{\mathsf{pk}}_C, a_2$ or the message $b_2, \ldots, \sigma_2$ was not delivered honestly by the environment. In that case, we either have $\mathsf{pre} \neq \mathsf{pre}'$, or T_S or σ_2 was not output by the simulator. In any of those cases, the client outputs $\textsc{Fail}$as of $\mathbf{G}_2$ or $\mathbf{G}_{13}$. Therefore, in this case the simulator gives input $(\textsc{CompleteRecC-DoS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$.
- Adversarial c: The simulator checks for a record $\langle x, \mathsf{ID_C}, n_C, \bar{\mathsf{pk}}_C, a_2, 1, \mathsf{shk}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2, T'_C, c' \rangle$ with $x = \{\textsc{Rec}, \textsc{MalRec}\}$. If $x = \textsc{MalRec}$, we argue below (on $(\textsc{RecResult}, \mathsf{ID_C}, T'_C)$) that the simulator will not produce c with overwhelming probability. So it does not change the distribution of the game if $\textsc{Sim}$ ignores the message when such a record is found. Next, if the simulator computes a value c on behalf of the HSM but the corrupt server provides a different c' to the client, a real-world client outputs $\textsc{Fail}$ as the decryption fails and thus $\textsc{Sim}$ gives input $(\textsc{CompleteRecC-DoS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$ in that case. If c' decrypts without error, the simulator aborts as of $\mathbf{G}_{15}$ (event E^1_{AE}). Finally, if the received ciphertext is

benign, $\textsc{Sim}$ gives input $(\textsc{CompleteRecC}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$ in order to make $\mathsf{ID_C}$ output a key. Note that $\mathcal{F}_{\mathsf{PPKR}}$ makes $\mathsf{ID_C}$ output a key $K \xleftarrow{\$} \{0,1\}^\lambda$. This key is distributed identically as the real-world output of an honest client. If the password of the initialization matched the password of recovery, the same key is output to the client.

- $(\textsc{RecS}, \mathsf{aid}, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*)$: As the real-world HSM does, the simulator starts by checking for a previously stored record and makes the server output $\textsc{Fail}$ if no such record is found for aid. Then $\textsc{Sim}$ uses its internal $\textsc{Sim}_{\mathsf{OPRF}}$ to simulate the OPRF response b_2 and its internal $\textsc{Sim}_{\mathsf{3DH}}$ and $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ to simulate the key exchange messages. This leads to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ outputting a session key $s = (K_S^{\mathsf{MAC}}, K_C^{\mathsf{MAC}}, \mathsf{shk})$.
  Then the simulator checks if there is a record $\langle \textsc{Rec}, \mathsf{ID_C}^*, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^* \rangle$. This record is produced by $\textsc{Sim}$ when it simulated the first message for an honest client. If no such record is found it means that the corrupt server sent the $\textsc{RecS}$ message to the HSM without receiving a corresponding message from a client. In that case, it stores a record $\langle \textsc{MalRec}, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*, \mathsf{aid}^*, e, \mathsf{shk}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2 \rangle$ that allows to distinguish this interaction later from a completely honest recovery.
  If such a record is found, $\textsc{Sim}$ sends $(\textsc{RecS}, \mathsf{sid}, \mathsf{ID_C}^*, \mathsf{aid}^*)$ to $\mathcal{F}_{\mathsf{PPKR}}$. This makes the functionality answer with a bit indicating whether the passwords in initialization and recovery did match. Note that in this game $\mathbf{G}_{21}$, $\mathcal{F}_{\mathsf{PPKR}}$ always responds, as we did not introduce the counter yet. Also in this game, $\textsc{Sim}$ could still check if the passwords match without $\mathcal{F}_{\mathsf{PPKR}}$ but in later game hops we remove the inputs pw from the simulator and it will only rely on this bit. $\textsc{Sim}$ stores this bit along with the other produced messages for later use. The simulator sends $\mathsf{aid}^*, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$ to the corrupt S as reply from $\mathcal{F}_{\mathsf{HSM}}$, as the real-world HSM would do. Finally, the simulator removes old records to ensure that new recovery sessions can be made.
- $(\textsc{RecResult}, \mathsf{aid}, T_C')$: $\textsc{Sim}$ first checks for $\langle x, n_C, \bar{\mathsf{pk}}_C, a_2, \mathsf{aid}, b, e, \mathsf{shk}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2, T_C' \rangle$ with $x \in \{\textsc{Rec}, \textsc{MalRec}\}$, and optional $\mathsf{ID_C}, b, T_C'$. If such a record exists with $x = \textsc{Rec}$ and T_C' set, it means that the simulator computed T_C' before to simulate an honest client. By construction, the simulator only does this, when the password guess bit was set to $b = 1$, indicating a successful recovery. In $\mathbf{G}_{20}$, if T_C' was output by $\textsc{Sim}$, it outputs the ciphertext c. Thus, here $\textsc{Sim}$ simulates c as an encryption of 0 under shk. This does not change the distribution as it is already done in $\mathbf{G}_{17}$.
  If T_C' was not computed by $\textsc{Sim}$, the above record exists without T_C' set. Hence, in this case $\textsc{Sim}$ gives input $(\textsc{CompleteRecC-DoS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$, as it would in $\mathbf{G}_{14}$.
  If the record exists with $x = \textsc{MalRec}$ and without $\mathsf{ID_C}, T_C'$ and b set, it means that the corrupt server played the role of the client in this whole interaction. In that case, $\textsc{Sim}$ has to check if T_C' is a valid MAC. If it is not valid, $\textsc{Sim}$ outputs $(\textsc{RecResult}, \mathsf{aid}, \textsc{Fail})$, which is the same behavior as in $\mathbf{G}_{20}$. Otherwise, $\textsc{Sim}$ checks whether a record $\langle \mathsf{prog}, \mathsf{aid}, [y'], [K] \rangle$ exists, which indicates that the corrupt server has queried H_2 with the correct password. $\textsc{Sim}$ then uses the prog record to compute c as it would in $\mathbf{G}_{20}$. Even if the corrupt server never queried H_2 with the correct password, it may still be able to compute a valid T_C' by reusing the same y' it used in the corresponding initialization. If that is the case, there is a record $\mathsf{e} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e$ stored in the HSM that is marked $\textsc{Broken}$. Then, $\textsc{Sim}$ encrypts the e from that record under shk, which ensures that the corrupt server recovers the correct key K.
  If the corrupt server never queried H_2 on the password of this recovery attempt and did not reuse the same y from the corresponding initialization, then y is uniformly random and thus K^{auth}

as well. Consequently, the key exchange key sk_C is information-theoretically hidden from the adversary. By the AKE security, this means that the key K_C^{MAC} is uniformly random to the adversary. If that is the case, the MAC T'_C will not verify with overwhelming probability by the sEUF-CMA security of the MAC. With a similar argument one can see that sk_C remains hidden from the environment and thus T'_C will not verify if the adversary only queried H_2 on a wrong password.

Overall we get

$$|\Pr[\mathbf{G}_{21}] - \Pr[\mathbf{G}_{20}]| \leq Q_{\text{REC}}\mathbf{Adv}^{\mathsf{sEUF\text{-}CMA}}_{\mathcal{A},\mathsf{MAC}}(\lambda).$$

Game $\mathbf{G}_{22}$: Introduce counter to $\mathcal{F}$. In this game we introduce the counter $\mathsf{tx}_{\mathsf{sid}}$ to the functionality.

As of this game hop the counter is not only maintained by the simulator (using its internal HSM) anymore but additionally by the functionality. In the real-world a counter for a file record is initially set to 10 and on every computed OPRF exponentiation $(\cdot)^{K_{\mathsf{aid}}^{\mathsf{PRF}}}$ in a retrieval phase the counter is decremented. If the counter is zero, the HSM outputs DELREC. If a recovery attempt is successful, the counter is reset to 10. In the ideal world, the functionality initializes the counter to 10 when it receives a COMPLETEINITS message from the ideal-world adversary, i.e., the simulator. On a RECS message, $\mathcal{F}_{\mathsf{PPKR}}$ checks if the counter reached 0 and outputs DELREC. If the counter is not 0 in a RECS call, the counter is decremented. On a COMPLETERECS message the functionality checks if the passwords match and resets the counter to 10 if they do. Further the functionality also maintains the counter in MALICIOUSREC calls, i.e., it checks if the counter is 0, decrements the counter by one and resets it if the password guess was correct. Similarly, $\mathcal{F}_{\mathsf{PPKR}}$ initializes the counter to 10 on a MALICIOUSINIT message. The main difference is that the functionality keeps a counter per $\mathsf{ID_C}$ (and sid) while the real-world HSM keeps the counters per aid.

We must argue that $\mathcal{F}_{\mathsf{PPKR}}$ never refuses to respond to a MALICIOUSREC or a RECS input because of the counter being 0. First, we note that $\mathcal{F}_{\mathsf{PPKR}}$'s counter and SIM's internal HSM counter are set and reset to 10 under the exact same conditions. The HSM initializes a record with counter 10 when it receives a $(\text{FILE}, \mathsf{aid}_{\mathsf{new}}, \mathsf{E})$ message where the decryption of E contains a matching transcript. In this situation, SIM gives input COMPLETEINITS to $\mathcal{F}_{\mathsf{PPKR}}$. SIM only does this for honest clients where it is clear that transcripts $\mathsf{tr_{HSM}}$ and $\mathsf{tr_C}$ match. But the initialization could also be finished successfully by a corrupt server without involving an honest client. SIM also checks if it made a MALINIT record. Remember that SIM makes these records when there was an initialization started without involving an honest client. If such a record exists and the corrupt server queried H_2 on a corresponding password, SIM gives input MALICIOUSINIT to the functionality, which leads to the counter being initialized as well.

Whenever SIM's internal HSM has a RECS query with counter 0, it outputs DELREC. Now assume that $\mathcal{F}_{\mathsf{PPKR}}$'s counter reaches 0 before the HSM's counter reaches 0. Then there was at least one MALICIOUSREC input or one RECS input to $\mathcal{F}_{\mathsf{PPKR}}$ for which the HSM did not decrease its counter. MALICIOUSREC is an adversarial interface and, as we assume a corrupted server, RECS is also given to $\mathcal{F}_{\mathsf{PPKR}}$ by the simulator.

SIM sends RECS to $\mathcal{F}_{\mathsf{PPKR}}$ when it receives a query $(\text{RECS}, \mathsf{aid}^*, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*)$ to $\mathcal{F}_{\mathsf{HSM}}$ for a client-initiated recovery. The first thing that SIM does when receiving such a query is running its internal HSM on $(\text{RECS}, \mathsf{aid}^*, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*)$, which will decrement the counter of the HSM.

SIM sends MALICIOUSREC in only one case and that is if there is a H_2 query with a corresponding MALREC record. Such a record is created by SIM when the corrupt server started a recovery with the

HSM but without an honest client interaction. Note here that the HSM's counter was decremented already when the HSM received a query $(\textsc{RecS}, \mathsf{aid}^*, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*)$ to $\mathcal{F}_{\mathsf{HSM}}$ for an adversarial recovery. That means the HSM counter got decremented before $\mathcal{F}_{\mathsf{PPKR}}$'s counter. It can even happen that $\mathcal{Z}$ never queries H_2 on the corresponding password. In that case only the HSM's counter gets decreased but not $\mathcal{F}_{\mathsf{PPKR}}$'s counter. Overall, it can never happen that $\mathcal{F}_{\mathsf{PPKR}}$'s counter gets decreased without HSM's counter being decreased. Hence, we have

$$\Pr[\mathbf{G}_{22}] = \Pr[\mathbf{G}_{21}].$$

Game $\mathbf{G}_{23}$: Remove password forwarding. In this game we take away the additional information about input that our simulator still gets from the functionality. More precisely we change IC.3 "Send $(\textsc{InitC}, \mathsf{sid}, \mathsf{ID}_\mathsf{C}, \mathsf{pw})$ to $\mathcal{A}$ and to S" to "Send $(\textsc{InitC}, \mathsf{sid}, \mathsf{ID}_\mathsf{C})$ to $\mathcal{A}$ and to S", and we change "RC.2 Send $(\textsc{RecC}, \mathsf{sid}, \mathsf{ID}_\mathsf{C}, \mathsf{pw}')$ to $\mathcal{A}$ and S" to "RC.2 Send $(\textsc{RecC}, \mathsf{sid}, \mathsf{ID}_\mathsf{C})$ to $\mathcal{A}$ and S". Finally, we take away the dummy interfaces from the functionality that allowed the simulator to make any party output whatever the simulator wanted.

As the simulator did not use the password-inputs anymore in $\mathbf{G}_{17}$, the distribution of the experiment does not change when the simulator does not get the information and we have

$$\Pr[\mathbf{G}_{23}] = \Pr[\mathbf{G}_{22}].$$

We now have $\mathcal{F} = \mathcal{F}_{\mathsf{PPKR}}$ from Figures 7 to 8.

E.1 Simulating (honest ID_C, corrupt S) – initialization phase

Because clients are authenticated, in the (honest ID_C, corrupt S) case there are no adversarial messages from the client towards the server. The simulator receives outputs of $\mathcal{F}_{\mathsf{PPKR}}$ intended towards the server, because the server is corrupt. The adversarial interfaces at $\mathcal{F}_{\mathsf{PPKR}}$ that are available for simulation are: MaliciousInit, MaliciousRec, InitS, CompleteInitC, CompleteInitS, CompleteInitC-DoS, CompleteInitS-DoS, RecS, CompleteRecC, CompleteRecS, CompleteRecC-DoS, CompleteRecS-DoS.

Sim maintains records Init,MalInit for each ID_C containing the state of an ongoing initialization session that has still the potential of terminating. Such session was either started by the honest ID_C with honest a_1 (Sim records this in Init) or by the adversary with adversarial a_1 (Sim uses MalInit). Sim only extends these records with either HSM-generated values, or values generated by the honest ID_C. For example, if b_2 appears in any record, b_2 was output by $\mathcal{F}_{\mathsf{HSM}}$, and if T'_C appears in the record, it was generated by ID_C.

[OPRFsim:] The simulator runs an instance of the OPRF simulator $\textsc{Sim}_{\mathsf{OPRF}}(H_1, H_2, N)$ of Figure 12, where N is the total number of InitC and RecC queries with respect to this key, and an instance of $\mathcal{F}_{\mathsf{OPRF}}$ of Figure 10 that interacts with $\textsc{Sim}_{\mathsf{OPRF}}$, and which never checks evaluation tickets in case of $\textsc{Sim}_{\mathsf{OPRF}}$ evaluating for honest key identifiers. Execute step 1 of $\textsc{Sim}_{\mathsf{OPRF}}$. **[endOPRFsim][AKEsim:]** Similarly, the simulator runs an instance of the AKE simulator $\textsc{Sim}_{\mathsf{3DH}}$ of fig. 16 and an instance of $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ of fig. 15 that interacts with $\textsc{Sim}_{\mathsf{3DH}}$.**[endAKEsim]**

<u>Calls to $\mathcal{F}_{\mathsf{HSM}}$</u> On anybody querying $(\textsc{GetPK})$ to $\mathcal{F}_{\mathsf{HSM}}$:

- If no record $\langle \mathsf{PublicKey}, *, * \rangle$ exists, **[AKEsim:]** send Init to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ from $\mathcal{F}_{\mathsf{HSM}}$, which triggers step 1 of $\textsc{Sim}_{\mathsf{3DH}}$, and obtain $\mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}$.**[endAKEsim]**

Compute $(\mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{Enc}}_{\mathsf{HSM}}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$ and $(\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{Sig}}_{\mathsf{HSM}}) \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$ and set $\mathsf{sk}_{\mathsf{HSM}} = \{\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}, \mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}, \perp, K^{\mathsf{Enc}}_{\mathsf{HSM}}\}$ and $\mathsf{pk}_{\mathsf{HSM}} = \{\mathsf{pk}^{\mathsf{Sig}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{Enc}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}\}$. Store $\langle \mathsf{PublicKey}, \mathsf{pk}_{\mathsf{HSM}}, \mathsf{sk}_{\mathsf{HSM}}\rangle$. Else retrieve $\langle \mathsf{PublicKey}, \mathsf{pk}_{\mathsf{HSM}}, *\rangle$
- Reply with $\mathsf{pk}_{\mathsf{HSM}}$.

On corrupt server querying $(\text{INITS}, \mathsf{aid}^*_{\mathsf{new}}, \mathsf{aid}^*_{\mathsf{old}}, a^*_1)$ to $\mathcal{F}_{\mathsf{HSM}}$:

- Run HSM code on the input $(\text{INITS}, \mathsf{aid}^*_{\mathsf{new}}, \mathsf{aid}^*_{\mathsf{old}}, a^*_1)$. If the HSM outputs $(\text{INITRESULT}, \mathsf{aid}^*_{\mathsf{new}}, \text{FAIL})$ give this to S as output of $\mathcal{F}_{\mathsf{HSM}}$, otherwise discard the output.
  // Just for aid treatment.
- [Initialization started by client] If there exists a record $\langle \text{INIT}, [\mathsf{ID_C}^*], a^*_1\rangle$, **[OPRFsim:]** Retrieve $\langle \mathsf{ID_C}^*, \mathsf{kid}, \perp\rangle$ and replace $\perp$ by $\mathsf{aid}^*_{\mathsf{new}}$. Run $\mathcal{F}_{\mathsf{OPRF}}$ on the input $(\text{INIT}, \mathsf{sid}, \mathsf{kid})$ from $\mathcal{F}_{\mathsf{HSM}}$, which triggers step 2 of $\text{SIM}_{\mathsf{OPRF}}$ and thus results in records $\langle F, \mathcal{F}_{\mathsf{HSM}}, \mathsf{aid}^*_{\mathsf{new}}, K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}, g^{K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}}\rangle$ and $\langle \mathcal{F}_{\mathsf{HSM}}, \mathsf{aid}^*_{\mathsf{new}}\rangle$ for $K^{\mathsf{PRF}}_{\mathsf{aid_{new}}} \xleftarrow{\$} \mathbb{Z}_q$. Compute $b_1 \leftarrow (a^*_1)^{K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}}$ as in step 6 of $\text{SIM}_{\mathsf{OPRF}}$. **[endOPRFsim]**// Equivalent to how $\mathcal{F}_{\mathsf{HSM}}$ would compute b_1. Choose $n_1 \xleftarrow{\$} \{0,1\}^\lambda$ and compute $\sigma_1 \leftarrow \mathsf{Sign}(\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}, b_1 \| n_1)$. Append $\mathsf{aid}^*_{\mathsf{new}}, b_1, n_1, \sigma_1$ to it. Give input $(\text{INITS}, \mathsf{sid}, \mathsf{ID_C}^*, \mathsf{aid}^*_{\mathsf{new}})$ to $\mathcal{F}_{\mathsf{PPKR}}$.
- [Adversarial initialization] If no record with a^*_1 exists so far, **[OPRFsim:]** Run $\mathcal{F}_{\mathsf{OPRF}}$ on the input $(\text{INIT}, \mathsf{sid}, \mathsf{aid}^*)$ from $\mathcal{F}_{\mathsf{HSM}}$, which triggers step 2 of $\text{SIM}_{\mathsf{OPRF}}$ and thus results in records $\langle F, \mathcal{F}_{\mathsf{HSM}}, \mathsf{aid}^*_{\mathsf{new}}, K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}, g^{K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}}\rangle$ and $\langle \mathcal{F}_{\mathsf{HSM}}, \mathsf{aid}^*_{\mathsf{new}}\rangle$ for $K^{\mathsf{PRF}}_{\mathsf{aid_{new}}} \xleftarrow{\$} \mathbb{Z}_q$. Compute $b_1 \leftarrow (a^*_1)^{K^{\mathsf{PRF}}_{\mathsf{aid_{new}}}}$ as in step 6 of $\text{SIM}_{\mathsf{OPRF}}$. **[endOPRFsim]**// Equivalent to how $\mathcal{F}_{\mathsf{HSM}}$ would compute b_1. Choose $n_1 \xleftarrow{\$} \{0,1\}^\lambda$ and compute $\sigma_1 \leftarrow \mathsf{Sign}(\mathsf{sk}^{\mathsf{Sig}}_{\mathsf{HSM}}, b_1 \| n_1)$. Create a new record $\langle \text{MALINIT}, \perp, a^*_1, \mathsf{aid}^*_{\mathsf{new}}, b_1, n_1, \sigma_1\rangle$ // Corrupt server can initialize on behalf of any $\mathsf{aid}^*_{\mathsf{new}}$.
- Create record $\langle \text{REPLAY}, a^*_1, b_1, n_1, \sigma_1\rangle$ and remove all records $\langle x, *, *, \mathsf{aid}^*_{\mathsf{old}}, ...\rangle$ with $x \in \{\text{INIT}, \text{MALINIT}\}$ and $\langle *, *, \mathsf{aid}^*_{\mathsf{old}}\rangle$ that have not been modified or newly created by the above. // HSM discards older init sessions of $\mathsf{aid}^*_{\mathsf{old}}$
- Give $\mathsf{aid}^*_{\mathsf{new}}, b_1, n_1, \sigma_1$ as output of $\mathcal{F}_{\mathsf{HSM}}$ to the corrupt server.

On corrupt server querying $(\text{FILE}, \mathsf{aid_{new}}, \mathsf{E})$ to $\mathcal{F}_{\mathsf{HSM}}$:

- [HSM does not expect E] Look for a record $\langle x, [\mathsf{ID_C}], [a_1], \mathsf{aid_{new}}, [b_1], [n_1], *, [\mathsf{E}']\rangle$ with $x \in \{\text{INIT}, \text{MALINIT}\}$ and optional E'. If no such record exists, ignore the query
- Run HSM code on the input $(\text{FILE}, \mathsf{aid_{new}}, \mathsf{E})$. If the HSM outputs $(\text{INITRESULT}, \mathsf{aid_{new}}, \text{FAIL})$, give this to S as output of $\mathcal{F}_{\mathsf{HSM}}$ and give input $(\text{COMPLETEINITS-DOS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$.
  // Transcript mismatch.
- [Adversary initializes] If a record was found with $x = \text{MALINIT}$, do:
  - Decrypt E with $\mathsf{sk}^{\mathsf{Enc}}_{\mathsf{HSM}}$ to obtain $m \leftarrow \mathsf{e} \| \mathsf{pk}_C \| \mathsf{tr_C} \| \mathsf{e\_cred} \| n_e \| T_e$
  - Find a record $\langle \mathsf{H}_2, \mathsf{pw}, u, \mathsf{aid_{new}}, y\rangle$ which was used to compute E, i.e., for $(K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}}) \leftarrow \mathsf{KDF}_1(y, n_e)$ it holds that (1) $\text{FAIL} \neq K \leftarrow \mathsf{AE.Dec}(K^{\mathsf{export}}; \mathsf{e})$, (2) $\mathsf{e\_cred} \oplus K^{\mathsf{mask}} = \mathsf{sk}_C$ corresponds to pk_C, (3) $T_e = \mathsf{MAC.Tag}(K^{\mathsf{auth}}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}} \| n_e \| \mathsf{e\_cred})$, and (4) $\mathsf{tr_C} \leftarrow \mathsf{H}_3(a_1, b_1, n_1)$. If such a record is found, send $(\text{MALICIOUSINIT}, \mathsf{sid}, \mathsf{aid_{new}}, \mathsf{pw}, K)$ to $\mathcal{F}_{\mathsf{PPKR}}$. // Valid password file!
  - If no such record is found, append $\mathsf{e}, \mathsf{pk}_C, \mathsf{e\_cred}, n_e, T_e$ to record $\langle \text{MALINIT}, \perp, a_1, \mathsf{aid_{new}}, b_1, n_1, \sigma_1\rangle$. Store $m \leftarrow \mathsf{e} \| \mathsf{pk}_C \| \mathsf{e\_cred} \| n_e \| T_e$ in the internal HSM and mark this file as BROKEN.
    // We cannot find a password matching E. This could be due to (1) $\mathcal{Z}$ never queried H_2 and must have used some own y or (2) it modified some part of m. But it could memorize whatever it used and do the same in recovery which we need to be able to simulate.
- [Client initializes] If a record was found with $x = \text{INIT}$, do:

- [E generated by honest client] If a record $\langle \textsc{Replay}, [a_1'], [b_1'], [n_1'], *, \mathsf{E}\rangle$ exists, do:
  - [Honest initialization by $\mathsf{ID_C}$] If $\mathsf{E}' = \mathsf{E}$, give input $(\textsc{CompleteInitS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$
  - [Replay with high probability] If $\mathsf{E}' \neq \mathsf{E}$, do:
    - If $a_1 \neq a_1'$, $b_1 \neq b_1'$, or $n_1 \neq n_1'$, abort the simulation. // event E^1_{Coll}
    - Otherwise, give input $(\textsc{CompleteInitS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$
- [Adversarial E] Give input $(\textsc{MaliciousInit}, \mathsf{sid}, \mathsf{ID_C}, \perp, \perp)$ to $\mathcal{F}_{\mathsf{PPKR}}$. // Tampered E such that key will be unrecoverable

Calls to random oracles // This part is the simulator of [22, Figure 3], working together with an emulated $\mathcal{F}_{\mathsf{OPRF}}$ of Figure 10.

On $\mathcal{A}$ querying $\mathsf{H}_1(x)$:

- If this query happened before, give the same answer.
- **[OPRFsim:]** Otherwise, run step 4 of $\textsc{Sim}_{\mathsf{OPRF}}$. **[endOPRFsim]**

On $\mathcal{A}$ querying $\mathsf{H}_2(x, u)$:

- If this query happened before, give the same answer.
- **[OPRFsim:]** Otherwise, run step 8 of $\textsc{Sim}_{\mathsf{OPRF}}$ and let y denote its output and, if applicable, kid the key identifier corresponding to y **[endOPRFsim]**
- [H_2 call belongs to init session] If there is a record $\langle \textsc{MalInit}, \ldots, \mathsf{aid}, \ldots\rangle$ with $\mathsf{aid} = \mathsf{kid}$, then create record $\langle \mathsf{H}_2, x, u, \mathsf{kid}, y\rangle$ and reply with y.
- [H_2 call belongs to recovery session] If there is a record $\langle \textsc{MalRec}, \ldots, \mathsf{aid}, \ldots\rangle$ with $\mathsf{aid} = \mathsf{kid}$, see Appendix E.2.
- [H_2 call wrt honest key that was not generated with the help of $\mathcal{F}_{\mathsf{HSM}}$] If none of the two cases above apply, if $\mathsf{kid} = \mathsf{aid}$ for any aid that $\mathcal{F}_{\mathsf{HSM}}$ has received an input $(\textsc{InitS}, \mathsf{aid}, *, *)$ for, then $\textsc{Sim}$ aborts.
- [H_2 call with malicious key] Create record $\langle \mathsf{H}_2, x, u, \mathsf{kid}, y\rangle$ and reply with y.

Adversarial messages

On message b_1, n_1, σ_1 towards $\mathsf{ID_C}$:

- [$\mathsf{ID_C}$ does not expect b_1, n_1, σ_1] Look for record $\langle \textsc{Init}, \mathsf{ID_C}, *\rangle$ or $\langle \textsc{Init}, \mathsf{ID_C}, *, *, *, *, *\rangle$. If no such record exists, ignore the query.
- [HSM-generated message] If a record $\langle \textsc{Replay}, *, b_1, n_1, \sigma_1\rangle$ exists, do:
  - Look for record $\langle \textsc{Init}, \mathsf{ID_C}, [a_1], [\mathsf{aid}], b_1, n_1, \sigma_1\rangle$. If no such record exists, look for records $\langle \textsc{Init}, \mathsf{ID_C}, [a_1]\rangle$ and $\langle \mathsf{ID_C}, [\mathsf{kid}], *\rangle$ and set $\mathsf{aid} \leftarrow \mathsf{kid}$. // If message is not replayed, first case will occur. If message is replayed, second case will occur
  - **[OPRFsim:]** Receive y as output from $\mathcal{F}_{\mathsf{OPRF}}$ **[endOPRFsim]**and create record $\langle \textsc{Prog}, \mathsf{aid}, y\rangle$ **[AKEsim:]** Send $\textsc{Init}$ to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ from $\mathsf{ID_C}$, which triggers step 1 of $\textsc{Sim}_{\mathsf{3DH}}$, to obtain pk_C. Run the simulated client using y as output of H_2, $\mathsf{e} \leftarrow 0$, $\mathsf{e\_cred} \xleftarrow{\$} \{0,1\}^\lambda$, and pk_C to produce E. Provide input $(\textsc{CompleteInitC}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$. // Let client output key, independent of replay or non-replay
  - [Replayed message] If no record $\langle \textsc{Init}, \mathsf{ID_C}, a_1, \mathsf{aid}, b_1, n_1, \sigma_1\rangle$ exists, append $\perp, b_1, n_1\sigma_1$ to record $\langle \textsc{Init}, \mathsf{ID_C}, a_1\rangle$. // Internal HSM will later fail because of transcript mismatch.
  - Append E to records $\langle \textsc{Init}, \mathsf{ID_C}, a_1, *, b_1, n_1, \sigma_1\rangle$ and $\langle \textsc{Replay}, *, b_1, n_1, \sigma_1\rangle$, and send $\mathsf{ID_C}, \mathsf{E}$ to $\mathcal{A}$ as message from $\mathsf{ID_C}$ to S
- [Adversarial message] If no such record exists, provide input $(\textsc{CompleteInitC-DoS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$. // event E^1_{Sig}

Messages from $\mathcal{F}_{\mathsf{PPKR}}$ Below we list all messages from $\mathcal{F}_{\mathsf{PPKR}}$ that are not replies to adversarial queries. See other parts of the simulation on how $\textsc{Sim}$ reacts to responses from $\mathcal{F}_{\mathsf{PPKR}}$ to its own queries, such as

$(\textsc{RecS}, \mathsf{sid}, \mathsf{ID_C}, b)$, "wrong/correct guess", "success", etc.

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

- **[OPRFsim:]** Choose fresh ssid, kid and a dummy password pw_D and give input $(\textsc{Eval}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathcal{F}_{\mathsf{HSM}}, \mathsf{pw}_D)$ to $\mathcal{F}_{\mathsf{OPRF}}$ from simulated $\mathsf{ID_C}$. $\mathcal{F}_{\mathsf{OPRF}}$ sends $(\textsc{Eval}, \mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, \mathsf{ID_C}, \mathcal{F}_{\mathsf{HSM}})$ to $\textsc{Sim}_{\mathsf{OPRF}}$, which will execute step 5 and return a message $(\mathsf{sid}, \mathsf{kid}, \mathsf{ssid}, a_1)$. **[endOPRFsim]**
- Create a new record $\langle \textsc{Init}, \mathsf{ID_C}, a_1\rangle$. Also create a record $\langle \mathsf{ID_C}, \mathsf{kid}, \perp\rangle$. Send a_1 to S.

E.2 Simulating (honest $\mathsf{ID_C}$, corrupt S) – recovery phase

The adversarial interfaces at $\mathcal{F}_{\mathsf{PPKR}}$ that are available for simulation are: MaliciousRec, MaliciousInit, CompleteInitC, CompleteInitS, RecS, CompleteInitC-DoS, CompleteInitS-DoS, ChangeUser, CompleteRecC, CompleteRecC-DoS, CompleteRecS, CompleteRecS-DoS.

Sim maintains records Rec,MalRec for each $\mathsf{ID_C}$ containing the state of an ongoing initialization or recovery session that has still the potential of terminating. Such session was either started by the honest $\mathsf{ID_C}$ with honest a_1 (Sim records this in Rec) or by the adversary with adversarial a_1 (Sim uses MalRec). Sim only extends these records with either HSM-generated values, or values generated by the honest $\mathsf{ID_C}$. For example, if b_2 appears in any record, b_2 was output by $\mathcal{F}_{\mathsf{HSM}}$, and if T'_C appears in the record, it was generated by $\mathsf{ID_C}$.

Calls to $\mathcal{F}_{\mathsf{HSM}}$

On anybody querying $(\textsc{GetPK})$ to $\mathcal{F}_{\mathsf{HSM}}$: as in Appendix E.1.

On corrupt server querying $(\textsc{RecS}, \mathsf{aid}^*, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*)$ to $\mathcal{F}_{\mathsf{HSM}}$:

- Run HSM code on input $(\textsc{RecS}, \mathsf{aid}^*, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*)$ to obtain $\mathsf{e}, \mathsf{pk}_C, K_C^{\mathsf{MAC}}, \mathsf{shk}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$, but with the following modifications:
  - Replace the computation of b_2 with the following: **[OPRFsim:]** Compute $b_2 \leftarrow (a_2^*)^{K_{\mathsf{aid}}^{\mathsf{PRF}}}$ as in step 6 of $\textsc{Sim}_{\mathsf{OPRF}}$ **[endOPRFsim]**. Here aid is either kid from record $\langle *, [\mathsf{kid}], \mathsf{aid}^*\rangle$ if the OPRF key was generated upon an honest client initiating, and otherwise $\mathsf{aid} \leftarrow \mathsf{aid}^*$.
  - Replace the computation of $(K_S^{\mathsf{MAC}}, K_C^{\mathsf{MAC}}, \mathsf{shk})$ with the following: Set $\mathsf{aux} \leftarrow \mathsf{pre}$. If a record $\langle [\mathsf{ID_C}], *, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*\rangle$ exists, set $\mathsf{P} \leftarrow \mathsf{ID_C}$, otherwise, set $\mathsf{P} \leftarrow \mathsf{pk}_C$. **[AKEsim:]** Give input $(\textsc{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P})$ as input to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ from $\mathcal{F}_{\mathsf{HSM}}$ for a fresh ssid, which triggers step 2 of $\textsc{Sim}_{\mathsf{3DH}}$. Receive $\bar{\mathsf{pk}}_S, \mathsf{pk}_{\mathsf{HSM}}^{\mathsf{DH}}, \perp$ as response from $\textsc{Sim}_{\mathsf{3DH}}$. Give message $\bar{\mathsf{pk}}_C^*, \mathsf{pk}_C, \mathsf{aux}$ to $\textsc{Sim}_{\mathsf{3DH}}$ as adversarial message on behalf of P to $\mathcal{F}_{\mathsf{HSM}}$, which triggers step 3 of $\textsc{Sim}_{\mathsf{3DH}}$. Receive an output s from $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ that we interpret as $(K_S^{\mathsf{MAC}}, K_C^{\mathsf{MAC}}, \mathsf{shk})$. **[endAKEsim]**

  If the HSM outputs $(\textsc{RecResult}, \mathsf{aid}, \textsc{Fail})$ or $(\textsc{DelRec}, \mathsf{aid}^*)$ give this to S as output of $\mathcal{F}_{\mathsf{HSM}}$.
- [Adversarial recovery] If there exists no record $\langle \textsc{Rec}, *, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*\rangle$, create a record $\langle \textsc{MalRec}, n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*, \mathsf{aid}^*, \mathsf{e}, K_C^{\mathsf{MAC}}, \mathsf{shk}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2\rangle$. // $\mathcal{F}_{\mathsf{PPKR}}$ counter is not decreased here, which is okay since $\mathcal{F}_{\mathsf{HSM}}$ maintains the counter
- [Client initiated recovery] Else if there exists such a record $\langle \textsc{Rec}, [\mathsf{ID_C}^*], n_C^*, \bar{\mathsf{pk}}_C^*, a_2^*\rangle$, give input $(\textsc{RecS}, \mathsf{sid}, \mathsf{ID_C}^*, \mathsf{aid}^*)$ to $\mathcal{F}_{\mathsf{PPKR}}$.
  - [aid* has exceeded his attempts] Else if $\mathcal{F}_{\mathsf{PPKR}}$ sends $(\textsc{DelRec}, \mathsf{sid}, \mathsf{aid}^*)$ to S, send $(\textsc{DelRec}, \mathsf{aid}^*)$ to the corrupt S as reply from $\mathcal{F}_{\mathsf{HSM}}$. // this case never happens, HSM would have already failed.
  - [There is a file for aid*, counter ok] On $\mathcal{F}_{\mathsf{PPKR}}$ replying with $(\textsc{RecS}, \mathsf{sid}, \mathsf{aid}^*, b)$, append $\mathsf{aid}^*, b, \mathsf{e}, K_C^{\mathsf{MAC}}, \mathsf{shk}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$ to record $\langle \textsc{Rec}, \mathsf{ID_C}^*, n_C, \bar{\mathsf{pk}}_C, a_2\rangle$.

- Output $\mathsf{aid}^*, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$ to the corrupt S as reply from $\mathcal{F}_{\mathsf{HSM}}$ and remove all records $\langle x, ..., \mathsf{aid}^*, ...\rangle$ with $x \in \{\text{REC},\text{MALREC}\}$ that have not been modified or newly created by the above. // HSM discards older recovery sessions of aid*

On corrupt server querying $(\text{RECRESULT}, \mathsf{aid}, T'_C)$ to $\mathcal{F}_{\mathsf{HSM}}$:

- Look for a record $\langle x, [\mathsf{ID_C}], [n_C], [\bar{\mathsf{pk}}_C], [a_2], \mathsf{aid}, [b], [\mathsf{e}], [K_C^{\mathsf{MAC}}], [\mathsf{shk}], [b_2], [\mathsf{e\_cred}], [n_e], [T_e], [n_S], [\bar{\mathsf{pk}}_S], [T_S], [\sigma_2], T'_C\rangle$ with $x \in \{\text{REC}, \text{MALREC}\}$, and optional $\mathsf{ID_C}, b, T'_C$.
  - [HSM does not expect T'_C] If no such record exists, ignore the query
  - [Server acts honestly, matching passwords] If this record exists with $x = \text{REC}$ and T'_C set, SIM computes $c \leftarrow \mathsf{AE.Enc}(\mathsf{shk}; 0)$, appends c to the record and gives aid, c to S as output from $\mathcal{F}_{\mathsf{HSM}}$.
  - [$\mathsf{ID_C}$ started recovery but T'_C adversarial] If this record exists with $x = \text{REC}$ but without T'_C set, give $(\text{COMPLETERECS-DOS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$
  - [Adversarial recovery without client involvement] If this record exists with $x = \text{MALREC}$, and without $\mathsf{ID_C}, T'_C$ and b, do:
    * If $T'_C \neq \mathsf{MAC.Tag}(K_C^{\mathsf{MAC}}, \mathsf{H}_3(\mathsf{pre} \parallel T_S))$, output $(\text{RECRESULT}, \mathsf{aid}, \text{FAIL})$ to S as output from $\mathcal{F}_{\mathsf{HSM}}$.
    * Otherwise, look for a record $\langle \text{PROG}, \mathsf{aid}, [y'], [K]\rangle$. If it exists, set $(K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}}) \leftarrow \mathsf{KDF}_1(y', n_e)$. $\mathsf{e} \leftarrow \mathsf{AE.Enc}(K^{\mathsf{export}}; K)$, $c \leftarrow \mathsf{AE.Enc}(\mathsf{shk}; \mathsf{e})$, append T'_C, c to the MALREC record and send aid, c to S as output of $\mathcal{F}_{\mathsf{HSM}}$.
      // Make sure $\mathcal{A}$ recovers the key chosen by $\mathcal{F}_{\mathsf{PPKR}}$ for aid.
    * If the record $\langle \text{PROG}, \mathsf{aid}, y', K\rangle$ does not exist, then check if the HSM's record $\mathsf{e} \parallel \mathsf{pk}_C \parallel \mathsf{e\_cred} \parallel n_e \parallel T_e$ is marked as BROKEN. If this is the case, set $c \leftarrow \mathsf{AE.Enc}(\mathsf{shk}, \mathsf{e})$, append T'_C, c to the MALREC record, and send aid, c to S. Otherwise, abort the simulation.

Calls to random oracles

On $\mathcal{A}$ querying $\mathsf{H}_1(x)$: As in Appendix E.1.

On $\mathcal{A}$ querying $\mathsf{H}_2(x, u)$: As in Appendix E.1, filled with the following, denoting with kid, y the output of $\text{SIM}_{\mathsf{OPRF}}$.

- [H_2 call belongs to recovery session] If there is a record $\langle \text{MALREC}, n_C, \bar{\mathsf{pk}}_C, a_2^*, \mathsf{aid}, \mathsf{e}, *, \mathsf{shk}, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2\rangle$ with $\mathsf{aid} = \mathsf{kid}$, send $(\text{MALICIOUSREC}, \mathsf{sid}, \mathsf{aid}, x)$ to $\mathcal{F}_{\mathsf{PPKR}}$.
  - On $\mathcal{F}_{\mathsf{PPKR}}$'s reply "wrong guess", create record $\langle \mathsf{H}_2, x, u, \mathsf{kid}, y\rangle$ and reply to $\mathcal{A}$ with y.
  - On $\mathcal{F}_{\mathsf{PPKR}}$'s reply ("correct guess", K), retrieve record $\langle \text{PROG}, \mathsf{aid}, y'\rangle$, append K to it and reply to $\mathcal{A}$ with y'.. Give input $(\text{COMPROMISE}, \mathsf{sid}, \mathsf{aid})$ to $\text{SIM}_{\mathsf{3DH}}$ and receive sk_C as answer. Then program $\mathsf{KDF}_2(y', n_e) \leftarrow K_1 \parallel K_2 \parallel K_3$ where $K_2 = \mathsf{sk}_C \oplus \mathsf{e\_cred}$ and $K_1, K_2 \leftarrow \{0,1\}^\lambda$.// leaks AKE sk to env.

Adversarial messages

On message $b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2$ towards $\mathsf{ID_C}$:

- [Server acts honestly] If a record $\langle \text{REC}, \mathsf{ID_C}, [n_C], [\bar{\mathsf{pk}}_C], [a_2], [b], *, *, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2\rangle$ with $x \in \{\text{REC}, \text{MALREC}\}$ exists, do: // All values in the record are simulated, either for client or HSM
  - [Passwords match] If $b = 1$,
    * **[AKEsim:]** Set $\mathsf{aux} \leftarrow \mathsf{pre}$. Give $\bar{\mathsf{pk}}_S, \mathsf{pk}_{\mathsf{HSM}}^{\mathsf{DH}}, \mathsf{aux}$ as adversarial message to $\text{SIM}_{\mathsf{3DH}}$ from $\mathsf{ID_C}$ to $\mathcal{F}_{\mathsf{HSM}}$, which triggers step 3 of $\text{SIM}_{\mathsf{3DH}}$. Receive s as output from $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$. **[endAKEsim]**

    * Interpret s as $(K_S^{\mathsf{MAC}}, K_C^{\mathsf{MAC}}, \mathsf{shk})$ and compute the tag $T'_C \leftarrow \mathsf{MAC.Tag}(K_C^{\mathsf{MAC}}, \mathsf{H}_3(\mathsf{pre}, T'_S))$. Append T'_C to record $\langle\textsc{Rec}, \mathsf{ID_C}, n_C, \bar{\mathsf{pk}}_C, a_2, b, *, *, b_2, \mathsf{e\_cred}, n_e, T_e, n_S, \bar{\mathsf{pk}}_S, T_S, \sigma_2\rangle$ and send $(\mathsf{ID_C}, T'_C)$ to S.
  - [Passwords do not match] Otherwise, the client uses a mismatching password ($b = 0$), and SIM provides input $(\textsc{CompleteRecC-DoS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$.
- [Adversarial message] If no such record exists, but a record $\langle\textsc{Rec}, \mathsf{ID_C}, *, *, *\rangle$ exists, give input $(\textsc{CompleteRecC-DoS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$.
- [$\mathsf{ID_C}$ does not expect this message] Otherwise, ignore this message.

On message c towards $\mathsf{ID_C}$:

- Look for record $\langle x, \mathsf{ID_C}, *, *, *, [b], *, *, *, *, *, *, *, *, *, *, *, c'\rangle$ with $x = \{\textsc{Rec}, \textsc{MalRec}\}$ and optional $\mathsf{ID_C}, b$.
  - [$\mathsf{ID_C}$ is not expecting c.] If this record exists with $x = \textsc{MalRec}$, or it does not exist at all, SIM ignores the query.
  - [$\mathsf{ID_C}$ started recovery, but c' adversarial] If this record exists with $x = \textsc{Rec}$ and $c \neq c'$, give input $(\textsc{CompleteRecC-DoS}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$.// (event E^1_{AE})
  - [$\mathsf{ID_C}$ started recovery, S acts honestly] If this record exists with $x = \textsc{Rec}$ and $c = c'$, send $(\textsc{CompleteRecC}, \mathsf{sid}, \mathsf{ID_C})$ to $\mathcal{F}_{\mathsf{PPKR}}$.

Messages from $\mathcal{F}_{\mathsf{PPKR}}$ Below we list all messages from $\mathcal{F}_{\mathsf{PPKR}}$ that are not replies to adversarial queries. See other parts of the simulation on how SIM reacts to responses from $\mathcal{F}_{\mathsf{PPKR}}$ to its own queries, such as $(\textsc{RecS}, \mathsf{sid}, \mathsf{ID_C}, b)$, "wrong/correct guess", "success", etc.

On message $(\textsc{RecC}, \mathsf{sid}, \mathsf{ID_C})$ from $\mathcal{F}_{\mathsf{PPKR}}$:

- **[OPRFsim:]** Retrieve $\langle\mathsf{ID_C}, \mathsf{kid}, *\rangle$. If no such record exists, ignores this message. Choose a fresh ssid and a dummy password pw_D and give input $(\textsc{Eval}, \mathsf{kid}, \mathsf{ssid}, \mathcal{F}_{\mathsf{HSM}}, \mathsf{pw}_D)$ to $\mathcal{F}_{\mathsf{OPRF}}$ from simulated $\mathsf{ID_C}$. $\mathcal{F}_{\mathsf{OPRF}}$ sends $(\textsc{Eval}, \mathsf{kid}, \mathsf{ssid}, \mathsf{ID_C}, \mathcal{F}_{\mathsf{HSM}})$ to $\textsc{Sim}_{\mathsf{OPRF}}$, which will execute step 5 and return a message $(\mathsf{kid}, \mathsf{ssid}, a_1)$. **[endOPRFsim]**
- **[AKEsim:]** Give $(\textsc{NewSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}})$ as input to $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ from $\mathsf{ID_C}$, which triggers step 2 of $\textsc{Sim}_{\mathsf{3DH}}$. Receive $\bar{\mathsf{pk}}_C, \mathsf{pk}_C, \perp$ as response from $\textsc{Sim}_{\mathsf{3DH}}$. **[endAKEsim]**
- $n_C \leftarrow \{0,1\}^\lambda$.
- Record $\langle\textsc{Rec}, \mathsf{ID_C}, n_C, \bar{\mathsf{pk}}_C, a_2\rangle$ and send $\mathsf{ID_C}, n_C, \bar{\mathsf{pk}}_C, a_2$ to S

F On unrestricted adaptive client corruptions

We introduce the restriction on client corruptions in Theorem 1 due to the following reason. Imagine that the environment instructs some honest client $\mathsf{ID_C}$ to first initialize with some password pw and afterwards instructs $\mathsf{ID_C}$ to start a recovery phase, again using pw. Then, after the server produced its final message c, but before the environment allows c to be delivered to $\mathsf{ID_C}$, it instructs the adversary to corrupt $\mathsf{ID_C}$. It now learns the entire state of $\mathsf{ID_C}$, and in particular the keys K^{export} and shk that $\mathsf{ID_C}$ would use to obtain K. Thus, it can now check whether the ciphertext c was produced according to the protocol description, namely by computing $\mathsf{e} \leftarrow \mathsf{AE.Dec}(\mathsf{shk}; c)$, $K \leftarrow \mathsf{AE.Dec}(K^{\mathsf{export}}; \mathsf{e})$, and checking whether $K \overset{?}{=} K'$, where K' is the key that was output by $\mathsf{ID_C}$ in the initialization phase.

However, in the ideal world, the simulator could not produce c according to the protocol description, as it knows neither shk, K^{export}, nor K. Instead it can only output a simulated ciphertext $c' \leftarrow \mathsf{AE.Enc}(\mathsf{shk}', 0)$ with uniformly random shk'. Therefore, upon corrupting $\mathsf{ID_C}$ the environment would be able to efficiently distinguish between the real and ideal world. In order to avoid this, we disallow the corruption of clients during ongoing initialization or recovery sessions, which prevents the environment from learning shk and checking whether c was produced according to the protocol description since $\mathsf{ID_C}$ deletes shk before outputting K at the end of the recovery phase.

We believe that even with the added restriction, this still provides a reasonably realistic modeling of corruptions. In practice we expect a full intialization or recovery session to take only a few seconds and thus we expect it to be very difficult for an adversary to corrupt a client in this short timeframe. Note that an adversary could theoretically extend this "corruption timeframe" by e.g. recording c and dropping the message from the network. However, we assume that in practice all protocol participants implement some timeout mechanism, which terminates the session if no response arrived within a short timeframe, as is standard in any networked application.

An alternative solution to the problem outlined above could be to require the authenticated encryption scheme AE to be *equivocable*. Then, the simulator can output an equivocal ciphertext c and, upon learning e due to the corruption of $\mathsf{ID_C}$, compute shk such that $\mathsf{e} = \mathsf{AE.Dec}(\mathsf{shk}; c)$ holds. Note that this task is still made more difficult for the simulator by the fact that shk also has to be consistent with all prior messages the simulator produced in this recovery session, such as the Diffie–Hellman shares $\bar{\mathsf{pk}}_C$ and $\bar{\mathsf{pk}}_S$. As described by Jarecki *et al.*[24], it is easy to show that GCM, which is deployed in the WBP, is equivocable in the ideal cipher model.

G On not Using any Proven OPAQUE Guarantees

The WBP relies on the strong asymmetric password-authenticated key exchange (saPAKE) protocol OPAQUE [24], which comes with a security analysis in the UC framework. This immediately raises the question whether one could modularize the analysis and leverage the UC composition theorem to obtain our main result. The approach would be as follows:

1. Prove that OPAQUE UC-emulates functionality $\mathcal{F}_{\mathsf{saPAKE}}$ (proven already in [24]).
2. Prove that the WBP using $\mathcal{F}_{\mathsf{saPAKE}}$ instead of OPAQUE UC-emulates $\mathcal{F}_{\mathsf{PPKR}}$ (presumably a simpler proof than proving security using OPAQUE).
3. Invoke the UC composition theorem: it yields that from 1. and 2. above it directly follows that WBP using OPAQUE UC-emulates $\mathcal{F}_{\mathsf{PPKR}}$.

However, this is not the road that we were able to take in this paper. Instead, we had to prove the statement in item 3 above from scratch because of the following reason: The OPAQUE version that is proven secure in [24, Fig. 8], differs from Internet Draft v03 [26] that the WBP is using.

In order to modularly use an saPAKE functionality, we would need to formally prove which exact functionality Internet Draft v03 [26] (or, more specific, the OPAQUE protocol as implemented in Figures 4 and 5) UC-emulates. This however seems overkill, because a) the list of differences is quite extensive, and b) we do not even rely on the "full" OPAQUE security: we only rely on OPAQUE being secure against a malicious client, because the OPAQUE server code is run on an incorruptible HSM.

For completeness, we list below the differences between the proven OPAQUE protocol and the one deployed in WBP.

- OPAQUE from Internet Draft v03 [26] does not separate hash domains of the different 2HashDH PRFs with domain separators. [24] only analyzes the security of OPAQUE where the hash domains are separate on a per-PRF-key basis.

- OPAQUE from Internet Draft v03 [26] has a long-term public key pair of the server (`server_private_key`, `server_public_key`), which is used for the generation of every password file. The paper version of OPAQUE [24, Fig. 8] lets the server choose a fresh key pair (p_S, P_S) for every password file.
- OPAQUE from Internet Draft v03 [26] has an interactive registration phase where the server does not learn the password. The paper version of OPAQUE [24, Fig. 8] has a registration phase where the server gets the cleartext password as input. This difference was already pointed out in [6], where it is claimed that (1) interactive registration only adds to the security proven in [24], and (2) an upcoming paper analyzes the interactive phase. .
- OPAQUE from Internet Draft v03 [26] puts the client secret key sk_C in the password file in form of $\mathsf{e\_cred} \leftarrow \mathsf{sk}_C \oplus K^{\mathsf{mask}}$. In the paper version of OPAQUE [24, Fig. 8], a password file contains password-encrypted credentials $\mathsf{AuthEnc}_{rw}(p_u, P_u, P_S)$, where rw is the PRF value for pw. A user can retrieve their key pair p_u, P_u from the file by decrypting it with rw. Draft v09 [6] motivates this change by (1) smaller password files and (2) no need for applications to provide AKE key material to the client, but instead deal with key generation within OPAQUE. (2) is actually only a difference between v09 and older versions of the draft, because in [24] OPAQUE (more specific: the server) is generating the client's AKE key pairs. Draft v09 [6] also says that this change is analyzed in an upcoming paper.
- OPAQUE from Internet Draft v03 [26] lets the client output an additional export key `export_key` that is not present in [24].
- OPAQUE from Internet Draft v03 [26] uses 3DH as AKE while [24] only shows that HMQV can be used.
- OPAQUE from Internet Draft v03 [26] uses a superset of transcripts used in [24].

H Comparison to the OPAQUE Internet Draft Notation

In Table 1 we list naming differences between our description of the WBP and the notation used in the OPAQUE draft. In the following we summarize where our protocol description in the Figures 4 and 5 differs from the OPAQUE draft [26], and justify these changes.

- In the OPAQUE draft the server sends its public key in its first message in both phases (see Sections 3.3.2 and 4.1.2.2 in [26]). This public key is already hardcoded into the WhatsApp client and thus is not sent by the HSM.

- In the OPAQUE draft $K^{\mathsf{export}}, K^{\mathsf{mask}}, K^{\mathsf{auth}}$ are derived via a memory-hard function and an HKDF (see Section 3.3.3 in [26]). These steps are simplified into one computation via KDF_1 in Figures 4 and 5. It is well-known that both HKDF.Extract and HKDF.Expand are PRFs [27]. We therefore can treat KDF_1 as a PRF.

- According to the OPAQUE draft, pre contains either the identities of the client and the server, or their respective public keys (see Section 6.2 in [26]). In the WBP, pre contains only the public key of the HSM and neither the client's identity nor its public key. However, pre contains e_cred, which is an encryption of sk_C. Since T_e ensures the integrity and authenticty of e_cred, e_cred uniquely determines the client's public key, which means that it still is implicitly contained in pre.

- In the OPAQUE draft $K_S^{\mathsf{MAC}}, K_C^{\mathsf{MAC}}$, shk are derived via a series of HKDF computations (see Section

4.2.2.2 in [26]). These steps are simplified into one computation via KDF_2 in Figure 5. We can treat KDF_2 as a PRF if the $\mathsf{HMAC}(0, \$)$-$\$$ assumption [18] holds.

Table 1: Names and variables as they are referred to in our protocol description and in the OPAQUE Internet Draft [26].

Our notation	**OPAQUE Internet Draft notation**
S	`Server`
$\mathsf{ID_C}$	`Client`
K^{export}	`export_key`
r_1 and r_2	`blind`
a_1 and a_2	`request` and `M`
$(\mathsf{pk}_C, \mathsf{sk}_C)$	`creds`
$(\mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}, \mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}})$	`(server_private_key, server_public_key)`
$\mathsf{sk}^{\mathsf{DH}}_{\mathsf{HSM}}$	`secret_creds`
$\mathsf{pk}^{\mathsf{DH}}_{\mathsf{HSM}}$	`cleartext_creds`
pw	`password`
$K^{\mathsf{PRF}}_{\mathsf{aid}}$	`oprf_key`
n_e	`envelope_nonce`
K^{mask}	`pseudorandom_pad`
K^{auth}	`auth_key`
$\mathsf{e\_cred}$	`encrypted_creds`
T_e	`auth_tag`
$(n_e, \mathsf{e\_cred}, T_e)$	`envelope`
$(K^{\mathsf{PRF}}_{\mathsf{aid}}, \mathsf{pk}_C, (n_e, \mathsf{e\_cred}, T_e))$	`credential_file`
T'_e	`expected_tag`
n_C	`client_nonce`
$\bar{\mathsf{pk}}_C$	`client_keyshare` and `epkU`
$\bar{\mathsf{pk}}_S$	`server_keyshare` and `epkS`
$\bar{\mathsf{sk}}_C$	`eskU`
$\bar{\mathsf{sk}}_S$	`eskS`
n_S	`server_nonce`
T_S or T'_C	`mac`
K^{MAC}_S	`server_mac_key`
K^{MAC}_C	`client_mac_key`
ikm	`IKM`
shk	`session_key`