Towards a Formally Secure and Practical PAKE Protocol

Ali Utkan Sahin

School of Computer and Communication Sciences

Master Thesis

August 2024

Responsible	Supervisor	Supervisor
Prof. Serge Vaudenay	Sebastian Faller	Julia Hesse
EPFL / LASEC	IBM Research	IBM Research

Acknowledgements

I consider myself lucky to be supervised by Sebastian Faller, who has helped me tremendously at every step of this project. I thank Julia Hesse whose expert feedback and suggestions has been invaluable. I would also like to thank Serge Vaudenay for giving me the opportunity to work under his supervision. Without them, this thesis would not be possible.

I thank my mom, my dad, and my dear friends who have supported me throughout this arduous process. My special thanks to Ekin Gurgen, Quentin Bordier and Can Kirimca for their feedbacks on the non-technical parts of the write-up.

Abstract

How to achieve strong security guarantees while providing a simple interface for the end-user is a key question while designing user authentication solutions. A draft for an "asymmetric" password-authenticated key exchange (PAKE) protocol is currently being worked on by the IRTF working group, which aims to answer exactly this question. In PAKE, the users authenticate themselves only by a low-entropy, easy-to-remember password. The main advantage of this protocol over the predominantly deployed password-over-TLS method is that the passwords never leave the device of the users, as implied by the term "asymmetric". However, the draft introduces several key differences compared to the construction that it is based on – OPAQUE. Although OPAQUE itself is proven to be secure, it is unclear whether the draft's additions present a vulnerability. In this paper, we bridge this gap by proposing a protocol that accommodates the most noteworthy changes introduced in the IRTF draft, and we prove its security in the Universal Composability framework. By doing this, we aim to contribute towards a formally secure and practical PAKE solution.

Contents

Chapter 1

Introduction

Authentication, i.e., verification of identity, is a key problem in security. The predominant authentication solutions requires the parties to maintain high-entropy keys [33]. However, this is infeasible in the context of *user* authentication, since most users do not have the means to remember or securely store such keys. In such cases, password-based authentication methods are suggested [35]. Although there are pushes towards more secure ways of user authentication like biometrics [34] or physical tokens [10], these require specialized hardware, and even then, they are usually deployed in conjunction with passwords. Due to their simplicity, passwords still remain and will likely remain to be the de-facto way of user authentication.

The simplicity of passwords comes at a cost. Since they are low-entropy and far from uniformly random, they are easier to guess [32]. Moreover, users tend to reuse passwords [11] which results in a password compromise to affect multiple services. Therefore, constructing password-based user authentication solutions that minimize such risks is of notable concern.

In this paper, we consider the setting where potentially multiple users wish to authenticate themselves with a server. Common implementations use the *password-over-TLS* method, where the password is sent to the server through a secure TLS channel. In this method, the server maintains a long-term database of the users' hashed passwords, usually along with a salt to avoid precomputation attacks in case of a server compromise [26]. Password-authenticated key exchange (PAKE) is an alternative method for realizing password-based authentication, in which two parties agree on a common key only if the user supplies the correct password [35]. In response to a demand for alternative user authentication methods, the recent efforts by the IRTF committee has lead to a draft which receives frequent updates [7].

The IRTF draft [7] is based on OPAQUE [26], a PAKE protocol that ensures the secrecy of the passwords while maintaining the security properties achieved by the aforementioned password-over-TLS method. An advantage of OPAQUE over password-over-TLS is that the user password is never transmitted to the server, hence the server never sees the password in clear-text form. This is thanks to OPAQUE's "asymmetric" property in which the server and the client take different inputs. As a result, establishing a secure channel (to transfer the passwords) is not required, avoiding the pitfalls of PKI [26].

OPAQUE itself is shown to be secure [26], but the IRTF draft [7] has significant differences. Notably, the original protocol utilizes globally unique "subsession identifiers" and is only for a single user that is known to be registered at the server, both of which cannot be assumed in a practical PAKE solution. To our knowledge, there is no formal proof for the current IRTF draft in the literature. In this paper, we seek to bridge the gap between the theory and practice by proving the security of a protocol (which we call augmented OPAQUE) that accommodates these differences in the current version of the draft. We provide our security proof in the Universal Composability (UC) framework, which makes our security results hold both in conjunction with other protocols and multiple instances of itself [8]. This is especially important in the context of PAKEs, since their utility comes from their usage in conjunction with other protocols.

Outline. We first give the preliminaries in Section 2. Then, we introduce the relevant cryptographic constructions in Section 3. We define our ideal functionality that captures our security requirements in Section 4. After that, we present the augmented OPAQUE in Section 5, which we prove to be secure in Section 6. Finally, we provide our conclusion in Section 7.

1.1 Our Results

Although the differences between the IRTF draft and OPAQUE are numerous, we identify and tackle some of them that we deem the most important. In particular, we focus on the nonces, multi-user servers, lack of global state, and user enumeration mitigation, which are introduced only in the draft.

First, we introduce the IRTF draft in Section 3.5 and analyze the notable differences of the draft compared to OPAQUE in Section 3.5.3. Since our proof is in the Universal Composability framework, we aim to model our security requirements by defining an *ideal functionality* machine that acts as a trusted-third party whose programming captures the expected behavior of a protocol. In our preliminaries we give an introduction to UC in Section 2.1. To that end, we augment the ideal functionality given in [26], so that (1) multiple users can initiate subsessions with a single server, and (2) the global subsession identifiers *ssid* are removed from the inputs. We let our ideal functionality accommodate for multiple users by adding the user id *uid* (which might correspond to the unique usernames in a service) to inputs and changing the global variables such that they can be indexed by multiple user ids. Moreover, we handle the case where the associated file record does not exist. We are able to remove *ssid* from the inputs by letting our ideal functionality output an identifier which uniquely identifies the particular server or user initialization. Our ideal functionality is explained in more detail in Section 4.

Then, we augment OPAQUE and introduce the *augmented OPAQUE* in Section 5 which realizes the ideal functionality that we introduced. We construct this protocol by removing *ssid* from the party inputs, introducing nonces, allowing the server to keep track of multiple users, and implementing user enumeration mitigation. We replace *ssid* in the original protocol with the pair of nonces introduced in the draft, which can uniquely identify a subsession except with negligible probability. We implement user enumeration mitigation by letting the server maintain an auxiliary password file that is populated on the fly in response to an authentication request for an unknown user. The produced response leads to an authentication failure except with negligible probability.

The main contribution of this paper is in Section 6, where we show that augmented OPAQUE realizes the ideal functionality that we introduced. This is done in the UC framework, which ensures that our results that are achieved in isolation hold even if multiple instances of the protocol are executed, or when augmented OPAQUE is used in combination with other protocols [8]. The approach we took in our proof is as follows: In presence of an active environment machine, we construct a simulator that interacts with our ideal functionality (i.e., the ideal world) such that it simulates any adversary that interacts with the protocol (i.e., the real world). We show that the environment machine cannot distinguish whether it is interacting with the real world or the ideal world. Our simulator fully simulates the client and server inputs, which ensures input secrecy.

1.2 Related Works

We note the initiative to formally prove the security of the IRTF draft [13] as part of the Whatsapp backup protocol, but the recent updates to the draft have made additional modifications (e.g., addition of nonces) that remain to be considered. Moreover, [13] is in a different setting than ours, i.e., key-retrieval security, for which OPAQUE is used. Therefore, their results do not directly translate into our setting. However, some of the approaches they took during their security proof of 3DH (a component of OPAQUE) are used in our proof in Section 6.

One of the goals of this paper is to remove the global subsession identifiers both from the ideal functionality and from the protocol itself. In [3], the authors propose an ideal functionality that takes only password as input. Notably, they utilize a method to remove global identifiers from the ideal functionality. In particular, the requirement to maintain a global identifier as an input to the ideal functionality interfaces is removed by letting the unique identifiers be an output of the ideal functionality instead. In this paper, we use a similar approach while building our ideal functionality in Section 4. However, [3] only considers symmetric PAKE, whereas we are concerned with asymmetric PAKEs.

In [14], the authors introduce a multi-user ideal functionality by introducing user ids, similar to our approach in Section 4. However, their construction employs globally unique subsession identifiers [14], which we aim to avoid in this paper. Moreover, they show a vulnerability in the current IRTF draft when the server uses the same OPRF key across different users (which is allowed by the draft) [14]. The OPRF keys are chosen fresh per user in the augmented OPAQUE protocol introduced in this paper, hence it remains unaffected by this finding.

Chapter 2

Preliminaries

In this section, we introduce the preliminary concepts along with the relevant notation that we use. In the remaining of this paper, we use the following standard notation:

- ▷ $x \leftarrow\$ \mathcal{S}$: Uniformly randomly sample x from some set $\mathcal{S}$.
- ▷ $x := \mathsf{p}$: Assign the output of a deterministic process p to x.
- ▷ $x \leftarrow \mathsf{p}$: Assign the output of a probabilistic process p to x.

2.1 Universal Composability

In this section, we give a brief introduction to the Universal Composability framework [8].

2.1.1 Motivation

While analyzing cryptographic protocols, we wish to achieve security guarantees that hold in as many situations as possible. In particular, a protocol that is proven to be secure in isolation should remain secure if it is executed multiple times sequentially or in a parallel manner. Moreover, it should remain secure if it is used as part of another protocol. *Universal Composability* (UC) is a framework that gives us the tools to realize such general proofs [8]. This is achieved by extending simulation proofs with an *active environment machine* that captures the possible adversarial interactions of a protocol with itself or with other protocols.

2.1.2 Interactive Turing Machines

An interactive Turing machine (ITM) is a Turing machine [36] that has the means to communicate with other ITMs. This is achieved by introducing "tapes" that ITMs can write to and read from. An ITM can either be used as a subroutine of another ITM, or it can be a participant in a protocol. As described in [8], an ITM μ has the following tapes:

1. **Identity tape:** This is a read-only tape that contains the *code* of the program and the identity of μ. By code, we mean a description of the algorithm the ITM executes. In classical Turing machine model, this is the state transition function and the initial tape contents.
2. **Outgoing message tape:** This is a writable tape to which μ writes the outgoing messages that it produces. Along with each message, μ must write a delivery information (e.g., the identity of the target ITM).
3. **Input tape:** This is a read-only (i.e., μ cannot write) and read-once (i.e., μ can only read the next bit at once) tape to which external ITMs can write to.
4. **Subroutine-output tape:** This is a read-only and read-once tape that contains the output of the subroutine ITMs invoked by μ.

5. **Backdoor tape:** This is a read-only and read-once tape that is used for communication with the adversary. Only the adversary can write to this tape.
6. **Activation tape:** This is a one-bit tape that contains '1' if and only if the ITM is currently executing.

We give the following ITM instructions from [8] that are defined in addition to conventional Turing machine instructions [36]:

1. **External write:** Once μ invokes this, the message written on the *outgoing message* tape is written to the destination ITM's *input* tape.
2. **Next message:** This is invoked along with the name of a tape which can be either *input*, *subroutine-output*, or *backdoor*. Once invoked by μ, the reading-head of the specified tape jumps to the next message.

Notation for ITMs

In our paper, we describe the algorithms executed by ITMs (i.e., the content of the *identity* tape) as blocks of pseudocode. Each block of pseudocode starts with a description denoting at which condition it is executed, e.g., **On** m **from P** means that the block of pseudocode that follows it is executed once m is received from a machine with identifier P. We call these descriptions *interfaces*.

Messages. A message is denoted as a tuple of arbitrary length with optionally a type, surrounded by parentheses. For example, $(\textsc{Request}, a, b)$ is a message of type $\textsc{Request}$ with two fields: a, b.

Records. In addition to variables, we let ITMs maintain state in the form of *records* which are tuples of arbitrary length with optionally a type, surrounded by angled brackets. For example, $\langle\textsc{Session}, a, b\rangle$ is a record of type $\textsc{Session}$, and it contains two fields: a, b.

Shorthand notations. We employ several shorthand notations while working with the fields of messages and records. We give basic examples of their usage and explain what they mean:

▷ $\langle\textsc{Session}, a, *, c\rangle$ means a $\textsc{Session}$ record such that its first field is equal to a, and its third field is equal to c.

▷ $\langle\textsc{Session}, a, \ldots\rangle$ means a $\textsc{Session}$ record such that its first field is equal to a.

▷ $\langle\textsc{Session}, a, \{\mathbf{b}\}\rangle$ means a $\textsc{Session}$ record such that its first field is equal to a, and we let b denote its second field from now on.

2.1.3 Ideal Functionalities

An *ideal functionality*, denoted as $\mathcal{F}$ is an ITM that represents a specification, i.e., it captures the behavior and security properties expected of a protocol Π. The security of Π can be demonstrated by showing that any execution by a set of parties executing Π can be simulated by a simulator that invokes the appropriate interfaces in $\mathcal{F}$. An execution with an ideal functionality is referred as the *ideal world*, whereas an execution with Π is referred as the *real world*.

Hybrid model. A protocol may be shown to be secure in some $(\mathcal{F}_1, \ldots, \mathcal{F}_n)$-hybrid model. By this, we mean that our proof assumes the presence of ITMs $\mathcal{F}_1, \ldots, \mathcal{F}_n$. For example, a proof in the random oracle model (see Section 2.4) would be a proof in $\mathcal{F}_{\mathsf{RO}}$-hybrid model where $\mathcal{F}_{\mathsf{RO}}$ is the random oracle ideal functionality (see Section 2.4.1).

2.1.4 Execution Model

In this section, we explain how ITMs interact in a protocol execution. Let Π be an n-party protocol that is to be executed by parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ where P_i is an ITM. We define the adversary $\mathcal{A}$ as an ITM that can write to the backdoor tape of P_i. Finally, we define the environment ε as an ITM that can communicate with $\mathcal{A}$ freely.

Steps of execution. The execution of a protocol is initiated with the environment, which can communicate with the adversary at any point during the execution of the protocol. The parties run the algorithms as described by Π, possibly sending each other messages. In the end, the environment outputs a bit denoting whether it is in the real world or in the ideal world. Note that the environment acts as an active distinguisher. Output of the environment ε in such an execution is represented by the random variable $\mathsf{Exec}^{\Pi}_{\mathcal{A},\varepsilon}$.

Powers of the adversary. The adversary machine can communicate with ε and any party during the execution of the protocol. To send a message to P_i, $\mathcal{A}$ writes to P_i's *backdoor* tape. On a message $(\textsc{Corrupt}, \mathsf{P}_i)$ from ε, we allow the adversary to take complete control of P_i throughout the remaining of the execution. In an *adaptive corruption* model, ε can send $\textsc{Corrupt}$ messages at any point. In a *static corruption* model, ε can send $\textsc{Corrupt}$ messages only before the start of the execution.

Modeling insecure asynchronous networks. To capture insecure asynchronous networks, we let $\mathcal{A}$ act as a *router* for the messages, i.e., if P_i wishes to send a message m to P_j, P_i sends (m, P_j) to $\mathcal{A}$. The adversary has full control over what it can do with this request. For example, the adversary may (1) deliver the message to P_j at some point in the future, (2) alter the message, (3) deliver the message to a different party, (4) drop the message altogether. An ideal functionality of this communication model is given in [8].

2.1.5 Proving Security

As we briefly discussed in Section 2.1.3, proving the security of a protocol comes down to showing that there exists a probabilistic polynomial time (PPT) simulator that can simulate an execution of the protocol with the ideal functionality. Recall from Section 2.1.4 that $\mathsf{Exec}^{\Pi}_{\mathcal{A},\varepsilon}$ is a random variable representing the output of the environment ε with protocol Π and adversary $\mathcal{A}$. We give the formal definition of *UC-emulation* from [8]:

Definition 1 (UC-emulation)**.** *Protocol* Π_1 *UC-emulates protocol* Π_2 *if*

$$\forall \text{PPT } \mathcal{A} : \exists \text{PPT } \mathcal{S} : \forall \text{PPT } \varepsilon : \mathit{Exec}^{\Pi_1}_{\mathcal{A},\varepsilon} \approx \mathit{Exec}^{\Pi_2}_{\mathcal{S},\varepsilon}$$

where $\approx$ *denotes indistinguishability.*

To be able to utilize ideal functionalities, we must define what it means for a protocol to adhere to a specification captured by an ideal functionality. This is explained through *UC-realization* [8]:

Definition 2 (UC-realization)**.** *Protocol* Π *UC-realizes ideal functionality* $\mathcal{F}$ *if* Π *UC-emulates the ideal protocol* $\mathsf{Ideal}_{\mathcal{F}}$*.*

Note that we define the ideal protocol $\mathsf{Ideal}_{\mathcal{F}}$ as a protocol with the ITM $\mathcal{F}$ and with m *dummy parties* whose identities correspond to the parties in Π where m is the number of parties in Π. A dummy party simply forwards messages to and from $\mathcal{F}$ [8]. Introducing these are necessary, because the environment trivially distinguishes an execution if the number or the identity of parties are different in the ideal world. Then, proving the security (with respect to $\mathcal{F}$) of Π is showing that Π UC-realizes $\mathcal{F}$. From Definition 2 and Definition 1, we say that protocol Π UC-realizes protocol $\mathcal{F}$ if

$$\forall \text{PPT } \mathcal{A} : \exists \text{PPT } \mathcal{S} : \forall \text{PPT } \varepsilon : \mathsf{Exec}^{\Pi}_{\mathcal{A},\varepsilon} \approx \mathsf{Exec}^{\mathsf{Ideal}_{\mathcal{F}}}_{\mathcal{S},\varepsilon} \tag{2.1}$$

Dummy adversaries. In order to satisfy Equation 2.1, a simulator needs to be constructed for every adversary. However, in [8] it is shown that, without loss of generality, we can only consider the *dummy adversary*. A dummy adversary, denoted as $\mathcal{A}_\perp$ is an ITM that forwards the messages to and from the environment and does nothing else. Intuitively, we can do this because quantifying over all environments means that the environments encapsulate all possible adversarial actions. In turn, the simulator needs to only consider environment as the "black-box challenger". Then, Equation 2.1 is equivalent to the following:

$$\exists \text{PPT } \mathcal{S} : \forall \text{PPT } \varepsilon : \mathsf{Exec}^{\Pi}_{\mathcal{A}_\perp,\varepsilon} \approx \mathsf{Exec}^{\mathsf{Ideal}_{\mathcal{F}}}_{\mathcal{S},\varepsilon} \tag{2.2}$$

2.1.6 Composition Theorem

We give the *universal composition* operation from [8] that lets us denote protocols where a subroutine is substituted with another one:

Definition 3 (Universal composition operation). *Let ρ, ϕ, π be protocols such that*

1. *ϕ is a subroutine protocol of ρ,*
2. *π and ϕ are compatible,*
3. *No machine in π has the same identity as any machine in $\rho \backslash \phi$.*

The composed protocol, denoted $\rho^{\phi\to\pi}$, is identical to ρ, except that the subroutine protocol ϕ is replaced by protocol π.

Notice that the protocols need to be *compatible* (as claimed in definition), which means that their interfaces must be identical.

Finally, we give the universal composition theorem and a corollary from [8] which captures the power of the UC framework:

Theorem 1 (Universal composition in restricted model). *Let ρ, ϕ, π be protocols such that ϕ is a subroutine of ρ, π UC-emulates ϕ, and π is identity-compatible with ρ and ϕ. Then protocol $\rho^{\phi\to\pi}$ UC-emulates ρ.*

Corollary 1. *If π UC-realizes an ideal functionality $\mathcal{F}$, and ρ uses $\mathcal{F}$ as a subroutine, then $\rho^{\mathcal{F}\to\pi}$ UC-emulates ρ.*

This means that if it is proven that a protocol Π UC-realizes some $\mathcal{F}$, it can be freely composed with other protocols that are proven to be secure in $\mathcal{F}$-hybrid model. A protocol that is proven to be secure in isolation remains secure in combination with other protocols.

2.2 Negligibility

Computational security arguments take the real-world feasibility of attacks into consideration. However, the computational power available to us is not constant – it constantly grows. We define *negligible functions* that asymptotically decrease "very quickly" with the security parameter. Then, in our proofs, we show that the probability of undesirable outcomes belongs to this class of functions. This allows us to increase the security parameter and still maintain the security in response to increased computational power.

We give the definition of a negligible function directly from [27]:

Definition 4 (Negligible function)**.** *A function f is negligible if for every polynomial $p(\cdot)$ there exists an N such that for all integers $n > N$ we have $f(n) < \frac{1}{p(n)}$*

Throughout this paper, we use the notation $\mathsf{negl}(\lambda)$ to denote a negligible function in λ where λ is a security parameter.

2.3 Hash Functions

Cryptographic hash functions are used to map an arbitrary length input string $\{0,1\}^*$ to a fixed length output $\{0,1\}^l$ for some l [27]. Throughout this paper, we use hash functions extensively.

Collision resistance. This is a property of hash functions which asserts that given a hash function H, no PPT (i.e., probabilistic polynomial time) adversary can find pairs of inputs $x \neq x'$ such that $\mathsf{H}(x) = \mathsf{H}(x')$ except with negligible probability [27].

Pre-image resistance. This is a weaker property of hash functions which asserts that given $y = \mathsf{H}(x)$ for some unknown x, no PPT adversary can find x' such that $\mathsf{H}(x') = y$ except with negligible probability [27]. Hash functions that are pre-image resistant can be considered as *one-way functions* [27]. Notice that collision resistance implies pre-image resistance: An adversary that can "invert" $\mathsf{H}(x)$ and get x' such that $\mathsf{H}(x) = \mathsf{H}(x')$ breaks collision resistance (i.e., we have $x \neq x'$) with high probability, since the domain of H is infinite [27].

Notation for Hash Functions

Throughout this paper, we denote hash functions as H_i for some string i. Unless noted otherwise, H_i takes an input from $\{0,1\}^*$. For simplicity, we assume that the inputs provided to H_i are automatically converted to their binary representation and padded appropriately. If multiple inputs are given to H_i, those inputs' padded binary representations are concatenated. Moreover, protocols might utilize multiple hash functions with the same range. In these cases, we implicitly utilize *domain separation*: The instances are separated by a domain separation identity, which corresponds to i in H_i in our notation. In an implementation with a hash function H invocations of $\mathsf{H}_i(x)$ (that have the same range as H) are equivalent to $\mathsf{H}(x \parallel i)$.

2.4 Random Oracle Model

The proof that we provide in Section 6 is in *the random oracle model* (ROM) [4], in which we assume that there exists a public dictionary of values with keys from $\{0,1\}^*$ and values from some set $\mathcal{R}$ where each value is initialized to a uniformly randomly chosen value from $\mathcal{R}$ when it is first queried.

Note that a practical implementation of a random oracle does not exist [27], yet it is an immensely useful tool that allows us to prove the security of a protocol in an idealized setting. In a real instantiation of the protocol, random oracle invocations are replaced with an appropriate hash function which simulates a random oracle except with negligible probability [27].

Programmability. In our proof in Section 6, we make use of a *programmable* ROM model. By programmable, we mean a ROM for which the outputs for some inputs can be programmed by the adversary before their invocation [23]. In [23], it is shown that this assumption is a necessity for proving the security of asymmetric password authenticated key exchange protocols in the UC framework.

```
▷ On ({x}):
  1: If $\mathsf{H}[x] = \bot$, then do:
    1.1: Choose $h \leftarrow\!\$\ \mathcal{R}$.
    1.2: Set $\mathsf{H}[x] := h$.
  2: Send $\mathsf{H}[x]$ to $\mathsf{P}$.
```

Figure 2.1: $\mathcal{F}_{\mathsf{RO}}$ ideal functionality

2.4.1 Ideal Functionality

In Figure 2.1, we give $\mathcal{F}_{\mathsf{RO}}$ which captures the random oracle functionality. Our functionality maintains a dictionary H with keys from $\{0,1\}^*$ and values from $\mathcal{R}$. In the definition, $\mathsf{H}[x] = \bot$ means that the value associated with x is uninitialized.

2.4.2 Joint State Theorem

Proofs in hybrid models (see Section 2.1.3) pose a problem once we consider parallel and sequential composition of protocols. In particular, even though the ideal functionalities can be invoked from different instances of protocols, the state that the ideal functionalities maintain must be independent for each independent instance [9]. To deal with this in our formal analysis, we extend our UC-model with *multi-session extensions* such that a multi-session extension of an ideal functionality $\mathcal{F}$ (i.e., $\hat{\mathcal{F}}$) is defined as an ITM that maintains multiple instances of $\mathcal{F}$ internally and can forward the messages to the correct instance [9]. We give the joint-state theorem from [9]:

Theorem 2 (UC with joint-state)**.** *Let $\mathcal{F}$ be an ideal functionality. Let Π_1 be a protocol in the $\mathcal{F}$-hybrid model, and let $\hat{\Pi}_2$ be a protocol that securely realizes $\hat{\mathcal{F}}$, the multi-session extension of $\mathcal{F}$, in the real-life model. Then the composed protocol $\Pi_1^{[\hat{\Pi}_2]}$ in the real-life model emulates protocol Π_1 in the $\mathcal{F}$-hybrid model.*

2.5 Cryptographic Assumptions

The security of modern cryptographic protocols depends on the computational infeasibility of some fundamental problems. Here we give the cryptographic assumptions that are relevant to this paper. It is important to note that the problems listed here are *believed* to be hard, i.e., there exists no known algorithm (with classical computers) that can solve them in reasonable time.

2.5.1 Diffie-Hellman Class of Problems

We discuss the Diffie-Hellman class of problems, which are essential to the constructions that we analyze in this paper. In the remaining of this section, we work in a cyclic group $\mathbb{G} = \langle g \rangle$ with prime order p.

Computational Diffie-Hellman problem. Let a, b be uniformly random values from $\mathbb{Z}_p$. The CDH problem is to compute g^{ab} while only knowing g^a and g^b [27]. For certain groups, this is believed to be infeasible [27]. In other words, there is no PPT adversary $\mathcal{A}$ that can win $\mathrm{CDH}^{\mathcal{A}}_{(\mathbb{G},g,p)}$ (shown in Figure 2.2) except with negligible probability.

```
$\mathrm{CDH}^{\mathcal{A}}_{(\mathbb{G},g,p)}(\lambda)$
1 : $a, b \leftarrow\$ \mathbb{Z}_p$
2 : $z \leftarrow \mathcal{A}(1^\lambda, g^a, g^b)$
3 : $\textbf{return } z = g^{ab}$
```

```
$\mathrm{GapDH}^{\mathcal{A}}_{(\mathbb{G},g,p)}(\lambda)$
1 : $a, b \leftarrow\$ \mathbb{Z}_p$
2 : $z \leftarrow \mathcal{A}^{\mathsf{DDH}(\cdot,\cdot,\cdot)}(1^\lambda, g^a, g^b)$
3 : $\textbf{return } z = g^{ab}$
```

```
$\mathrm{DDH}^{\mathcal{A}}_{(\mathbb{G},g,p)}(\lambda)$
1 : $a, b, c \leftarrow\$ \mathbb{Z}_p$
2 : $z_0 := g^{ab}; z_1 := g^c$
3 : $b \leftarrow\$ \{0, 1\}$
4 : $b' \leftarrow \mathcal{A}(1^\lambda, g^a, g^b, z_b)$
5 : $\textbf{return } b = b'$
```

Figure 2.2: From left to right: (1) CDH game, (2) GapDH game, (3) DDH game

Decisional Diffie-Hellman problem. Let a, b, c be uniformly random values from $\mathbb{Z}_p$. The DDH problem is to distinguish between g^c and g^{ab} [27]. For certain groups, this is believed to be infeasible [27]. In other words, there is no PPT adversary $\mathcal{A}$ that can win $\mathrm{DDH}^{\mathcal{A}}_{(\mathbb{G},g,p)}$ (shown in Figure 2.2) except with negligible probability. Notice that if CDH is *easy* (i.e., an adversary exists with non-negligible advantage) in some group, then DDH is easy in that group as well, but the converse is not true [27].

Gap Diffie-Hellman problem. There exist some groups in which DDH is easy but CDH is hard. Such groups are useful in the construction of some protocols [5,31]. Let a, b be uniformly random values from $\mathbb{Z}_p$. The GapDH problem is to compute g^{ab} while only knowing g^a and g^b and having access to a decisional Diffie-Hellman oracle [5]. The associated game is given in Figure 2.2, denoted as GapDH. Note that as opposed to the CDH game, we equip $\mathcal{A}$ with an oracle DDH that solves the decisional Diffie-Hellman problem, i.e., $\mathsf{DDH}(\cdot,\cdot,\cdot)$ such that $\mathsf{DDH}(g^a, g^b, g^c)$ returns 1 if and only if $g^{ab} = g^c$.

The advantage of an adversary $\mathcal{A}$ is given as

$$\mathsf{Adv}^{\mathrm{GapDH}}_{\mathcal{A},(\mathbb{G},g,p)}(\lambda) = \Pr\Big[\mathrm{GapDH}^{\mathcal{A}}_{(\mathbb{G},g,p)}(\lambda) = 1\Big].$$

Then, we say that GapDH is hard in $G = \langle g \rangle$ if

$$\forall \mathrm{PPT}\ \mathcal{A} : \mathsf{Adv}^{\mathrm{GapDH}}_{\mathcal{A},(\mathbb{G},g,p)}(\lambda) \leq \mathsf{negl}(\lambda).$$

One-More Diffie-Hellman problem. Informally, in a group that admits OMDH, it is infeasible to come up with g^k without knowing k, even if some exponentiations g_i^k such that $g_i \neq g$ are known [25]. We show the associated security game in Figure 2.3. Note that we equip $\mathcal{A}$ with a limited exponentiation oracle Exp in addition to a decisional Diffie-Hellman oracle DDH. An invocation of the limited exponentiation oracle $\mathsf{Exp}^k_Q(g)$ returns g^k up until Q unique queries. After that, it always returns $\perp$.

The advantage of an adversary $\mathcal{A}$ is given as

$$\mathsf{Adv}^{\mathrm{OMDH(N,\ Q)}}_{\mathcal{A},(\mathbb{G},g,p)}(\lambda) = \Pr\Big[\mathrm{OMDH(N,\ Q)}^{\mathcal{A}}_{(\mathbb{G},g,p)}(\lambda) = 1\Big].$$

Then, we say that OMDH(N, Q) is hard in $\mathbb{G} = \langle g \rangle$ of order p if

$$\forall \mathrm{PPT}\ \mathcal{A} : \mathsf{Adv}^{\mathrm{OMDH(N,\ Q)}}_{\mathcal{A},(\mathbb{G},g,p)}(\lambda) \leq \mathsf{negl}(\lambda).$$

```
$\mathrm{OMDH(N,\ Q)}^{\mathcal{A}}_{(\mathbb{G},g,p)}(\lambda)$
1 :  $k \leftarrow\!\$\ \mathbb{Z}_p$
2 :  $g_1, \dots, g_N \leftarrow\!\$\ \mathbb{G}$
3 :  $((g_1, e_1), \dots, (g_{Q+1}, e_{Q+1})) \leftarrow \mathcal{A}^{\mathsf{Exp}^k_Q(\cdot),\mathsf{DDH}(\cdot,\cdot,\cdot)}(1^\lambda, g^k, \{g_i\}_{i\in[N]})$
4 :  return $\wedge_{i\in[Q+1]}\ e'_i = {g'_i}^k$
```

Figure 2.3: OMDH(N, Q) game

```
$\mathrm{Exp\text{-}PRF}^{\mathcal{A}}_F(\lambda)$
1 :  $k \leftarrow\!\$\ \mathcal{K}; f_0 := F_k$
2 :  $f_1 \leftarrow\!\$\ \mathcal{F}$
3 :  $b \leftarrow\!\$\ \{0,1\}$
4 :  $\mathsf{Eval} := \mathsf{Eval}_{f_b}$
5 :  $b' \leftarrow \mathcal{A}^{\mathsf{Eval}(\cdot)}(1^\lambda)$
6 :  return $b' = b$
```

Figure 2.4: Exp-PRF game

2.6 Pseudorandom Functions

Given two sets $\mathcal{D}$ and $\mathcal{R}$, let $\mathcal{F}$ denote all the functions from $\mathcal{D}$ to $\mathcal{R}$. A pseudorandom function (PRF) family is a special set of functions $F \subseteq \mathcal{F}$ indexed by a key from $\mathcal{K}$. For a uniformly random key $k \in \mathcal{K}$, it must be infeasible to distinguish between $F_k(\cdot)$ and a uniformly random function chosen from $\mathcal{F}$ [27]. The game that captures this notion is given in Figure 2.4, denoted as Exp-PRF. Note that we equip the adversary with a function evaluation oracle Eval_f that returns $f(x)$ on a query x. Advantage of a distinguisher $\mathcal{A}$ is defined as

$$\mathsf{Adv}^{\text{Exp-PRF}}_{\mathcal{A},F}(\lambda) = \left|\frac{1}{2} - \Pr\big[\text{Exp-PRF}^{\mathcal{A}}_F = 1\big]\right|.$$

We say that F is a pseudorandom function family if and only if

$$\forall \text{PPT } \mathcal{A} : \mathsf{Adv}^{\text{Exp-PRF}}_{\mathcal{A},F}(\lambda) \leq \mathsf{negl}(\lambda).$$

2.7 Oblivious Pseudorandom Functions

Oblivious pseudorandom functions (OPRFs) are two-party protocols that allow a client to receive the evaluation of a PRF from the server without revealing its input. There are multiple definitions of OPRF, with varying degree of information leak allowed [19]. Roughly speaking, the security requirement is that the server doesn't learn the client's input, and the client doesn't learn the PRF key.

In [26], the authors use the definition captured by the ideal functionality $\mathcal{F}_{\mathsf{OPRF}}$ which is described in Section 2.7.1. We are specifically concerned with the 2HashDH protocol [25] which realizes $\mathcal{F}_{\mathsf{OPRF}}$ under some assumptions [25], as explained in Section 3.3.

2.7.1 Ideal Functionality

In Figure 2.5, we present the functionality $\mathcal{F}_{\mathsf{OPRF}}$ from [26] with only syntactic changes. In the figure, l denotes the OPRF output length.

Usage. With two honest parties U and S, $\mathcal{F}_{\mathsf{OPRF}}$ might be used as follows:

1. The server S first sends $(\textsc{Init}, sid)$ to initialize the session.
2. To request an evaluation from the server, the client U sends $(\textsc{Eval}, sid, ssid, \mathsf{S}, x)$ where x is its input.
3. Once the server sees the request, it sends $(\textsc{SndComplete}, sid, ssid)$ which represents sending back an evaluation.
4. Once the adversary $\mathcal{A}$ sends $(\textsc{RcvComplete}, sid, ssid, \mathsf{U}, \mathsf{S})$, the evaluation $F_{sid,\mathsf{S}}(x)$ is revealed to U.

PRF as random oracle. The functionality maintains a table $F_{sid,*}$ for every sid. For an identifier i and an input x, the value of $F_{sid,i}(x)$ is initialized to a uniformly random value from $\{0,1\}^l$ when it is referenced the first time. The subsequent references to $F_{sid,i}(x)$ return the same value. Notice that for any i, $F_{sid,i}$ is exactly a random oracle (see $\mathcal{F}_{\mathsf{RO}}$ in Figure 2.1).

OPRF prefix. Notice that the ideal functionality asks for a *prefix* information from the adversary. This allows us to precisely pinpoint at which point $\mathcal{A}$ can mount an active attack, so that it can request an evaluation of its choice in that subsession. We give the description directly from [26]:

Description 1 (OPRF prefix)**.** *If some subsession of server S shares a protocol prefix with some subsession of user U, then the only party which can compute function $F_k(\cdot)$ on some input x due to this interaction is that U's subsession, and not e.g. the adversary.*

Transaction counter. The ideal functionality maintains a transaction counter tx, which is incremented by one every time the server gives out an evaluation to $\mathcal{A}$, and it is decremented by one every time $\mathcal{A}$ receives an evaluation. Note that the counter quantifies the information leakage to $\mathcal{A}$. This observation is used in the security proof of 2HashDH in [25].

Offline evaluation and adaptive compromise. The ideal functionality $\mathcal{F}_{\mathsf{OPRF}}$ is equipped with a $\textsc{Compromise}$ interface. This captures the compromise of a server, e.g., compromise of an OPRF key, which results in an adversary being able to locally run evaluations. If the server is compromised, the $\textsc{OfflineEval}$ interface allows an adversary to run evaluations outside of a subsession context, i.e., locally. On the other hand, the server is allowed to use the $\textsc{OfflineEval}$ interface without any restrictions.

2.8 Encryption

Encryption is a cryptographic tool that provides *data confidentiality*, i.e., the data security property which asserts that only the intended recipient of a message should be able to read it. An encryption scheme is defined as $\mathsf{ENC} := (\mathsf{Enc}, \mathsf{Dec})$ over a message, key and ciphertext space, where Enc is the PPT encryption algorithm, and Dec is the *deterministic* polynomial time decryption algorithm [6]. Given a message space $\mathcal{M}$ and a key space $\mathcal{K}$, the correctness of a symmetric encryption scheme is given as follows [6]:

$$\forall m \in \mathcal{M} : \Pr_{k \in \mathcal{K}}[\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m] = 1.$$

Note that it is possible to define key generation as a separate algorithm [27], but in this paper we only consider the encryption schemes in which the key generation is uniform sampling from $\mathcal{K}$.

```
▷ On (INIT, {sid}) from S:
  1: If this is the first such message, then do the following:
    1.1: Associate $sid$ with the machine S.
    1.2: Set $\mathsf{tx}[sid] := 0$.
    1.3: Send (INIT, $sid$, S) to $\mathcal{A}$.

▷ On (COMPROMISE, {sid}) from $\mathcal{A}$:
  1: Get the associated server S.
  2: Mark the server S as Compromised.

▷ On (OFFLINEEVAL, {sid}, {i}, {x}) from P or $\mathcal{A}$:
  1: Get the associated server S.
  2: If S is corrupted, or ($\mathsf{P} = \mathsf{S}$ and $i = \mathsf{S}$), or ($\mathsf{P} = \mathcal{A}$ and $i \neq \mathsf{S}$), or ($\mathsf{P} = \mathcal{A}$ and S is
     marked as Compromised), then send (OFFLINEOUT, $sid$, $F_{sid,i}(x)$) to P.

▷ On (EVAL, {sid}, {ssid}, {S'}, {x}) from P or $\mathcal{A}$:
  1: Send (EVAL, $sid$, $ssid$, P, S') to $\mathcal{A}$, and let prfx be the response.
  2: If prfx was used before, then ignore the message.
  3: Else, do the following:
    3.1: Record ⟨SESSION, $ssid$, P, $x$, Await(prfx)⟩.
    3.2: Send (PREFIX, $sid$, $ssid$, prfx) to P.

▷ On (SNDCOMPLETE, {sid}, {ssid}) from the associated server S:
  1: Send (SNDCOMPLETE, $sid$, $ssid$, S) to $\mathcal{A}$, and let prfx' be the response.
  2: Send (PREFIX, $sid$, $ssid$, prfx') to S.
  3: If there is a record ⟨SESSION, $ssid$, {P}, $x$, {st}⟩ where $\mathsf{P} \neq \mathcal{A}$ and st = Await(prfx'),
     then update st := Ok.
  4: Else, update $\mathsf{tx}[sid] = \mathsf{tx}[sid] + 1$.

▷ On (RCVCOMPLETE, {sid}, {ssid}, {P}, {i}) from $\mathcal{A}$:
  1: Retrieve ⟨SESSION, $ssid$, P, $x$, {st}⟩, or ignore the message.
  2: If $i = \mathsf{S}$ and $\mathsf{tx}[sid] = 0$ and st ≠ Ok, then ignore the message.
  3: Send (OUT, $sid$, $ssid$, $F_{sid,i}(x)$) to P.
  4: If $i = \mathsf{S}$ and st ≠ Ok, then update $\mathsf{tx}[sid] = \mathsf{tx}[sid] - 1$.
```

Figure 2.5: $\mathcal{F}_{\mathsf{OPRF}}$ ideal functionality [26]

Indistinguishability. Indistinguishability is a key security notion for encryption schemes. Informally, it asserts that an adversary should not be able to distinguish between the encryptions of two different messages [27].

Equivocability. Equivocability (EQV) is another security notion related to encryption schemes. In an *equivocable* encryption scheme, for some ciphertext c and plaintext m, there exists an efficiently computable k such that $\mathsf{Dec}_k(c) = m$. Such encryption schemes exist in the random-oracle model [26].

As in [26], we introduce an equivocability simulator $\mathsf{SIM}_{\mathrm{EQV}}$ that allows us to choose a ciphertext first and find a key that decrypts it to some known plaintext later. As opposed to [26], we let $\mathsf{SIM}_{\mathrm{EQV}}$ be stateless by letting it output an internal state st after its first invocation. The second invocation (that *equivocates* a key) takes this state as input.

1. On $|m|$, $\mathsf{SIM}_{\mathrm{EQV}}$ returns a ciphertext c and a state st. Notice that c can contain no information about m apart from its length.
2. On (m, st) where st is the state returned from an invocation of $|m|$ along with c, $\mathsf{SIM}_{\mathrm{EQV}}$ returns a key k such that $\mathsf{Dec}_k(m) = c$.

In an equivocable encryption scheme, we must have $\mathsf{SIM}_{\mathrm{EQV}}$ such that it is infeasible to distinguish its outputs from the *usual* case, i.e., choosing a key first and encrypting it later. The game that captures this notion is given in Figure 2.6, denoted as Exp-EQV. Advantage of an adversary $\mathcal{A}$ is defined as

$$\mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{A},\mathsf{ENC}}(\lambda) = \left|\frac{1}{2} - \Pr\left[\text{Exp-EQV}^{\mathcal{A}}_{\mathsf{ENC}}(\lambda) = 1\right]\right|.$$

Then, we say that ENC is equivocable if and only if

$$\exists \text{PPT } \mathsf{SIM}_{\mathrm{EQV}} : \forall \text{PPT } \mathcal{A} : \mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{A},\mathsf{ENC}}(\lambda) \leq \mathsf{negl}(\lambda). \tag{2.3}$$

Note that our definition Equation 2.3 is different from the one in [26], where the quantifiers are reversed, i.e., for all adversaries, a simulator must exist. However, this poses an issue for the UC framework, where we construct a simulator for all possible environments. Hence, we augment the definition to make it in accordance with the UC framework.

2.9 Message Authentication Codes

Message authentication code (MAC) schemes are cryptographic tools that provide authenticity and integrity. A MAC scheme is defined as $\mathsf{MAC} := (\mathsf{Tag}, \mathsf{Vrfy})$ over a message, key and tag space, where Tag is a PPT tag generation algorithm and Vrfy is a PPT verification algorithm that outputs a bit denoting the validity of the given tag [6]. A party can generate a tag $t \leftarrow \mathsf{Tag}_k(m)$ and distribute it, which can later be verified $\mathsf{Vrfy}_k(t, m) = 1$ to ensure that neither the tag nor the message was altered. Given a message space $\mathcal{M}$ and a key space $\mathcal{K}$, the correctness of a MAC scheme is given as follows [6]:

$$\forall m \in \mathcal{M} : \Pr_{k \in \mathcal{K}}[\mathsf{Vrfy}_k(\mathsf{Tag}_k(m), m) = 1] = 1.$$

Note that, as in encryption, it is possible to define key generation as a separate algorithm [27], but in this paper we only consider the MAC schemes in which the key generation is uniform sampling from $\mathcal{K}$.

Unforgeability. Unforgeability is a key security notion related to MACs, which states that a MAC scheme is secure if an adversary cannot *forge* (i.e., come up with) a tag on a message that correctly verifies without knowing the symmetric key [27].

```
$\text{Exp-EQV}^{\mathcal{A}}_{\mathsf{ENC}}(\lambda)$
1 : $(m, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^{\lambda})$
2 : $b \leftarrow\!\!\$ \{0,1\}$
3 : $k_0 \leftarrow \mathcal{K}$
4 : $c_0 \leftarrow \mathsf{Enc}_{k_0}(m)$
5 : $(c_1, st_{\mathsf{Sim}}) \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(|m|)$
6 : $k_1 \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m, st_{\mathsf{Sim}})$
7 : $b' \leftarrow \mathcal{A}(1^{\lambda}, st_{\mathcal{A}}, (c_b, k_b))$
8 : return $b' = b$
```

```
$\text{Exp-RKR}^{\mathcal{A}}_{\mathsf{AE}}(\lambda)$
1 : $k_1 \leftarrow\!\!\$ \mathcal{K}$
2 : $k_2 \leftarrow\!\!\$ \mathcal{K}$
3 : $c \leftarrow \mathcal{A}(1^{\lambda}, (k_1, k_2))$
4 : $p_1 := \mathsf{AuthDec}_{k_1}(c)$
5 : $p_2 := \mathsf{AuthDec}_{k_2}(c)$
6 : return $p_1 \neq \perp$ and $p_2 \neq \perp$
```

Figure 2.6: From left to right: (1) Exp-EQV game, (2) Exp-RKR game.

Construction. A MAC that satisfies unforgeability under *chosen-message attacks* (i.e., adversary has access to an oracle that outputs tags for messages of its choice) can be trivially constructed from a PRF [27]. Given a PRF F, we define:

$$\mathsf{Tag}_k(m) = F_k(m)$$
$$\mathsf{Vrfy}_k(t, m) = \begin{cases} 1 & \text{if } F_k(m) = t \\ 0 & \text{if } F_k(m) \neq t \end{cases}$$

2.10 Authenticated Encryption

In addition to data confidentiality, authenticated encryption (AE) schemes provide data authenticity and integrity. The schemes that we consider in this paper enforce that a ciphertext can be decrypted only if it is the message sent by the expected sender.

An authenticated encryption scheme is defined as $\mathsf{AE} := (\mathsf{AuthEnc}, \mathsf{AuthDec})$ over a message, key and ciphertext space, similar to ENC [6]. Unlike ENC, if a ciphertext c is not authentic, $\mathsf{AuthDec}$ might *fail* to decrypt [6]. We denote this case as $\mathsf{AuthDec}(c) = \perp$.

Ciphertext integrity. Ciphertext integrity is a security notion related to AE schemes. Roughly speaking, it asserts that for a key k uniformly randomly chosen by the challenger, even if the adversary has access to an encryption oracle, it cannot output a fresh ciphertext (i.e., a ciphertext not returned by the oracle) c encrypted under k such that $\mathsf{AuthDec}_k(c) \neq \perp$ [6].

Random-key robustness. Random-key robustness (RKR) is another security notion related to AE schemes. An authenticated encryption scheme is random-key robust if it is infeasible to come up with a ciphertext that decrypt under two uniformly random keys [18]. In this paper, we are concerned with a specific version of RKR where the keys are chosen uniformly randomly by the challenger [26].

The game that captures this notion is given in Figure 2.6, denoted as Exp-RKR where $\mathcal{K}$ denotes the key set. Advantage of an adversary $\mathcal{A}$ is defined as

$$\mathsf{Adv}^{\text{Exp-RKR}}_{\mathcal{A},\mathsf{AE}}(\lambda) = \Pr\left[\text{Exp-RKR}^{\mathcal{A}}_{\mathsf{AE}}(\lambda) = 1\right].$$

Then, we say that AE is random-key robust if and only if

$$\forall \text{PPT } \mathcal{A} : \mathsf{Adv}^{\text{Exp-RKR}}_{\mathcal{A},\mathsf{AE}}(\lambda) \leq \mathsf{negl}(\lambda).$$

Construction. An authenticated encryption scheme can be constructed by extending an encryption scheme ENC with a MAC scheme MAC with *encrypt-then-MAC* method [27], as follows:

$$\mathsf{AuthEnc}_k(m) = (\mathsf{ENC.Enc}_k(m), \mathsf{MAC.Tag}_k(\mathsf{ENC.Enc}_k(m)))$$

$$\mathsf{AuthDec}_k((c', t)) = \begin{cases} \mathsf{ENC.Dec}_k(c') & \text{if } \mathsf{MAC.Vrfy}_k(t, c') = 1 \\ \perp & \text{if } \mathsf{MAC.Vrfy}_k(t, c') = 0 \end{cases}$$

2.11 Authenticated Key Exchange

Key exchange (KE) protocols allows parties to establish a common key over an insecure channel [27]. Roughly speaking, the security of KE dictates that if a key is established between honest parties, then the adversary has no knowledge of it [27]. Accordingly, KE may be used to establish common keys after which point the parties communicate with a symmetric cipher [33]. A well-known protocol that realizes this is Diffie-Hellman key exchange protocol [15], explained in Section 3.1.

Authenticated key exchange (AKE) protocols additionally have the property that only the intended parties can establish a common key [16]. This is achieved by using long-term asymmetric keys that are used to identify the parties. We give the ideal functionality $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ defined in [26], and we present it in Section 2.11.1. We are specifically concerned with the Triple Diffie-Hellman (3DH) protocol [31] given in Section 3.2, which UC-realizes $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ [13].

Forward secrecy. Forward secrecy is a security requirement on KE schemes. Assume that we are in a setting where two parties communicate over an insecure channel by encrypting their messages with a shared secret key established with some key agreement protocol. Also assume that an adversary eavesdrops on the encrypted communication. Now, once the adversary steals the secret key (i.e., the key is *compromised*), the communication up to that point is revealed, since it can decrypt the encrypted communication material it had collected up to that point. Forward secrecy is a requirement that aims to mitigate such attacks. We give the following description of forward secrecy directly from [16]:

Description 2 (Forward secrecy)**.** *An authenticated key exchange protocol provides perfect forward secrecy if disclosure of long-term secret keying material does not compromise the secrecy of the exchanged keys from earlier runs.*

Key-compromise impersonation security. KCI security is another security requirement on AKE schemes. In our explanation of forward secrecy, we gave an attack that the adversary can mount after it compromises the key. Now, we consider another attack: *key-compromise impersonation* (KCI). In this attack, once the adversary steals the keys that belong to a party $\mathcal{P}$, it can *impersonate* $\mathcal{P}$ (i.e., act on behalf of $\mathcal{P}$) to other honest parties from that point on. KCI security ensures that even after key compromise, the honest interactions (i.e., interactions where all parties are honest) remain secure. We give the following description of KCI security directly from [26]:

Description 3 (KCI Security)**.** *The notion of KCI (for "key-compromise impersonation") security for KE protocols, concerns an attacker A who learns party P's long-term keys but otherwise does not actively control P. Resistance to KCI attacks, or "KCI security" for short, postulates that even though A can impersonate P to other parties, sessions which P itself runs with honest peers are unaffected and remain secure.*

```
▷ On $(\textsc{UsrSession}, \{\textbf{sid}\}, \{\textbf{ssid}\}, \{\mathsf{S}\})$ from $\mathsf{U}$:
  1: Send $(\textsc{UsrSession}, sid, ssid, \mathsf{U}, \mathsf{S})$ to $\mathcal{A}$.
  2: If $ssid$ was not used before by $\mathsf{U}$, then record $\langle \textsc{Session}, ssid, \mathsf{U}, \mathsf{S}, \texttt{Fresh} \rangle$.

▷ On $(\textsc{SvrSession}, \{\textbf{sid}\}, \{\textbf{ssid}\}, \{\mathsf{U}\})$ from $\mathsf{S}$:
  1: Send $(\textsc{SvrSession}, sid, ssid, \mathsf{U}, \mathsf{S})$ to $\mathcal{A}$.
  2: If $ssid$ was not used before by $\mathsf{S}$, then record $\langle \textsc{Session}, ssid, \mathsf{S}, \mathsf{U}, \texttt{Fresh} \rangle$.

▷ On $(\textsc{Compromise}, \{\textbf{sid}\}, \{\mathsf{P}\})$ from $\mathcal{A}$:
  1: Mark $\mathsf{P}$ as $\texttt{Compromised}$.

▷ On $(\textsc{Impersonate}, \{\textbf{sid}\}, \{\textbf{ssid}\}, \{\mathsf{P}\})$ from $\mathcal{A}$:
  1: If $\mathsf{P}$ is marked as $\texttt{Compromised}$ and $\langle \textsc{Session}, ssid, *, \mathsf{P}, \{\mathsf{st}\} \rangle$ where $\mathsf{st} = \texttt{Fresh}$ exists,
     then update $\mathsf{st} := \texttt{Compromised}$.

▷ On $(\textsc{NewKey}, \{\textbf{sid}\}, \{\textbf{ssid}\}, \{\mathsf{P}\}, \{\hat{\mathsf{SK}}\})$ where $|\hat{\mathsf{SK}}| = \lambda$ from $\mathcal{A}$:
  1: If $\langle \textsc{Session}, ssid, \mathsf{P}, \{\mathsf{P'}\}, \{\mathsf{st}\} \rangle$ where $\mathsf{st} \neq \texttt{Completed}$ exists, then do:
    1.1: If $\mathsf{st} = \texttt{Compromised}$, then set $\mathsf{SK} := \hat{\mathsf{SK}}$.
    1.2: Else if $\mathsf{st} = \texttt{Fresh}$ and $(\textsc{Output}, sid, ssid, \{\mathsf{SK'}\})$ was sent to $\mathsf{P'}$ when there was a
         record $\langle \textsc{Session}, ssid, \mathsf{P'}, \mathsf{P}, \texttt{Fresh} \rangle$, then set $\mathsf{SK} = \mathsf{SK'}$.
    1.3: Else, choose $\mathsf{SK} \leftarrow\!\$ \{0,1\}^\lambda$.
    1.4: Update $\mathsf{st} := \texttt{Completed}$.
    1.5: Send $(\textsc{Output}, sid, ssid, \mathsf{SK})$ to $\mathsf{P}$.
```

Figure 2.7: $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ ideal functionality [26]

2.11.1 Ideal Functionality

In Figure 2.7, we present the functionality $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ from [26] with only syntactic changes. In the figure, λ denotes the security parameter.

Usage. In a setting with two honest parties U and S and no active attacks, $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ can be used as follows:

1. The server S sends $(\textsc{SvrSession}, sid, ssid, \mathsf{U})$.
2. The client U sends $(\textsc{UsrSession}, sid, ssid, \mathsf{S})$.
3. The adversary $\mathcal{A}$ sends $(\textsc{NewKey}, sid, ssid, \mathsf{U}, \hat{\mathsf{SK}})$ with some session key $\hat{\mathsf{SK}}$ which reveals a uniformly random session key SK to the client U.
4. The adversary $\mathcal{A}$ sends $(\textsc{NewKey}, sid, ssid, \mathsf{S}, \hat{\mathsf{SK}})$ with some session key $\hat{\mathsf{SK}}$ which reveals SK to the server S.

Adaptive compromise and impersonation attacks. We equip the ideal functionality with COMPROMISE and IMPERSONATE interfaces. The adversary $\mathcal{A}$ can invoke COMPROMISE to compromise an honest server, after which it can invoke IMPERSONATE to impersonate as that honest server in any subsession of its choice.

Choice of session keys. Notice that in an honest setting, the ideal functionality chooses a common uniformly random key. In essence, this captures the security property that $\mathcal{A}$ has no knowledge of the session key outputted by the honest parties. However, if $\mathcal{A}$ mounts an impersonation attack, it has the power to set the session key.

KCI security. The ideal functionality captures the KCI security notion that we defined earlier. Notice that even after a server compromise, the subsessions in which $\mathcal{A}$ doesn't mount an impersonation attack remain secure.

2.12 Towards saPAKEs

In this section, we introduce *strong asymmetric password-authenticated KE* (saPAKE) protocols. We do this by continuing our discussion of authenticated KE protocols that we first introduced in Section 2.11 and expanding it with passwords, asymmetry, and strong-ness. The meanings of these concepts are explained in the corresponding sections.

2.12.1 Motivation

Currently, services that require user authentication predominantly utilize password-over-TLS with salted hashes (along with the salt) stored at the server [26]. We wish to move away from this model for two main reasons:

1. **Reliance on PKI:** TLS requires public-key infrastructure (PKI) [33]. In particular, the users have to implicitly trust that the certificate authorities in the certificate chain are trustworthy. However, there have been PKI attacks that incline us to shy away from such a setup [26].
2. **Exposure of the clear-text password to the server:** During authentication, even though the password is sent through a secure tunnel, the server sees it in the clear. Considering that most people tend to reuse passwords [11], or some servers might log these passwords, this is a risk that should be avoided.

While introducing a new solution, we wish to maintain the security properties that are provided by password-over-TLS with salted hashes. In particular, only the following attacks should be possible: In a single subsession, only a single *online dictionary attack* should be possible, and an *offline dictionary attack* should be possible only after compromising a server. We consider these attacks to be unpreventable in the context of PAKEs [23], because the authenticity is ensured only by the password. Now, we briefly explain these attacks.

Online dictionary attack. Let spw be the actual password of a user. In this attack, the adversary can check the correctness of a password by using the server as an oracle. The adversary can do this by initiating an authentication request with its guessed password tpw. The authentication fails if and only if $\mathsf{spw} \neq \mathsf{tpw}$.

Offline dictionary attack. Let spw be the actual password of a user. In this attack, the adversary can check the correctness of a password *locally*. The adversary can do this by compromising a server and stealing the salt and the salted hash h of a user. Then, to check whether a guessed password tpw is correct, the adversary computes the salted hash of tpw h' and compares it with h. We have $h = h'$ if and only if $\mathsf{tpw} = \mathsf{spw}$.

2.12.2 Introducing Passwords

In Section 3.2, we give an AKE implementation, which makes it possible for two parties to establish a common key with authenticity guarantees if both parties securely maintain long-term keys. However, it is unreasonable to expect users to memorize cryptographic keys, and neither can we assume that they have the hardware necessary to securely store them.

We introduce password-AKEs (PAKEs), where the user authenticates itself by a low-entropy string that it chooses during a registration process (i.e., a password) [1]. In this setting, we say that an authenticated client is a client that knows the password.

Issue with PAKEs. Current PAKE implementations [2, 24] do not work well as a user authentication mechanism, since the password is an input to the server. If the passwords are stored in plaintext at the server side, the adversary can trivially impersonate the users after a server compromise. If the password is sent to the server from the user, a secure channel must be established. This can be done securely by utilizing TLS, but this would contradict our goal of avoiding PKI.

2.12.3 Introducing Asymmetry

In this setting, we remove the trust from the server. More precisely, we allow the adversary to compromise (i.e., steal the password files) or corrupt (i.e., take full control of) the server. Asymmetric PAKE (aPAKE) is a type of PAKE in which the client and the server have different inputs and run different protocols [26]. In aPAKE protocols, the server only needs to store a one-way function [27] (e.g., hash) of the password, and it doesn't see the password in plaintext. Recall from Section 2.3 that a pre-image resistant hash function is a one-way function.

Issue with aPAKEs. These protocols are weak towards *pre-computation attacks* [26]. More specifically, an adversary can choose a subset of all possible passwords $\mathcal{D}$ and build a table locally before compromising the server:

$$\forall \mathsf{pw} \in \mathcal{D} : (\mathsf{pw}, \mathsf{H}(\mathsf{pw}))$$

where H is the pre-image resistant hash function used in the protocol. Then, the adversary can steal the password of any user whose password is in $\mathcal{D}$ in $\mathcal{O}(1)$ time by computing $\mathsf{H}(\mathsf{tpw})$ and performing a lookup on this table.

Note that there exist aPAKE protocols that utilize salts, but they are transmitted from the server to the user in plaintext [20, 21].

2.12.4 Introducing Strong-ness

We refer to the definition of strong aPAKE (saPAKE) [26] that mitigates pre-computation attacks. In this setting, the server maintain a salted hash such that the salt is only visible to the server [26]. This ensures that only dictionary attacks remain possible [26]. These protocols give us "best of the both worlds", i.e., the following goals are achieved:

1. No reliance on PKI, just like in aPAKEs.

2. No password transmission in cleartext, just like in aPAKEs.

3. No pre-computation attacks, just like in TLS.

In [26], the authors propose a basic compiler that can turn any aPAKE protocol into an saPAKE protocol by combining it with an OPRF protocol. However, we consider their second construction (i.e., OPAQUE) which is achieved by a concurrent execution of OPRF and AKE [26]. More specifically, the client uses OPRF (with its password as the input) to derive a decryption key for a ciphertext stored in the server. If the password is correct, the ciphertext decrypts into the necessary asymmetric keys for it to participate in an AKE with the server. Ultimately, the common session keys are established by AKE.

Chapter 3

Constructions

In this section, we explain the constructions that we use in the protocol we describe in Section 5. We do not provide formal security proofs for the constructions but instead refer to the papers that contain them.

3.1 Diffie-Hellman Key Exchange

In Figure 3.1, we give Diffie-Hellman key exchange (DH), a two-party key exchange protocol that provides forward secrecy [15].

Let $\mathbb{G}_p$ be a cyclic group of prime order p with a generator g_p. In this protocol, both parties choose a uniformly random value $x \leftarrow\$ \mathbb{Z}_p$, referred to as their *private key*. Then, they compute their *public keys* $X \coloneqq g_p^x$ and share it with each other. Finally, a party that has received X' from the other party computes the common key $K \coloneqq X'^x$.

Assuming that DDH is hard in $\mathbb{G}_p$, then the adversary doesn't learn anything about the established key among two honest parties [27]. Notice that DH is not authenticated. In other words, an active man-in-the-middle adversary can establish common secrets with a party of its choosing.

3.2 Triple Diffie-Hellman

In Figure 3.2, we give triple Diffie-Hellman (3DH), a two-party authenticated key exchange protocol with forward secrecy and KCI security [31].

Let $\mathbb{G}_p$ be a cyclic group of prime order p with a generator g_p. In this protocol, the parties have *long-term keys* and *ephemeral keys*. We assume that the long-term public keys $P_a = g_p^{p_a}$ and $P_b = g_p^{p_b}$ are public. In each execution of the protocol, the parties choose and exchange

Alice		**Bob**
$x_a \leftarrow\$ \mathbb{Z}_p$		$x_b \leftarrow\$ \mathbb{Z}_p$
$X_a \coloneqq g_p^{x_a}$		$X_b \coloneqq g_p^{x_b}$
	$\xrightarrow{X_a}$	
	$\xleftarrow{X_b}$	
$K_a \coloneqq X_b^{x_a}$		$K_b \coloneqq X_a^{x_b}$
return K_a		**return** K_b

Figure 3.1: Diffie-Hellman Key Exchange (DH) Protocol

$$
\begin{array}{lcl}
\textbf{Alice}(p_a, P_a, P_b) & & \textbf{Bob}(p_b, P_b, P_a) \\
\hline
x_a \leftarrow\!\!\$\ \mathbb{Z}_p & & x_b \leftarrow\!\!\$\ \mathbb{Z}_p \\
X_a := g_p^{x_a} & & X_b := g_p^{x_b} \\
 & \xrightarrow{\qquad X_a \qquad} & \\
 & \xleftarrow{\qquad X_b \qquad} & \\
K_a := \mathsf{H}(X_b^{x_a}, X_b^{p_a}, P_b^{x_a}) & & K_b := \mathsf{H}(X_a^{x_b}, P_a^{x_b}, X_a^{p_b}) \\
\textbf{return } K_a & & \textbf{return } K_b
\end{array}
$$

Figure 3.2: Triple Diffie-Hellman Key Exchange (3DH) Protocol

fresh ephemeral keys, as in the regular DH key exchange. Different from DH, the final key is computed by combining the ephemeral keys with long-term keys through a random oracle H. Note that the usage of ephemeral keys provides forward secrecy whereas the long-term keys ensure authenticity.

It is shown in [13] that 3DH UC-realizes $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ (see Section 2.11.1) in $\mathcal{F}_{\mathsf{RO}}$-hybrid model if GapDH is hard in $\mathbb{G}_p$.

3.3 2HashDH

In Figure 3.3, we give 2HashDH, a two-party OPRF protocol [25]. In this protocol, the pseudo-random function family is defined as $F_k(x) = \mathsf{H}_2(x, \mathsf{H}_1(x)^k)$ where $\mathsf{H}_1, \mathsf{H}_2$ are random oracles.

Let $\mathbb{G}_q$ be a cyclic group of prime order q with a generator g_q. Let $\mathsf{H}_1, \mathsf{H}_2$ be random oracles that map to $\mathbb{G}_p$. In this protocol, the client with input x first chooses a blinding factor $r \leftarrow\!\!\$\ \mathbb{Z}_q$ and blinds its input $x_b := \mathsf{H}_1(x)^r$. The blinded input is then sent to the server with input k, who in turn computes the blinded evaluation $y_b := x_b^k$ and sends it back. Finally, the client unblinds the evaluation with the blinding factor $y := \mathsf{H}_2(x, y_b^{1/r})$ to get $\mathsf{H}_2(x, \mathsf{H}_1(x)^k) = F_k(x)$. Notice that the server doesn't learn x, and the client doesn't learn k.

It is shown in [25] that 2HashDH UC-realizes $\mathcal{F}_{\mathsf{OPRF}}$ (see Section 2.7.1) in $\mathcal{F}_{\mathsf{RO}}$-hybrid model if OMDH(N, Q) is hard in $\mathbb{G}_q$, where Q is the total number of 2HashDH executions and $N = Q + q_1$ where q_1 is the number of H_1 evaluations.

OPRF prefix. Recall from Section 2.7.1 that protocols which UC-realize $\mathcal{F}_{\mathsf{OPRF}}$ have a "prefix". Notice that, in a subsession, $\mathcal{A}$ cannot ask for an evaluation of its choice after the server receives α sent by the client, since the input of the client is fixed after that point. Hence, the prefix corresponds to the blinded input (i.e., α) sent by the client in this protocol [26].

3.4 OPAQUE

In this section, we give the protocol diagram of OPAQUE from [26] only with syntactic differences. The protocol consists of a concurrent execution of 2HashDH (OPRF) [25] and AKE. The registration phase, which is assumed to happen over secure channels, is shown in Figure 3.4. In [26], an alternative registration phase that doesn't assume secure channels is explained, which we give in Appendix B. The authentication phase is shown in Figure 3.5.

Briefly, the client with input password pwd, asks for an OPRF evaluation y which is used as a key for the ciphertext c provided by the server. The evaluation y successfully decrypts c into long-term AKE keys p_u, P_u, P_s if pwd is correct. Then, the client and the server compute the session keys with the key computation algorithm KE.

$$
\begin{array}{lcr}
\textbf{Client}(x) & & \textbf{Server}(k) \\
\hline
r \leftarrow_{\$} \mathbb{Z}_q & & \\
x_b := \mathsf{H}_1(x)^r & & \\
 & \xrightarrow{\quad x_b \quad} & \\
 & & y_b := x_b^k \\
 & \xleftarrow{\quad y_b \quad} & \\
y := \mathsf{H}_2(x, y_b^{1/r}) & & \\
\textbf{return } y & &
\end{array}
$$

Figure 3.3: 2HashDH Protocol

$$
\begin{array}{lcl}
\textbf{Client}(sid, ssid, \mathsf{pwd}) & & \textbf{Server}(sid, ssid, \mathsf{file}) \\
\hline
 & \xrightarrow{\quad \mathsf{pwd} \quad} & \\
 & & k_s \leftarrow_{\$} \mathbb{Z}_q \\
 & & y := \mathsf{H}_2(\mathsf{pwd}, \mathsf{H}_1(\mathsf{pwd})^{k_s}) \\
 & & \textbf{discard } \mathsf{pwd} \\
 & & p_s \leftarrow_{\$} \mathbb{Z}_p; P_s := g_p^{p_s} \\
 & & p_u \leftarrow_{\$} \mathbb{Z}_p; P_u := g_p^{p_u} \\
 & & c \leftarrow \mathsf{AuthEnc}_y(p_u \parallel P_u \parallel P_s) \\
 & & \textbf{discard } p_u \\
 & & \mathsf{file}[sid] := (p_s, P_u, c, k_s)
\end{array}
$$

Figure 3.4: OPAQUE Registration Phase

Client$(sid, ssid, \mathsf{pwd})$		**Server**$(sid, ssid, \mathsf{file})$
$r \leftarrow\!\!\$\; \mathbb{Z}_q; \alpha \coloneqq \mathsf{H}_1(\mathsf{pwd})^r$		
$x_u \leftarrow\!\!\$\; \mathbb{Z}_p; X_u \coloneqq g_p^{x_u}$		
	$\xrightarrow{X_u, \alpha}$	
		$x_s \leftarrow\!\!\$\; \mathbb{Z}_p; X_s \coloneqq g_p^{x_s}$
		$(p_s, P_u, c, k_s) \coloneqq \mathsf{file}[sid]$
		$\beta \coloneqq \alpha^{k_s}$
		$K \coloneqq \mathsf{KE}(p_s, x_s, P_u, X_u)$
		$\mathsf{tx} \coloneqq \mathsf{GenTx}(sid \parallel ssid \parallel \alpha)$
		$\mathsf{SK} \coloneqq F_K^{(0)}(\mathsf{tx})$
		$A_s \coloneqq F_K^{(1)}(\mathsf{tx}); A_u \coloneqq F_K^{(2)}(\mathsf{tx})$
	$\xleftarrow{X_s, \beta, c, A_s}$	
$y' \coloneqq \mathsf{H}_2(\mathsf{H}_1(\mathsf{pwd}), \beta^{1/r})$		
$(p'_u, P'_u, P'_s) \coloneqq \mathsf{AuthDec}_{y'}(c)$ **or abort**		
$K' \coloneqq \mathsf{KE}(p_u, x_u, P_s, X_s)$		
$\mathsf{tx'} \coloneqq \mathsf{GenTx}(sid \parallel ssid \parallel \alpha)$		
$\mathsf{SK'} \coloneqq F_{K'}^{(0)}(\mathsf{tx'})$		
$A'_s \coloneqq F_{K'}^{(1)}(\mathsf{tx'}); A'_u \coloneqq F_{K'}^{(2)}(\mathsf{tx'})$		
assert $A'_s = A_s$		
	$\xrightarrow{A'_u}$	
return $(sid, ssid, \mathsf{SK'})$		
		assert $A'_u = A_u$
		return $(sid, ssid, \mathsf{SK})$

Figure 3.5: OPAQUE Authentication Phase

Notation and parameters. We give the parameters directly from [26], but the notation is slightly augmented for consistency. The OPRF is executed over the cyclic group $\mathbb{G}_q = \langle g_q \rangle$ of prime degree q such that $|q| = 2\lambda$ where λ is the security parameter [26]. The AKE is executed over the cyclic group $\mathbb{G}_p = \langle p \rangle$ of prime degree p such that $|p| = 2\lambda$. The long-term keys are denoted as p_u, p_s, P_u, P_s, and the ephemeral keys are denoted as x_u, x_s, X_u, X_s. We have random oracles H_1, H_2 where H_1 maps to $\mathbb{G}_q$ and H_2 maps to $\{0,1\}^{2\lambda}$. For $i \in \{1,2,3\}$ and $k \in \{0,1\}^{2\lambda}$, $F_k^{(i)}$ is a PRF with range $\{0,1\}^{2\lambda}$. Finally, $\mathsf{AuthEnc}$ and $\mathsf{AuthDec}$ are the authenticated encryption and decryption algorithms respectively.

Protocol transcript tx. The protocol transcript is computed by applying a random oracle GenTx on *sid*, *ssid*, and α. This makes the session key dependent on the particular subsession and the OPRF prefix in that subsession.

Implicit usage of MACs. As discussed in Section 2.9, a MAC scheme can be directly constructed from a PRF. Notice that A_s and A_u are MAC tags on the transcript tx computed by PRFs $F_k^{(1)}(\cdot)$ and $F_k^{(2)}(\cdot)$ respectively.

Key computation KE algorithm. In Figure 3.5, KE represents the session key computation algorithm of an AKE. In the paper, the authors demonstrate HMQV [30], but any AKE protocol that UC-realizes $\mathcal{F}_{\mathsf{AKE\text{-}KCI}}$ can be used [26]. We give the KE algorithm for 3DH [31] (explained in Section 3.2), which is also used in the IRTF draft given in Section 3.5 and the protocol given in Section 5:

$$\text{(At client): } \mathsf{KE}(p_u, x_u, P_s, X_s) = \mathsf{H}(X_s^{x_u}, X_s^{p_u}, P_s^{x_u})$$
$$\text{(At server): } \mathsf{KE}(p_s, x_s, P_u, X_u) = \mathsf{H}(X_u^{x_s}, P_u^{x_s}, X_u^{p_s})$$

where H is a random oracle. Note that 3DH requires that GapDH is hard in $\mathbb{G}_p$.

Authenticated encryption scheme. Any authenticated encryption scheme that is random-key robust and equivocable can be used [26]. See Section 2.10 for the explanation of these security notions. We refer to [26] for more details on how such a scheme can be constructed.

3.5 IRTF Draft

In this section, we give an overview of the authentication phase of the protocol described in the IRTF draft [7] which is based on [26]. A simplified protocol diagram is given in Figure 3.6.

3.5.1 Building Blocks

The protocol in Figure 3.6 uses multiple subroutines which are explained in this section. For detailed explanation on how to securely configure the parameters, we refer to the original draft [7].

Length parameters. Two important global parameters are N_n (which denotes the nonce length) and N_s (which denotes the seed length) [7]. Both of these parameters are set to be 32 bytes [7].

Nonce generation. Both the client and the server generate nonces by calling $\mathsf{NewNonce}()$. The returned value must be chosen uniformly randomly from $\{0,1\}^{N_n}$.

```
$\textbf{Client}(uid, \mathsf{pwd})$                                   $\textbf{Server}(p_s, P_s, s_{\mathsf{OPRF}}, \mathsf{file})$

$(\alpha, r) \leftarrow \mathsf{OPRF.Blind}(\mathsf{pwd})$
$(x_u, X_u) \leftarrow \mathsf{DH.Gen}()$
$\delta_c \leftarrow \mathsf{NewNonce}()$

                         $\xrightarrow{uid, \delta_c, X_u, \alpha}$

                                                                       $(P_u, k_{\mathsf{mask}}, n_e, t_e) \coloneqq \mathsf{file}[uid]$

                                                                       $s' \coloneqq s_{\mathsf{OPRF}} \parallel uid$
                                                                       $(\_, k_{\mathsf{OPRF}}) \coloneqq \mathsf{DH.GenFrom}(s')$
                                                                       $\beta \coloneqq \mathsf{OPRF.BlindEval}(k_{\mathsf{OPRF}}, \alpha)$

                                                                       $\delta_{\mathsf{mask}} \leftarrow \mathsf{NewNonce}()$
                                                                       $\Omega \coloneqq \mathsf{DrvMask}(k_{\mathsf{mask}}, \delta_{\mathsf{mask}})$
                                                                           $\oplus (P_s \parallel n_e \parallel t_e)$

                                                                       $(x_s, X_s) \leftarrow \mathsf{DH.Gen}()$
                                                                       $m_2' \coloneqq \delta_s \parallel \delta_{\mathsf{mask}} \parallel \beta \parallel \Omega \parallel X_s$
                                                                       $\mathsf{pre} \coloneqq P_s \parallel P_u \parallel m_1 \parallel m_2'$
                                                                       $\mathsf{ikm} \coloneqq (X_u)^{x_s} \parallel (X_u)^{p_s} \parallel (P_u)^{x_s}$
                                                                       $(k_0, k_1, k_s) \coloneqq \mathsf{DrvKeys}(\mathsf{ikm}, \mathsf{pre})$

                                                                       $t_s \leftarrow \mathsf{MAC.Tag}_{k_0}(\mathsf{H_p}(\mathsf{pre}))$
                                                                       $\delta_s \leftarrow \mathsf{NewNonce}()$

                         $\xleftarrow{\delta_s, X_s, \beta, \delta_{\mathsf{mask}}, \Omega, t_s}$

$y \coloneqq \mathsf{OPRF.Finalize}(\mathsf{pwd}, r, \beta)$

$k_{\mathsf{mask}} \coloneqq \mathsf{ExpMaskKey}(y)$
$(P_s \parallel n_e \parallel t_e) \coloneqq \Omega \oplus \mathsf{DrvMask}(k_{\mathsf{mask}}, \delta_{\mathsf{mask}})$
$(k_{\mathsf{auth}}, K_e, s_c) \coloneqq \mathsf{ExpEnv}(y, n_e)$
$(P_u, p_u) \coloneqq \mathsf{DH.GenFrom}(s_c)$
$\textbf{assert } \mathsf{MAC2.Vrfy}_{k_{\mathsf{auth}}}(t_e, n_e \parallel P_s \parallel P_u) = 1$

$m_2' \coloneqq \delta_s \parallel \delta_{\mathsf{mask}} \parallel \beta \parallel \Omega \parallel X_s$
$\mathsf{pre} \coloneqq P_s \parallel P_u \parallel m_1 \parallel m_2'$
$\mathsf{ikm} \coloneqq (X_s)^{x_u} \parallel (P_s)^{x_u} \parallel (X_s)^{p_u}$
$(k_0, k_1, k_s) \coloneqq \mathsf{DrvKeys}(\mathsf{ikm}, \mathsf{pre})$

$\textbf{assert } \mathsf{MAC.Vrfy}_{k_0}(t_s, \mathsf{H_p}(\mathsf{pre})) = 1$
$t_c \leftarrow \mathsf{MAC.Tag}_{k_1}(\mathsf{H_p}(\mathsf{pre} \parallel t_s))$
$\textbf{return } k_s$

                         $\xrightarrow{t_c}$

                                                                       $\textbf{assert}$
                                                                           $\mathsf{MAC.Vrfy}_{k_1}(t_c, \mathsf{H_p}(\mathsf{pre} \parallel t_s)) = 1$
                                                                       $\textbf{return } k_s$
```

Figure 3.6: Simplified IRTF draft (version 16) [7]. User id *uid* corresponds to a registered user.

Hash function. The hash function invocations are denoted as H_s for a domain separation identity s such that $\mathsf{H}_s(x)$ returns $\mathsf{H}(x \,\|\, s)$ where H is a collision-resistant hash function. In [7], the authors suggest the usage of SHA-256 or SHA-512 [22].

Key derivation function. The functions with names $\mathsf{Drv*}$ and $\mathsf{Exp*}$ correspond to functions that internally use a key derivation function (KDF) that we omit here for simplicity. Note that these functions are modelled as random oracles. In [7], the authors suggest the usage of HKDF [29].

- ▷ The function ExpEnv is used to expand an *envelope* into the material for which the client can use to complete the authentication. We explain the meaning of an envelope in Section 3.5.2.
- ▷ The function $\mathsf{ExpMaskKey}$ is used to expand a randomized password y to the masking key k_{mask}. The masking key is used as an input to $\mathsf{DrvMask}$.
- ▷ The function $\mathsf{DrvKeys}$ is used to generate the session key k_s and the MAC keys k_0, k_1.
- ▷ The function $\mathsf{DrvMask}$ is used to generate a mask that the server uses to blind the authentication response.

DH module. The DH module implements the Diffie-Hellman key exchange protocol on an appropriately chosen cyclic group of prime order [7].

- ▷ $\mathsf{DH.Gen}()$: Returns a pair (x, X) where x is a uniformly randomly chosen DH private key and X is the corresponding public key.
- ▷ $\mathsf{DH.GenFrom}(s)$: Same as $\mathsf{DH.Gen}$ except the choice of private key is deterministic on the given seed s, i.e., it uses the explicit randomness s.

OPRF module. The OPRF module implements an OPRF protocol. In [7], the authors suggest the usage of the protocol described in [12].

- ▷ $\mathsf{OPRF.Blind}(x)$: Returns α, r where α is the blinded input x, and r is the blinding factor that can be used to unblind an evaluation of α.
- ▷ $\mathsf{OPRF.BlindEval}(k, \alpha)$: Returns the blinded evaluation of α with key k, i.e., blinded version of the value $F_k(x)$ for a fixed PRF F.
- ▷ $\mathsf{OPRF.Finalize}(x, r, \beta)$: Given a blinded evaluation β of input x with some key k, unblinds it with r and returns $F_k(x)$ for a fixed PRF F.

MAC module(s). The $\mathsf{MAC2}$ module implements a random-key robust MAC scheme, i.e., given two random keys, it is infeasible to come up with a message that maps to the same tag. The MAC module implements a MAC scheme that is not necessarily random-key robust. In [7], the authors suggest HMAC [28] for both modules.

- ▷ $\mathsf{MAC/MAC2.Tag}_k(m)$: Returns a tag t on the message m with key k.
- ▷ $\mathsf{MAC/MAC2.Vrfy}_k(t, m)$: Returns 1 if and only if t is a tag on m with key k.

3.5.2 Overview

In essence, the protocol in Figure 3.6 is a concurrent execution of an OPRF (whose interfaces are denoted as OPRF) and an AKE. OPRF module is treated as a black box, but for AKE, a modified version of 3DH [31] with long-term keys P_s, P_u, p_u, p_s, and ephemeral keys x_u, X_u, x_s, X_s is used. An explanation of 3DH is given in Section 3.2.

Party inputs. The client takes as input a *uid* (which identifies the user) and its input password pwd. In turn, the server takes as input its long-term keys p_s, P_s, an OPRF seed s_{OPRF}, and the password file file. Server uses s_{OPRF} to compute the OPRF key by calling $\mathsf{DH.GenFrom}(s_{\mathsf{OPRF}} \parallel uid)$. This seed can either be chosen during the registration phase per user, or a single seed can be reused across different users [7], but it should be noted that a recent work [14] presents an attack when the same seed is reused. The long-term keys p_s, P_s are used to compute the *input key material* (ikm).

Utility of nonces. The nonces δ_c and δ_s together uniquely identify a subsession except with negligible probability. Although it is possible to use the ephemeral keys X_u, X_s as implicit nonces, adding nonces gives us "separation of concerns" which is arguably more important in an implementation draft than having the succinctness that we might strive for in theory.

Input key material. Both parties compute an input key material ikm which is a combination of the parties' long-term and ephemeral keys, as in 3DH [31]. In absence of active attacks, the client and the server compute the same ikm. As explained in Section 3.2, this ensures forward secrecy and KCI security.

Transcripts. Both parties compute a transcript of the protocol pre, which is dependent on the messages sent throughout the subsession, i.e., a single execution of the protocol. The transcript is used for key derivation, i.e., as input to $\mathsf{DrvKeys}$. Roughly speaking, this does two things: (1) binds the AKE and OPRF executions, and (2) ensures that the produced keys are directly dependent on fresh random values (e.g., the nonces) which uniquely identifies a subsession except with negligible probability.

Envelopes. The server stores an envelope, n_e along with its MAC tag t_e. Together with a correct OPRF evaluation y, the client invokes $\mathsf{ExpEnv}(y, n_e)$ which expands into a triplet $(k_{\mathsf{auth}}, K_e, s_c)$ where k_{auth} is the MAC key for the envelope's tag t_e for a random-key robust MAC scheme, K_e is the *export key*, and s_c is the seed from which user's long-term keys can be calculated. Note that the export key is a key that is the same across all invocations for a client who enters the correct password, and it can be used for application-specific use cases [7]. The envelope along with its MAC tag serves the same purpose as the ciphertext c in the protocol described in Section 3.4.

Masking. On an authentication request from the client, the server returns an authentication response that is XORed with a *mask* computed from $\mathsf{DrvMask}(k_{\mathsf{mask}}, \delta_{\mathsf{mask}})$. Notice that δ_{mask} is a fresh nonce per subsession, which means the mask is fresh per subsession. The authentication response is hidden as such in order to prevent user enumeration attacks in which an adversary aims to check if a user is registered on a server or not [7]. Accordingly, if *uid* corresponds to an invalid user, the server must return a bogus authentication response [7]. Note that this case is omitted in Figure 3.6.

3.5.3 Differences From OPAQUE

In this section, we identify the noteworthy differences of the IRTF draft from [26].

- ▷ The protocol in [26] utilizes *ssid*, i.e., global subsession ids, which are assumed to be unique per subsession. However, this requires either (1) synchronization between the client and the server, or (2) a 2-way handshake before the beginning of the protocol. To mitigate this, in the draft, nonces are used instead of subsession ids [7]. Note that a pair of nonces (one from the client and one from the server) can uniquely identify a subsession, except with negligible probability.

- ▷ The protocol in the draft accommodates multiple users [7]. However, [26] is only for a single user that is assumed to be registered at the server.
- ▷ The protocol in the draft utilizes user enumeration mitigation, as explained in Section 3.5.2 [7]. Such a mechanism is not present in [26].
- ▷ The protocol in the draft uses 3DH [31] for AKE, whereas [26] treats AKE as a black-box [7].
- ▷ The protocol in the draft treats OPRF as a black-box, whereas [26] uses 2HashDH [7, 25].

Chapter 4

Practical saPAKE Functionality

In this section, we describe our ideal functionality that was adapted from [26]. It is *practical* because (1) it does not take any global identities as input, and (2) it allows for multiple users. First, we introduce the concept of partial subsessions. Then, we give an overview of the changes we made compared to [26]. Finally, we go over the definition. The full definition is given in Appendix A.

4.1 Partial Subsessions

To circumvent the usage of global subsession ids, we introduce the concept of *partial subsessions.* For a set of subsession ids $\mathcal{J}$, we define the set of partial subsession identifiers $\mathcal{P}$ and the mapping $f_{\mathcal{P}}$ from $(\mathcal{P} \times \mathcal{P})$ to $\mathcal{J}$ that satisfies

$$\forall i \in \mathcal{J} : \exists \rho_1, \rho_2 \in \mathcal{P} : f_{\mathcal{P}}(\rho_1, \rho_2) = f_{\mathcal{P}}(\rho_2, \rho_1) = i.$$

We decompose the subsession ids defined in [26] to partial subsession ids. For every new client or server initialization, we let our ideal functionality choose and return a unique partial subsession id ρ. We assume that these subsession ids have total ordering property. Then we implicitly set

$$f_{\mathcal{P}}(\rho_1, \rho_2) = (\mathsf{min}(\rho_1, \rho_2), \mathsf{max}(\rho_1, \rho_2))$$

such that two partial subsession ids uniquely identify a subsession. A similar approach is used in [3].

4.2 Overview of Changes

Our proposed ideal functionality $\mathcal{F}_{\mathsf{psaPAKE}}$ (i.e., practical strong asymmetric PAKE functionality) is adapted from [26] with the following noteworthy changes:

▷ Partial subsessions are identified by unique partial subsession ids, denoted as ρ.

▷ The subsession id *ssid* is removed from interfaces as input. These interfaces now either take a partial subsession id or a pair of them (to identify a subsession) as inputs.

▷ User ids that identify different users for the same session are introduced as an input. We denote this new identity as *uid*. As a result, the session identifier *sid* uniquely identifies the server only.

▷ The File and Session records contain *sid* and *uid* to indicate for which server-user pair they are created for.

▷ The NewKey interface outputs $(sid, uid, \rho_1, \rho_2, \mathsf{SK})$ where SK is the session key. For a subsession among two machines $(\mathsf{P}_1, \mathsf{P}_2)$, ρ_i denotes the partial subsession id of P_i.

- ▷ The flag that keeps track of password compromise is replaced with pwstolen, which is a dictionary of flags indexed by (sid, uid).
- ▷ The delayed password attack flag dPT is now a dictionary of flags indexed by the triplet (sid, uid, ρ).
- ▷ The SESSION record contains a value denoting if it belongs to the server (i.e., `Svr`) or a user (i.e., `Usr`).
- ▷ On a SVRSESSION message, if the FILE record associated with the given uid does not exist, the password value of the partial subsession is set to $\perp$.

4.3 Definition

We define $\mathcal{F}_{\mathsf{psaPAKE}}$ as an ITM that captures the behavior and security guarantees of a strong asymmetric PAKE. The functionality is used by a server machine S and potentially multiple users. We allow the ITM to maintain the following global variables:

- ▷ pwstolen: For a given pair (sid, uid), we have $\mathsf{pwstolen}[(sid, uid)] = 1$ if and only if the password of the user uid registered with the server sid is compromised.
- ▷ dPT: For a given partial subsession (sid, uid, ρ), we have $\mathsf{dPT}[(sid, uid, \rho)] = 1$ if and only if the adversary is allowed to mount a delayed password test attack in that partial subsession.
- ▷ rcount: Maintains the number of created SESSION records and is initialized to 0.

Password Registration

The single server S registers the users by sending a STOREPWDFILE message, at which point an `Uncompromised` file record is created.

```
▷ On (StorePwdFile, {sid}, {uid}, {pwd}) from S:
  1: If this is the first (StorePwdFile, $sid, uid, *$) message, then do:
    1.1: Record $\langle$File, $sid, uid$, pwd, Uncompromised$\rangle$.
    1.2: Set $\mathsf{pwstolen}[(sid, uid)] := 0$.
```

Initialization

The interfaces USRSESSION and SVRSESSION are used to create the SESSION records for a client machine U and the server machine S respectively. Notice that USRSESSION interface takes as input the client's input password pwd, whereas SVRSESSION uses the associated FILE record's password value. Each new SESSION record is given a unique partial subsession id ρ, which is sent to the adversary, so that it can be used to refer to the records in the subsequent calls.

```
▷ On (UsrSession, {sid}, {uid}, {pwd}) from some user machine U:
  1: Set $\rho :=$ rcount, and update rcount := rcount + 1.
  2: Record $\langle$Session, $sid, uid, \rho$, Usr, pwd, Fresh, P$\rangle$.
  3: Send $\langle$UsrSession, $sid, uid, \rho$, P$\rangle$ to $\mathcal{A}$.

▷ On (SvrSession, {sid}, {uid}) from S:
  1: Set $\rho :=$ rcount, and update rcount := rcount + 1.
  2: Try to retrieve $\langle$File, $sid, uid$, {pwd}, $*\rangle$ or do:
    2.1: Set pwd := $\perp$.
```

```
3: Record $\langle \textsc{Session}, sid, uid, \rho, \texttt{Svr}, \mathsf{pwd}, \texttt{Fresh}, \mathsf{P}\rangle$.
4: Send $\langle \textsc{SvrSession}, sid, uid, \rho, \mathsf{P}\rangle$ to $\mathcal{A}$.
```

Adaptive Compromise

We define the compromise interfaces that can be invoked only on a request from the environment.

1. StealPwdFile: Marks the associated File record as `Compromised`. This interface models the real-life scenario of server compromise.
2. OfflineTestPwd: Allows the adversary to mount offline dictionary attacks only if the associated File record is marked as `Compromised`. We allow ε to issue OfflineTestPwd requests only if it has already issued the corresponding StealPwdFile request.

```
▷ On $\boxed{(\textsc{StealPwdFile}, \{\mathbf{sid}\}, \{\mathbf{uid}\})}$ from $\mathcal{A}$:
  1: If $\langle \textsc{File}, sid, uid, *, *\rangle$ exists, then do:
    1.1: Update the file record's status to Compromised.
    1.2: Send "password file stolen" to $\mathcal{A}$.
  2: Else, send "no password file" to $\mathcal{A}$.

▷ On $\boxed{(\textsc{OfflineTestPwd}, \{\mathbf{sid}\}, \{\mathbf{uid}\}, \{\mathsf{tpw}\})}$ from $\mathcal{A}$:
  1: If $\langle \textsc{File}, sid, uid, \{\mathsf{pwd}\}, \texttt{Compromised}\rangle$ exists, then do:
    1.1: If $\mathsf{tpw} = \mathsf{pwd}$, then do:
      1.1.1: Set $\mathsf{pwstolen}[(sid, uid)] := 1$.
      1.1.2: Send "correct guess" to $\mathcal{A}$.
    1.2: Else, send "wrong guess" to $\mathcal{A}$.
```

Active Session Attacks

Now, we define the active attacks that $\mathcal{A}$ can mount. The interface TestPwd captures an online dictionary attack. The interface Impersonate captures an impersonation attack, where the adversary who has compromised the server impersonates an honest server.

```
▷ On $(\textsc{Interrupt}, \{\mathbf{sid}\}, \{\mathbf{uid}\}, \{\rho\})$ from $\mathcal{A}$:
  1: If $\langle \textsc{Session}, sid, uid, \rho, \texttt{Svr}, *, \texttt{Fresh}, *\rangle$ exists, then do:
    1.1: Update the record's status to Interrupted.
    1.2: Set $\mathsf{dPT}[(sid, uid, \rho)] := 1$.

▷ On $(\textsc{TestPwd}, \{\mathbf{sid}\}, \{\mathbf{uid}\}, \{\rho\}, \{\mathsf{tpw}\})$ from $\mathcal{A}$:
  1: Try to retrieve $\langle \textsc{Session}, sid, uid, \rho, \{\mathsf{ty}\}, \{\mathsf{pwd}\}, \{\mathsf{st}\}, *\rangle$ or drop the message.
  2: If $\mathsf{ty} = \texttt{Svr}$ and $\mathsf{tpw} = \mathsf{pwd}$, set $\mathsf{pwstolen}[(sid, uid)] := 1$.
  3: If $\mathsf{st} = \texttt{Fresh}$, then do:
    3.1: If $\mathsf{tpw} = \mathsf{pwd}$, then do:
      3.1.1: Update the Session record's status to Compromised.
      3.1.2: Send "correct guess" to $\mathcal{A}$.
    3.2: Else, do:
      3.2.1: Update the Session record's status to Interrupted.
      3.2.2: Send "wrong guess" to $\mathcal{A}$.
  4: Else if $\mathsf{ty} = \texttt{Svr}$ and $\mathsf{dPT}[(sid, uid, \rho)] = 1$, then do:
    4.1: Set $\mathsf{dPT}[(sid, uid, \rho)] := 0$
    4.2: If $\mathsf{tpw} = \mathsf{pwd}$ send "correct guess" to $\mathcal{A}$.
    4.3: Else, send "wrong guess" to $\mathcal{A}$.

▷ On (IMPERSONATE, {sid}, {uid}, {$\rho$}) from $\mathcal{A}$:
  1: Try to retrieve $\langle$SESSION, $sid, uid, \rho$, Usr, {pwd}, Fresh, $*\rangle$ or drop the message.
  2: If there is a file record $\langle$FILE, $sid, uid$, pwd, Compromised$\rangle$, then do:
    2.1: Update the SESSION record's status to Compromised.
    2.2: Send "correct guess" to $\mathcal{A}$.
  3: Else, do:
    3.1: Update the SESSION record's status to Interrupted.
    3.2: Send "wrong guess" to $\mathcal{A}$.
```

Key Generation and Authentication

We finally define the key generation and authentication interfaces. In certain cases, e.g., when the adversary corrupts a party, the NEWKEY interface outputs exactly the given session key. Otherwise, the outputted session key is chosen uniformly randomly by the ideal functionality. The abort interface TESTABORT returns "success" in case the subsession represented by ρ_1, ρ_2 will succeed. By this, we mean that (1) client has entered the correct password, and (2) there has been no active attacks in that subsession.

```
▷ On (NEWKEY, {sid}, {uid}, {$\hat{\mathsf{SK}}$}, {$\rho_1$}, {$\rho_2$}) where $|\hat{\mathsf{SK}}| = \lambda$ from $\mathcal{A}$:
  1: Try to retrieve $\langle$SESSION, $sid, uid, \rho_2, *, *, *$, {$\mathsf{P_2}$}$\rangle$ or drop the message.
  2: If $\langle$BINDING, $\rho_2$, {$\rho'_1$}$\rangle$ exists and $\rho'_1 \neq \rho_1$, then abort.
  3: Else, record $\langle$BINDING, $\rho_1, \rho_2\rangle$.
  4: If $\langle$SESSION, $sid, uid, \rho_1$, {ty}, {pwd}, {st}, {$\mathsf{P_1}$}$\rangle$ where st $\neq$ Completed exists, then do:
    4.1: If ty = Svr, then set ty' := Usr, else set ty' := Svr.
    4.2: If st = Compromised,
         or (st = Interrupted, and ty = Svr, and pwstolen[$(sid, uid)$] = 1),
         or ($\mathsf{P}_1$ or $\mathsf{P}_2$ is corrupted),
         then set SK := $\hat{\mathsf{SK}}$.
    4.3: Else if st = Fresh, and (OUTPUT, $sid, uid, \rho_2, \rho_1$, {SK'}) was sent to $\mathsf{P}_2$ when there
         was a record $\langle$SESSION, $sid, uid, \rho_2$, ty', pwd, Fresh, $\mathsf{P}_2\rangle$,
         then set SK := SK'.
    4.4: Else, choose SK $\leftarrow\$ \{0,1\}^\lambda$.
    4.5: Update the SESSION record's status to Completed.
    4.6: Send (OUTPUT, $sid, uid, \rho_1, \rho_2$, SK) to $\mathsf{P}_1$.

▷ On (TESTABORT, {sid}, {uid}, {$\rho_1$}, {$\rho_2$}) from $\mathcal{A}$:
  1: If $\langle$BINDING, $\rho_2$, {$\rho'_1$}$\rangle$ exists and $\rho'_1 \neq \rho_1$, then abort.
  2: Else, record $\langle$BINDING, $\rho_1, \rho_2\rangle$.
  3: If $\langle$SESSION, $sid, uid, \rho_1$, {ty}, {pwd}, {st}, {$\mathsf{P_1}$}$\rangle$ where st $\neq$ Completed exists, then do:
    3.1: If ty = Svr, then set ty' := Usr, else set ty' := Svr.
    3.2: If st = Fresh, and $\langle$SESSION, $sid, uid, \rho_2$, ty', pwd, $*, *\rangle$ exists,
         then send "success" to $\mathcal{A}$.
    3.3: Else if st = Fresh, and ty = Svr, and $\langle$FILE, $sid, uid$, pwd, $*\rangle$ exists,
         then send "success" to $\mathcal{A}$.
    3.4: Else, do:
      3.4.1: Update the SESSION record's status to Completed.
      3.4.2: Send "failure" to $\mathcal{A}$.
      3.4.3: Send $\langle$ABORT, $sid, uid, \rho_1\rangle$ to $\mathsf{P}_1$.
```

Chapter 5

Augmented OPAQUE

In this section, we introduce the augmented OPAQUE protocol, which is adapted from [26]. The protocol is composed of a registration phase and an authentication phase. For the remaining of this paper, we denote this protocol as $\Pi_\star$.

5.1 Notation

We denote the security parameter as λ. For the key exchange protocol, we work in the cyclic group $\mathbb{G}_p$, which has prime order p with a generator g_p. For the OPRF protocol, we work in the cyclic group $\mathbb{G}_q$, which has prime order q with a generator g_q. For $i \in \{0,1,2\}$ and a key $k \in \mathcal{K}$, we assume that $F_k^{(i)}$ is a PRF over $\{0,1\}^\lambda$. We assume that H_1, H_2 and H_3 are random oracles. Among these,

- ▷ H_1 maps from $\{0,1\}^*$ to $\mathbb{G}_q$,
- ▷ H_2 maps from $\{0,1\}^* \times \mathbb{G}_q$ to $\{0,1\}^\lambda$,
- ▷ H_3 maps from $\mathbb{G}_p \times \mathbb{G}_p \times \mathbb{G}_p$ to $\mathcal{K}$ where $\mathcal{K}$ is the PRF key space.

5.2 Overview of Changes

We augment the protocol from [26] with the following noteworthy changes:

- ▷ Neither the server nor the client take a subsession id as an input.
- ▷ The password file at the server, denoted as file, is additionally indexed by the user id *uid* to accommodate multiple users. Accordingly, the client sends its user id along with its password in the registration phase.
- ▷ The client generates a random nonce δ_c and includes it in the first message. Similarly, the server generates a random nonce δ_s and includes it in the second message.
- ▷ Rather than generating a transcript by hashing the session id *sid*, subsession id *ssid*, we now generate a transcript tx by simply concatenating the public message contents.
- ▷ The protocol no longer outputs *ssid* along with the session key SK. Instead, it outputs the nonces along with SK to identify the subsession.
- ▷ We implement user enumeration mitigation, which is explained in Section 5.3.2.

5.3 Protocol Overview

In this section, we explain the protocol $\Pi_\star$.

Client(sid, uid, pwd) **Server**(sid, file)

$$\xrightarrow{uid,\ \mathsf{pwd}}$$

$k_s \leftarrow\$\ \mathbb{Z}_q$

$y := \mathsf{H}_2(\mathsf{pwd}, \mathsf{H}_1(\mathsf{pwd})^{k_s})$

discard pwd

$p_s \leftarrow\$\ \mathbb{Z}_p; P_s := g_p^{p_s}$

$p_u \leftarrow\$\ \mathbb{Z}_p; P_u := g_p^{p_u}$

$c \leftarrow \mathsf{AE.AuthEnc}_y(p_u \parallel P_u \parallel P_s)$

discard p_u

$\mathsf{file}[(sid, uid)] := (p_s, P_u, c, k_s)$

Figure 5.1: Augmented OPAQUE Registration Phase

5.3.1 Registration Phase

Registration phase is shown in Figure 5.1. During the registration phase, the client sends their password over to the server, and the server generates the credentials. Note that the registration message is assumed to be sent over a confidential and authenticated channel with integrity protections in place. Hence, the confidentiality, integrity, and the authenticity of the registration message is ensured. We additionally require that the server securely discards the user's long-term private key p_u and its password pwd.

5.3.2 Authentication Phase

Authentication phase is shown in Figure 5.2. Once a client and a server complete a registration phase, the client is able to initiate the authentication phase to authenticate itself to the server with the password that they had supplied during the registration phase. As opposed to the registration phase, authentication happens over an insecure channel where the adversary has full control over the network.

Explicit key agreement. We implement *explicit key agreement*, as in [26] and the IRTF draft [7]. This means that we consider it an authentication failure when a party aborts on a tag verification. In *implicit* key agreement, the authentication is considered to be failed when two parties output different keys, and this would only be noticed when parties proceed with these outputs, e.g., to establish a secure channel. Notice that when the session keys are different, then so are the tags, and the checks will fail except with negligible probability.

User enumeration mitigation. Since we work in multi-user setting, we need to consider *user enumeration attacks* as in the IRTF draft [7], in which an adversary aims to check if a user is registered on a server or not. To not leak this information, we do the following: On an unknown uid, the server responds with a set of values that are indistinguishable from the ones of a correct user. However, the ciphertext is produced in such a way that the client won't be able to decrypt it, except for negligible probability. Consequently, a malicious (or an honest) client doesn't know whether it has provided an invalid password or an invalid user id. Note that this mitigation is performed at the authentication phase only. If required and applicable, user enumeration mitigation during registration must be considered at the application level.

```
$\mathbf{Client}(sid, uid, \mathsf{pwd})$                                   $\mathbf{Server}(sid, \mathsf{file}, \mathsf{fakefile})$

$\delta_c \leftarrow\$ \{0,1\}^\lambda$
$r \leftarrow\$ \mathbb{Z}_q; \alpha \leftarrow \mathsf{H}_1(\mathsf{pwd})^r$
$x_u \leftarrow\$ \mathbb{Z}_p; X_u := g_p^{x_u}$

                      $\xrightarrow{uid, \delta_c, X_u, \alpha}$

                                                         $\delta_s \leftarrow\$ \{0,1\}^\lambda$
                                                         $x_s \leftarrow\$ \mathbb{Z}_p; X_s := g_p^{x_s}$
                                                         $\mathbf{if}\ \mathsf{file}[(sid, uid)] \neq \bot\ \mathbf{then}$
                                                           $(p_s, P_u, c, k_s) := \mathsf{file}[(sid, uid)]$
                                                         $\mathbf{elseif}\ \mathsf{fakefile}[uid] \neq \bot\ \mathbf{then}$
                                                           $(p_s, P_u, c, k_s) := \mathsf{fakefile}[(sid, uid)]$
                                                         $\mathbf{else}$
                                                           $k_s \leftarrow\$ \mathbb{Z}_q$
                                                           $p_s \leftarrow\$ \mathbb{Z}_p; P_u \leftarrow\$ \mathbb{G}_p$
                                                           $y \leftarrow\$ \{0,1\}^\lambda$
                                                           $c \leftarrow \mathsf{AE.AuthEnc}_y(0 \ldots 0)$
                                                           $\mathsf{fakefile}[(sid, uid)] := (p_s, P_u, c, k_s)$
                                                         $\mathbf{fi}$
                                                         $\beta := \alpha^{k_s}$
                                                         $K := \mathsf{H}_3(X_u^{x_s}, X_u^{p_s}, P_u^{x_s})$
                                                         $\mathsf{tx} := sid \parallel uid \parallel \delta_c \parallel \delta_s$
                                                           $\parallel X_u \parallel X_s \parallel \alpha \parallel \beta \parallel c$
                                                         $\mathsf{SK} := F_K^{(0)}(\mathsf{tx})$
                                                         $A_s := F_K^{(1)}(\mathsf{tx}); A_u := F_K^{(2)}(\mathsf{tx})$

                      $\xleftarrow{\delta_s, X_s, \beta, c, A_s}$

$y' := \mathsf{H}_2(\mathsf{H}_1(\mathsf{pwd}), \beta^{1/r})$
$(p'_u, P'_u, P'_s) := \mathsf{AE.AuthDec}_{y'}(c)$ $\mathbf{or\ abort}$
$K' := \mathsf{H}_3(X_s^{x_u}, P_s'^{x_u}, X_s^{p'_u})$
$\mathsf{tx'} := sid \parallel uid \parallel \delta_c \parallel \delta_s$
  $\parallel X_u \parallel X_s \parallel \alpha \parallel \beta \parallel c$
$\mathsf{SK'} := F_{K'}^{(0)}(\mathsf{tx'})$
$A'_s := F_{K'}^{(1)}(\mathsf{tx'}); A'_u := F_{K'}^{(2)}(\mathsf{tx'})$
$\mathbf{assert}\ A'_s = A_s$

                      $\xrightarrow{A'_u}$

$\mathbf{return}\ (sid, uid, \delta_c, \delta_s, \mathsf{SK'})$
                                                         $\mathbf{assert}\ A'_u = A_u$
                                                         $\mathbf{return}\ (sid, uid, \delta_s, \delta_c, \mathsf{SK})$
```

Figure 5.2: Augmented OPAQUE Authentication Phase

5.4 Message Multiplexing

An important consideration is establishment of duplex channels over adversarial networks. A server might receive multiple authentication requests, which could be implemented by executing each subsession in a different process. The server should be able to handle a response reliably by dispatching it to the correct process. Widely deployed transport layer protocols like TCP tackle this issue by using *sockets*, which is a concatenation of a host's unique identifier (e.g., IP address) and the port (denoting the process) [17]. However, assuming the existence of a secure duplex channel has two problems:

1. The implementations assume globally unique host identifiers, e.g., IP addresses.
2. It implicitly assumes that $\mathcal{A}$ cannot modify the certain parts (e.g., source/destination sockets) of a message.

To avoid this, we do the following: Once an ITM P receives a *response* r, we allow P to ask the adversary: "r was generated in respond to which *request*?" Clearly, $\mathcal{A}$ maintains full control over the request that P sees. This ability is embedded into our ITMs through the interface definitions of form "**On** m_1 **in response to** m_2" Indeed, our ITMs enhanced with this ability can be converted into ITMs in the standard UC model [8] e.g., by letting each party include in their response which message they are responding to.

5.5 Adversarial Model

We let the adversary corrupt a party in-between two authentication phases after which point it takes full control of the party. Moreover, on a request from the environment in-between two authentication phases, we let the adversary compromise the server by sending StealPwdFile messages, after which point it can ask the adversary to perform an offline dictionary attacks by sending OfflineTestPwd messages. Finally, we assume that the environment can initiate a registration phase only before any authentication phase is initiated.

Secure registrations. As opposed to the authentication phase, we assume that the registration phase happens over secure channels, which is modeled in the ideal functionality by the StorePwdFile interface. We describe an alternate registration phase from [26] in Appendix B, which doesn't need this assumption.

5.6 Security Theorem

In this section, we introduce the theorem that captures the security properties of the protocol introduced in Section 5.3, denoted as $\Pi_\star$.

Theorem 3 (Security). *Let H_1, H_2, H_3 be random oracles. Let $\mathbb{G}_q = \langle g_q \rangle$ be a cyclic group of prime order q such that $q \geq 2^\lambda$ and* OMDH(N, Q) *is hard in $\mathbb{G}_q$ where N is the sum of number of fresh H_1 queries and the number of authentication responses produced by a honest server, and Q is the number of authentication responses produced by a honest server. Let $\mathbb{G}_p = \langle g_p \rangle$ be a cyclic group of prime order p such that $p \geq 2^\lambda$ and* GapDH *is hard in $\mathbb{G}_p$. Let $F^{(0)}_{\cdot}$, $F^{(1)}_{\cdot}$, $F^{(2)}_{\cdot}$ be secure PRFs with domain $\{0,1\}^*$, range $\{0,1\}^\lambda$ and key space $\{0,1\}^\lambda$. Let $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ be a random-key robust and equivocable authenticated encryption scheme. Then, the protocol $\Pi_\star$ UC-realizes the $\mathcal{F}_{\mathsf{psaPAKE}}$ functionality in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model under the adversarial assumptions described in Section 5.5 except with at most the probability given in Section 6.2.3, which is negligible in the security parameter λ.*

5.7 Intuition for Security

In this section, we aim to give the reader an intuition on the security guarantees of $\Pi_\star$ as captured by the ideal functionality. To that end, we describe some attacks and how $\Pi_\star$ responds.

Impersonation attack. If both parties are honest, and $\mathcal{A}$ has compromised the server, it can mount an impersonation attack. By an impersonation attack, we mean that $\mathcal{A}$ acts as an honest server to the honest client. It can do this by choosing a password apw and responding to the authentication requests using the stolen OPRF key k_s. If $\mathsf{apw} = \mathsf{upw}$ where upw is client's input password, the client successfully authenticates, even though it might be that $\mathsf{upw} \neq \mathsf{spw}$ where spw is the user's actual password. The attack is shown in Figure 5.3. This attack is captured in $\mathcal{F}_{\mathsf{psaPAKE}}$ by the IMPERSONATE interface.

Offline dictionary attack. Now, we explain how $\mathcal{A}$ checks $\mathsf{tpw} = \mathsf{spw}$ where tpw is $\mathcal{A}$'s password guess, and spw is the user's actual password. It can do so by locally computing $y' := \mathsf{H}_2(\mathsf{tpw}, \mathsf{H}_1(\mathsf{tpw})^{k_s})$ where k_s is the stolen OPRF key, and c is the stolen ciphertext. Since the authenticated encryption scheme admits random key robustness, if $\mathsf{AE.AuthDec}_{y'}(c) \neq \perp$, then the adversary infers that $\mathsf{tpw} = \mathsf{spw}$ except with negligible probability. The attack is shown in Figure 5.4. This attack is captured in $\mathcal{F}_{\mathsf{psaPAKE}}$ by the OFFLINETESTPWD interface.

Server-side online dictionary attack. Assume that $\mathcal{A}$ would like to check $\mathsf{tpw} = \mathsf{spw}$ where tpw is $\mathcal{A}$'s password guess, and spw is the user's actual password. It can do so by playing the role of a client and receiving an evaluation $e = \mathsf{H}_1(\mathsf{tpw})^{k_s}$ and the ciphertext c from $\mathcal{P}_s$. Then, if $\mathsf{AE.AuthDec}_y(c) \neq \perp$ where $y = \mathsf{H}_2(\mathsf{tpw}, e)$, then $\mathcal{A}$ infers that $\mathsf{tpw} = \mathsf{spw}$ with overwhelming probability. The attack is shown in Figure 5.5. This attack is captured in $\mathcal{F}_{\mathsf{psaPAKE}}$ by the TESTPWD interface.

Client-side online dictionary attack. Assume that $\mathcal{A}$ would like to check $\mathsf{tpw} = \mathsf{upw}$ where tpw is $\mathcal{A}$'s password guess, and upw is the client's password input. It can do so by playing the role of a server and simulating a response with an arbitrary OPRF key k and a password guess tpw. If the client does not abort, then $\mathsf{H}_2(\mathsf{upw}, \mathsf{H}_1(\mathsf{tpw})^k) = \mathsf{H}_2(\mathsf{tpw}, \mathsf{H}_1(\mathsf{tpw})^k)$ except with negligible probability, and $\mathcal{A}$ infers that its guess is correct. The attack is shown in Figure 5.6. This attack is captured in $\mathcal{F}_{\mathsf{psaPAKE}}$ by the TESTPWD interface.

Mix-and-match attack. The adversarial power we introduced in Section 5.4 allows an adversary to forward a message for a particular subsession to another subsession of its choice. An observation we make is that such an attack is equivalent to an active attack on all the other values in a message. For this reason, our security proof in Section 6 does not explicitly consider mix-and-match attacks.

Replay attack. The public ephemeral keys computed by honest parties X_s and X_u are fresh per subsession. These values are used in the computation of K which is used to derive the final keys. Then, the output of every execution of the protocol is unique, except with negligible probability. In our security proof in Section 6, we show precisely this, which means we mitigate replay attacks.

```
 1: Retrieve $(\delta_c, X_u, \alpha)$ from $\mathcal{P}_c$.
 2: Choose any $\delta_s$ from $\{0,1\}^\lambda$. // Not necessarily random
 3: Choose any $x_s$ from $\mathbb{Z}_p$. // Not necessarily random
 4: Choose any $p_u$, $p_s$ from $\mathbb{Z}_p$// Not necessarily random
 5: Set $P_u \coloneqq g_p^{p_u}$, $P_s \coloneqq g_p^{p_s}$.
 6: Retrieve the stolen $k_s$.
 7: Set $\beta \coloneqq \alpha^{k_s}$. // Identical to what honest $\mathcal{P}_s$ would compute.
 8: Choose a password $\mathsf{apw}$.
 9: Set $y' \coloneqq \mathsf{H}_2(\mathsf{apw}, \mathsf{H}_1(\mathsf{apw})^{k_s})$.
10: Set $c' \coloneqq \mathsf{AuthEnc}_{y'}(p_u \parallel P_u \parallel P_s)$.
11: Set $\mathsf{tx} \coloneqq sid \parallel uid \parallel \delta_c \parallel \delta_s \parallel X_u \parallel X_s \parallel \alpha \parallel \beta \parallel c$.
12: Set $K \coloneqq \mathsf{H}_3(X_u^{x_s}, X_u^{p_s}, P_u^{x_s})$.
13: Set $\mathsf{SK} \coloneqq F_K^{(0)}(\mathsf{tx}), A_s \coloneqq F_K^{(1)}(\mathsf{tx})$.
14: Send $(\delta_s, g_p^{x_s}, c', A_s, \beta)$ to $\mathcal{P}_c$. // $\mathcal{P}_c$ will output $\mathsf{SK}$ if $\mathsf{apw} = \mathsf{upw}$.
```

Figure 5.3: Impersonation attack

```
1: Retrieve the stolen $c$ and $k_s$.
2: Set $y' \coloneqq \mathsf{H}_2(\mathsf{tpw}, \mathsf{H}_1(\mathsf{tpw})^{k_s})$.
3: If $\mathsf{AuthDec}_{y'}(c) \neq \perp$, then return "correct password".
4: Else, return "wrong password".
```

Figure 5.4: Offline password test attack

```
1: Choose any $r$ from $\mathbb{Z}_q$. // Not necessarily random
2: Choose any $X_u$ from $\mathbb{G}_p$. // Not necessarily random
3: Choose any $\delta_c$ from $\{0,1\}^\lambda$. // Not necessarily random
4: Set $\alpha \coloneqq \mathsf{H}_1(\mathsf{tpw})^r$.
5: Send $(\delta_c, X_u, \alpha)$ to $\mathcal{P}_s$, and receive $(*, *, \{\mathbf{c}\}, *, \{\beta\})$.
6: Set $e \coloneqq \beta^{1/r}$. // Unblind the evaluation
7: Set $y' \coloneqq \mathsf{H}_2(\mathsf{tpw}, e)$.
8: If $\mathsf{AuthDec}_{y'}(c) \neq \perp$, then return "correct password".
9: Else, return "wrong password".
```

Figure 5.5: Server-side online password test attack

```
1: Receive $(\delta_c, X_u, \alpha)$ from $\mathcal{P}_c$.
2: Choose any $\delta_s$ from $\{0,1\}^\lambda$. // Not necessarily random
3: Choose any $k$ from $\mathbb{Z}_q$. // Not necessarily random
4: Choose any $p_u$, $p_s$ from $\mathbb{Z}_p$// Not necessarily random
5: Set $P_u \coloneqq g_p^{p_u}$, $P_s \coloneqq g_p^{p_s}$.
6: Choose any $x_s$ from $\mathbb{Z}_p$. // Not necessarily random
7: Set $\beta \coloneqq \alpha^k$.
8: Set $y \coloneqq \mathsf{H}_2(\mathsf{tpw}, \mathsf{H}_1(\mathsf{tpw})^k)$.
9: Set $c \coloneqq \mathsf{AuthEnc}_y(p_u \parallel P_u \parallel P_s)$.
10: Set $\mathsf{tx} \coloneqq sid \parallel uid \parallel \delta_c \parallel \delta_s \parallel X_u \parallel X_s \parallel \alpha \parallel \beta \parallel c$.
11: Set $K \coloneqq \mathsf{H}_3(X_u^{x_s}, X_u^{p_s}, P_u^{x_s})$.
12: Set $A_s \coloneqq F_K^{(1)}(\mathsf{tx})$.
13: Send $(\delta_s, g_p^{x_s}, c, A_s, \beta)$ to $\mathcal{P}_c$.
14: If $\mathcal{P}_c$ aborts, then return “wrong password”.
15: Else, return “correct password”.
```

Figure 5.6: Client-side online password test attack

Chapter 6

Proof of Security

In this section, we prove Theorem 3. For any environment ε and the dummy adversary $\mathcal{A}_\perp$ that interacts with $\Pi_\star$, we first build the simulator ITM $\mathcal{S}$ that interacts with $\mathcal{F}_{\mathsf{psaPAKE}}$. By Equation 2.2, we argue that

$$\forall \text{PPT } \varepsilon : \exists \text{PPT } \mathcal{S} : \mathsf{Exec}^{\Pi_\star}_{\varepsilon,\mathcal{A}_\perp} \overset{c}{\approx} \mathsf{Exec}^{\mathsf{Ideal}_{\mathcal{F}_{\mathsf{psaPAKE}}}}_{\varepsilon,\mathcal{S}}$$

where $\overset{c}{\approx}$ denotes computational indistinguishability.

We denote the server as $\mathcal{P}_s$ and a client as $\mathcal{P}_c$. Throughout the proof, we may use the terms *real world* and *ideal world*. These terms correspond to the execution that is represented by $\mathsf{Exec}^{\Pi_\star}_{\varepsilon,\mathcal{A}_\perp}$ and $\mathsf{Exec}^{\mathsf{Ideal}_{\mathcal{F}_{\mathsf{psaPAKE}}}}_{\varepsilon,\mathcal{S}}$ respectively.

Capturing registrations. Recall that for simplicity, we assume that registration happens over secure channels. To capture the registration phase in our proof, we do the following: Before any authentication phase, the environment can register the users. Accordingly, we let the server in $\Pi_\star$ execute the corresponding registration phase on the receipt of a StorePwdFile message. In our hybrid argument, in Game 23, we ensure that $\mathcal{S}$ does not read the registration messages.

6.1 The Simulator

In this section, we go over the simulator $\mathcal{S}$ for Theorem 3. The full definition is given in Appendix C.

6.1.1 Random Oracle

In the simulation of H_1, for every output e of a fresh query tpw, we keep track of its exponent j such that $e = g_q^j$. We do this by first choosing j uniformly randomly from $\mathbb{Z}_q$ and returning g_q^j, which is shown to be uniformly random in Game 3.

In H_3, we abort the simulation if the adversary is able to query it with some input that is stored in a corresponding ForbiddenInput record. We show that an environment that can query H_3 in such a way wins GapDH in Game 28, Game 29, Game 30.

The dictionary attacks are captured in the simulation of H_2. For a key k, we call g_q^k its corresponding *key id*. Notice that for $\mathsf{H}_1(\mathsf{tpw}) = g_q^j$, if we have $\mathsf{H}_2(\mathsf{tpw}, b)$ where $b^{1/j} = g_q^k$ for some k, then this is an OPRF evaluation with some key k. If this is the case, and the evaluation is correct (i.e., tpw matches the user's actual password), we send back a key that decrypts the simulated ciphertext into the simulated keys using the equivocability simulator $\mathsf{SIM}_{\text{EQV}}$ (see Section 2.10).

```
▷ On ({tpw}) to H_1:
  1: If $\mathsf{H}_1[\mathsf{tpw}] = \bot$, then do:
    1.1: Choose $j \leftarrow\!\!\$\ \mathbb{Z}_q$.
    1.2: [Collision] If $\langle \textsc{HashedPwd}, *, j, * \rangle$ exists, then abort.
    1.3: Set $\mathsf{H}_1[\mathsf{tpw}] := g_q^j$.
    1.4: Record $\langle \textsc{HashedPwd}, \mathsf{tpw}, j, g_q^j \rangle$.
  2: Send $\mathsf{H}_1[\mathsf{tpw}]$ to $\mathcal{P}$.

▷ On ({tpw}, {b}) to H_2:
  1: Set $\mathsf{offline} :=$ (this query is made due to $\boxed{(\textsc{OfflineTestPwd}, sid, uid, \mathsf{tpw})}$ from $\varepsilon$).
  2: If $\mathsf{H}_2[(\mathsf{tpw}, b)] = \bot$, then do:
    2.1: Call $\mathsf{H}_1(\mathsf{tpw})$. // Ensure the HashedPwd record exists
    2.2: Get $\langle \textsc{HashedPwd}, \mathsf{tpw}, \{\mathbf{j}\}, \{\mathbf{h_1}\} \rangle$,
    2.3: [Key id extraction] Set $l := b^{1/j}$. // If $\exists k : b = \mathsf{H}_1(\mathsf{tpw})^k$, then $l = g_q^k$.
    2.4: If $\langle \textsc{RndPwd}, sid, \{\mathbf{uid}\}, \{\mathbf{k_s^*}\}, \{\mathbf{st}\} \rangle$ where $l = g_q^{k_s^*}$ exists, then do:
      2.4.1: If $\mathsf{offline} = \mathtt{true}$, then do:
        2.4.1.1: Send $(\textsc{OfflineTestPwd}, sid, uid, \mathsf{tpw})$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
      2.4.2: Else, do:
        2.4.2.1: Pop the first $\langle \textsc{BlindedEval}, sid, uid, l, \{\rho_\mathbf{s}\} \rangle$ or abort.
        2.4.2.2: Send $(\textsc{TestPwd}, sid, uid, \rho_s, \mathsf{tpw})$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
      2.4.3: [$\mathcal{A}$ guessed the password] If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "correct guess", then do:
        2.4.3.1: Record $\langle \textsc{StolenPwd}, sid, uid, \mathsf{tpw} \rangle$.
        2.4.3.2: Retrieve $\langle \textsc{UsrCreds}, sid, uid, \{\mathbf{p_u^*}\}, *, \{\mathbf{P_u^*}\}, \{\mathbf{P_s^*}\}, * \rangle$ or abort.
        2.4.3.3: Set $m := p_u^* \parallel P_u^* \parallel P_s^*$.
        2.4.3.4: Set $h_2 \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m, st)$.
    2.5: Else, do:
      2.5.1: Choose $h_2 \leftarrow\!\!\$\ \{0,1\}^\lambda$.
      2.5.2: [Future collision] If $\langle \textsc{RndPwd}, sid, \ldots, h_2 \rangle$ exists, then abort.
    2.6: [Collision] If $\mathsf{H}_2[(\mathsf{tpw}', b')] = h_2$ for some $(\mathsf{tpw}', b')$, then abort.
    2.7: Record $\langle \textsc{OprfEval}, \mathsf{tpw}, l, h_2 \rangle$.
    2.8: Set $\mathsf{H}_2[(\mathsf{tpw}, b)] := h_2$.
  3: Send $\mathsf{H}_2[(\mathsf{tpw}, b)]$ to $\mathcal{P}$.

▷ On ({x}, {y}, {z}) to H_3:
  1: If $\mathsf{H}_3[(x, y, z)] = \bot$, then do:
    1.1: If $\langle \textsc{ForbiddenInput}, sid, *, (x, \bot, \bot) \rangle$ exists, and both $\mathcal{P}_c$ and $\mathcal{P}_s$ are honest,
         then abort.
    1.2: Else if $\langle \textsc{ForbiddenInput}, sid, *, (\bot, y, \bot) \rangle$ exists, then abort.
    1.3: Else if $\langle \textsc{ForbiddenInput}, sid, \{\mathbf{uid}\}, (\bot, \bot, z) \rangle$ and $\langle \textsc{StolenPwd}, sid, uid, * \rangle$ ex-
         ists, then abort.
    1.4: Else if either $\langle \textsc{SvrPrfKey}, sid, (x, y, z), \{\mathbf{k}\} \rangle$ or $\langle \textsc{UsrPrfKey}, sid, *, (x, y, z), \{\mathbf{k}\} \rangle$
         exists, then set $h_3 := k$.
    1.5: Else, choose $h_3 \leftarrow\!\!\$\ \mathcal{K}$.
    1.6: Set $\mathsf{H}_3[(x, y, z)] := h_3$.
  2: Set $k := \mathsf{H}_3[(x, y, z)]$.
  3: Send $k$ to $\mathcal{P}$.
```

6.1.2 Honest Client

Note that we keep track of all fresh H_2 invocations in the form of OprfEval records. We use this information to decide whether the ciphertext received from the server would decrypt, and if so, to what values, without knowing the honest client's input. We also capture the impersonation attacks during the simulation of an honest client.

```
▷ On $(\textsc{UsrSession}, \{\mathbf{sid}\}, \{\mathbf{uid}\}, \{\rho_\mathbf{c}\}, *)$ from $\mathcal{F}_{\mathsf{psaPAKE}}$ to $\mathcal{A}_\perp$:
  1: Record $sid, uid$.
  2: Choose $\delta_c \leftarrow\!\!\$ \{0,1\}^\lambda$.
  3: If $\delta_c$ was chosen before, abort.
  4: Record $\langle\textsc{ClientRecord}, sid, uid, \rho_c, \delta_c\rangle$.
  5: Choose $x_u^* \leftarrow\!\!\$ \mathbb{Z}_p$, and set $X_u^* := g_p^{x_u^*}$.
  6: If $\langle\textsc{UsrCreds}, sid, uid, *, \{\mathbf{p_s^*}\}, \ldots\rangle$ exists, then do:
    6.1: Record $\langle\textsc{ForbiddenInput}, sid, uid, (\perp, X_u^{*p_s^*}, \perp)\rangle$
  7: Record $\langle\textsc{ClientState}, sid, uid, \delta_c, \rho_c, x_u^*, X_u^*\rangle$.
  8: Choose $r^* \leftarrow\!\!\$ \mathbb{Z}_q$, set $\alpha^* := g_q^{r^*}$
  9: Record $\langle\textsc{BlindedPwd}, sid, uid, \delta_c, r^*, \alpha^*\rangle$.
  10: Send $(\delta_c, X_u^*, \alpha^*)$ to $\mathcal{P}_s$.

▷ On $(\{\hat{\delta_\mathbf{s}}\}, \{\hat{\mathbf{X}}_\mathbf{s}\}, \{\hat{\beta}\}, \{\hat{\mathbf{c}}\}, \{\hat{\mathbf{A}}_\mathbf{s}\})$ from $\mathcal{A}_\perp$in response to $(\hat{\delta_c}, \ldots)$:
  1: Try to retrieve $\langle\textsc{ClientState}, sid, uid, \hat{\delta_c}, \{\mathbf{x_u^*}\}, \{\mathbf{X_u^*}\}\rangle$ or drop the message.
  2: Try to retrieve $\langle\textsc{BlindedPwd}, sid, uid, \hat{\delta_c}, \{\mathbf{r^*}\}, \{\alpha^*\}\rangle$ or abort.
  3: Try to retrieve $\langle\textsc{ClientRecord}, sid, uid, \{\rho_\mathbf{c}\}, \hat{\delta_c}\rangle$ or abort.
  4: [Ensure server session exists] Try to retrieve $\langle\textsc{ServerRecord}, sid, uid, \{\rho_\mathbf{s}\}, \hat{\delta_s}\rangle$ or do:
    4.1: Send $(\textsc{SvrSession}, sid, uid)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$, and receive $(\textsc{SvrSession}, sid, uid, \{\rho_\mathbf{s}\}, *)$.
    4.2: Record $\langle\textsc{ServerRecord}, sid, uid, \rho_s, \hat{\delta_s}\rangle$.
  5: [Honest execution] If $\mathcal{P}_s$ is honest
     and $\langle\textsc{RndPwd}, sid, uid, \{\mathbf{k_s^*}\}, \{\mathbf{st}\}\rangle$ where $\hat{\beta} = \alpha^{*k_s^*}$ exists,
     and $\langle\textsc{UsrCreds}, sid, uid, \{\mathbf{p_u^*}\}, *, \{\mathbf{P_u^*}\}, \{\mathbf{P_s^*}\}, *, \hat{c}\rangle$ exists,
     and $\langle\textsc{StolenKey}, sid, uid, \ldots\rangle$ doesn't exist, then do:
    5.1: [Check $\mathsf{upw} = \mathsf{spw}$] Send $(\textsc{TestAbort}, sid, uid, \rho_c, \rho_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
    5.2: [$\mathsf{upw} = \mathsf{spw}$] If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "success", then do:
      5.2.1: [$\mathcal{A}$ stole the password] If $\langle\textsc{StolenPwd}, sid, uid, \{\mathsf{spw}\}\rangle$ exists, then do:
        5.2.1.1: Send $(\textsc{TestPwd}, sid, uid, \rho_c, \mathsf{spw})$ to $\mathcal{F}_{\mathsf{psaPAKE}}$. // Compromise the user
      5.2.2: Jump to line 9.
    5.3: [$\mathsf{upw} \neq \mathsf{spw}$] Else, stop. // Already aborted by $\mathcal{F}_{\mathsf{psaPAKE}}$
  6: [Adversarial execution] Else, do:
    6.1: Find the randomized password $y$ such that:
      6.1.1: At least one of the following hold:
        6.1.1.1: [$\hat{\beta}$ is well-formed] $\langle\textsc{OprfEval}, \{\mathbf{x}\}, *, \{\mathbf{y}\}\rangle$ where $y = \mathsf{H}_2[(x, \hat{\beta}^{1/r^*})]$ exists.
        6.1.1.2: [$\hat{\beta}$ is from honest $\mathcal{P}_s$] $\langle\textsc{RndPwd}, sid, uid, \{\mathbf{k_s^*}\}, *\rangle$ where $\alpha^{*k_s^*} = \hat{\beta}$ exists.
      6.1.2: $\mathsf{AE.AuthDec}_y(\hat{c}) \neq \perp$.
    6.2: [Random-key robustness] If multiple different such values exist, then abort.
    6.3: [$\hat{c}$ decrypts with queried $y$] Else if $\langle\textsc{OprfEval}, \{\mathsf{apw}\}, *, \{\mathbf{y'}\}\rangle$ exists, then do:
      6.3.1: Set $y = y'$.
      6.3.2: [Check $\mathsf{upw} = \mathsf{apw}$] Send $(\textsc{TestPwd}, sid, uid, \rho_c, \mathsf{apw})$ to $\mathcal{F}_{\mathsf{psaPAKE}}$
      6.3.3: [$\mathsf{upw} \neq \mathsf{apw} \rightarrow \mathcal{P}_c$ won't get $y$] If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "wrong guess",
             then set $y := \perp$.
    6.4: [$\hat{c}$ decrypts with $y^*$] Else if only $\langle\textsc{RndPwd}, sid, uid, *, \{\mathbf{st}\}\rangle$ exists, then do:
      6.4.1: Retrieve $\langle \textsc{UsrCreds}, sid, uid, \{\mathbf{p_u^*}\}, *, \{\mathbf{P_u^*}\}, \{\mathbf{P_s^*}\}, * \rangle$ or abort.
      6.4.2: Set $\hat{p_u} := p_u^*$, $\hat{P}_u := P_u^*$, $\hat{P}_s := P_s^*$.
      6.4.3: [Check $\mathsf{upw} = \mathsf{spw}$] Send $(\textsc{Impersonate}, sid, uid, \rho_c)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
      6.4.4: [$\mathsf{upw} \neq \mathsf{spw} \to \mathcal{P}_c$ can't decrypt] If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "wrong guess",
             then set $y := \perp$.
      6.4.5: Else, jump to line 9.
 7: [No $y$ can decrypt $\hat{c}$ with $\mathsf{upw}$] If $y = \perp$, then do:
   7.1: Send $(\textsc{TestPwd}, sid, uid, \rho_c, \perp)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$. // Interrupt the user "session".
   7.2: Send $(\textsc{TestAbort}, sid, uid, \rho_c, \rho_s, \mathcal{P}_c)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$, and stop.
 8: Try to decrypt $(\hat{p_u}, \hat{P}_u, \hat{P}_s) := \mathsf{AE.AuthDec}_y(\hat{c})$ or abort.
 9: Set $\mathsf{in\_keys} := (\hat{X_s}^{x_u^*}, \hat{P_s}^{x_u^*}, \hat{X_s}^{\hat{p_u}})$.
10: If $\langle \textsc{SvrPrfKey}, sid, \mathsf{in\_keys}, \{\mathbf{k}\} \rangle$ exists, then set $K^* := k$.
11: Else, choose $K^* \leftarrow\!\$\ \mathcal{K}$.
12: Record $\langle \textsc{UsrPrfKey}, sid, uid, \mathsf{in\_keys}, K^* \rangle$.
13: If $\langle \textsc{SvrState}, sid, uid, \hat{\delta_s}, \delta_c, *, \{\mathbf{X_s^*}\}, \ldots \rangle$ exists and $X_s^* \neq \hat{X}_s$,
    then send $(\textsc{TestPwd}, sid, uid, \delta_c, \perp)$ to $\mathcal{F}$.
14: Set $\mathsf{tx} := (sid \parallel uid \parallel \delta_c \parallel \hat{\delta_s} \parallel \alpha^*)$.
15: Set $SK^* := F_{K^*}^{(0)}(\mathsf{tx})$.
16: If $\mathcal{P}_s$ is honest, then do:
   16.1: Try to retrieve $\langle \textsc{SvrTag}, sid, uid, K^*, \mathsf{tx}, \{\mathbf{A_s^*}\} \rangle$ or set $A_s^* := \perp$.
   16.2: Choose $A_u^* \leftarrow\!\$\ \{0,1\}^\lambda$.
   16.3: Record $\langle \textsc{UsrTag}, sid, uid, \rho_c, \rho_s, A_u^* \rangle$.
17: Else, set $A_s^* := F_{K^*}^{(1)}(\mathsf{tx})$, $A_u^* := F_{K^*}^{(2)}(\mathsf{tx})$.
18: If $A_s^* \neq \hat{A}_s$, then do:
   18.1: Send $(\textsc{TestPwd}, sid, uid, \rho_c, \perp)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
   18.2: Send $(\textsc{TestAbort}, sid, uid, \rho_c, \rho_s, \mathcal{P}_c)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$, and stop.
19: Send $(\textsc{NewKey}, sid, uid, \rho_c, \rho_s, \mathsf{SK}^*, \mathcal{P}_c, \mathcal{P}_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
20: Send $A_u^*$ to $\mathcal{P}_s$.
```

6.1.3 Honest Server

During the simulation of an honest server, we capture the server interruption attacks, in which the adversary delivers a faulty user tag A_u that would make an honest server abort. Additionally, when the adversary modifies the user ephemeral key X_u sent by an honest client, we consider this as a server interruption attack as well.

```
▷ On $\boxed{(\textsc{StealPwdFile}, sid, \{\mathbf{uid}\})}$ from $\mathcal{A}_\perp$:

 1: Send $\langle \textsc{StealPwdFile}, sid, uid \rangle$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
 2: If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "password file stolen", then do:
   2.1: Try to retrieve $\langle \textsc{RndPwd}, sid, uid, \{\mathbf{k_s^*}\}, * \rangle$ or abort.
   2.2: Record $\langle \textsc{StolenKey}, sid, uid, k_s^*, g_q^{k_s^*} \rangle$.
 3: Forward $\mathcal{F}_{\mathsf{psaPAKE}}$'s response to $\mathcal{A}_\perp$.

▷ On $(\textsc{SvrSession}, sid, \{\mathbf{uid}\}, \{\rho_\mathbf{s}\})$ from $\mathcal{F}_{\mathsf{psaPAKE}}$ to $\mathcal{A}_\perp$:

 1: Choose $\delta_s \leftarrow\!\$\ \{0,1\}^\lambda$, and associate $\rho_s$ with $\delta_s$.
 2: If $\delta_s$ was chosen before, abort.
 3: Record $\langle \textsc{ServerRecord}, sid, uid, \rho_s, \delta_s \rangle$.
 4: Record $\langle \textsc{NewSvrSession}, sid, uid, \rho_s, \delta_s \rangle$.
 5: If $\langle \textsc{UsrCreds}, sid, uid, ... \rangle$ does not exist, then do:
    5.1: Choose $p_u^*, p_s^* \leftarrow\!\!\$\ \mathbb{Z}_p$, and set $P_s^* := g_p^{p_s^*}$, $P_u^* := g_p^{p_u^*}$.
    5.2: Set $m^* := p_u^* \parallel P_u^* \parallel P_s^*$.
    5.3: Set $(c^*, st) := \mathsf{SIM}_{\mathrm{EQV}}(|m|)$.
    5.4: Record $\langle \textsc{UsrCreds}, sid, uid, p_u^*, p_s^*, P_u^*, P_s^*, c^* \rangle$.
    5.5: Choose $k_s^* \leftarrow\!\!\$\ \mathbb{Z}_q$.
    5.6: Record $\langle \textsc{RndPwd}, sid, uid, k_s^*, st \rangle$.

▷ On $(\{\mathbf{uid}\}, \{\hat{\delta}_\mathbf{c}\}, \{\hat{\mathbf{X}}_\mathbf{u}\}, \{\hat{\alpha}\})$ from $\mathcal{A}_\perp$:
  1: Send $(\textsc{SvrSession}, sid, uid)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
  2: Pop the first $\langle \textsc{NewSvrSession}, sid, uid, \{\rho_\mathbf{s}\}, \{\delta_\mathbf{s}\} \rangle$.
  3: Try to retrieve $\langle \textsc{UsrCreds}, sid, uid, \{\mathbf{p_u^*}\}, \{\mathbf{p_s^*}\}, \{\mathbf{P_u^*}\}, \{\mathbf{P_s^*}\}, \{\mathbf{c}^*\} \rangle$ or abort.
  4: Try to retrieve $\langle \textsc{RndPwd}, sid, uid, \{\mathbf{k_s^*}\}, * \rangle$ or abort.
  5: [Ensure user session exists] Try to retrieve $\langle \textsc{ClientRecord}, sid, uid, \{\rho_\mathbf{c}\}, \hat{\delta}_c \rangle$ or do:
    5.1: Send $(\textsc{UsrSession}, sid, uid, \perp)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$,
         and receive $(\textsc{UsrSession}, sid, uid, \{\rho_\mathbf{c}\}, *)$.
    5.2: Record $\langle \textsc{ClientRecord}, sid, uid, \rho_c, \hat{\delta}_c \rangle$.
  6: Set $\beta^* := \hat{\alpha}^{k_s^*}$.
  7: Choose $x_s^* \leftarrow\!\!\$\ \mathbb{Z}_p$, and set $X_s^* := g_p^{x_s^*}$.
  8: If $\langle \textsc{ClientState}, sid, uid, \hat{\delta}_c, *, *, \{\mathbf{X_u^*}\} \rangle$ exists, then do:
    8.1: Record $\langle \textsc{ForbiddenInput}, sid, uid, (X_u^{*x_s^*}, \perp, \perp) \rangle$.
    8.2: If $X_u^* \neq \hat{X}_u$, then send $(\textsc{Interrupt}, sid, uid, \rho_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
  9: Record $\langle \textsc{ForbiddenInput}, sid, uid, (\perp, \perp, P_u^{*x_s^*}) \rangle$.
 10: Set $\mathsf{in\_keys} := (\hat{X}_u^{\,x_s^*}, \hat{X}_u^{\,p_s^*}, P_u^{*x_s^*})$.
 11: Choose $K^* \leftarrow\!\!\$\ \mathcal{K}$.
 12: Record $\langle \textsc{SvrPrfKey}, sid, \mathsf{in\_keys}, K^* \rangle$.
 13: Set $\mathsf{tx} := (sid \parallel uid \parallel \hat{\delta}_c \parallel \delta_s \parallel \hat{\alpha})$,
 14: If $\mathcal{P}_c$ is honest, then do:
   14.1: Choose $A_s^* \leftarrow\!\!\$\ \{0,1\}^\lambda$.
   14.2: Record $\langle \textsc{SvrTag}, sid, uid, \rho_c, \rho_s, A_u^* \rangle$.
 15: Else, set $A_s^* := F_{K^*}^{(1)}(\mathsf{tx})$.
 16: Record $\langle \textsc{SvrState}, sid, uid, \delta_s, \hat{\delta}_c, c^*, X_s^*, K^*, \mathsf{tx}, A_s^* \rangle$.
 17: Record $\langle \textsc{BlindedEval}, sid, uid, g_q^{k_s^*}, \rho_s \rangle$.
 18: Send $(\delta_s, X_s^*, c^*, A_s^*, \beta^*)$ to $\mathcal{P}_c$.

▷ On $(\{\hat{\mathbf{A}}_\mathbf{u}\})$ from $\mathcal{A}_\perp$in response to $(\hat{\delta}_s, \ldots)$:
  1: Try to retrieve $\langle \textsc{SvrState}, sid, uid, \hat{\delta}_s, \{\delta_\mathbf{c}\}, *, *, \{\mathbf{K}^*\}, \{\mathsf{tx}\}, * \rangle$ or drop the message.
  2: Try to retrieve $\langle \textsc{ServerRecord}, sid, uid, \{\rho_\mathbf{s}\}, \hat{\delta}_s \rangle$ or abort.
  3: Try to retrieve $\langle \textsc{ClientRecord}, sid, uid, \{\rho_\mathbf{c}\}, \delta_c \rangle$ or abort.
  4: If $\mathcal{P}_c$ is honest, then try to retrieve $\langle \textsc{UsrTag}, sid, uid, K^*, \mathsf{tx}, \{\mathbf{A_u^*}\} \rangle$ or set $A_u^* := \perp$.
  5: Else, set $A_u^* := F_{K^*}^{(2)}(\mathsf{tx})$
  6: Set $\mathsf{SK}^* := F_{K^*}^{(0)}(\mathsf{tx})$.
  7: If $A_u^* = \hat{A}_u$, then do:
    7.1: Send $(\textsc{NewKey}, sid, uid, \mathsf{SK}^*, \rho_s, \rho_c, \mathcal{P}_s, \mathcal{P}_c)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
  8: Else, do:
    8.1: Send $(\textsc{Interrupt}, sid, uid, \rho_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
    8.2: Send $(\textsc{TestAbort}, sid, uid, \rho_s, \rho_c, \mathcal{P}_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
```

6.2 Indistinguishability Argument

In this section, we show that the simulator introduced in Section 6.1 satisfies Theorem 3. To prove the secrecy of the password, we let $\mathcal{S}$ discard the password in Game 33. To capture the fact that registration phase happens over secure channels, we let $\mathcal{S}$ fully simulate the password file in Game 20 and merely forward the registration messages in Game 23.

6.2.1 Notation

For a game numbered i, $\Pr[\mathbf{G}_i]$ denotes the output of ε being 1 for that game. Accordingly, for a game numbered $i > 0$, the *gain* in the distinguishing advantage of ε in that game is

$$|\Pr[\mathbf{G}_i] - \Pr[\mathbf{G}_{i-1}]|.$$

If we have

$$|\Pr[\mathbf{G}_i] - \Pr[\mathbf{G}_{i-1}]| = 0,$$

then we say that game i is *perfectly indistinguishable* from the previous one.

6.2.2 Distinguishing Aborts

In this argument, we use the term *graceful abort* to distinguish an abort specified in the protocol description (e.g., when $\mathcal{P}_c$ fails to decrypt the ciphertext) from an abort in the simulation definition, which ends the simulation altogether. A party that gracefully aborts stops executing until the next subsession, whereas in the case of a simulation abort, no code is executed at all after that point. We additionally assume that, on a graceful abort, the parties output an appropriate ABORT message to the environment. An environment that observes the abort of a simulation infers with certainty that it is not in the real world. For this reason, in many of our game hops, our aim is to provide an upper bound on the probability of simulation aborts.

6.2.3 Game Hops

In this section, we give the game hops. In each game hop, either we give a bound on the introduced distinguishing advantage, or we argue that the modifications lead to no distinguishing advantage, i.e., perfectly indistinguishable.

Game 0 We start with a game that captures the real world execution. To that end, we define the environment machine ε, the random oracle functionality $\mathcal{F}_{\mathsf{RO}}$, and the parties of $\Pi_\star$ as $\mathcal{P}_s$ and $\mathcal{P}_c$ that can freely communicate with each other. We let ε provide the inputs to $\mathcal{P}_s$ and $\mathcal{P}_c$. In turn, the parties execute $\Pi_\star$ and return their outputs to ε.
We include an adversary machine $\mathcal{A}$ that can intercept the messages among the parties and freely communicates with ε over a two-way back-channel. Before the start of a subsession, ε may request $\mathcal{A}$ to corrupt a party $\mathcal{P}$. Then, $\mathcal{A}$ takes full control over $\mathcal{P}$ for the remaining subsessions.

Game 1 We introduce two new machines: $\mathcal{F}$ and $\mathcal{S}$. We set $\mathcal{F}$ to be an empty machine. We move the machines $\mathcal{F}_{\mathsf{RO}}$, $\mathcal{P}_s$, $\mathcal{P}_c$, and $\mathcal{A}$ into $\mathcal{S}$. We let $\mathcal{S}$ run these machines internally and forward the messages to their destination. Since $\mathcal{F}$ does nothing and $\mathcal{S}$ merely forwards the messages, this game is perfectly indistinguishable from the previous game.
Note that the inputs and the registration messages are known to $\mathcal{S}$ at this point.

Game 2 We let $\mathcal{S}$ respond to the H_1, H_2, H_3 queries. To do that, we update $\mathcal{S}$ as follows:

```
▷ On internal call $\mathsf{sample}(\{\mathcal{S}\})$:
  1: Choose $v \leftarrow\!\!\$\ \mathcal{S}$.
  2: Return $v$.
▷ On $(\{\mathbf{x}\})$ to $\mathsf{H}_1$:
  1: If $\mathsf{H}_1[x] = \bot$, then do:
    1.1: Set $h_1 \leftarrow \mathsf{sample}(\mathbb{G}_q)$.
    1.2: Set $\mathsf{H}_1[x] := h_1$.
  2: Send $\mathsf{H}_1[x]$ to $\mathcal{P}$.
▷ On $(\{\mathbf{x}\}, \{\mathbf{y}\})$ to $\mathsf{H}_2$:
  1: If $\mathsf{H}_2[(x, y)] = \bot$, then do:
    1.1: Set $h_2 \leftarrow \mathsf{sample}(\{0,1\}^\lambda)$.
    1.2: Set $\mathsf{H}_2[(x, y)] := h_2$.
  2: Send $\mathsf{H}_2[(x, y)]$ to $\mathcal{P}$.
▷ On $(\{\mathbf{x}\}, \{\mathbf{y}\}, \{\mathbf{z}\})$ to $\mathsf{H}_3$:
  1: If $\mathsf{H}_3[(x, y, z)] = \bot$, then do:
    1.1: Choose $h_3 \leftarrow\!\!\$\ \mathcal{K}$.
    1.2: Set $\mathsf{H}_3[(x, y, z)] := h_3$.
  2: Send $\mathsf{H}_3[(x, y, z)]$ to $\mathcal{P}$.
```

Since $\mathcal{S}$ simulates the exact specification of $\mathcal{F}_{\mathsf{RO}}$, this game is perfectly indistinguishable from the previous game.

Game 3 We add a new internal procedure $\mathsf{sample\_exp}$ and modify the simulation of H_1 as follows:

```
▷ On internal call $\mathsf{sample\_exp}(\{\mathbf{g}\}, \{\mathcal{S}\})$:
  1: Choose $v \leftarrow\!\!\$\ \mathcal{S}$.
  2: Return $(v, g^v)$.
▷ On $(\{\mathbf{x}\})$ to $\mathsf{H}_1$:
  1: If $\mathsf{H}_1[x] = \bot$, then do:
    1.1: Set $(j, h_1) \leftarrow \mathsf{sample\_exp}(g_q, \mathbb{Z}_q)$.
    1.2: Record $\langle \textsc{HashedPwd}, j, h_1 \rangle$.
    1.3: Set $\mathsf{H}_1[x] := h_1$.
  2: Send $\mathsf{H}_1[x]$ to $\mathcal{P}$.
```

Note that g_q generates $\mathbb{G}_q$, i.e., $f(i) = g_q^i$ is a bijective function from $\mathbb{Z}_q$ to $\mathbb{G}_q$. Since f is a bijective function, and keeping internal records do not change the outputs, this game is perfectly indistinguishable from the previous one.

Game 4 We modify the simulation of H_2 as follows: For a new input (x, y), we get the exponent j from the record $\langle \textsc{HashedPwd}, \{\mathbf{j}\}, \mathsf{H}_1(x) \rangle$, i.e., $\mathsf{H}_1(x) = g_q^j$. Then, we set $l := y^{1/j}$. Finally, we record $\langle \textsc{OprfEval}, x, l, h_2 \rangle$ and set $\mathsf{H}_2(x, y) := h_2$ where h_2 is the output of sample as before. Note that these changes do not change the output of H_2. Furthermore, $\mathcal{S}$ does not read $\textsc{OprfEval}$ records at this point. Hence, this game is perfectly indistinguishable from the previous one.

Game 5 To the simulation of H_1 and H_2 we add a collision check: If an output that we generated was already chosen for some other input, i.e., we encounter a collision, we abort the simulation.

▷ Let h_i denote the number of *unique* queries made to H_i.

- ▷ Let $h = \sum_i h_i$ be the total number of unique random oracle queries.
- ▷ Let abort_i denote the event of an abort due to a collision at H_i.
- ▷ Let abort denote the event of an abort due to a collision.

Due to birthday bound, we have

$$\Pr[\mathsf{abort}_1] \leq \frac{h_1^2}{\|\mathbb{G}_q\|}, \Pr[\mathsf{abort}_2] \leq \frac{h_2^2}{2^\lambda},$$

Due to union bound and $\|\mathbb{G}_q\| \geq 2^\lambda$, we have

$$\begin{aligned} |\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| &= \Pr[\mathsf{abort}] \\ &\leq \Pr[\mathsf{abort}_1] + \Pr[\mathsf{abort}_2] \\ &\leq \frac{h^2}{2^\lambda}. \end{aligned}$$

Game 6 Once $\mathcal{P}_c$ receives a ciphertext $\hat{c}$, we do the following:
If there exists $\langle \textsc{OprfEval}, x_1, *, y_1 \rangle$ and $\langle \textsc{OprfEval}, x_2, *, y_2 \rangle$ such that $x_1 \neq x_2$ and

$$\mathsf{AuthDec}_{y_1}(\hat{c}) \neq \bot \ \textbf{and} \ \mathsf{AuthDec}_{y_2}(\hat{c}) \neq \bot$$

then we abort the simulation. Notice that due to Game 5, we have $y_1 \neq y_2$.
We modify the simulator from the previous game to build $\mathcal{A}_{\mathsf{RKR}}$ such that it plays Exp-RKR (see Figure 2.6) as follows:

```
▷ On ({k1}, {k2}) from the challenger:
  1: For each $i \in [h_2]$, set $r_i := \bot$.
  2: Choose $i, j \leftarrow\$ [h_2]$ such that $i \neq j$.
  3: Update $r_i := k_1$ and $r_j := k_2$.
  4: Start the simulation of the previous game.
  5: Wait for $\mathcal{P}_c$ to receive $\hat{c}$.
  6: Return $\hat{c}$.
▷ On internal call sample($\{0,1\}^\lambda$) during $k$th fresh invocation of $\mathsf{H}_2$:
  1: If $r_k \neq \bot$, then return $r_k$. // $r_k$ is either $k_1$ or $k_2$, chosen randomly by the challenger.
  2: Else, return a uniformly random value from $\{0,1\}^\lambda$.
```

Notice that $\mathcal{A}_{\mathsf{RKR}}$ wins if all the following hold:

1. The first subsession uses $\mathsf{H}_2[x_1] = k_1$ and $\mathsf{H}_2[x_2] = k_2$ for some x_1, x_2.
2. $\mathcal{A}$ sends a $\hat{c}$ that would lead to an abort in this game.

Let abort denote the event that we abort in this game. The advantage of $\mathcal{A}_{\mathsf{RKR}}$ is then bounded as

$$\mathsf{Adv}^{\text{Exp-RKR}}_{\mathcal{A}_{\mathsf{RKR}},\mathsf{AE}}(\lambda) \geq \binom{h_2}{2}^{-1} \cdot \Pr[\mathsf{abort}].$$

Then we have,

$$\begin{aligned} |\Pr[\mathbf{G}_6] - \Pr[\mathbf{G}_5]| &= \Pr[\mathsf{abort}] \\ &\leq \binom{h_2}{2} \cdot \mathsf{Adv}^{\text{Exp-RKR}}_{\mathcal{A}_{\mathsf{RKR}},\mathsf{AE}}(\lambda) \\ &\leq h_2^2 \cdot \mathsf{Adv}^{\text{Exp-RKR}}_{\mathcal{A}_{\mathsf{RKR}},\mathsf{AE}}(\lambda) \end{aligned}$$

Game 7 We first add a new internal procedure $\mathsf{extract\_key\_id}$ to $\mathcal{S}$:

```
▷ On internal call $\mathsf{extract\_key\_id}(\{\mathsf{tpw}\}, \{\mathbf{b}\})$:
  1: Call $\mathsf{H}_1(\mathsf{tpw})$. // Ensure HashedPwd exists
  2: Retrieve $\langle$HashedPwd$, \{\mathbf{j}\}, \mathsf{H}_1[\mathsf{tpw}]\rangle$.
  3: Return $b^{1/j}$.
```

Then we make the following modifications to $\mathcal{S}$:

- ▷ **At the simulation of honest $\mathcal{P}_s$,** Once $\mathcal{S}$ gives out a blinded evaluation $\beta = \alpha^k$ for an OPRF key k generated for the user *uid*, we record $\langle$BlindedEval$, sid, uid, g_q^k, \rho_s\rangle$ where ρ_s is the associated partial subsession identifier for the honest server.
- ▷ **At the simulation of H_2,** Once H_2 receives a fresh query (tpw, b) not in response to a OfflineTestPwd message, we set $l := \mathsf{extract\_key\_id}(\mathsf{tpw}, b)$. Notice that if $b = \mathsf{H}_1(\mathsf{tpw})^k$ for some k, then we have $l = g_q^k$. Then we pop the first $\langle$BlindedEval$, sid, uid, l, *\rangle$ record. If no such record exists, then we continue as usual.

In this game, we only create and remove records which do not augment the observable behavior of the simulation. Hence, this game is perfectly indistinguishable from the previous one.

Game 8 Now, we add a new abort condition: During the simulation of H_2, when $\mathcal{S}$ tries to pop the first BlindEval record, and it does not exist, we abort the simulation.
We modify the simulator from the previous game to build $\mathcal{A}_{\mathsf{OMDH}}$ that plays OMDH(N, Q) (see Figure 2.3) as follows:

```
▷ On $(\{\mathbf{g_q}\}, \{\mathbf{g_q^k}\}, \{\mathbf{g_1}\}, \ldots, \{\mathbf{g_N}\})$ from the challenger:
  1: If $N < h_1$, then abort.
  2: For each $i \in [h_1]$, set $r_i := \bot$.
  3: Choose unique $\mathsf{idx}[1], \ldots, \mathsf{idx}[N] \leftarrow\!\$\ [h_1]$.
  4: For each $i \in [N]$, update $r_{\mathsf{idx}[i]} := g_i$.
  5: Set $\mathsf{issued} := 0$.
  6: Start the simulation of the previous game.
  7: Wait until $\mathsf{issued} >= Q$.
  8: For each $\mathsf{H}_2[(*, b)] = y$ such that $\mathsf{DDH}(g_i, g_q^k, b) = 1$, do:
    8.1: Set $e_i := b$.
  9: Return $\{(g_i, e_i)\}_i$.
▷ On internal call $\mathsf{sample\_exp}(g_q, \mathbb{Z}_q)$ during $k$th fresh invocation of $\mathsf{H}_1$:
  1: If $r_k \neq \bot$, then return $(\bot, r_k)$. // $r_k = g_i$ for some $i$ randomly chosen by the chal.
  2: Else, do:
    2.1: Choose $j \leftarrow\!\$\ \mathbb{Z}_q$.
    2.2: Return $(j, g_q^j)$.
▷ On internal call $\mathsf{extract\_key\_id}(\{\mathsf{tpw}\}, \{\mathbf{b}\})$:
  1: Call $\mathsf{H}_1(\mathsf{tpw})$.
  2: Retrieve $\langle$HashedPwd$, \{\mathbf{j}\}, \mathsf{H}_1[\mathsf{tpw}]\rangle$.
  3: If $j \neq \bot$, then return $b^{1/j}$.
  4: Else, return $g_q^k$. // We must have $\mathsf{H}_1[\mathsf{tpw}] = g_i$ for some $i$.
▷ On $(*, *, *, \{\alpha\})$ from $\mathcal{P}_c$:
  1: If $\mathsf{issued} < Q$, then do:
    1.1: Update $\mathsf{issued} := \mathsf{issued} + 1$.
    1.2: Run $\mathcal{S}$ to get the response of $\mathcal{P}_s$, $(*, *, \{\beta\}, *, *)$.
    1.3: Retrieve the new record $\langle$BlindedEval$, *, *, \{\mathbf{l}\}, *\rangle$, and update $l := g_q^k$.
    1.4: Update $\beta := \mathsf{Exp}_Q^k(\alpha)$.
    1.5: Send the updated response to $\mathcal{P}_c$.
```

- ▷ Let s_s denote the number of honest server partial subsessions.

▷ Let s_u denote the number of honest user partial subsessions.

Notice that s_s coincides with the total number of generated BLINDEVAL records, and s_u coincides with the maximum number of consumed BLINDEVAL records. We let $N = h_1 + s_u$ and $Q = s_s$. Then, we see that $\mathcal{A}_{\mathsf{OMDH}}$ wins OMDH($h_1 + s_u$, s_s) if all the following hold:

1. In the first s_s many subsessions, ε uses as input one of $x_1, \ldots, x_{h_1+s_u}$.
2. There are $s_s + 1$ many unique $\mathsf{H}_2[(*, b)] = y$ such that $b = g_i^k$.

Let abort denote the event that we abort in this game. This happens when we "consume" one more than the number of BLINDEVAL records created, which is precisely cond. §2. The advantage of $\mathcal{A}_{\mathsf{OMDH}}$ is then bounded as

$$\mathsf{Adv}^{\mathrm{OMDH}(h_1 + s_u,\, s_s)}_{\mathcal{A}_{\mathsf{OMDH}},(\mathbb{G}_q,g_q,q)}(\lambda) \geq \binom{h_1 + s_u}{s_s}^{-1} \cdot \Pr[\mathsf{abort}].$$

Hence, we have

$$|\Pr[\mathbf{G}_8] - \Pr[\mathbf{G}_7]| \leq \binom{h_1 + s_u}{s_s} \cdot \mathsf{Adv}^{\mathrm{OMDH}(h_1 + s_u,\, s_s)}_{\mathcal{A}_{\mathsf{OMDH}},(\mathbb{G}_q,g_q,q)}(\lambda).$$

Game 9 In this game, we let $\mathcal{S}$ create records during the simulation of honest $\mathcal{P}_c$, $\mathcal{P}_s$.

▷ **During the simulation of honest $\mathcal{P}_c$,** right before $\mathcal{S}$ sends the first message of a subsession, $\mathcal{S}$ creates a BLINDEDPWD record to store the chosen blinding exponent r and the blinded password α. At the same time, it stores the chosen ephemeral private key x_s and the corresponding public key X_s in a CLIENTSTATE record.

▷ **During the simulation of honest $\mathcal{P}_s$,** for every registered user uid, $\mathcal{S}$ keeps track of their OPRF key k_s and randomized password y by creating a $\langle \text{RNDPWD}, sid, uid, k_s, y\rangle$ record on user registration. Notice that k_s is already included in plaintext in the server file, and y can be computed from the user's password, which is at this point known to $\mathcal{S}$. Furthermore, we let $\mathcal{S}$ keep track of the long-term keys p_u, p_s, P_u, P_s and the associated ciphertext $c = \mathsf{AuthEnc}_y(p_u \parallel P_u \parallel P_s)$ in the form of USRCREDS records, in which the values are similarly known to $\mathcal{S}$.

In this game, we only create records and never read them. Therefore, this game is perfectly indistinguishable from the previous one.

Game 10 We directly add the USRSESSION, SVRSESSION, STOREPWDFILE, OFFLINETESTPWD, and STEALPWDFILE interfaces of $\mathcal{F}_{\mathsf{psaPAKE}}$ to $\mathcal{F}$. We make the following modifications to $\mathcal{S}$:

▷ **Registration:** Once the environment ε sends a STOREPWDFILE message to $\mathcal{P}_s$, we let $\mathcal{S}$ forward this message to both $\mathcal{P}_s$ and $\mathcal{F}$. This ensures that $\mathcal{F}$ has all the registered users in its internal FILE records. Since $\mathcal{F}$ does nothing with these records, and $\mathcal{P}_s$ also receives this message as before, this change is perfectly indistinguishable.

▷ **Adaptive compromise:** Once the environment ε sends a STEALPWDFILE or OFFLINETESTPWD message to $\mathcal{A}$, we let $\mathcal{S}$ forward this message to both $\mathcal{A}$ and $\mathcal{F}$, and then $\mathcal{S}$ returns the answer of $\mathcal{F}$ to ε on behalf of $\mathcal{A}$. Since $\mathcal{F}$ now has the correct FILE records, $\mathcal{F}$ produces the same output as $\mathcal{A}$ used to do in the previous game. This change is therefore perfectly indistinguishable. Additionally, on a STEALPWDFILE request, we let $\mathcal{S}$ record $\langle \text{STOLENKEY}, sid, uid, k_s, g^{k_s}\rangle$ where k_s is the stolen OPRF key. Since this doesn't lead to a change in the behavior of $\mathcal{S}$, this change is perfectly indistinguishable.

▷ **Honest initialization:** On the construction of honest $\mathcal{P}_c$, $\mathcal{S}$ sends the corresponding USRSESSION message to $\mathcal{F}$. Once honest $\mathcal{P}_s$ receives an authentication request, $\mathcal{S}$ sends the corresponding SVRSESSION message to $\mathcal{F}$. This ensures that $\mathcal{F}$ has all the honest partial subsessions internally recorded as SESSION records. We let $\mathcal{S}$ associate the ρ

values returned from the $\mathcal{F}$ with the generated nonces in the form of ServerRecord and ClientRecord records. Since this doesn't lead to a change in the behavior of $\mathcal{S}$, this change is perfectly indistinguishable.

- ▷ **Adversarial initialization:** Once honest $\mathcal{P}_s$ receives an authentication request with an unknown nonce, $\mathcal{S}$ sends the corresponding UsrSession message to $\mathcal{F}$. Once honest $\mathcal{P}_c$ receives an authentication response with an unknown nonce, $\mathcal{S}$ sends the corresponding SvrSession message to $\mathcal{F}$. This ensures that $\mathcal{F}$ has all the adversarial partial subsessions internally recorded. We let $\mathcal{S}$ associate ρ values returned from $\mathcal{F}$ with these adversarial nonces in the form of ServerRecord and ClientRecord records. Since this doesn't lead to a change in the behavior of $\mathcal{S}$, this change is perfectly indistinguishable.

Game 11 If $\mathcal{A}$ has compromised the file associated with the user *uid*, i.e., $\langle$StolenKey, $sid, uid, \ldots\rangle$ exists, we now respond differently to the queries to H_2 as follows: On an input (tpw, y), we check if $y = \mathsf{H}_1(\mathsf{tpw})^{k_s}$ for some OPRF key k_s for a user *uid*. If that's the case and the query is being made in response to a (OfflineTestPwd, sid, uid, tpw) message, then y is allowed to be *locally* computed with the stolen k_s. Hence, we forward (OfflineTestPwd, sid, uid, tpw) to $\mathcal{F}$.
Note that ε can issue an OfflineTestPwd only after stealing the password, i.e., File record is marked as `Compromised`. Accordingly, $\mathcal{F}$ returns "correct guess" if and only if $\mathsf{tpw} = \mathsf{spw}$ where spw is the password for *uid* registered at the server.
This change does not lead to a change in the observable behavior of the simulation, so this game is perfectly indistinguishable from the previous game.

Game 12 Now, we add the active session attack interfaces to $\mathcal{F}$. We add Impersonate directly from $\mathcal{F}_{\mathsf{psaPAKE}}$ to $\mathcal{F}$. We add the following version of Interrupt, and TestPwd to $\mathcal{F}$ which are exactly as in $\mathcal{F}_{\mathsf{psaPAKE}}$ but the parts concerning the dPT flag are stripped:

```
▷ On (Interrupt, {sid}, {uid}, {$\delta$}) from $\mathcal{A}$:
  1: If ⟨Session, $sid, uid, \delta, *,$ Fresh⟩ exists, then do:
    1.1: Update the record's status to Interrupted.
▷ On (TestPwd, {sid}, {uid}, {$\delta$}, {tpw}) from $\mathcal{A}$:
  1: Try to retrieve the session record ⟨Session, $sid, uid, \delta$, {ty}, {pw}, {st}⟩ or drop the message.
  2: If $\mathsf{ty} = $ Svr and $\mathsf{tpw} = \mathsf{pw}$, set $\mathsf{pwstolen}[(sid, uid)] := 1$.
  3: If $\mathsf{st} = $ Fresh, then do:
    3.1: If $\mathsf{tpw} = \mathsf{pw}$, then do:
      3.1.1: Update the Session record's status to Compromised.
      3.1.2: Send "correct guess" to $\mathcal{A}$.
    3.2: Else, do:
      3.2.1: Update the Session record's status to Interrupted.
      3.2.2: Send "wrong guess" to $\mathcal{A}$.
```

Game 13 Once $\mathcal{S}$ receives a user tag $\hat{A}_u$ meant for honest $\mathcal{P}_s$, if the check (i.e., $A_u \neq \hat{A}_u$) fails, $\mathcal{S}$ now sends (Interrupt, sid, uid, ρ_s) to $\mathcal{F}$ before gracefully aborting. In turn, $\mathcal{F}$ marks the Session record of the server as `Interrupted`.
This change does not lead to a change in the observable behavior of the simulation, so this game is perfectly indistinguishable from the previous game.
Even though now it is possible for $\mathcal{F}$ to have Session records that are `Interrupted`, Interrupt interface itself is the only one concerned with the status of the Session records, and it is called *only by* the honest server *only once* per subsession.

Game 14 Once $\mathcal{S}$ receives a server tag $\hat{A}_s$ meant for honest $\mathcal{P}_c$, if the check (i.e., $A_s \neq \hat{A}_s$) fails, $\mathcal{S}$ now sends $(\textsc{TestPwd}, sid, uid, \rho_c, \perp)$ to $\mathcal{F}$ before gracefully aborting. In turn, $\mathcal{F}$ marks the SESSION record of the user as `Interrupted`.
This change does not lead to a change in the observable behavior of the simulation, so this game is perfectly indistinguishable from the previous game.

Game 15 We capture the impersonation attacks in this game.
Let α be the blinded password sent by $\mathcal{P}_c$, and let $\hat{c}$, $\hat{\beta}$ be the received values in response. If all the following hold, then $\mathcal{S}$ sends an $(\textsc{Impersonate}, sid, uid, \rho_c)$ message to $\mathcal{F}$ where ρ_c is the partial subsession id of the client:

1. Both $\mathcal{P}_c$ and $\mathcal{P}_s$ are honest.
2. $\langle \textsc{StolenKey}, sid, uid, \ldots \rangle$ exists. Note that this means that the relevant FILE record at $\mathcal{F}$ is marked as `Compromised`.
3. We have $\hat{\beta} = \alpha^{k_s}$ where k_s is the OPRF key chosen by $\mathcal{S}$ for the user *uid*.
4. For k_s in cond. §3, the record $\langle \textsc{RndPwd}, sid, uid, k_s, \{\mathbf{y_h}\} \rangle$ exists such that $\hat{c}$ decrypts with y_h (i.e., $\mathsf{AuthDec}_{y_h}(\hat{c}) \neq \perp$.)
5. No $\langle \textsc{OprfEval}, x', *, y' \rangle$ such that $\mathsf{H}_2[(x', \hat{\beta}^{1/r})] = y'$ and $\mathsf{AuthDec}_{y'}(\hat{c}) \neq \perp$ exists.

Note that cond. §4 implies $y_h = \mathsf{H}_2(\mathsf{spw}, \mathsf{H}_1(\mathsf{spw})^{k_s})$ where spw is the password registered for *uid* during the registration phase. Now, assume that cond. §3 holds, i.e., $\mathcal{P}_c$ receives β such that $\beta = \alpha^{k_s} = \mathsf{H}_1(\mathsf{upw})^{r * k_s}$. Then, $\mathcal{P}_c$ sets $y := \mathsf{H}_2(\mathsf{upw}, \beta^{1/r}) = \mathsf{H}_2(\mathsf{upw}, \mathsf{H}_1(\mathsf{upw})^{k_s})$ where upw is $\mathcal{P}_c$'s input password and r is the blinding factor chosen by $\mathcal{P}_c$. Then, due to cond. §4 and cond. §2, if $\mathcal{F}$ returns "correct guess" (i.e., $\mathsf{upw} = \mathsf{spw}$), then we have $y = y_h$, and $\mathsf{AuthDec}_y(\hat{c}) \neq \perp$. Hence, in this case, we directly use y_h during the simulation of $\mathcal{P}_c$. On the other hand, if $\mathcal{F}$ returns "wrong guess", then $y \neq y_h$ due to Game 5:

$$\begin{aligned}&\Pr[y = y_h \mid \mathcal{F} \text{ returns "wrong guess"}]\\&= \Pr\Big[\mathsf{H}_2(\mathsf{upw}, \mathsf{H}_1(\mathsf{upw})^{k_s}) = \mathsf{H}_2(\mathsf{spw}, \mathsf{H}_1(\mathsf{spw})^{k_s}) \mid \mathsf{upw} \neq \mathsf{spw}\Big]\\&= 0\end{aligned}$$

Moreover, if $y \neq y_h$ and cond. §4 holds, then $\hat{c}$ cannot decrypt with y due to Game 6.
Since we know $\mathcal{P}_c$ will fail to decrypt, we let $\mathcal{S}$ gracefully abort $\mathcal{P}_c$ early if $\mathcal{F}$ returns "wrong guess" on a response to IMPERSONATE message, and this game is perfectly indistinguishable from the previous one.
Due to this change, it is now possible for $\mathcal{F}$ to have SESSION records that are `Compromised`. Yet, an IMPERSONATE message is sent only by the honest client only once. Since INTERRUPT message is only sent by the server for its own partial subsession, and there are no server SESSION records that are `Compromised`, all calls to INTERRUPT will make the associated partial subsession `Interrupted`, just as before.

Game 16 We partially capture the *server side* online password test attacks in this game.
On a fresh invocation of H_2 with input $(\mathsf{tpw}, \mathsf{H}_1(\mathsf{tpw})^k)$ that is not in response to a OFFLINETESTPWD request from ε, we send $(\textsc{TestPwd}, sid, uid, \rho_s, \mathsf{tpw})$ to $\mathcal{F}$ after we pop a $\langle \textsc{BlindedEval}, sid, uid, g_q^k, \{\rho_\mathbf{s}\} \rangle$ record.
In turn, $\mathcal{F}$ will mark the server's `Fresh` session record as `Compromised` if $\mathsf{tpw} = \mathsf{spw}$, or as `Interrupted` if $\mathsf{tpw} \neq \mathsf{spw}$. Notice that, if the server has aborted, server's record would be `Interrupted` (see Game 13). In this case, $\mathcal{F}$ leaves the record as `Interrupted` and always returns "wrong guess" on subsequent calls. This is not a problem, since we do not utilize the response of $\mathcal{F}$ at this point.
After this modification, the behavior of $\mathcal{F}$ does not change on an IMPERSONATE message, since this interface is only concerned with SESSION records of a client. However, a call to

INTERRUPT might have different outcomes depending on when it is called:

- ▷ When INTERRUPT is sent before TESTPWD:
  Server's SESSION record is `Interrupted`
- ▷ When INTERRUPT is sent after TESTPWD with $\mathsf{tpw} \neq \mathsf{spw}$:
  Server's SESSION record is `Interrupted`.
- ▷ When INTERRUPT is sent after TESTPWD with $\mathsf{tpw} = \mathsf{spw}$:
  Server's SESSION record is `Compromised`.

Still, this ambiguity does not lead to a change in the behavior of $\mathcal{P}_s$ or $\mathcal{P}_c$. Hence, this game is perfectly indistinguishable from the previous one.

Game 17 We capture the server side online password test attacks that are *delayed* in this game. Note that in the given algorithm in Figure 5.5, the adversary has all the necessary information to perform the attack after line 5. Even though the server expects a user tag from the client, the adversary may intentionally make the server abort, or make it wait infinitely. At any point in the future, this attack might be performed, i.e., the adversary may *delay* the attack as long as it wishes.
At this point, $\mathcal{F}$ does not allow $\mathcal{A}$ to mount a password test attack after the server has aborted, as discussed in Game 16. In order to remedy this, we replace the INTERRUPT and TESTPWD interfaces with the ones defined in $\mathcal{F}_{\mathsf{psaPAKE}}$ (with the dPT variable).
This ensures that an invocation of TESTPWD at $\mathcal{F}$ will return "correct guess" if and only if $\mathsf{tpw} = \mathsf{spw}$ where tpw is the queried password (i.e., the input of H_2), and spw is the password for *uid* registered at the server. Since we still do not use the response of $\mathcal{F}$, this game is perfectly indistinguishable from the previous one.

Game 18 Note that at this point, on a $(\textsc{TestPwd}, sid, uid, *, \mathsf{tpw})$ message sent as a result of the modifications in Game 16, $\mathcal{F}$ returns "correct guess" if and only if $\mathsf{spw} = \mathsf{tpw}$ where spw is the registered password. Accordingly, if $\mathcal{F}$ responds with "correct guess", then we consider that particular *uid* compromised and record $\langle \textsc{StolenPwd}, sid, uid, \mathsf{tpw}\rangle$. Since $\mathcal{S}$ doesn't read these records, this game is perfectly indistinguishable from the previous one.

Game 19 We capture the *client side* online password test attacks in this game.
On the receipt of a $\hat{\beta}, \hat{c}$ intended for $\mathcal{P}_c$ in response to $\alpha^* = \mathsf{H}_1(\mathsf{upw})^r$, the simulator checks if the record $\langle \textsc{OprfEval}, \{\mathsf{apw}\}, *, \{\mathbf{y}\}\rangle$ where $\mathsf{H}_2[(\mathsf{apw}, \hat{\beta}^{1/r})] = y$ and $\mathsf{AuthDec}_y(\hat{c}) \neq \perp$ exists. If so, then $\mathcal{S}$ sends a $(\textsc{TestPwd}, sid, uid, \rho_c, \mathsf{apw})$ message to $\mathcal{F}$, where ρ_c is client's partial subsession id. In turn, $\mathcal{F}$ will mark the server's `Fresh` session record as `Compromised` if $\mathsf{apw} = \mathsf{upw}$, or as `Interrupted` if $\mathsf{apw} \neq \mathsf{upw}$. If $\mathcal{F}$ returns "correct guess", i.e., if $\mathsf{apw} = \mathsf{upw}$, then we use y for the simulation of honest $\mathcal{P}_c$. Otherwise, i.e., if $\mathsf{apw} \neq \mathsf{upw}$, $\mathcal{P}_c$ wouldn't get y that decrypts $\hat{c}$, due to the collision resistance of H_2 introduced in Game 5, hence we gracefully abort $\mathcal{P}_c$ early, which is perfectly indistinguishable from the previous game.
Recall from Game 4 that we create a new OPRFEVAL record on every fresh H_2 invocation. If no such OPRFEVAL record exists and $\hat{\beta}$ is modified by $\mathcal{A}$, i.e., $\hat{\beta} \neq \alpha^{*k_s^*}$ where k_s^* is the OPRF key chosen by $\mathcal{S}$ for the user, then that can mean two things:

1. We have an *ill-formed* $\hat{\beta}, \hat{c}$ pair. By ill-formed, we mean $\mathsf{AuthDec}_y(\hat{c}) = \perp$ where $y = \mathsf{H}_2(\mathsf{upw}, \beta^{1/r})$.
2. $\mathcal{A}$ was able to come up with an OPRF evaluation for upw without invoking H_2.

We have

$$\begin{aligned}\Pr[\text{cond. §2 happens}] &= \sum_{y \in \{0,1\}^\lambda} \Pr[\mathcal{S} \text{ samples } y \mid \mathcal{A} \text{ chooses } y] \cdot \Pr[\mathcal{A} \text{ chooses } y] \\ &\leq \frac{1}{2^\lambda}.\end{aligned}$$

Hence, on an unexpected $\hat{\beta}$ and no OPRFEVAL record, we assume cond. §1, send (TESTPWD, $sid, uid, \rho_c, \perp$) to $\mathcal{F}$, and gracefully abort $\mathcal{P}_c$ early. Let bad denote the event of gracefully aborting whereas c would decrypt. Then we get

$$\begin{aligned}|\Pr[\mathbf{G}_{19}] - \Pr[\mathbf{G}_{18}]| &= \Pr[\mathsf{bad}] = \Pr[\text{cond. §2 happens}]\\ &\leq \frac{1}{2^\lambda}.\end{aligned}$$

Notice that in Game 15, we only send IMPERSONATE if a record $\langle$OPRFEVAL$, x', *, y'\rangle$ such that $\mathsf{H}_2[(x', \hat{\beta}^{1/r})] = y'$ and $\mathsf{AuthDec}_{y'}(\hat{c}) \neq \perp$ doesn't exist. As a result, we either send an IMPERSONATE or TESTPWD, so each message is sent for a user SESSION record that is `Fresh`.

Game 20 We define the following internal procedures:

```
▷ On internal call create_keys({uid}):
  1: Choose $k_s^* \leftarrow\!\$\ \mathbb{Z}_q$.
  2: Choose $p_s^* \leftarrow\!\$\ \mathbb{Z}_p$, and set $P_s^* := g_p^{p_s^*}$.
  3: Choose $p_u^* \leftarrow\!\$\ \mathbb{Z}_p$, and set $P_u^* := g_p^{p_u^*}$.
  4: Return $(k_s^*, p_s^*, p_u^*, P_s^*, P_u^*)$.
▷ On internal call register_user({uid}):
  1: Set $(k_s^*, p_s^*, p_u^*, P_s^*, P_u^*) \leftarrow \mathsf{create\_keys}(uid)$.
  2: Set $m := p_u^* \parallel P_u^* \parallel P_s^*$.
  3: Set $(c^*, st) \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(|m|)$.
  4: Set $y^* \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(st, m)$.
  5: Record $\langle$RNDPWD$, sid, uid, k_s^*, y^*\rangle$.
  6: Record $\langle$USRCREDS$, sid, uid, p_u^*, p_s^*, P_u^*, P_s^*, c^*\rangle$.
  7: Return $(p_s^*, P_u^*, c^*, k_s^*)$.
```

On a registration request for user uid intended for honest $\mathcal{P}_s$, we let $\mathcal{S}$ call $\mathsf{register\_user}(uid)$ and place the returned values in $\mathsf{file}[uid]$.
Notice that register_user is exactly the registration procedure ran in the previous game except for one crucial difference: Now we use the output of $\mathsf{SIM}_{\mathrm{EQV}}$ for c^* and y^* instead of setting $y := \mathsf{H}_2(\mathsf{spw}, \mathsf{H}_1(\mathsf{spw})^k)$ and $c \leftarrow \mathsf{AuthEnc}_y(p_u^* \parallel P_u^* \parallel P_s^*)$.

Programming H_2. On a fresh query (tpw, b) to H_2, if $\mathcal{F}$ responds with "correct guess" either in response to a TESTPWD or an OFFLINETESTPWD query, we program the output to be the simulated decryption key, i.e., $\mathsf{H}_2(\mathsf{tpw}, b) = y^*$. Recall that due to the changes made in Game 17 and Game 11, $\mathcal{F}$ returns "correct guess" if and only if $\mathsf{tpw} = \mathsf{spw}$ where spw is the password registered for uid.

Equivocability advantage. Now we turn our attention to the environment advantage introduced after utilizing $\mathsf{SIM}_{\mathrm{EQV}}$ in the creation of c, y.

▷ Let u denote the number of registered users.

▷ We enumerate the registered users as $uid_1, \ldots, uid_u$.

We let $\mathbf{G}_{19.0} = \mathbf{G}_{19}$ and build the sub-games $\mathbf{G}_{19.1}, \ldots, \mathbf{G}_{19.u}$.

Game 19.i We modify $\mathbf{G}_{19.i-1}$, so that for the registration of uid_i, $\mathcal{S}$ now utilizes the register_user procedure. We modify the simulator from $\mathbf{G}_{19.i-1}$ and build $\mathcal{S}_i'$ that plays Exp-EQV (see Figure 2.6) as follows:

```
▷ On INIT from the challenger:
  1: Set $(k_s^*, p_s^*, p_u^*, P_s^*, P_u^*) \leftarrow \mathsf{create\_keys}(uid)$.
  2: Record $k_s, p_s, p_u, P_s, P_u$.
  3: Set $m := p_u \parallel P_u \parallel P_s$.
  4: Return $m$ to the challenger.
▷ On $(\{\mathbf{c}\}, \{\mathbf{y}\})$ from the challenger:
  1: Record $c, y$.
  2: Start the simulation of the previous game.
  3: Forward the output of $\varepsilon$ to the challenger.
▷ On $i$th internal call $\mathsf{register\_user}(\{\mathbf{uid}\})$:
  1: Retrieve $k_s, p_s, p_u, P_s, P_u$.
  2: Retrieve $c, y$.
  3: Record $\langle \textsc{RndPwd}, sid, uid, k_s, y \rangle$.
  4: Record $\langle \textsc{UsrCreds}, sid, uid, p_u, p_s, P_u, P_s, c \rangle$.
  5: Return $(p_s, P_u, c, k_s)$.
```

Assume that an authentication phase is for uid_i. For this subsession, we have the following two cases:

1. If $b = 0$, then the challenger has sent $y \leftarrow \mathsf{Gen}(1^\lambda)$ and $c \leftarrow \mathsf{AuthEnc}_y(m)$, and what $\mathcal{S}_i'$ runs is exactly the previous game.

2. If $b = 1$, then the challenger has sent $c \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(|m|)$ and $y \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m, ...)$, and what $\mathcal{S}_i'$ runs is exactly this game.

Then, if the environment correctly distinguishes $\mathbf{G}_{19.i}$ from $\mathbf{G}_{19.i-1}$, the environment outputs b, and $\mathcal{S}_i'$ wins in an authentication phase. Additionally, the environment can distinguish this game from the previous one if it encounters a collision on H_2, since in the previous game (due to Game 5) we did not have collisions in a graceful execution (i.e., an execution with no simulation aborts). Hence, we have the following bounds:

$$|\Pr[\mathbf{G}_{19.i}] - \Pr[\mathbf{G}_{19.i-1}]| \leq \mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{S}_i',\mathsf{AE}}(\lambda)$$
$$\mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{S}_i',\mathsf{AE}}(\lambda) \geq \mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{S}_{i-1}',\mathsf{AE}}(\lambda)$$

Note that we have $\mathbf{G}_{19.u} = \mathbf{G}_{20}$. We let $\mathcal{S}_u' = \mathcal{A}_{\mathsf{EQV}}$. Hence,

$$\begin{aligned}|\Pr[\mathbf{G}_{19}] - \Pr[\mathbf{G}_{19.u}]| &= |\Pr[\mathbf{G}_{19}] - \Pr[\mathbf{G}_{20}]| \\ &\leq \sum_{i\in[u]} \mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{S}_i',\mathsf{AE}}(\lambda) \\ &\leq u \cdot \mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{A}_{\mathsf{EQV}},\mathsf{AE}}(\lambda)\end{aligned}$$

Game 21 We modify $\mathbf{G}_{20}$, and define $\mathsf{sim\_unknown\_user}$ as follows:

```
▷ On internal call $\mathsf{sim\_unknown\_user}(\{\mathbf{uid}\})$:
  1: If $\langle \textsc{UsrCreds}, sid, uid, \ldots \rangle$ doesn't exist, then do:
    1.1: Set $(k_s^*, p_s^*, p_u^*, P_s^*, P_u^*) \leftarrow \mathsf{create\_keys}(uid)$.
    1.2: Set $m := 0 \ldots 0$ such that $|m| = |p_u^* \parallel P_u^* \parallel P_s^*|$.
    1.3: $(c^*, st) \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(|m|)$
    1.4: $y^* \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m, st)$.
    1.5: Record $\langle \textsc{RndPwd}, sid, uid, k_s^*, y^* \rangle$.
    1.6: Record $\langle \textsc{UsrCreds}, sid, uid, p_u^*, p_s^*, P_u^*, P_s^*, c^* \rangle$.
```

```
2: Retrieve $\langle \textsc{RndPwd}, sid, uid, k_s^*, * \rangle$.
3: Retrieve $\langle \textsc{UsrCreds}, sid, uid, *, \{\mathbf{p_s^*}\}, \{\mathbf{P_u^*}\}, \{\mathbf{P_s^*}\}, \{\mathbf{c^*}\} \rangle$.
4: Return $(p_s^*, P_u^*, c^*, k_s^*)$.
```

On an authentication request, if uid was not registered before, we let $\mathcal{S}$ call $\mathsf{sim\_unknown\_user}$ and treat its output as the tuple recorded at $\mathsf{fakefile}[uid]$.
This game is the same as the previous one except for the user enumeration mitigation: In the previous game, the simulator picks a random key y_r and sets $c = \mathsf{AuthEnc}_{y_r}(m)$ where m is a string of zeroes of length $|p_u^* \parallel P_u^* \parallel P_s^*|$. Now, we have $c^* \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(|m|)$ and $y^* \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m, ...)$.

- ▷ Let $\tilde{u}$ denote the number of unknown uids that was sent in at least one authentication request.
- ▷ We enumerate these unknown user ids as $\tilde{uid}_1, ..., \tilde{uid}_{\tilde{u}}$.
- ▷ Let $\tilde{y_i}$ be the key generated for $\tilde{uid}_i$.
- ▷ Let y_i be the key generated for uid_i.

We let $\mathbf{G}_{20.0} = \mathbf{G}_{20}$ and build the sub-games $\mathbf{G}_{20.1}, \ldots, \mathbf{G}_{20.\tilde{u}}$.

Game 20.i We modify $\mathbf{G}_{20.i-1}$, so that on an authentication request for user $\tilde{uid}_i$, we let $\mathcal{S}$ call $\mathsf{sim\_unknown\_user}(\tilde{uid}_i)$ and use its output as if it is the output of $\mathsf{fakefile}[\tilde{uid}_i]$.
We modify the simulator from $\mathbf{G}_{20.i-1}$ and build $\mathcal{S}'_i$ that plays Exp-EQV (see Figure 2.6) as follows:

```
▷ On Init from the challenger:
  1: Set $(k_s^*, p_s^*, p_u^*, P_s^*, P_u^*) \leftarrow \mathsf{create\_keys}(uid)$.
  2: Record $k_s, p_s, p_u, P_s, P_u$.
  3: Set $m := 0 \ldots 0$ such that $|m| = |p_u \parallel P_u \parallel P_s|$.
  4: Return $m$ to the challenger.
▷ On $(\{\mathbf{c}\}, \{\mathbf{y}\})$ from the challenger:
  1: Record $c, y$.
  2: Start the simulation of the previous game.
  3: Forward the output of $\varepsilon$ to the challenger.
▷ On an authentication request for $\tilde{uid}_i$:
  1: Set $uid := \tilde{uid}_i$.
  2: If $\langle \textsc{UsrCreds}, sid, uid, \ldots \rangle$ doesn't exist, then do:
    2.1: Retrieve $k_s, p_s, p_u, P_s, P_u$.
    2.2: Retrieve $c, y$.
    2.3: Record $\langle \textsc{RndPwd}, sid, uid, k_s, y \rangle$.
    2.4: Record $\langle \textsc{UsrCreds}, sid, uid, p_u, p_s, P_u, P_s, c \rangle$.
  3: Retrieve $\langle \textsc{RndPwd}, sid, uid, \{\mathbf{k_s}\}, * \rangle$.
  4: Retrieve $\langle \textsc{UsrCreds}, sid, uid, *, \{\mathbf{p_s}\}, \{\mathbf{P_u}\}, *, \{\mathbf{c}\} \rangle$
  5: Return $(p_s, P_u, c, k_s)$.
```

By a similar argument as in Game 20, the environment directly acts as a distinguisher for the ciphertext and key produced for $\tilde{uid}_i$. Then we have the following bounds:

$$|\Pr[\mathbf{G}_{20.i}] - \Pr[\mathbf{G}_{20.i-1}]| \leq \mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{S}'_i, \mathsf{ENC}}(\lambda)$$
$$\mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{S}'_i, \mathsf{ENC}}(\lambda) \geq \mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{S}'_{i-1}, \mathsf{ENC}}(\lambda)$$

Since $\mathbf{G}_{20.\tilde{u}} = \mathbf{G}_{21}$, we have

$$|\Pr[\mathbf{G}_{21}] - \Pr[\mathbf{G}_{20}]| \leq \tilde{u} \cdot \mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{S}'_{\tilde{u}}, \mathsf{ENC}}(\lambda)$$

Game 22 We move the generation of the encryption key $y^* \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m, st)$ at $\mathsf{register\_user}$ to the simulation of H_2. More specifically, we do the following changes:

- ▷ We now record $\langle \textsc{RndPwd}, sid, uid, k_s, st\rangle$ during $\mathsf{register\_user}(uid)$ where st is the state of $\mathsf{SIM}_{\mathrm{EQV}}$ outputted alongside with c^*.
- ▷ At the invocation of a fresh $\mathsf{H}_2(x, y)$ query, if $\mathcal{F}$ returns "correct guess" either in response to an $\textsc{TestPwd}$ or $\textsc{OfflineTestPwd}$ message, we retrieve st, compute $y^* \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m, st)$, and set $\mathsf{H}_2(x, y) = y^*$.

In the previous game, let $\mathsf{H}_2(x, y) = y^*$ for some x, y. In this game, we still have $\mathsf{H}_2(x, y) = y^*$. The only difference is we delay the computation of y^* until it needs to be returned from H_2. Hence, this game is perfectly indistinguishable from the previous one.

Game 23 In this game, we let $\mathcal{S}$ only forward the registration requests to $\mathcal{F}$ (and never read their contents), since the registration happens over secure channels. To that end, we define the following internal procedure $\mathsf{simulate\_user}$:

```
▷ On internal call $\mathsf{simulate\_user}(\{\mathbf{uid}\})$:
  1: If $\langle \textsc{UsrCreds}, sid, uid, \ldots\rangle$ doesn't exist, then do:
    1.1: Call $\mathsf{register\_user}(uid)$.
  2: Retrieve $\langle \textsc{RndPwd}, sid, uid, k_s^*, *\rangle$.
  3: Retrieve $\langle \textsc{UsrCreds}, sid, uid, *, \{\mathbf{p_s^*}\}, \{\mathbf{P_u^*}\}, \{\mathbf{P_s^*}\}, \{\mathbf{c^*}\}\rangle$.
  4: Return $(p_s^*, P_u^*, c^*, k_s^*)$.
```

We let $\mathcal{S}$ call $\mathsf{simulate\_user}$ every time $\mathcal{P}_s$ receives an authentication request with uid. Instead of using $\mathsf{file}[uid]$ or $\mathsf{sim\_unknown\_user}(uid)$, $\mathcal{S}$ now directly uses the output of $\mathsf{simulate\_user}(uid)$ as the user's file record. Additionally, at the registration of uid, $\mathcal{S}$ only forwards the received $\textsc{StorePwdFile}$ to $\mathcal{F}$, without calling $\mathsf{register\_user}$.

The case when *uid* was registered before. In this case, this game is exactly the same as the previous game, except the call to $\mathsf{register\_user}$ happens on the first authentication attempt for uid by $\mathcal{P}_c$. This doesn't lead to a change in the behavior of $\mathcal{S}$. In this case, this game is perfectly indistinguishable from the previous one.

The case when *uid* was not registered before. In this case, we had $c \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(|m|)$ and $y \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m, ...)$ where m is a string of zeroes of length $|p_u \parallel P_u \parallel P_s|$ in the previous game where p_u, P_u, P_s are the randomly chosen keys for the particular uid. Now we have $c' \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(|m'|)$ and $y' \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m', ...)$ where $m' = p_u \parallel P_u \parallel P_s$.
Since $|m'| = |m|$, we have

$$\mathsf{Dist}(c) = \mathsf{Dist}(c'). \tag{6.1}$$

Now we argue that neither y nor y' is leaked to the adversary:
Recall from Section 4 that for an invalid uid, $\mathcal{F}$ sets the password value of the user's $\textsc{Session}$ record to $\perp$. Therefore, $\mathcal{F}$ never returns "correct guess" on a $(\textsc{TestPwd}, sid, uid, \mathsf{tpw}, \rho_c)$ for any tpw where ρ_c is a user partial subsession identifier, and $\mathcal{S}$ never returns y' due to a "correct" password guess. Moreover, in a graceful execution (i.e., an execution with no simulation aborts), $\mathcal{S}$ never samples y' on a fresh H_2 query, because we abort the simulation on collisions. Hence, no query to H_2 can return y'. Additionally, any other value sent by $\mathcal{S}$ is computed independently of y'. The same argument holds for y in the previous game. This means $\mathcal{S}$ never leaks y' as in the previous game, and due to Equation 6.1, this game is perfectly indistinguishable from the previous one.

Game 24 We modify the invocations of H_3 as follows:

- ▷ **At the simulation of honest $\mathcal{P}_s$:** Instead of setting the simulated PRF key K^* to be the output of the invocation $\mathsf{H}_3({X_u}^{x_s}, {X_u}^{p_s}, {P_s}^{x_s})$, we now choose $K^* \leftarrow\$ \mathcal{K}$ and record $\langle \textsc{SvrPrfKey}, sid, ({X_u}^{x_s}, {X_u}^{p_s}, {P_s}^{x_s}), K^* \rangle$.
- ▷ **At the simulation of honest $\mathcal{P}_c$:** Instead of setting the simulated PRF key K^* to be the output of the invocation $\mathsf{H}_3(X_s^{x_u}, P_s^{x_u}, X_s^{p_u})$, we now look for a record $\langle \textsc{SvrPrfKey}, sid, (X_s^{x_u}, P_s^{x_u}, X_s^{p_u}), \{\mathbf{K}\} \rangle$. If it exists, we set $K^* := K$. Otherwise, we choose $K^* \leftarrow\$ \mathcal{K}$. Finally, we record $\langle \textsc{UsrPrfKey}, sid, uid, (X_s^{x_u}, P_s^{x_u}, X_s^{p_u}), K^* \rangle$.
- ▷ **At the simulation of H_3:** On a fresh query $\mathsf{H}_3(x, y, z)$ for some x, y, z, if there exists a record $\langle \textsc{SvrPrfKey}, \ldots, (x, y, z), \{\mathbf{K}^*\} \rangle$ or there exists a record $\langle \textsc{UsrPrfKey}, \ldots, (x, y, z), \{\mathbf{K}^*\} \rangle$, then we set $\mathsf{H}_3[(x, y, z)] = K^*$ and return K^*. Otherwise, we proceed as in the previous game.

In this game, every fresh H_3 query is sampled from a uniform distribution, as in the previous game. The only difference is that for some inputs, we sample the outputs beforehand, which does not change the observed outputs. Finally, since the honest $\mathcal{P}_c$ computes K^* after honest $\mathcal{P}_s$ in $\Pi_\star$, in an honest execution with no active attacks, there is an appropriate $\textsc{SvrPrfKey}$ record when $\mathcal{S}$ computes the PRF key in the simulation of honest $\mathcal{P}_c$. Hence, $\mathcal{S}$ gets the same K^* for both honest parties as before in an honest execution with no active attacks. Therefore, this game is perfectly indistinguishable from the previous one.

Game 25 We add the following internal procedures to $\mathcal{S}$:

```
▷ On internal call create_eph_keys({uid}, {Y}):
1: Choose $x \leftarrow\$ \mathbb{Z}_p$, and set $X := g_p^x$.
2: Return $(x, X)$.
```

We let $\mathcal{S}$ invoke $\mathsf{create\_eph\_keys}(uid, \perp)$ during the simulation of honest $\mathcal{P}_c$ to generate the client's ephemeral keys where uid is the client's user id input. In response to an authentication request for uid that contains the client's ephemeral key $\hat{X}_u$, we let $\mathcal{S}$ invoke $\mathsf{create\_eph\_keys}(uid, \hat{X}_u)$ during the simulation of honest $\mathcal{P}_s$ to generate the server's ephemeral keys.

Since $\mathsf{create\_eph\_keys}$ chooses a uniformly random private key and computes the public key exactly as in the previous game, this game is perfectly indistinguishable from the previous one.

Game 26 We make the following modifications to the simulator in an authentication phase. Let uid denote the user id of the authentication phase for the rest of this game hop.

- ▷ **During the simulation of honest $\mathcal{P}_c$:** Once $\mathcal{S}$ chooses a ephemeral private user key x_u^* and computes the public key X_u^*, if $\mathcal{P}_s$ is honest, then we record $\langle \textsc{ForbiddenInput}, sid, uid, (\perp, y', \perp) \rangle$ where y' is set to be $X_u^{*p_s^*}$ or $P_s^{*x_u^*}$ (if $p_s^* = \perp$) such that P_s^*, p_s^* are the long-term server keys chosen by $\mathcal{S}$.
- ▷ **During the simulation of honest $\mathcal{P}_s$:** Once $\mathcal{S}$ chooses an ephemeral private server key x_s^* and computes the public key X_s^*, we record $\langle \textsc{ForbiddenInput}, sid, uid, (\perp, \perp, z') \rangle$ where z' is set to be $P_u^{*x_s^*}$ or $X_s^{*p_u^*}$ (if $x_s^* = \perp$) such that p_u^*, P_u^* are the long-term user keys. Additionally, if $\mathcal{P}_c$ is honest, we record $\langle \textsc{ForbiddenInput}, sid, uid, (x', \perp, \perp) \rangle$ where x' is set to be $X_s^{*x_u^*}$ or $X_u^{*x_s^*}$ (if $x_u^* = \perp$) such that x_u^*, X_u^* are the user ephemeral keys chosen by $\mathcal{S}$.

In this game, we only create $\textsc{ForbiddenInput}$ records and never read them. Hence, this game is perfectly indistinguishable from the previous one.

Game 27 In this game, we ensure that a $\textsc{Session}$ record at $\mathcal{F}$ is marked as `Fresh` only if $\mathcal{A}$ has not mounted an active attack during the key exchange.

- ▷ **During the simulation of honest $\mathcal{P}_s$:** During an authentication session for uid where ρ_s is server's partial subsession id, if $\mathcal{P}_c$ is honest and $\hat{X}_u \neq X^*_u$, then we send (INTERRUPT, sid, uid, ρ_s) to $\mathcal{F}$.
- ▷ **During the simulation of honest $\mathcal{P}_c$:** During an authentication session for uid where ρ_c is user's partial subsession id, if $\mathcal{P}_s$ is honest and $\hat{X}_s \neq X^*_s$, then we send (TESTPWD, $sid, uid, \rho_c, \perp$) to $\mathcal{F}$.

These changes do not lead to an observable change in the behavior, hence this game is perfectly indistinguishable from the previous one.

Game 28 On a fresh invocation of $\mathsf{H}_3(x, y, z)$, if $\langle$FORBIDDENINPUT$, sid, uid, (x, \perp, \perp)\rangle$ exists, then we abort the simulation. Note that this implies that we have $x = g_p^{x^*_s x^*_u}$ where x^*_s and x^*_u are the ephemeral private keys chosen by $\mathcal{S}$ that are unknown to the adversary.
We modify the simulator from the previous game and build $\mathcal{A}_{\mathsf{GapDH}}$ that plays GapDH (see Figure 2.2) as follows:

```
▷ On ({g_p^a}, {g_p^b}) from the challenger:
1: Record $g_p^a, g_p^b$.
2: Choose $i \leftarrow\$ [s_u]$ and record $i$.
3: Start the simulation of the previous game.
▷ On ith internal call to create_eph_keys($*, \perp$) during the simulation of honest $\mathcal{P}_c$:
1: Set $X_u := g_p^a$.
2: Return $(\perp, X_u)$ // $X_u$ observed by $\varepsilon$ is still uniformly random
▷ On internal call to create_eph_keys($*, g_p^a$) during the simulation of honest $\mathcal{P}_s$:
1: Set $X_s := g_p^b$.
2: Return $(\perp, X_s)$ // $X_s$ observed by $\varepsilon$ is still uniformly random
▷ On a fresh query $\mathsf{H}_3(\{x\}, \{y\}, \{z\})$:
1: If $\mathsf{DDH}(g_p^a, g_p^b, x) = 1$, then send $x$ to the challenger.
2: Return $\mathsf{H}_3(x, y, z)$.
```

Notice that if the environment is able to trigger the abort condition specified in this game, and we choose the right user subsession (out of s_u) to replace, then $\mathcal{A}_{\mathsf{GapDH}}$ wins GapDH. We have,

$$\begin{aligned} |\Pr[\mathbf{G}_{28}] - \Pr[\mathbf{G}_{27}]| &\leq \Pr[\mathsf{abort}] \\ &\leq s_u \cdot \mathsf{Adv}^{\mathrm{GapDH}}_{\mathcal{A}_{\mathsf{GapDH}},(\mathbb{G}_p,g_p,p)}(\lambda). \end{aligned}$$

Game 29 On a fresh invocation of $\mathsf{H}_3(x, y, z)$, if $\langle$FORBIDDENINPUT$, sid, \{\mathbf{uid}\}, (\perp, y, \perp)\rangle$ exists, then we abort the simulation. Note that this implies that we have $y = g^{p^*_s x^*_u}$ where x^*_u is the client's ephemeral private key and p^*_s is the server's long-term private key for uid. We know that both x^*_u and p^*_s are unknown to the environment.
We modify the simulator from the previous game and build $\mathcal{A}_{\mathsf{GapDH}}$ that plays GapDH (see Figure 2.2) as follows:

```
▷ On ({g_p^a}, {g_p^b}) from the challenger:
1: Record $g_p^a, g_p^b$.
2: Choose $i \leftarrow\$ [u]$ and record $uid_i$.
3: Start the simulation of the previous game.
▷ On internal call to create_keys($uid_i$) during the simulation of honest $\mathcal{P}_s$:
```

```
1: Choose $k_s \leftarrow\!\!\$\; \mathbb{Z}_q$.
2: Choose $p_u \leftarrow\!\!\$\; \mathbb{Z}_p$, and set $P_u := g_p^{p_u}$.
3: Set $P_s := g_p^b$.
4: Return $(k_s, \bot, p_u, P_s, P_u)$.
▷ On internal call to $\mathsf{create\_eph\_keys}(uid_i, \bot)$ during the simulation of honest $\mathcal{P}_c$:
1: Set $X_u := g_p^a$.
2: Return $(\bot, X_u)$ // $X_u$ observed by $\varepsilon$ is still uniformly random
▷ On $\mathsf{H}_3(\{\mathbf{x}\}, \{\mathbf{y}\}, \{\mathbf{z}\})$:
1: If $\mathsf{DDH}(g_p^a, g_p^b, y) = 1$, then send $y$ to the challenger.
2: Return $\mathsf{H}_3(x, y, z)$.
```

The advantage argument is the same as in Game 28 except the bound is now dependent on the number of registered users (i.e., u). Hence, we have

$$\begin{aligned}|\Pr[\mathbf{G}_{29}] - \Pr[\mathbf{G}_{28}]| &\leq \Pr[\mathsf{abort}]\\ &\leq u \cdot \mathsf{Adv}^{\mathrm{GapDH}}_{\mathcal{A}_{\mathsf{GapDH}},(\mathbb{G}_p,g_p,p)}(\lambda).\end{aligned}$$

Game 30 On a fresh invocation of $\mathsf{H}_3(x, y, z)$, if all the following conditions hold, then we abort:

1. $\langle \textsc{ForbiddenInput}, sid, \{\mathbf{uid}\}, (\bot, \bot, z)\rangle$ exists.

2. $\langle \textsc{StolenPwd}, sid, uid, *\rangle$ doesn't exist.

Note that cond. §1 implies that we have $z = g^{p_u^* x_s^*}$ where x_s^* is the server's ephemeral private key unknown to the environment and p_u^* is the user's long-term private key for *uid*. Moreover, cond. §2 implies that user's long-term keys (i.e., p_u^* and P_u^*) are unknown to the environment. We modify the simulator from the previous game and build $\mathcal{A}_{\mathsf{GapDH}}$ that plays GapDH (see Figure 2.2) as follows:

```
On $(\{\mathbf{g_p^a}\}, \{\mathbf{g_p^b}\})$ from the challenger:
1: Record $g_p^a, g_p^b$.
2: Choose $i \leftarrow\!\!\$\; [u]$ and record $uid_i$.
3: Start the simulation of the previous game.
On internal call to $\mathsf{create\_keys}(uid_i)$ during the simulation of honest $\mathcal{P}_s$:
1: Choose $k_s \leftarrow\!\!\$\; \mathbb{Z}_q$.
2: Choose $p_s \leftarrow\!\!\$\; \mathbb{Z}_p$, and set $P_s := g_p^{p_s}$.
3: Set $P_u := g_p^b$. // Indistinguishable to $\varepsilon$ if the password is not stolen.
4: Return $(k_s, p_s, \bot, P_s, P_u)$.
On internal call to $\mathsf{create\_eph\_keys}(uid_i, \bot)$ during the simulation of honest $\mathcal{P}_s$:
1: Set $X_s := g_p^a$.
2: Return $(\bot, X_s)$ // $X_s$ observed by $\varepsilon$ is still uniformly random
On $\mathsf{H}_3(\{\mathbf{x}\}, \{\mathbf{y}\}, \{\mathbf{z}\})$:
1: If $\mathsf{DDH}(g_p^a, g_p^b, z) = 1$, then send $z$ to the challenger.
2: Return $\mathsf{H}_3(x, y, z)$.
```

The advantage argument is the same as in Game 28 except the bound is now dependent on the number of registered users (i.e., u). Hence, we have

$$\begin{aligned}|\Pr[\mathbf{G}_{30}] - \Pr[\mathbf{G}_{29}]| &\leq \Pr[\mathsf{abort}]\\ &\leq u \cdot \mathsf{Adv}^{\mathrm{GapDH}}_{\mathcal{A}_{\mathsf{GapDH}},(\mathbb{G}_p,g_p,p)}(\lambda).\end{aligned}$$

Game 31 We directly add the NEWKEY, TESTABORT interfaces of $\mathcal{F}_{\mathsf{psaPAKE}}$ to $\mathcal{F}$. Since these interfaces are never invoked, this game is perfectly indistinguishable from the previous game.

Game 32 We capture the honest execution aborts in this game. Once $\mathcal{S}$ receives $\hat{\beta}, \hat{c}$ for honest $\mathcal{P}_c$, before doing anything else, $\mathcal{S}$ checks if all the following hold:

1. $\mathcal{P}_s$ is honest.
2. $\hat{\beta} = \alpha^{*k_s^*}$ where α^* is the α value computed by the $\mathcal{S}$, and k_s^* is the simulated OPRF key for the user (first introduced in Game 20).
3. $\hat{c} = c^*$ where c^* is the simulated ciphertext for the user (first introduced in Game 20).
4. $\langle \text{STOLENKEY}, sid, uid, \ldots \rangle$ doesn't exist.

If so, we send a $(\text{TESTABORT}, sid, uid, \rho_c, \rho_s)$ to $\mathcal{F}$. In turn, $\mathcal{F}$ returns "success" if $\mathsf{upw} = \mathsf{spw}$ or outputs an ABORT message otherwise. Recall that $\mathcal{S}$ forwards ABORT messages directly to ε.
If $\mathcal{F}$ returns "success" and $\langle \text{STOLENPWD}, sid, uid, \{\mathbf{spw}\} \rangle$ exists, we send a (TESTPWD, $sid, uid, \rho_c, \mathsf{spw}$) message to $\mathcal{F}$, so that the user's session record is marked as `Compromised`.
Note that if all the above conditions hold, then $\mathcal{S}$ correctly decrypts c^* if and only if $\mathsf{upw} = \mathsf{spw}$. Then, in the previous game, $\mathcal{S}$ gracefully aborts the simulation of honest $\mathcal{P}_c$ if $\mathsf{upw} \neq \mathsf{spw}$. Hence, this game is perfectly indistinguishable from the previous one.

Game 33 In this game, we let $\mathcal{S}$ discard the client's input password upw.
At the simulation of honest $\mathcal{P}_c$, after choosing the blinding factor r, instead of computing $\alpha = \mathsf{H}_1(\mathsf{upw})^r$, we now compute $\alpha^* = g_q^r$ and record $\langle \text{BLINDEDPWD}, sid, uid, \delta_c, r, \alpha^* \rangle$ where δ_c is the chosen nonce and uid is the user id.
Notice that the modifications in Game 15, Game 19 and Game 32 ensure that there is no execution path where $\mathcal{S}$ invokes $\mathsf{H}_2(\mathsf{upw}, \ldots)$ during the simulation of honest $\mathcal{P}_c$. We use the TESTABORT, TESTPWD and IMPERSONATE interfaces of $\mathcal{F}$ to check whether upw is the correct password, and depending on the response, we either abort gracefully or continue the execution. Hence, this game is perfectly indistinguishable from the previous one.

Game 34 We ensure that the outputs of the simulated honest parties are provided by $\mathcal{F}$ as follows: Once $\mathcal{S}$ is about to output a computed session key cSK on behalf of honest $\mathcal{P}$, it now sends the corresponding $(\text{NEWKEY}, sid, uid, \mathsf{cSK}, ...)$ message to $\mathcal{F}$, so that $\mathcal{F}$ outputs on behalf of $\mathcal{P}$. Let SK be the key outputted by $\mathcal{F}$ on behalf of $\mathcal{P}$. Notice that if either $\mathcal{P}_c$ or $\mathcal{P}_s$ is corrupt, then $\mathsf{SK} = \mathsf{cSK}$, and the environment observes cSK exactly as in the previous game. We now assume that both parties are honest and consider the cases $\mathcal{P} = \mathcal{P}_c$ and $\mathcal{P} = \mathcal{P}_s$ separately.

The first case $\mathcal{P} = \mathcal{P}_c$. The first NEWKEY message is sent during the simulation of honest $\mathcal{P}_c$. Then, we either have $\mathsf{SK} = \mathsf{cSK}$ or SK is chosen uniformly randomly by $\mathcal{F}$. If $\mathcal{F}$ outputs cSK, then the environment sees cSK exactly as in the previous game. Now, consider the case where $\mathcal{F}$ chooses SK uniformly randomly. This only happens when the associated SESSION record is marked as `Fresh`. This means that there was no TESTPWD or IMPERSONATE message sent targeting the user's SESSION record. In particular, we know that the following conditions hold:

1. We must have $\mathsf{upw} = \mathsf{spw}$ and $\hat{\beta} = \alpha^{*k_s^*}$ where upw is the client's input password, spw is the user's registered password, $\hat{\beta}$ is the β value received by $\mathcal{S}$, α^* is the α computed by $\mathcal{S}$, and k_s^* is the the simulated OPRF key for the user. Otherwise, $\mathcal{S}$ would have sent either IMPERSONATE or TESTPWD.

2. We must have $\hat{c} = c^*$ where $\hat{c}$ is the ciphertext received by $\mathcal{S}$, and c^* is the ciphertext computed for the user by $\mathcal{S}$. Otherwise, $\mathcal{S}$ would have sent either IMPERSONATE or TESTPWD.

Due to cond. §1 and cond. §2, the simulator gets $\hat{p_u} = p_u^*$, $\hat{P_u} = P_u^*$, and $\hat{P_s} = p_s^*$ during the simulation of honest $\mathcal{P}_c$. The PRF key for the client K^* is chosen uniformly randomly by $\mathcal{S}$ either during the simulation of honest $\mathcal{P}_c$ (when $\hat{X}_s \neq X_s^*$) or during the simulation of honest $\mathcal{P}_s$ (when $\hat{X}_s = X_s^*$). In both cases, $\mathcal{S}$ programs $\mathsf{H}_3(x, P_s^{*x_u^*}, z) = K^*$ for some x, z and records $\langle$FORBIDDENINPUT$, \ldots, (\perp, P_s^{*x_u^*}, \perp)\rangle$ during the simulation of honest $\mathcal{P}_c$. Hence, K^* is never leaked to the environment due to the abort condition introduced in Game 30.
We modify the previous game and build $\mathcal{A}_{\mathsf{PRF}}$ that plays Exp-PRF (see Figure 2.4) as follows:

```
On INIT from the challenger:
1: Run the simulation of the previous game.
2: Forward the output of $\varepsilon$ to the challenger.
On internal call to $F_K^{(0)}(\{\mathbf{x}\})$:
1: Return $\mathsf{Eval}(x)$.
```

1. If $b = 0$, $\mathsf{Eval}(x)$ returns $F_{K'}^{(0)}(x)$ for a fixed uniformly random key K', and the environment sees $\mathsf{SK} = F_{K'}^{(0)}(x)$ for a K' unknown to the environment. This is perfectly indistinguishable from the previous game.
2. If $b = 1$, $\mathsf{Eval}(x)$ returns $f(x)$ for a fixed uniformly random function f, and the environment sees $\mathsf{SK} = f(x)$ where SK is perfectly indistinguishable from a uniformly randomly chosen value. This is perfectly indistinguishable from this game.

In the previous game, we compute the output $\mathsf{SK} = F_{K^*}^{(0)}(\mathsf{tx})$ for a K^* that is unknown to ε. In this game, the output is chosen uniformly randomly. Then, if ε correctly distinguishes this game from the previous one, it outputs b, and consequently $\mathcal{A}_{\mathsf{PRF}}$ wins the Exp-PRF game.

The second case $\mathcal{P} = \mathcal{P}_s$. The second NEWKEY message is sent during the simulation of honest $\mathcal{P}_s$. If $\mathsf{SK} = \mathsf{cSK}$, then the environment sees cSK exactly as in the previous game. If $\mathcal{F}$ outputs a uniformly random session key, then either one of the following conditions must hold:

1. Both SESSION records are marked as `Fresh`, but we have $\mathsf{upw} \neq \mathsf{spw}$.
2. Server's SESSION record is marked as `Fresh` but not the user's record.
3. Server's SESSION record is marked as `Interrupted`, and $\langle$STOLENPWD$, sid, uid, *\rangle$ does not exist.

Assume that cond. §1 is the case. However, this never happens, since the only execution path leading to a NEWKEY query in which both SESSION records are `Fresh` is when we have a *honest execution* as implemented in Game 32. However, in this execution path, $\mathcal{S}$ sends a TESTABORT for the user's SESSION record, which leaves the record as `Completed` if we have $\mathsf{upw} \neq \mathsf{spw}$.
Assume that cond. §2 is the case. Then we have $\hat{X}_u = X_u^*$, since otherwise we would've sent an INTERRUPT query to $\mathcal{F}$, due to the changes made in Game 27. We know that $\mathcal{S}$ programs $\mathsf{H}_3(X_u^{*x_s^*}, y, z) = K^*$ for some y, z and records $\langle$FORBIDDENINPUT$, \ldots, (X_u^{*x_s^*}, \perp, \perp)\rangle$ during the simulation of honest $\mathcal{P}_s$. Due to the abort condition introduced in Game 28, K^* is never leaked to $\mathcal{A}$. As before, $\mathcal{A}_{\mathsf{PRF}}$ wins if the environment can distinguish between $\mathsf{cSK} = F_{K^*}^{(0)}$ and SK.
Assume that cond. §3 is the case. We know that $\mathcal{S}$ programs $\mathsf{H}_3(x, y, P_u^{*x_s^*}) = K^*$ for some x, y and records $\langle$FORBIDDENINPUT$, \ldots, (\perp, \perp, P_u^{*x_s^*})\rangle$ during the simulation of honest $\mathcal{P}_s$.

Since $\langle \textsc{StolenPwd}, sid, uid, *\rangle$ doesn't exist, and due to the abort condition introduced in Game 30, K^* is never leaked to $\mathcal{A}$. As before, $\mathcal{A}_{\mathsf{PRF}}$ wins if the environment can distinguish between $\mathsf{cSK} = F^{(0)}_{K^*}$ and SK.
Let SK' be the key outputted by $\mathcal{F}$ on behalf of $\mathcal{P}_c$. If we have $\mathsf{SK} = \mathsf{SK}$', then we know the following hold:

1. $\textsc{Session}$ record associated with $\mathcal{P}_s$ is fresh.
2. $\textsc{Session}$ record associated with $\mathcal{P}_c$ is fresh. This is because after $\mathcal{F}$ outputs a key on behalf of $\mathcal{P}_c$, we either send $\textsc{Interrupt}$ and $\textsc{TestAbort}$ to $\mathcal{F}$ and end the subsession (when $A^*_u \neq \hat{A}_u$), or we only send $\textsc{NewKey}$ to $\mathcal{F}$ (when $A^*_u = \hat{A}_u$).

Due to cond. §1, we know the following hold:

1. We have $\hat{X}_u = X^*_u$. Recall from Game 27 that when $\hat{X}_u \neq X^*_u$, we mark the server's $\textsc{Session}$ record as `Interrupted`.

Due to cond. §2, we know the following hold:

1. We have $\hat{X}_s = X^*_s$. Recall from Game 27 that when $\hat{X}_s \neq X^*_s$, we mark the user's $\textsc{Session}$ record as `Interrupted` by sending $(\textsc{TestPwd}, \dots, \perp)$.
2. We have $\hat{\alpha} = \alpha^*$, due to Game 19.
3. The simulated honest $\mathcal{P}_c$ has received p^*_u, P^*_u, P^*_s that was chosen by $\mathcal{S}$, due to the same reasoning as in the case where we had $\mathcal{P} = \mathcal{P}_c$.

Hence, in the previous game, after $\mathcal{S}$ outputs K on behalf of $\mathcal{P}_s$, $\mathcal{S}$ has outputs K on behalf of $\mathcal{P}_c$ too, due to the existence of $\langle \textsc{SvrPrfKey}, \dots, \mathsf{in\_keys}, \{\mathsf{K}\}\rangle$ where $\mathsf{in\_keys} = (X^{*x^*_u}_s, P^{*x^*_u}_s, X^{*p^*_u}_s)$. Therefore, the parties output the same key in this game if and only if they would have outputted the same key in the previous game.

Overall advantage. In this game, we have bound the advantage of the ε in case $\mathcal{F}$ chooses SK randomly, and we have shown that the parties output the same keys just like in the previous game. Then, by union bound, we have the following:

$$|\Pr[\mathbf{G}_{34}] - \Pr[\mathbf{G}_{33}]| \leq 3 \cdot \mathsf{Adv}^{\text{Exp-PRF}}_{\mathcal{A}_{\mathsf{PRF}}, F^{(0)}_{\cdot}}(\lambda).$$

Game 35 In this game, we simulate the authentication tags A_u and A_s.
We do this by replacing the tags with uniformly random values if both parties are honest. In particular, we introduce the following modifications:

- ▷ **At the simulation of honest $\mathcal{P}_s$:** If $\mathcal{P}_c$ is honest, we choose $A^*_s \leftarrow\$ \{0,1\}^\lambda$ and record $\langle \textsc{SvrTag}, sid, uid, K^*, \mathsf{tx}, A^*_s\rangle$. We construct the authentication response with A^*_s as the server tag. On the receipt of $\hat{A}_u$, if $\mathcal{P}_c$ is honest, we retrieve $\langle \textsc{UsrTag}, sid, uid, K^*, \mathsf{tx}, \{\mathbf{A^*_u}\}\rangle$ and continue the execution with A^*_u as the expected user tag.
- ▷ **At the simulation of honest $\mathcal{P}_c$:** On the receipt of $\hat{A}_s$, if $\mathcal{P}_s$ is honest, we retrieve $\langle \textsc{SvrTag}, sid, uid, K^*, \mathsf{tx}, \{\mathbf{A^*_s}\}\rangle$ and continue the execution with A^*_s as the expected server tag. If $\mathcal{P}_s$ is honest, we choose $A^*_u \leftarrow\$ \{0,1\}^\lambda$ and record $\langle \textsc{UsrTag}, sid, uid, K^*, \mathsf{tx}, A^*_u\rangle$. We send A^*_u in the last message of the protocol.
- ▷ **In both cases:** If we try to retrieve a $\textsc{SvrTag}$ or $\textsc{UsrTag}$ record but fail, then we set $A^*_u/A^*_s \coloneqq \perp$ instead. Note that this leads to a graceful abort in the corresponding verification check.

Notice that, by the same argument as in Game 34, K^* is never leaked to the adversary if both parties are honest. Accordingly, we bound the environment advantage with the Exp-PRF advantage.

Environment advantage introduced by replacing A_s with A_s^*. We modify the previous game and build $\mathcal{A}_{\mathsf{PRF1}}$ that plays Exp-PRF (see Figure 2.4) as follows:

```
On INIT from the challenger:
1: Run the simulation of the previous game.
2: Forward the output of $\varepsilon$ to the challenger.
On internal call to $F_K^{(1)}(\{\mathbf{x}\})$:
1: Return $\mathsf{Eval}(x)$.
```

1. If $b = 0$, $\mathsf{Eval}(x)$ returns $F_{K'}^{(1)}(x)$ for a fixed uniformly random key K', and the environment sees $A_s = F_{K'}^{(1)}(x)$ for a K' unknown to the environment. This is perfectly indistinguishable from the previous game.
2. If $b = 1$, $\mathsf{Eval}(x)$ returns $f(x)$ for a fixed uniformly random function f, and the environment sees $A_s' = f(x)$ where A_s' is perfectly indistinguishable from a uniformly randomly chosen value. This is perfectly indistinguishable from this game.

Environment advantage introduced by replacing A_u with A_u^*. We modify the previous game and build $\mathcal{A}_{\mathsf{PRF2}}$ that plays Exp-PRF (see Figure 2.4) as follows:

```
On INIT from the challenger:
1: Run the simulation of the previous game.
2: Forward the output of $\varepsilon$ to the challenger.
On internal call to $F_K^{(2)}(\{\mathbf{x}\})$:
1: Return $\mathsf{Eval}(x)$.
```

1. If $b = 0$, $\mathsf{Eval}(x)$ returns $F_{K'}^{(2)}(x)$ for a fixed uniformly random key K', and the environment sees $A_u = F_{K'}^{(2)}(x)$ for a K' unknown to the environment. This is perfectly indistinguishable from the previous game.
2. If $b = 1$, $\mathsf{Eval}(x)$ returns $f(x)$ for a fixed uniformly random function f, and the environment sees $A_u' = f(x)$ where A_u' is perfectly indistinguishable from a uniformly randomly chosen value. This is perfectly indistinguishable from this game.

Environment advantage introduced by setting $A_u^*/A_s^* := \perp$. We only check for the existence of a UsrTag or SvrTag record if both parties are honest. Then, if we cannot find such a record, then either tx or K^* is computed differently during the simulation of the parties.

Let k, k' be the PRF keys (i.e., K^* values) computed during the simulation of $\mathcal{P}_s$ and $\mathcal{P}_c$ respectively. Notice that k, k' are chosen uniformly randomly by the simulator and are never leaked to the adversary as we argued previously in this game. Let x, x' be the transcripts (i.e., tx values) computed during the simulation of $\mathcal{P}_s$ and $\mathcal{P}_c$ respectively. In both games we abort gracefully if for either $x \neq x'$ or $k_1 \neq k_2$, we have $F_k^{(1)}(x) \neq F_{k'}^{(1)}(x')$ or $F_k^{(2)}(x) \neq F_{k'}^{(2)}(x')$. Let bad denote the event that we gracefully abort in this game either during the simulation of $\mathcal{P}_c$ or $\mathcal{P}_s$ but not in the previous game. Then, by union bound, we have

$$\Pr[\mathsf{bad}] \leq \sum_{i \in \{1,2\}} \Pr\Big[F_k^{(i)}(x) = F_{k'}^{(i)}(x') \mid x \neq x' \vee k \neq k'\Big]$$

If $F_\cdot^{(1)}$ and $F_\cdot^{(2)}$ are PRFs with range $\{0,1\}^\lambda$, then we have

$$\Pr[\mathsf{bad}] \leq \frac{1}{2^{\lambda-1}}.$$

Overall advantage. By the two reductions above, and assuming that $F^{(1)}_{\cdot}$ and $F^{(2)}_{\cdot}$ are PRFs, we have

$$|\Pr[\mathbf{G}_{35}] - \Pr[\mathbf{G}_{34}]| \leq \mathsf{Adv}^{\text{Exp-PRF}}_{\mathcal{A}_{\mathsf{PRF1}},F^{(1)}_{\cdot}}(\lambda) + \mathsf{Adv}^{\text{Exp-PRF}}_{\mathcal{A}_{\mathsf{PRF2}},F^{(2)}_{\cdot}}(\lambda) + \frac{1}{2^{\lambda-1}}.$$

Game 36 At an honest party, we replace a graceful abort with a TestAbort message to $\mathcal{F}$. Additionally, we make sure that after sending a TestAbort message, the party stops executing for that subsession. Notice that $\mathcal{F}$ outputs "failure" to $\mathcal{A}$ followed by the corresponding Abort message to ε as in the previous game if the associated Session record is not marked as `Fresh`. Now, we go through every modification that includes a *graceful abort* up to this point and argue that at the point of that graceful abort, the relevant Session record is not marked as `Fresh`.

1. In Game 13: Before the graceful abort, $\mathcal{S}$ sends an Interrupt message for the server's Session record, which marks it as `Interrupted`.
2. In Game 14: Before the graceful abort, $\mathcal{S}$ sends a TestPwd message with password value $\perp$ for the user's Session record, which marks it as `Interrupted`.
3. In Game 15: Before the graceful abort, $\mathcal{S}$ sends an Impersonate message for the client's Session record, which marks it either as `Interrupted` or `Compromised`.
4. In Game 19: In either one of the two execution paths towards the graceful abort, $\mathcal{S}$ sends a TestPwd message for the client's Session record, which marks it either as `Interrupted` or `Compromised`.

Hence, this game is perfectly indistinguishable from the previous one.

Total Environment Advantage

We conclude our proof by giving a bound on the distinguishing advantage of ε:

$$\begin{aligned}
|\Pr[\mathbf{G}_0] - \Pr[\mathbf{G}_{36}]| \leq\ & \frac{h^2}{2^\lambda} + \frac{1}{2^\lambda} \\
& + h_2^2 \cdot \mathsf{Adv}^{\text{Exp-RKR}}_{\mathcal{A}_{\mathsf{RKR}},\mathsf{AE}}(\lambda) \\
& + \binom{h_1 + s_u}{s_s} \cdot \mathsf{Adv}^{\text{OMDH}(h_1 + s_u,\ s_s)}_{\mathcal{A}_{\mathsf{OMDH}},(\mathbb{G}_q,g_q,q)}(\lambda) \\
& + (u + \tilde{u}) \cdot \mathsf{Adv}^{\text{Exp-EQV}}_{\mathcal{A}_{\mathsf{EQV}},\mathsf{AE}}(\lambda) \\
& + (s_u + 2 \cdot u) \cdot \mathsf{Adv}^{\text{GapDH}}_{\mathcal{A}_{\mathsf{GapDH}},(\mathbb{G}_p,g_p,p)}(\lambda) \\
& + 3 \cdot \mathsf{Adv}^{\text{Exp-PRF}}_{\mathcal{A}_{\mathsf{PRF}},F^{(0)}_{\cdot}}(\lambda) \\
& + \mathsf{Adv}^{\text{Exp-PRF}}_{\mathcal{A}_{\mathsf{PRF1}},F^{(1)}_{\cdot}}(\lambda) + \mathsf{Adv}^{\text{Exp-PRF}}_{\mathcal{A}_{\mathsf{PRF2}},F^{(2)}_{\cdot}}(\lambda) + \frac{1}{2^{\lambda-1}}.
\end{aligned} \tag{6.2}$$

Chapter 7

Conclusion

In this work, we addressed some issues arising from the differences between OPAQUE [26], and the derivative IRTF draft [7]. We introduced the IRTF draft in Section 3.5 and explained the noteworthy differences in Section 3.5.3. In particular, we considered the multi-user setting and global state. To that end, we defined a new ideal functionality that accommodates these changes in Section 4 and provided the protocol that UC-realizes this functionality in Section 5. We gave our security proof in Section 6.

Future work. We now present some points that require further consideration.

- ▷ **Masking the authentication response:** In the IRTF draft, as described in Section 3.5, the authentication response produced by the server is masked with a fresh string. This is done so that the server's response to an authentication request always appears random, which provides user enumeration resistance even if the server uses the same "bogus" authentication response for all invalid authentication requests. In the augmented protocol in Section 5, we do not utilize masking. Therefore, the server must keep a database of all the invalid authentication requests as well. Clearly, the approach taken in the IRTF draft is more space efficient for the server, and their approach needs to be considered.
- ▷ **Treating OPRF as black-box:** In the IRTF draft, the OPRF functionality is used as a black-box, whereas we use the protocol described in Section 3.3 directly in augmented OPAQUE. Clearly, the approach taken in the IRTF draft leads to a more flexible construction. As explained in Section 3.3, the augmented protocol UC-realizes the ideal functionality that we give in Section 2.7.1. However, this protocol takes globally unique session identifiers as inputs which do not exist in augmented OPAQUE. To be able to treat OPRF as a black-box, this ideal functionality needs to be augmented, and the concrete implementations must be proven to UC-realize this new ideal functionality.
- ▷ **Authenticated encryption:** In the IRTF draft, the client derives its keys from a key derivation function (which is modelled as a random oracle) applied on an envelope. The authentication requirement is satisfied by using a random-key robust MAC scheme to construct a tag on the envelope (see $\mathsf{MAC2}$ in Section 3.5). In augmented OPAQUE, we use an authenticated encryption scheme directly. It is important to analyze whether their construction satisfies the security notions we need, e.g., equivocability.
- ▷ **Treating MAC as black-box:** In the IRTF draft, AKE tags (i.e., user and server tags) are implemented by a MAC module (see MAC in Section 3.5). In augmented OPAQUE, we use PRF functions directly. We note in Section 2.9 that our usage of PRFs correctly realizes a MAC with the required security properties. However, IRTF draft's approach is more flexible, and a protocol that treats MAC as a black-box would capture the draft's construction better.

Bibliography

[1] Michel Abdalla. Password-based authenticated key exchange: An overview. In Sherman S. M. Chow, Joseph K. Liu, Lucas C. K. Hui, and Siu-Ming Yiu, editors, *ProvSec 2014*, volume 8782 of *LNCS*, pages 1–9. Springer, Heidelberg, October 2014.

[2] Michel Abdalla, Björn Haase, and Julia Hesse. CPace, a balanced composable PAKE. Internet-Draft draft-irtf-cfrg-cpace-11, Internet Engineering Task Force, March 2024. Work in Progress.

[3] Manuel Barbosa, Kai Gellert, Julia Hesse, and Stanislaw Jarecki. Bare pake: universally composable key exchange from just passwords. In *Annual International Cryptology Conference*, pages 183–217. Springer, 2024.

[4] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

[5] Alexandra Boldyreva. Efficient threshold signature, multisignature and blind signature schemes based on the gap-Diffie-Hellman-group signature scheme. Cryptology ePrint Archive, Report 2002/118, 2002. `https://eprint.iacr.org/2002/118`.

[6] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography.* 2023.

[7] Daniel Bourdrez, Dr. Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. The OPAQUE Augmented PAKE Protocol. Internet-Draft draft-irtf-cfrg-opaque-16, Internet Engineering Task Force, June 2024. Work in Progress.

[8] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[9] Ran Canetti and Tal Rabin. Universal composition with joint state. Cryptology ePrint Archive, Report 2002/047, 2002. `https://eprint.iacr.org/2002/047`.

[10] Maissa Dammak, Omar Rafik Merad Boudia, Mohamed Ayoub Messous, Sidi Mohammed Senouci, and Christophe Gransart. Token-based lightweight authentication to secure iot networks. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–4. IEEE, 2019.

[11] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.

[12] Alex Davidson, Armando Faz-Hernandez, Nick Sullivan, and Christopher A. Wood. Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups. RFC 9497, December 2023.

[13] Gareth T. Davies, Sebastian H. Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the WhatsApp end-to-end encrypted backup protocol. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 330–361. Springer, Heidelberg, August 2023.

[14] Dennis Dayanikli and Anja Lehmann. (strong) apake revisited: Capturing multi-user security and salting. *Cryptology ePrint Archive*, 2024.

[15] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[16] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[17] Wesley Eddy. Transmission Control Protocol (TCP). RFC 9293, August 2022.

[18] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. Security of symmetric primitives under incorrect usage of keys. Cryptology ePrint Archive, Report 2017/288, 2017. `https://eprint.iacr.org/2017/288`.

[19] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[20] Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. KHAPE: Asymmetric PAKE from key-hiding key exchange. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 701–730, Virtual Event, August 2021. Springer, Heidelberg.

[21] Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. Cryptology ePrint Archive, Report 2018/286, 2018. `https://eprint.iacr.org/2018/286`.

[22] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, May 2011.

[23] Julia Hesse. Separating symmetric and asymmetric password-authenticated key exchange. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 579–599. Springer, Heidelberg, September 2020.

[24] David P Jablon. Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review*, 26(5):5–26, 1996.

[25] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). Cryptology ePrint Archive, Report 2016/144, 2016. `https://eprint.iacr.org/2016/144`.

[26] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Heidelberg, April / May 2018.

[27] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* CRC Press, second edition, 2014.

[28] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[29] Dr. Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[30] Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 546–566. Springer, Heidelberg, August 2005.

[31] Caroline Kudla and Kenneth G. Paterson. Modular security proofs for key agreement protocols. In Bimal K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 549–565. Springer, Heidelberg, December 2005.

[32] David Malone and Kevin Maher. Investigating the distribution of password choices. In *Proceedings of the 21st international conference on World Wide Web*, pages 301–310, 2012.

[33] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[34] Zhang Rui and Zheng Yan. A survey on biometric authentication: Toward secure and privacy-preserving identification. *IEEE access*, 7:5994–6009, 2018.

[35] Joern-Marc Schmidt. Requirements for Password-Authenticated Key Agreement (PAKE) Schemes. RFC 8125, April 2017.

[36] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

Appendix A

Practical saPAKE Functionality

We give the full definition of $\mathcal{F}_{\mathsf{psaPAKE}}$ that we introduce in Section 4. Note that $\boxed{\text{bordered}}$ interfaces can be invoked by the adversary only on a request from the environment.

```
▷ On (StorePwdFile, {sid}, {uid}, {pwd}) from S:
  1: If this is the first (StorePwdFile, $sid, uid, *$) message, then do:
    1.1: Record ⟨File, $sid, uid$, pwd, Uncompromised⟩.
    1.2: Set pwstolen[($sid, uid$)] := 0.

▷ On (UsrSession, {sid}, {uid}, {pwd}) from some user machine U:
  1: Set $\rho$ := rcount, and update rcount := rcount + 1.
  2: Record ⟨Session, $sid, uid, \rho$, Usr, pwd, Fresh, P⟩.
  3: Send ⟨UsrSession, $sid, uid, \rho$, P⟩ to $\mathcal{A}$.

▷ On (SvrSession, {sid}, {uid}) from S:
  1: Set $\rho$ := rcount, and update rcount := rcount + 1.
  2: Try to retrieve ⟨File, $sid, uid$, {pwd}, $*$⟩ or do:
    2.1: Set pwd := $\perp$.
  3: Record ⟨Session, $sid, uid, \rho$, Svr, pwd, Fresh, P⟩.
  4: Send ⟨SvrSession, $sid, uid, \rho$, P⟩ to $\mathcal{A}$.

▷ On $\boxed{\text{(StealPwdFile, \{sid\}, \{uid\})}}$ from $\mathcal{A}$:
  1: If ⟨File, $sid, uid, *, *$⟩ exists, then do:
    1.1: Update the file record's status to Compromised.
    1.2: Send "password file stolen" to $\mathcal{A}$.
  2: Else, send "no password file" to $\mathcal{A}$.

▷ On $\boxed{\text{(OfflineTestPwd, \{sid\}, \{uid\}, \{tpw\})}}$ from $\mathcal{A}$:
  1: If ⟨File, $sid, uid$, {pwd}, Compromised⟩ exists, then do:
    1.1: If tpw = pwd, then do:
      1.1.1: Set pwstolen[($sid, uid$)] := 1.
      1.1.2: Send "correct guess" to $\mathcal{A}$.
    1.2: Else, send "wrong guess" to $\mathcal{A}$.

▷ On (Interrupt, {sid}, {uid}, {$\rho$}) from $\mathcal{A}$:
  1: If ⟨Session, $sid, uid, \rho$, Svr, $*$, Fresh, $*$⟩ exists, then do:
    1.1: Update the record's status to Interrupted.
  1.2: Set $\mathsf{dPT}[(sid, uid, \rho)] := 1$.
```

▷ <u>On $(\textsc{TestPwd}, \{\mathbf{sid}\}, \{\mathbf{uid}\}, \{\rho\}, \{\mathsf{tpw}\})$ from $\mathcal{A}$:</u>

```
1: Try to retrieve $\langle \textsc{Session}, sid, uid, \rho, \{\mathsf{ty}\}, \{\mathsf{pwd}\}, \{\mathsf{st}\}, * \rangle$ or drop the message.
2: If $\mathsf{ty} = \mathtt{Svr}$ and $\mathsf{tpw} = \mathsf{pwd}$, set $\mathsf{pwstolen}[(sid, uid)] := 1$.
3: If $\mathsf{st} = \mathtt{Fresh}$, then do:
  3.1: If $\mathsf{tpw} = \mathsf{pwd}$, then do:
    3.1.1: Update the Session record's status to $\mathtt{Compromised}$.
    3.1.2: Send "correct guess" to $\mathcal{A}$.
  3.2: Else, do:
    3.2.1: Update the Session record's status to $\mathtt{Interrupted}$.
    3.2.2: Send "wrong guess" to $\mathcal{A}$.
4: Else if $\mathsf{ty} = \mathtt{Svr}$ and $\mathsf{dPT}[(sid, uid, \rho)] = 1$, then do:
  4.1: Set $\mathsf{dPT}[(sid, uid, \rho)] := 0$
  4.2: If $\mathsf{tpw} = \mathsf{pwd}$ send "correct guess" to $\mathcal{A}$.
  4.3: Else, send "wrong guess" to $\mathcal{A}$.
```

▷ <u>On $(\textsc{Impersonate}, \{\mathbf{sid}\}, \{\mathbf{uid}\}, \{\rho\})$ from $\mathcal{A}$:</u>

```
1: Try to retrieve $\langle \textsc{Session}, sid, uid, \rho, \mathtt{Usr}, \{\mathsf{pwd}\}, \mathtt{Fresh}, * \rangle$ or drop the message.
2: If there is a file record $\langle \textsc{File}, sid, uid, \mathsf{pwd}, \mathtt{Compromised} \rangle$, then do:
  2.1: Update the Session record's status to $\mathtt{Compromised}$.
  2.2: Send "correct guess" to $\mathcal{A}$.
3: Else, do:
  3.1: Update the Session record's status to $\mathtt{Interrupted}$.
  3.2: Send "wrong guess" to $\mathcal{A}$.
```

▷ <u>On $(\textsc{NewKey}, \{\mathbf{sid}\}, \{\mathbf{uid}\}, \{\hat{\mathsf{SK}}\}, \{\rho_1\}, \{\rho_2\})$ where $|\hat{\mathsf{SK}}| = \lambda$ from $\mathcal{A}$:</u>

```
1: Try to retrieve $\langle \textsc{Session}, sid, uid, \rho_2, *, *, *, \{\mathsf{P_2}\} \rangle$ or drop the message.
2: If $\langle \textsc{Binding}, \rho_2, \{\rho'_1\} \rangle$ exists and $\rho'_1 \neq \rho_1$, then abort.
3: Else, record $\langle \textsc{Binding}, \rho_1, \rho_2 \rangle$.
4: If $\langle \textsc{Session}, sid, uid, \rho_1, \{\mathsf{ty}\}, \{\mathsf{pwd}\}, \{\mathsf{st}\}, \{\mathsf{P_1}\} \rangle$ where $\mathsf{st} \neq \mathtt{Completed}$ exists, then do:
  4.1: If $\mathsf{ty} = \mathtt{Svr}$, then set $\mathsf{ty'} := \mathtt{Usr}$, else set $\mathsf{ty'} := \mathtt{Svr}$.
  4.2: If $\mathsf{st} = \mathtt{Compromised}$,
       or ($\mathsf{st} = \mathtt{Interrupted}$, and $\mathsf{ty} = \mathtt{Svr}$, and $\mathsf{pwstolen}[(sid, uid)] = 1$),
       or ($\mathsf{P}_1$ or $\mathsf{P}_2$ is corrupted),
       then set $\mathsf{SK} := \hat{\mathsf{SK}}$.
  4.3: Else if $\mathsf{st} = \mathtt{Fresh}$, and $(\textsc{Output}, sid, uid, \rho_2, \rho_1, \{\mathsf{SK'}\})$ was sent to $\mathsf{P}_2$ when there
       was a record $\langle \textsc{Session}, sid, uid, \rho_2, \mathsf{ty'}, \mathsf{pwd}, \mathtt{Fresh}, \mathsf{P}_2 \rangle$,
       then set $\mathsf{SK} := \mathsf{SK'}$.
  4.4: Else, choose $\mathsf{SK} \leftarrow\!\!\$ \{0,1\}^\lambda$.
  4.5: Update the Session record's status to $\mathtt{Completed}$.
  4.6: Send $(\textsc{Output}, sid, uid, \rho_1, \rho_2, \mathsf{SK})$ to $\mathsf{P}_1$.
```

▷ <u>On $(\textsc{TestAbort}, \{\mathbf{sid}\}, \{\mathbf{uid}\}, \{\rho_1\}, \{\rho_2\})$ from $\mathcal{A}$:</u>

```
1: If $\langle \textsc{Binding}, \rho_2, \{\rho'_1\} \rangle$ exists and $\rho'_1 \neq \rho_1$, then abort.
2: Else, record $\langle \textsc{Binding}, \rho_1, \rho_2 \rangle$.
3: If $\langle \textsc{Session}, sid, uid, \rho_1, \{\mathsf{ty}\}, \{\mathsf{pwd}\}, \{\mathsf{st}\}, \{\mathsf{P_1}\} \rangle$ where $\mathsf{st} \neq \mathtt{Completed}$ exists, then do:
  3.1: If $\mathsf{ty} = \mathtt{Svr}$, then set $\mathsf{ty'} := \mathtt{Usr}$, else set $\mathsf{ty'} := \mathtt{Svr}$.
  3.2: If $\mathsf{st} = \mathtt{Fresh}$, and $\langle \textsc{Session}, sid, uid, \rho_2, \mathsf{ty'}, \mathsf{pwd}, *, * \rangle$ exists,
       then send "success" to $\mathcal{A}$.
```

```
3.3: Else if $\mathsf{st} = \texttt{Fresh}$, and $\mathsf{ty} = \texttt{Svr}$, and $\langle \textsc{File}, sid, uid, \mathsf{pwd}, * \rangle$ exists,
     then send "success" to $\mathcal{A}$.
3.4: Else, do:
  3.4.1: Update the Session record's status to Completed.
  3.4.2: Send "failure" to $\mathcal{A}$.
  3.4.3: Send $\langle \textsc{Abort}, sid, uid, \rho_1 \rangle$ to $\mathsf{P}_1$.
```

Appendix B

Secure Registration in OPAQUE

We give an alternative registration phase for OPAQUE that does not assume secure channels, as described in [26]. In this protocol, the user doesn't learn the OPRF key k_s or the long-term private server key p_s, and the server doesn't learn the password pwd or the long-term private user key p_u.

Client$(sid, ssid, \mathsf{pwd})$		**Server**$(sid, ssid, \mathsf{file})$
$r \leftarrow\$ \mathbb{Z}_q$		
$\alpha := \mathsf{H}_1(\mathsf{pwd})^r$		
	$\xrightarrow{\quad \alpha \quad}$	
		$k_s \leftarrow\$ \mathbb{Z}_q$
		$\beta := \alpha^{k_s}$
		$p_s \leftarrow\$ \mathbb{Z}_p; P_s := g_p^{p_s}$
	$\xleftarrow{\quad \beta, P_s \quad}$	
$y := \mathsf{H}_2(\mathsf{pwd}, \beta^{1/r})$		
$p_u \leftarrow\$ \mathbb{Z}_p; P_u := g_p^{p_u}$		
$c \leftarrow \mathsf{AuthEnc}_y(p_u \parallel P_u \parallel P_s)$		
	$\xrightarrow{\quad c, P_u \quad}$	
		$\mathsf{file}[sid] := (p_s, P_u, c, k_s)$

Appendix C

Simulator for The Security Theorem

We give the full definition of the simulator $\mathcal{S}$ explained in Section 6.

```
▷ On ({tpw}) to H_1:
  1: If $\mathsf{H}_1[\mathsf{tpw}] = \bot$, then do:
    1.1: Choose $j \leftarrow\!\!\$\ \mathbb{Z}_q$.
    1.2: [Collision] If $\langle \textsc{HashedPwd}, *, j, * \rangle$ exists, then abort.
    1.3: Set $\mathsf{H}_1[\mathsf{tpw}] := g_q^j$.
    1.4: Record $\langle \textsc{HashedPwd}, \mathsf{tpw}, j, g_q^j \rangle$.
  2: Send $\mathsf{H}_1[\mathsf{tpw}]$ to $\mathcal{P}$.

▷ On ({tpw}, {b}) to H_2:
  1: Set $\mathsf{offline} :=$ (this query is made due to $\boxed{(\textsc{OfflineTestPwd}, sid, uid, \mathsf{tpw})}$ from $\varepsilon$).
  2: If $\mathsf{H}_2[(\mathsf{tpw}, b)] = \bot$, then do:
    2.1: Call $\mathsf{H}_1(\mathsf{tpw})$. // Ensure the HashedPwd record exists
    2.2: Get $\langle \textsc{HashedPwd}, \mathsf{tpw}, \{\mathbf{j}\}, \{\mathbf{h_1}\} \rangle$,
    2.3: [Key id extraction] Set $l := b^{1/j}$. // If $\exists k : b = \mathsf{H}_1(\mathsf{tpw})^k$, then $l = g_q^k$.
    2.4: If $\langle \textsc{RndPwd}, sid, \{\mathbf{uid}\}, \{\mathbf{k_s^*}\}, \{\mathbf{st}\} \rangle$ where $l = g_q^{k_s^*}$ exists, then do:
      2.4.1: If $\mathsf{offline} = \mathtt{true}$, then do:
        2.4.1.1: Send $(\textsc{OfflineTestPwd}, sid, uid, \mathsf{tpw})$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
      2.4.2: Else, do:
        2.4.2.1: Pop the first $\langle \textsc{BlindedEval}, sid, uid, l, \{\rho_\mathbf{s}\} \rangle$ or abort.
        2.4.2.2: Send $(\textsc{TestPwd}, sid, uid, \rho_s, \mathsf{tpw})$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
      2.4.3: [$\mathcal{A}$ guessed the password] If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "correct guess", then do:
        2.4.3.1: Record $\langle \textsc{StolenPwd}, sid, uid, \mathsf{tpw} \rangle$.
        2.4.3.2: Retrieve $\langle \textsc{UsrCreds}, sid, uid, \{\mathbf{p_u^*}\}, *, \{\mathbf{P_u^*}\}, \{\mathbf{P_s^*}\}, * \rangle$ or abort.
        2.4.3.3: Set $m := p_u^* \parallel P_u^* \parallel P_s^*$.
        2.4.3.4: Set $h_2 \leftarrow \mathsf{SIM}_{\mathrm{EQV}}(m, st)$.
    2.5: Else, do:
      2.5.1: Choose $h_2 \leftarrow\!\!\$\ \{0,1\}^\lambda$.
      2.5.2: [Future collision] If $\langle \textsc{RndPwd}, sid, \ldots, h_2 \rangle$ exists, then abort.
    2.6: [Collision] If $\mathsf{H}_2[(\mathsf{tpw}', b')] = h_2$ for some $(\mathsf{tpw}', b')$, then abort.
    2.7: Record $\langle \textsc{OprfEval}, \mathsf{tpw}, l, h_2 \rangle$.
    2.8: Set $\mathsf{H}_2[(\mathsf{tpw}, b)] := h_2$.
  3: Send $\mathsf{H}_2[(\mathsf{tpw}, b)]$ to $\mathcal{P}$.

▷ On ({x}, {y}, {z}) to H_3:
  1: If $\mathsf{H}_3[(x, y, z)] = \bot$, then do:
```

```
  1.1: If $\langle \textsc{ForbiddenInput}, sid, *, (x, \perp, \perp)\rangle$ exists, and both $\mathcal{P}_c$ and $\mathcal{P}_s$ are honest,
       then abort.
  1.2: Else if $\langle \textsc{ForbiddenInput}, sid, *, (\perp, y, \perp)\rangle$ exists, then abort.
  1.3: Else if $\langle \textsc{ForbiddenInput}, sid, \{\mathbf{uid}\}, (\perp, \perp, z)\rangle$ and $\langle \textsc{StolenPwd}, sid, uid, *\rangle$ exists, then abort.
  1.4: Else if either $\langle \textsc{SvrPrfKey}, sid, (x, y, z), \{\mathbf{k}\}\rangle$ or $\langle \textsc{UsrPrfKey}, sid, *, (x, y, z), \{\mathbf{k}\}\rangle$
       exists, then set $h_3 \coloneqq k$.
  1.5: Else, choose $h_3 \leftarrow\$ \mathcal{K}$.
  1.6: Set $\mathsf{H}_3[(x, y, z)] \coloneqq h_3$.
2: Set $k \coloneqq \mathsf{H}_3[(x, y, z)]$.
3: Send $k$ to $\mathcal{P}$.
```

▷ <u>On $(\textsc{UsrSession}, \{\mathbf{sid}\}, \{\mathbf{uid}\}, \{\rho_\mathbf{c}\}, *)$ from $\mathcal{F}_{\mathsf{psaPAKE}}$ to $\mathcal{A}_\perp$:</u>

```
1: Record $sid, uid$.
2: Choose $\delta_c \leftarrow\$ \{0,1\}^\lambda$.
3: If $\delta_c$ was chosen before, abort.
4: Record $\langle \textsc{ClientRecord}, sid, uid, \rho_c, \delta_c\rangle$.
5: Choose $x_u^* \leftarrow\$ \mathbb{Z}_p$, and set $X_u^* \coloneqq g_p^{x_u^*}$.
6: If $\langle \textsc{UsrCreds}, sid, uid, *, \{\mathbf{p_s^*}\}, \ldots\rangle$ exists, then do:
  6.1: Record $\langle \textsc{ForbiddenInput}, sid, uid, (\perp, X_u^{*p_s^*}, \perp)\rangle$
7: Record $\langle \textsc{ClientState}, sid, uid, \delta_c, \rho_c, x_u^*, X_u^*\rangle$.
8: Choose $r^* \leftarrow\$ \mathbb{Z}_q$, set $\alpha^* \coloneqq g_q^{r^*}$
9: Record $\langle \textsc{BlindedPwd}, sid, uid, \delta_c, r^*, \alpha^*\rangle$.
10: Send $(\delta_c, X_u^*, \alpha^*)$ to $\mathcal{P}_s$.
```

▷ <u>On $(\{\hat{\delta_\mathbf{s}}\}, \{\hat{\mathbf{X_s}}\}, \{\hat{\beta}\}, \{\hat{\mathbf{c}}\}, \{\hat{\mathbf{A_s}}\})$ from $\mathcal{A}_\perp$in response to $(\hat{\delta_c}, \ldots)$:</u>

```
1: Try to retrieve $\langle \textsc{ClientState}, sid, uid, \hat{\delta_c}, \{\mathbf{x_u^*}\}, \{\mathbf{X_u^*}\}\rangle$ or drop the message.
2: Try to retrieve $\langle \textsc{BlindedPwd}, sid, uid, \hat{\delta_c}, \{\mathbf{r^*}\}, \{\alpha^*\}\rangle$ or abort.
3: Try to retrieve $\langle \textsc{ClientRecord}, sid, uid, \{\rho_\mathbf{c}\}, \hat{\delta_c}\rangle$ or abort.
4: [Ensure server session exists] Try to retrieve $\langle \textsc{ServerRecord}, sid, uid, \{\rho_\mathbf{s}\}, \hat{\delta_s}\rangle$ or do:
  4.1: Send $(\textsc{SvrSession}, sid, uid)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$, and receive $(\textsc{SvrSession}, sid, uid, \{\rho_\mathbf{s}\}, *)$.
  4.2: Record $\langle \textsc{ServerRecord}, sid, uid, \rho_s, \hat{\delta_s}\rangle$.
5: [Honest execution] If $\mathcal{P}_s$ is honest
   and $\langle \textsc{RndPwd}, sid, uid, \{\mathbf{k_s^*}\}, \{\mathbf{st}\}\rangle$ where $\hat{\beta} = \alpha^{*k_s^*}$ exists,
   and $\langle \textsc{UsrCreds}, sid, uid, \{\mathbf{p_u^*}\}, *, \{\mathbf{P_u^*}\}, \{\mathbf{P_s^*}\}, *, \hat{c}\rangle$ exists,
   and $\langle \textsc{StolenKey}, sid, uid, \ldots\rangle$ doesn't exist, then do:
  5.1: [Check $\mathsf{upw} = \mathsf{spw}$] Send $(\textsc{TestAbort}, sid, uid, \rho_c, \rho_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
  5.2: [$\mathsf{upw} = \mathsf{spw}$] If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "success", then do:
    5.2.1: [$\mathcal{A}$ stole the password] If $\langle \textsc{StolenPwd}, sid, uid, \{\mathsf{spw}\}\rangle$ exists, then do:
      5.2.1.1: Send $(\textsc{TestPwd}, sid, uid, \rho_c, \mathsf{spw})$ to $\mathcal{F}_{\mathsf{psaPAKE}}$. // Compromise the user
    5.2.2: Jump to line 9.
  5.3: [$\mathsf{upw} \neq \mathsf{spw}$] Else, stop. // Already aborted by $\mathcal{F}_{\mathsf{psaPAKE}}$
6: [Adversarial execution] Else, do:
  6.1: Find the randomized password $y$ such that:
    6.1.1: At least one of the following hold:
      6.1.1.1: [$\hat{\beta}$ is well-formed] $\langle \textsc{OprfEval}, \{\mathbf{x}\}, *, \{\mathbf{y}\}\rangle$ where $y = \mathsf{H}_2[(x, \hat{\beta}^{1/r^*})]$ exists.
      6.1.1.2: [$\hat{\beta}$ is from honest $\mathcal{P}_s$] $\langle \textsc{RndPwd}, sid, uid, \{\mathbf{k_s^*}\}, *\rangle$ where $\alpha^{*k_s^*} = \hat{\beta}$ exists.
    6.1.2: $\mathsf{AE.AuthDec}_y(\hat{c}) \neq \perp$.
  6.2: [Random-key robustness] If multiple different such values exist, then abort.
  6.3: [$\hat{c}$ decrypts with queried $y$] Else if $\langle \textsc{OprfEval}, \{\mathsf{apw}\}, *, \{\mathbf{y'}\}\rangle$ exists, then do:
      6.3.1: Set $y = y'$.
      6.3.2: [Check upw = apw] Send $(\textsc{TestPwd}, sid, uid, \rho_c, \mathsf{apw})$ to $\mathcal{F}_{\mathsf{psaPAKE}}$
      6.3.3: [upw $\neq$ apw $\rightarrow \mathcal{P}_c$ won't get $y$] If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "wrong guess",
             then set $y := \perp$.
    6.4: [$\hat{c}$ decrypts with $y^*$] Else if *only* $\langle \textsc{RndPwd}, sid, uid, *, \{\mathbf{st}\}\rangle$ exists, then do:
      6.4.1: Retrieve $\langle \textsc{UsrCreds}, sid, uid, \{\mathbf{p_u^*}\}, *, \{\mathbf{P_u^*}\}, \{\mathbf{P_s^*}\}, *\rangle$ or abort.
      6.4.2: Set $\hat{p_u} := p_u^*$, $\hat{P}_u := P_u^*$, $\hat{P}_s := P_s^*$.
      6.4.3: [Check upw = spw] Send $(\textsc{Impersonate}, sid, uid, \rho_c)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
      6.4.4: [upw $\neq$ spw $\rightarrow \mathcal{P}_c$ can't decrypt] If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "wrong guess",
             then set $y := \perp$.
      6.4.5: Else, jump to line 9.
  7: [No $y$ can decrypt $\hat{c}$ with upw] If $y = \perp$, then do:
    7.1: Send $(\textsc{TestPwd}, sid, uid, \rho_c, \perp)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$. // Interrupt the user "session".
    7.2: Send $(\textsc{TestAbort}, sid, uid, \rho_c, \rho_s, \mathcal{P}_c)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$, and stop.
  8: Try to decrypt $(\hat{p_u}, \hat{P}_u, \hat{P}_s) := \mathsf{AE.AuthDec}_y(\hat{c})$ or abort.
  9: Set $\mathsf{in\_keys} := (\hat{X}_s{}^{x_u^*}, \hat{P}_s{}^{x_u^*}, \hat{X}_s{}^{\hat{p_u}})$.
 10: If $\langle \textsc{SvrPrfKey}, sid, \mathsf{in\_keys}, \{\mathbf{k}\}\rangle$ exists, then set $K^* := k$.
 11: Else, choose $K^* \leftarrow\!\$\ \mathcal{K}$.
 12: Record $\langle \textsc{UsrPrfKey}, sid, uid, \mathsf{in\_keys}, K^*\rangle$.
 13: If $\langle \textsc{SvrState}, sid, uid, \hat{\delta_s}, \delta_c, *, \{\mathbf{X_s^*}\}, \ldots\rangle$ exists and $X_s^* \neq \hat{X}_s$,
     then send $(\textsc{TestPwd}, sid, uid, \delta_c, \perp)$ to $\mathcal{F}$.
 14: Set $\mathsf{tx} := (sid \,\|\, uid \,\|\, \delta_c \,\|\, \hat{\delta_s} \,\|\, \alpha^*)$.
 15: Set $SK^* := F^{(0)}_{K^*}(\mathsf{tx})$.
 16: If $\mathcal{P}_s$ is honest, then do:
   16.1: Try to retrieve $\langle \textsc{SvrTag}, sid, uid, K^*, \mathsf{tx}, \{\mathbf{A_s^*}\}\rangle$ or set $A_s^* := \perp$.
   16.2: Choose $A_u^* \leftarrow\!\$\ \{0,1\}^\lambda$.
   16.3: Record $\langle \textsc{UsrTag}, sid, uid, \rho_c, \rho_s, A_u^*\rangle$.
 17: Else, set $A_s^* := F^{(1)}_{K^*}(\mathsf{tx})$, $A_u^* := F^{(2)}_{K^*}(\mathsf{tx})$.
 18: If $A_s^* \neq \hat{A}_s$, then do:
   18.1: Send $(\textsc{TestPwd}, sid, uid, \rho_c, \perp)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
   18.2: Send $(\textsc{TestAbort}, sid, uid, \rho_c, \rho_s, \mathcal{P}_c)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$, and stop.
 19: Send $(\textsc{NewKey}, sid, uid, \rho_c, \rho_s, \mathsf{SK}^*, \mathcal{P}_c, \mathcal{P}_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
 20: Send $A_u^*$ to $\mathcal{P}_s$.

▷ On $\boxed{(\textsc{StealPwdFile}, sid, \{\mathbf{uid}\})}$ from $\mathcal{A}_\perp$:

  1: Send $\langle \textsc{StealPwdFile}, sid, uid\rangle$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
  2: If $\mathcal{F}_{\mathsf{psaPAKE}}$ returns "password file stolen", then do:
    2.1: Try to retrieve $\langle \textsc{RndPwd}, sid, uid, \{\mathbf{k_s^*}\}, *\rangle$ or abort.
    2.2: Record $\langle \textsc{StolenKey}, sid, uid, k_s^*, g_q^{k_s^*}\rangle$.
  3: Forward $\mathcal{F}_{\mathsf{psaPAKE}}$'s response to $\mathcal{A}_\perp$.

▷ On $(\textsc{SvrSession}, sid, \{\mathbf{uid}\}, \{\rho_\mathbf{s}\})$ from $\mathcal{F}_{\mathsf{psaPAKE}}$ to $\mathcal{A}_\perp$:

  1: Choose $\delta_s \leftarrow\!\$\ \{0,1\}^\lambda$, and associate $\rho_s$ with $\delta_s$.
  2: If $\delta_s$ was chosen before, abort.
  3: Record $\langle \textsc{ServerRecord}, sid, uid, \rho_s, \delta_s\rangle$.
  4: Record $\langle \textsc{NewSvrSession}, sid, uid, \rho_s, \delta_s\rangle$.
  5: If $\langle \textsc{UsrCreds}, sid, uid, ...\rangle$ does not exist, then do:
    5.1: Choose $p_u^*, p_s^* \leftarrow\!\$\ \mathbb{Z}_p$, and set $P_s^* := g_p^{p_s^*}$, $P_u^* := g_p^{p_u^*}$.
    5.2: Set $m^* := p_u^* \,\|\, P_u^* \,\|\, P_s^*$.
    5.3: Set $(c^*, st) := \mathsf{SIM}_{\mathrm{EQV}}(|m|)$.
    5.4: Record $\langle \textsc{UsrCreds}, sid, uid, p_u^*, p_s^*, P_u^*, P_s^*, c^* \rangle$.
    5.5: Choose $k_s^* \leftarrow\$ \mathbb{Z}_q$.
    5.6: Record $\langle \textsc{RndPwd}, sid, uid, k_s^*, st \rangle$.

▷ On $(\{\mathbf{uid}\}, \{\hat{\delta}_{\mathbf{c}}\}, \{\hat{\mathbf{X}}_{\mathbf{u}}\}, \{\hat{\alpha}\})$ from $\mathcal{A}_\perp$:
  1: Send $(\textsc{SvrSession}, sid, uid)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
  2: Pop the first $\langle \textsc{NewSvrSession}, sid, uid, \{\rho_{\mathbf{s}}\}, \{\delta_{\mathbf{s}}\} \rangle$.
  3: Try to retrieve $\langle \textsc{UsrCreds}, sid, uid, \{\mathbf{p}_{\mathbf{u}}^*\}, \{\mathbf{p}_{\mathbf{s}}^*\}, \{\mathbf{P}_{\mathbf{u}}^*\}, \{\mathbf{P}_{\mathbf{s}}^*\}, \{\mathbf{c}^*\} \rangle$ or abort.
  4: Try to retrieve $\langle \textsc{RndPwd}, sid, uid, \{\mathbf{k}_{\mathbf{s}}^*\}, * \rangle$ or abort.
  5: [Ensure user session exists] Try to retrieve $\langle \textsc{ClientRecord}, sid, uid, \{\rho_{\mathbf{c}}\}, \hat{\delta}_c \rangle$ or do:
    5.1: Send $(\textsc{UsrSession}, sid, uid, \perp)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$,
         and receive $(\textsc{UsrSession}, sid, uid, \{\rho_{\mathbf{c}}\}, *)$.
    5.2: Record $\langle \textsc{ClientRecord}, sid, uid, \rho_c, \hat{\delta}_c \rangle$.
  6: Set $\beta^* := \hat{\alpha}^{k_s^*}$.
  7: Choose $x_s^* \leftarrow\$ \mathbb{Z}_p$, and set $X_s^* := g_p^{x_s^*}$.
  8: If $\langle \textsc{ClientState}, sid, uid, \hat{\delta}_c, *, *, \{\mathbf{X}_{\mathbf{u}}^*\} \rangle$ exists, then do:
    8.1: Record $\langle \textsc{ForbiddenInput}, sid, uid, (X_u^{*x_s^*}, \perp, \perp) \rangle$.
    8.2: If $X_u^* \neq \hat{X}_u$, then send $(\textsc{Interrupt}, sid, uid, \rho_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
  9: Record $\langle \textsc{ForbiddenInput}, sid, uid, (\perp, \perp, P_u^{*x_s^*}) \rangle$.
  10: Set $\mathsf{in\_keys} := (\hat{X}_u^{x_s^*}, \hat{X}_u^{p_s^*}, P_u^{*x_s^*})$.
  11: Choose $K^* \leftarrow\$ \mathcal{K}$.
  12: Record $\langle \textsc{SvrPrfKey}, sid, \mathsf{in\_keys}, K^* \rangle$.
  13: Set $\mathsf{tx} := (sid \parallel uid \parallel \hat{\delta}_c \parallel \delta_s \parallel \hat{\alpha})$,
  14: If $\mathcal{P}_c$ is honest, then do:
    14.1: Choose $A_s^* \leftarrow\$ \{0,1\}^\lambda$.
    14.2: Record $\langle \textsc{SvrTag}, sid, uid, \rho_c, \rho_s, A_u^* \rangle$.
  15: Else, set $A_s^* := F_{K^*}^{(1)}(\mathsf{tx})$.
  16: Record $\langle \textsc{SvrState}, sid, uid, \delta_s, \hat{\delta}_c, c^*, X_s^*, K^*, \mathsf{tx}, A_s^* \rangle$.
  17: Record $\langle \textsc{BlindedEval}, sid, uid, g_q^{k_s^*}, \rho_s \rangle$.
  18: Send $(\delta_s, X_s^*, c^*, A_s^*, \beta^*)$ to $\mathcal{P}_c$.

▷ On $(\{\hat{\mathbf{A}}_{\mathbf{u}}\})$ from $\mathcal{A}_\perp$in response to $(\hat{\delta}_s, \ldots)$:
  1: Try to retrieve $\langle \textsc{SvrState}, sid, uid, \hat{\delta}_s, \{\delta_{\mathbf{c}}\}, *, *, \{\mathbf{K}^*\}, \{\mathsf{tx}\}, * \rangle$ or drop the message.
  2: Try to retrieve $\langle \textsc{ServerRecord}, sid, uid, \{\rho_{\mathbf{s}}\}, \hat{\delta}_s \rangle$ or abort.
  3: Try to retrieve $\langle \textsc{ClientRecord}, sid, uid, \{\rho_{\mathbf{c}}\}, \delta_c \rangle$ or abort.
  4: If $\mathcal{P}_c$ is honest, then try to retrieve $\langle \textsc{UsrTag}, sid, uid, K^*, \mathsf{tx}, \{\mathbf{A}_{\mathbf{u}}^*\} \rangle$ or set $A_u^* := \perp$.
  5: Else, set $A_u^* := F_{K^*}^{(2)}(\mathsf{tx})$
  6: Set $\mathsf{SK}^* := F_{K^*}^{(0)}(\mathsf{tx})$.
  7: If $A_u^* = \hat{A}_u$, then do:
    7.1: Send $(\textsc{NewKey}, sid, uid, \mathsf{SK}^*, \rho_s, \rho_c, \mathcal{P}_s, \mathcal{P}_c)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
  8: Else, do:
    8.1: Send $(\textsc{Interrupt}, sid, uid, \rho_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
    8.2: Send $(\textsc{TestAbort}, sid, uid, \rho_s, \rho_c, \mathcal{P}_s)$ to $\mathcal{F}_{\mathsf{psaPAKE}}$.
```