Separating Standard and Asymmetric Password-Authenticated Key Exchange

Julia Hesse

IBM Research, Zurich, Switzerland
juliahesse2@gmail.com

Abstract

Password-Authenticated Key Exchange (PAKE) is a method to establish cryptographic keys between two users sharing a low-entropy password. In its asymmetric version, one of the users acts as a server and only stores some function of the password, e.g., a hash. Upon server compromise, the adversary learns $H(\mathsf{pw})$. Depending on the strength of the password, the attacker now has to invest more or less work to reconstruct pw from $H(\mathsf{pw})$. Intuitively, asymmetric PAKE seems more challenging than standard (symmetric) PAKE since the latter is not supposed to protect the password upon compromise. In this paper, we provide three contributions:

- **Separating standard and asymmetric PAKE.** We prove that a strong assumption like a programmable random oracle is necessary to achieve security of asymmetric PAKE in the Universal Composability (UC) framework. For standard PAKE, programmability is not required.
- **Revising the security definition.** We identify and close a gap in the UC security definition of 2-party asymmetric PAKE given by Gentry, MacKenzie and Ramzan (Crypto 2006). For this, we specify a natural corruption model for server compromise attacks. We further remove an undesirable weakness that lets parties wrongly believe in security of compromised session keys. We demonstrate usefulness by proving that the Ω-protocol proposed by Gentry et al. satisfies our new security notion for aPAKE.
- **Composable multi-party aPAKE.** We demonstrate that reliance on a programmable random oracle hinders construction of multi-party aPAKE protocols from 2-party protocols via UC composition. Namely, the resulting protocols offer such strong security guarantees that they become impractical in any application. We provide guidance on how to relax composable security notions for multi-party asymmetric aPAKE to obtain useful protocols.

Keywords: Asymmetric Password-Authenticated Key Exchange, Universal Composability

1 Introduction

Establishing secure communication channels in untrusted environments is an important measure to ensure privacy, authenticity or integrity on the internet. The most important cryptographic building blocks for securing channels are key exchange protocols. The exchanged keys can be used to, e.g., encrypt messages using a symmetric cipher, or to authenticate users. *Password-authenticated key exchange* (PAKE), introduced by Bellovin and Merrit [BM92a], is a method to establish cryptographic keys between two users sharing a *password*. A PAKE manages to "convert" this possibly low-entropy password into a random-looking key with high entropy, which is the same for both users if and only if they both used the same password. What makes these schemes interesting for practice is that they tie authentication solely to passwords, while other methods such as password-over-TLS involve more authentication material such as a certificate. The probably most prominent implementation of PAKE is the TLS-SRP ciphersuite[1], which is used by GnuTLS, OpenSSL and Apache.

[1] Specified in RFCs 2945 and 5054.

reference	type	sec. notion	model
[BPR00]	standard	BPR	non-prog. RO
[GL01]	standard	BPR	standard
[KOY01]	standard	BPR	standard
[BCL+11]	standard	BPR	standard
[KV11]	standard	BPR	standard
[BP13]	asymmetric	BPR	non-prog. RO/GGM
[PW17]	asymmetric	BPR	GGM
[CHK+05]	standard	UC	standard
[KV11]	standard	UC	standard
[BBC+13]	standard	UC	standard
[GMR06]	asymmetric	UC	progr. RO
[JR16]	asymmetric	UC	limited progr. RO
[HL18]	asymmetric	UC	progr. RO
[JKX18]	asymmetric	UC	progr. RO

Figure 1: Comparison of static security of different PAKE and aPAKE schemes. In the above, RO means random oracle and GGM means generic group model.

In most applications, users of a PAKE actually take quite different roles. Namely, some may act as servers, maintaining sessions with various clients, and storing passwords of clients in a file. For better security, it seems reasonable to not write the password to the file system in the clear, but store, e.g., a hash of the password. A PAKE protocol that lets users take the roles of a client or a server is called *asymmetric* or *augmented*[2] PAKE (aPAKE).

Security of PAKE. Since a password is potentially of low entropy, an attacker can always engage in a PAKE execution with another user by just trying a password, resulting in key agreement with non-negligible probability. Such an attack is called an *on-line dictionary attack* since the attacker only has one password guess per run of the protocol. A security requirement for standard PAKE is that an on-line dictionary attack is the "worst the adversary can do". Especially, the attacker should not be able to mount *off-line* dictionary attacks on the password, e.g., by deriving information about the password by just looking at the transcript. For an asymmetric PAKE, we can require more: if an attacker gets his hands on a password file, in which case the server is called *compromised*, the attacker should not learn the password directly. At least, some computation such as hashing password guesses is necessary.

Is aPAKE harder than PAKE? Intuitively, asymmetric PAKE seems more challenging than standard PAKE already in the setting of static security, since standard PAKE protocols are supposed to protect the password whenever the storage of a user is leaked to the adversary. This claim is supported by schemes from the literature: there are PAKE schemes that are BPR-secure (BPR is the most widely used game-based notion for PAKE, introduced by Bellare et al. [BPR00]) in the standard model, while current aPAKE schemes satisfying the asymmetric variant of BPR security are only proven in an idealized model such as the non-programmable random oracle or the generic group model. The situation is similar for security of PAKE/aPAKE in the Universal Composability (UC) framework of Canetti [Can01]. UC-secure PAKE protocols exist in the non-programmable random oracle model and even in the standard model, while proofs of current aPAKE schemes additionally rely on some form of programmability of the random oracle. See Figure 1 for a comparison.

However, to our knowledge, in none of the aforementioned models there exist any formal evidence for asymmetric PAKE being harder to achieve than standard PAKE. In particular, Figure 1 demonstrates that the "gap" in the model is much bigger in case of UC security. It thus seems reasonable to hope that UC secure asymmetric PAKE can also be constructed from rather mild versions of idealized assumptions, such as a non-programmable random oracle. This is particularly interesting since it is agreed upon in the literature that UC security is the right security notion for PAKE protocols. Besides composability guarantees, the most notable difference to BPR-security is that UC-secure PAKE protocols remain secure even when passwords

[2]To emphasize that we talk about a PAKE protocol without different roles, we write *standard* or *symmetric* PAKE.

are adversarially chosen.

1.1 Our Contributions

In this paper, we rule out the existence of UC-secure aPAKE protocols from assumptions that are enough to obtain (even adaptively) UC-secure standard PAKE. Namely, we show that aPAKE is impossible to achieve w.r.t a non-programmable random oracle. To our knowledge, this is the first formal evidence that universally composable aPAKE is harder to achieve than PAKE.

In preparation of this formal result, we revisit the ideal functionality $\mathcal{F}_{\mathsf{apwKE}}$ for asymmetric PAKE of Gentry, MacKenzie and Ramzan [GMR06]. In more detail, our contributions are as follows:

- We show that $\mathcal{F}_{\mathsf{apwKE}}$ is not realizable due to an incorrect modeling of server compromise attacks. We fix this by formally viewing server compromise as partial corruption of the server[3].
- We show that $\mathcal{F}_{\mathsf{apwKE}}$ allows attacks on explicit authentication[4]. An adversary exploiting this weakness can make parties believe in the security of adversarially chosen session keys. We exclude such attacks by introducing a stronger modeling of explicit authentication to the functionality.

We argue plausibility of our revisited functionality $\mathcal{F}_{\mathsf{aPAKE}}$ by showing that it is realized by the two-party version of the Ω-protocol from [GMR06]. By this, we show that the Ω-protocol actually provides stronger security guarantees regarding explicit authentication than originally claimed in [GMR06].

All aforementioned results hold for asymmetric PAKE protocols run between two users. As a further contribution, we investigate security of aPAKE protocols that are run by multiple users who want to bilaterally exchange password-authenticated session keys. Our contributions are as follows:

- We show that our result established above, namely reliance on a programmable random oracle, yields 2-party protocols that become quite impractical when run in a context with multiple parties. While it is well known that overly restrictive security requirements lead to impractical protocols, we demonstrate that, for aPAKE protocols, issues remain unnoticed in a 2-party scenario but become visible when the protocol is executed by more than two users.
- To alleviate the aforementioned issue, multi-party security notions for aPAKE need to be relaxed. We provide a formal property that such notions need to fulfill to allow for practical protocols.

Let us now explain our results in more detail.

Separating standard and asymmetric PAKE. As already mentioned, asymmetric PAKE protocols are supposed to provide some protection of the password in case of a server compromise. Let us elaborate on this kind of attack. Naturally, for a server compromise to make sense, we can assume that the server already stored a password file at the time it gets compromised. Formally, this means that the attacker is allowed to obtain some partial inner state of an honest party who already ran parts of the protocol. Note that the adversary is however not allowed to control the behaviour of the compromised server, which distinguishes compromising a server from corrupting a server. Nevertheless, a server compromise allows the attacker to mount an *adaptive* attack.

In simulation-based security notions such as notions stated in the UC model, adaptive attacks often impose a security problem. Such problems are often referred to as "commitment problem", since they require the simulator to explain how, e.g., a transcript that he simulated in the beginning of the protocol matches certain secrets of honest participants that are revealed at a later stage of the protocol run. The first mentioning of such a commitment problem is the work of Nielsen [Nie02], who showed that a non-programmable random oracle (NPRO) is not enough to obtain non-committing encryption. One contribution of the paper is to formalize non-programmable random oracles. In a nutshell, such an oracle is modeled as an external oracle that informs the adversary about all queries, but chooses values truly at random, especially not letting the adversary in any way influence the outputs of the oracle.

[3] This was already proposed but not enforced in [GMR06].

[4] In a authenticated key exchange protocol with explicit authentication, the parties are reliably informed about the outcome of the authentication.

Inspired by the work of Nielsen, we obtain the following result: UC-secure asymmetric PAKE is impossible to achieve in the NPRO model. Namely, we show that due to the adaptive nature of the server compromise attack, the simulator needs to commit to a password file without knowing the password that the file is containing. When the attacker tries to reconstruct the file from the true password, since accessing the external oracle is sufficient to compute the file, the simulator merely learns the true password but cannot influence the computation of the file anymore.

While the result itself is not very surprising, we stress that the techniques to prove it are actually completely different from the techniques used by Nielsen [Nie02]. In the case of non-committing encryption, the proof technique is to let the simulator commit to "too many" ciphertexts such that there simply does not exist a secret key of reasonable size to explain all these ciphertexts later. However, in asymmetric PAKE the situation is quite different, since there exists only one file per password. Indeed, there is a bit more hope for a simulator of asymmetric PAKE to actually find a good password file if he only guesses the password correctly. Our formal argument thus heavily relies on the fact that, in the UC model, the simulator does not have an arbitrary amount of runtime to simulate the password file. We formally prove that, with very high probability, he will exhaust before finding a good file.

Contrarily, it is known from the literature that UC-secure standard PAKE can be constructed in the NPRO model and even in the standard model [ACCP08, CHK$^+$05, KV11, DHP$^+$18]. We note that, while also the NPRO model suffers from the uninstantiability results of [CGH98], requiring programmability of a random oracle is crucially strengthening the model. In a security proof w.r.t an NPRO, the reduction does not need to determine any output values and thus could use, e.g., a hash function like SHA-3 to answer random oracle queries. This is not possible for a reduction that makes use of the programmability property of the random oracle. Thus, our results indicate that, while going from standard to asymmetric PAKE in the UC model, we are forced to move further away from realistic setup assumptions.

Modeling server compromise. Towards separating standard and asymmetric PAKE, we first carefully revisit the ideal functionality $\mathcal{F}_{\mathsf{apwKE}}$ for asymmetric PAKE from [GMR06], which adopts the ideal functionality for standard PAKE from [CHK$^+$05] to the asymmetric case. Besides assigning the roles of a server and a client to the users, $\mathcal{F}_{\mathsf{apwKE}}$ features interfaces that let the adversary mount a server compromise attack with all its consequences: now the adversary is able to (a) submit off-line password guesses and (b) impersonate the server. To model the fact that a server compromise is an attack that might help the UC environment $\mathcal{Z}$ distinguishing real and simulated protocol runs, $\mathcal{F}_{\mathsf{apwKE}}$ only allows the adversary to ask StealPwdFile queries upon instruction from $\mathcal{Z}$.

We first prove that restricting the ideal world adversary to only submit off-line password guesses upon instructions from $\mathcal{Z}$ results in a security notion that is impossible to realize. Secondly, putting restrictions such as "only ask query x if $\mathcal{Z}$ tells you" on the ideal world adversary is not conform with the UC framework and indeed invalidates important properties of the UC framework such as simulation with respect to the dummy adversary. Towards a better modeling, we first lift the aforementioned restriction on the ideal world adversary and argue why the resulting security notion captures what we expect from an asymmetric PAKE. Secondly, we propose a UC-conform modeling of server compromise attacks as "partial" corruption queries which, in the real execution of the protocol, leak a function of the internal state of an honest party to the adversary (e.g., the password file).

We call our revisited functionality for asymmetric PAKE $\mathcal{F}_{\mathsf{aPAKE}}$. It however differs in another aspect from $\mathcal{F}_{\mathsf{apwKE}}$ as mentioned above, which we will now explain in more detail.

Modeling explicit authentication. A protocol is said to have *explicit authentication* if the parties can learn whether the key agreement was successful, in which case they might opt for, e.g., reporting failure. $\mathcal{F}_{\mathsf{apwKE}}$ features a TestAbort interface which allows the adversary to obtain information about the authentication status and also to decide whether parties should abort if their computed session keys do not match. The idea behind modeling explicit authentication via an interface that the adversary may or may not decide to use is to keep $\mathcal{F}_{\mathsf{apwKE}}$ flexible: both protocols with or without explicit authentication can be proven to realize it. However, we show that this results in $\mathcal{F}_{\mathsf{apwKE}}$ providing very weak security guarantees regarding explicit authentication. One property that a protocol with authentication should have is that parties reliably abort if they detect authentication failure. However, $\mathcal{F}_{\mathsf{apwKE}}$ does not enforce this property since the adversary can simply decide *not* to use the TestAbort interface. We propose a stronger version of $\mathcal{F}_{\mathsf{aPAKE}}$ that enforces explicit authentication *within the functionality* and merely informs the adversary

about the status of authentication.

WHY IS IT IMPORTANT TO ANALYZE MULTI-PARTY UC SECURITY? One of the main benefits of the UC framework is that it comes with a composition theorem. The essence of this theorem is that whatever guarantee a protocol gives when considering an isolated run of it, it will maintain when running in an arbitrary context. For key exchange protocols, which are never the ultimate goal in any higher level application, secure composition with arbitrary other protocols is essential. Moreover, the same protocol suite is run by a large number of users and thus multiple instances of the key exchange protocol are executed. To name an example, consider protocols for user-friendly password authentication in a landscape of distributed servers and service providers (e.g., Single-Sign-On schemes or distributed authentication schemes). To facilitate security analysis of such a protocol suite which uses aPAKE as a building block, we want to assume that all the single aPAKE instances behave in an ideal way.

HOW TO GO FROM 2-PARTY APAKE TO MULTI-PARTY APAKE. $\mathcal{F}_{\mathsf{apwKE}}$ as well as our $\mathcal{F}_{\mathsf{aPAKE}}$ are *two-party* functionalities running with one client and one server. This means they let us only analyze security of a very small part of the whole protocol. This however is usually not a problem: the UC framework comes with a composition theorem, which allows to instantiate an arbitrary number of instances of the two-party functionality with the real protocol. E.g., in the above scenario, each client would invoke an instance of $\mathcal{F}_{\mathsf{aPAKE}}$, and the server participates in all of them. To avoid that all instances use their "own" setup, e.g., the server has to use a different hash function for each client, all functionalities could share their setups. This can be achieved by transforming $\mathcal{F}_{\mathsf{aPAKE}}$ to a multi-party functionality $\hat{\mathcal{F}}_{\mathsf{aPAKE}}$ that acts as a wrapper for multiple copies of $\mathcal{F}_{\mathsf{aPAKE}}$ and shares the random oracle between them. This approach is widely used and called UC with joint state (JUC) [CR03].

However, we demonstrate that, in case of asymmetric PAKE, leveraging 2-party security to multi-party security using a natural compositional approach such as the above is tricky. The reason lies in the composability guarantee itself: even within a larger context with multiple users, each 2-party aPAKE instance is guaranteed to keep its security guarantees. This isolation, however, is not realistic for aPAKE protocols, where Alice running a key exchange session with some server from her home computer should be entangled with Alice running the same protocol from her work computer. But since Alice is supposed to remember only her password, we cannot expect her to help linking the different sessions, e.g., by remembering her session id with the server and entering it into her work computer.

To remedy the situation, we identify the property that is missing from any multi-party aPAKE security notion that is composed from 2-party instances. In a nutshell, the security notion needs to allow Alice's password file at the server side to work for all her sessions. Formally, this can be seen as a "cross-session" impersonation attack requiring only the password file. Note that already 2-party aPAKE security notions need to incorporate vulnerability to impersonation attacks against the single session to be useful at all in practice (otherwise even an honest server would not be able to run the protocol). We conclude by claiming that any multi-user aPAKE protocol needs to additionally allow for cross-session impersonation in order to be useful in practice.

RELATED WORK. Canetti et al. [CHK+05] show impossibility of $\mathcal{F}_{\mathsf{apwKE}}$ in the plain model. [GMR06] show how to transform a UC-secure PAKE into a UC-secure asymmetric PAKE[5]. However, as we will show while proving security of their resulting aPAKE, their transformation seem to require a strong assumption such as a programmable random oracle. Thus, this does not contradict our separation result.

Assumption-wise, there is a distance between non-programmable and a programmable random oracles. [FLR+10] introduces models that are in between both: random oracles with "limited programmability". In a nutshell, such a random oracle allows the adversary to influence the mapping between queries and outputs, but the outputs are always randomly chosen. As our proof of security of the Ω-method demonstrates, for aPAKE influencing the mapping is sufficient. This means that our impossibility result for NPRO cannot be broadened to hold also for random oracles with limited programmability. And indeed, [JR16] propose an aPAKE that is secure w.r.t a limited programmability random oracle.

[CK02] analyze multi-session security of UC-secure key exchange protocols. Similar to us, they apply the JUC composition theorem and methods to leverage single-session security to multi-party security. However, in their work, a session refers to an exchange of a single key, while in UC notions of PAKE a session refers

[5]The opposite direction is trivial.

to a pair of users. And indeed, all PAKE functionalities already enable exchange of multiple keys via the use of subsession identifiers. On the other hand, PAKE functionalities handle only two users, while the KE functionality treated in [CK02] can deal with many users from the start. While the goal of [CK02] and our work is the same, namely achieving "multi-party + multi-session" UC security, the starting points are different.

ROADMAP. Section 2 gives details on how to model server compromise as partial corruption and states our revisited functionality $\mathcal{F}_{\mathsf{aPAKE}}$. Section 3 shows our separation result. In Section 4 we use our new model to make a stronger security statement of the Ω-protocol. In Section 5 we discuss composable multi-party security of asymmetric PAKE schemes. The appendix recalls ideal functionalities, gives some technical details of the UC model and contains the full proofs of security.

2 The Security Model

The notion of universally composable asymmetric PAKE was introduced in 2006 by Gentry, MacKenzie and Ramzan ([GMR06]). Their two-party functionality $\mathcal{F}_{\mathsf{apwKE}}$ augments the functionality for (symmetric) PAKE from [CHK+05] by adding an interface for server compromise attacks. The presentation is slightly more involved due to the different roles that the two participating users can take in the asymmetric version of PAKE: while the client can initiate multiple key exchange sessions by providing a fresh password each time, the server has to register a password file once which is then used in every key exchange session with the client. We recall $\mathcal{F}_{\mathsf{apwKE}}$ in Figure 8 in Appendix B.

STEALING PASSWORD DATA To model a server compromise attack, $\mathcal{F}_{\mathsf{apwKE}}$ provides three interfaces called STEALPWDFILE, OFFLINETESTPWD and IMPERSONATE, which we describe in more detail in the following.

- The STEALPWDFILE query initiates a server compromise attack. The output is a bit, depending on whether the server already registered a password file or not, and the query can only be made by the simulator if $\mathcal{Z}$ gives the instruction for it.
- The OFFLINETESTPWD query models an off-line dictionary attack. In this attack, the adversary tests whether a client password is contained in the password file. The output is a bit, depending on whether the password was correct or not. The attack is called "off-line" since it can be mounted by the adversary without engaging in a session with the client. The interface can only be used after a STEALPWDFILE query happended, and like STEALPWDFILE this attack can only be mounted by the simulator if $\mathcal{Z}$ instructs him to do so. Depending on whether one wants to allow pre-computation attacks or not, OFFLINETESTPWD can be either queried at any time or is restricted to be queried only after a STEALPWDFILE query was issued. In any case, if the attack is successful, the password is revealed to the adversary only after a STEALPWDFILE query happened.
- The IMPERSONATE query enables the adversary to engage in a key exchange session with the client, using the stolen password file as authenticating data. This query is necessary since a server compromise attack (i.e., STEALPWDFILE query) does not allow the adversary in any way to *control* the behavior of the server. Nonetheless, a network attacker is able to engage in a session with the client using the stolen password file, which he can do via IMPERSONATE.

DEFINITIONAL ISSUES WITH $\mathcal{F}_{\text{APWKE}}$ Let us make a few observations on these interfaces. Firstly, the restriction of letting $\mathcal{S}$ ask specific queries only upon receiving them from $\mathcal{Z}$ constitutes a change of the UC framework (the authors propose to change the control function of the UC framework to enforce it). This needs to be done carefully to not invalidate important properties of the framework such as the composition theorem and emulation w.r.t the dummy adversary. Without further restrictions, at least the latter does not hold anymore. To provide an example, consider an environment $\mathcal{Z}$ that asks an "encoded" STEALPWDFILE query, e.g., $(\texttt{ask-StealPwdFile-query}, \mathsf{sid})$ and sends it to $\mathcal{A}$, respectively to $\mathcal{S}$. A real-world adversary $\mathcal{A}$ can easily decode this query and perform the desired attack, while the simulator in the ideal world has to drop the message due to the wrong format. This way, $\mathcal{Z}$ can keep the simulator from using his interfaces, leaving him with no leverage to presume his simulation. Clearly, such a real-world adversary is worse than

a dummy adversary, who would have to drop the message just like $\mathcal{S}$. Thus, to get a meaningful definition that inherits all properties of the UC framework, such environments would have to be excluded.

Our second observation concerns the real execution of the protocol, where the adversary $\mathcal{A}$ obtains StealPwdFile and OfflineTestPwd queries from $\mathcal{Z}$. [GMR06] seem to implicitly assume that $\mathcal{A}$ now *mounts a server-compromise attack* and does not behave as the dummy adversary, as usually assumed in the UC framework (see [Can00], Section 4.4.1). Note that $\mathcal{A}$ only has influence on the communication channel and corrupted parties, and it seems that none of this helps him to, e.g., steal a password file from the server[6]. For proving UC-security of a protocol with respect to $\mathcal{F}_{\mathsf{apwKE}}$, it is however crucial to formally specify the outputs of $\mathcal{A}$ upon these queries, since $\mathcal{A}$'s output is what has to be simulated by $\mathcal{S}$[7].

2.1 Corruption Model

To address the aforementioned issues, we first reformulate StealPwdFile queries, which can be asked by $\mathcal{S}$ only upon getting instructions from $\mathcal{Z}$, as corruption queries. This possibility was already pointed out in [GMR06]. Modeling server compromise via corruption offers the following advantages:

- It captures the intuition that compromising the server, like Byzantine party corruption, constitutes an attack that the environment $\mathcal{Z}$ can mount to distinguish real and ideal execution.
- It takes care of definitional issues by using the special properties of corruption queries in UC, e.g., that they can only be asked by $\mathcal{S}$ and $\mathcal{A}$ if $\mathcal{Z}$ instructs them to do so. As a consequence, there is no need to adjust the control function or to put restrictions on the environment, nor to consider adversaries other than the dummy adversary.
- It leads to a flexible definition, since arbitrary corruption models can be integrated into the UC framework (see [Can00], section 6.1). E.g., one can choose whether upon compromising the server the adversary merely learns that a password file exists or even leak the whole file to him.

Formally, besides Byzantine party corruption, we allow $\mathcal{Z}$ to issue an additional corruption query called StealPwdFile. To formalize what happens upon corruption in the ideal world, we adopt the conventions for corruption queries from [Can00], Section 6.2, and let dummy parties in the ideal world completely ignore corruption messages. Instead, these queries are handeled by the ideal functionality, who receives them directly from $\mathcal{S}$. We now detail the structure and effect of the two types of corruption queries that we allow $\mathcal{Z}$ to ask.

Byzantine corruption

Real world: Upon receiving a message $(\text{CORRUPT}, \mathcal{P}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{A}$ delivers the message to $\mathcal{P}$ who immediately sends its internal state to $\mathcal{A}$ and, from that point on, is completely controlled by $\mathcal{A}$.
Ideal world: Upon receiving a message $(\text{CORRUPT}, \mathcal{P}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{S}$ delivers the message to $\mathcal{F}_{\mathsf{apwKE}}$, who marks $\mathcal{P}$ as `corrupted`. If $\mathcal{F}_{\mathsf{apwKE}}$ already received input from $\mathcal{P}$ or sent output to it, it sends all these values to $\mathcal{S}$. $\mathcal{F}_{\mathsf{apwKE}}$ further notifies $\mathcal{S}$ of all future inputs and outputs of $\mathcal{P}$ and lets $\mathcal{S}$ modify $\mathcal{P}$'s input values. We call a party that is corrupted in this way *corrupted*.

Server compromise

Real world: Let $f : \{0,1\}^* \to \{0,1\}^*$ be an efficiently computable function. We denote the internal state of $\mathcal{P}_S$ with $\mathsf{state} = (\mathsf{file}, \mathsf{pw}, \mathsf{state}')$, consisting of a password file file, the server's input pw and additional information state'. Upon receiving a message $(\text{StealPwdFile}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{A}$ delivers the message to $\mathcal{P}_S$, who immediately sends $f(\mathsf{state})$ to $\mathcal{A}$[8]. A natural choice is $f(\mathsf{state}) = \mathsf{file}$, which we will use throughout this

[6] The intention of $\mathcal{F}_{\mathsf{apwKE}}$ to model a server compromise attack is to leave the server honest and mount the attack via the provided interfaces instead.

[7] This requirement of adding explanation of the real-world adversary $\mathcal{A}$ to obtain a security notion usually does not occur in the UC framework as long as $\mathcal{Z}$ does not expect meaningful output from queries other than (a) modifying/introducing messages on communication tapes or (b) acting on behalf of corrupted parties and (c) messages to *hybrid* ideal functionalities (such as $\mathcal{F}_{\mathsf{RO}}$ used in this paper) that $\mathcal{Z}$ accesses through $\mathcal{A}$. The StealPwdFile and OfflineTestPwd queries are not of any of these two types.

[8] This definition of corruption resembles what [Can00] describes as *physical "side channel" attacks* since it results in leaking a function of the internal state of a party.

work.
Ideal world: Upon receiving a message (StealPwdFile, sid) from $\mathcal{Z}$, $\mathcal{S}$ sends this message to $\mathcal{F}_{\mathsf{apwKE}}$. $\mathcal{F}_{\mathsf{apwKE}}$ marks $\mathcal{P}_S$ as `compromised`. If there are records (file, $\mathcal{P}_C, \mathcal{P}_S$, pw) and (offline, pw), $\mathcal{F}_{\mathsf{apwKE}}$ sends pw to $\mathcal{S}$.

Let us emphasize that, while we now model StealPwdFile queries formally as corruption queries, this does not mean that the adversary gets to control the behavior of the server afterwards. The reader should keep in mind that there are different variants of corruption, some more severe and some less. As common in the UC model, we refer to a party having received a Byzantine corruption message as *corrupted* (acknowledging that Byzantine party corruption is the "default" type of corruption used in the literature). Also, we refer to a server having received a StealPwdFile corruption query as *compromised*. In line with [GMR06],[JKX18], we only consider static Byzantine corruption (meaning that Byzantine corruption messages are ignored after the first party obtained input from $\mathcal{Z}$). Contrarily, StealPwdFile corruption messages can be asked by $\mathcal{Z}$ at any time. This makes server compromise an *adaptive* attack. Obviously, static server compromise is not very interesting since it will result in leakage of an empty file.

2.2 Offline Attacks against the Password File

As soon as an adversary gets his hands on the password file of a server, he can try to figure out the password which was used to generate this file. Such an attack on the password file is called "off-line" to emphasize that there is no further interaction with the client required.

Such an attack is reflected in $\mathcal{F}_{\mathsf{apwKE}}$ via the OfflineTestPwd interface. This interface lets the adversary guess the password contained in the file, and can only be used after a StealPwdFile corruption query was asked. However, the security notion should offer means to tell a protocol with file = pw from a protocol with file $= H(pw)$ for some hash function H. The latter requires the adversary to compute hashes of passwords until it guesses the correct password, while the former directly leaks the password to the adversary without any computational effort. If we want to distinguish between these protocols, we need to make the number of OfflineTestPwd queries explicit to $\mathcal{Z}$.

To this end, [GMR06] define $\mathcal{F}_{\mathsf{aPAKE}}$ such that $\mathcal{A}$ is allowed to query OfflineTestPwd only upon getting instruction from $\mathcal{Z}$. Clearly, this gives $\mathcal{Z}$ a way to bound the number of OfflineTestPwd guesses and does not allow to prove a protocol with file = pw secure.

However, $\mathcal{F}_{\mathsf{apwKE}}$ with OfflineTestPwd instructed by $\mathcal{Z}$ is inherently impossible to realize for natural asymmetric PAKE protocols even under strong assumptions such as a programmable random oracle (see B for a definition of this assumption). To show this, we first need to formally capture what it means for a protocol formulated in the UC framework to have a verifiable password file. In a nutshell, the definition captures whether, after a server compromise attack happens, the adversary is provided with all necessary information and interfaces at hybrid functionalities to reconstruct the password file from only pw, i.e., to actually mount an off-line dictionary attack on file[9].

Definition 2.1 (Adversarial Verifiability)**.** *Let π be an asymmetric PAKE protocol in an $\mathcal{F}$-hybrid model, where $\mathcal{F}$ is an arbitrary set of ideal functionalities. We say that π is* adversarially verifiable *if the real-world adversary $\mathcal{A}$ can execute the server's code to compute* file *from* pw *by only interacting with the ideal functionalities $\mathcal{F}$. In this case, we also say that $\mathcal{A}$ can* efficiently verify correctness of (pw, file)*.*

We emphasize that it is the real-world adversary $\mathcal{A}$ who can verify correctness of the password file, and after giving the inputs no further help of the environment is required. This will become crucial in proving our impossibility result. Most asymmetric PAKE protocols formulated in the UC framework are adversarially verifiable [GMR06, HL18, JKX18].

Theorem 2.2. *Let $\mathcal{F}$ be a set of ideal functionalities. Then $\mathcal{F}_{apwKE}$ as depicted in Fig. 8 is not realizable in the $\mathcal{F}$-hybrid model by any protocol that is adversarially verifiable.*

[9] While it seems contradictory for an asymmetric PAKE protocol to not allow off-line dictionary attacks upon server compromise, one could indeed build such strong protocols by shielding information from the adversary using, e.g., a third party. If this party is involved in the registration phase but is not accessible by the adversary, off-line attacks can be prevented. Most notably, sharing the role of the server among several parties achieves this, and is known as *threshold PAKE* [FJ00]. While $\mathcal{F}_{\mathsf{apwKE}}$ allows to consider such protocols, we are not interested in them in this paper.

Proof. Let π be an adversarially verifiable protocol that UC-realizes $\mathcal{F}_{\mathsf{apwKE}}$. Consider the following environment $\mathcal{Z}$ running either with an adversary or the simulator.

- $\mathcal{Z}$ starts the protocol with (StorePwdFile, sid, $\mathcal{P}_C$, pw) as input to $\mathcal{P}_S$. All parties remain uncorrupted.
- $\mathcal{Z}$ sends (StealPwdFile, sid) to the adversary to mount a server compromise attack. It obtains file as answer from the adversary.
- $\mathcal{Z}$ flips a coin b. If $b = 0$, set $\mathsf{pw}' = \mathsf{pw}$, else draw $\mathsf{pw}' \neq \mathsf{pw}$ uniformly at random with the same length as pw. $\mathcal{Z}$ now sends $(\mathsf{pw}', \mathsf{file})$ to the adversary for verification.
- $\mathcal{Z}$ outputs 0 if verification succeeds, else it outputs 1.

If $b = 0$, then due to the adversarial verifiability of the protocol, $\mathcal{Z}$ will always output 0 in the real execution. If $b = 1$, verification will succeed only with negligible probability[10]. To see this, consider an environment $\mathcal{Z}_{rand}$ that starts the server with pw, then corrupts the client, honestly executes the client's code using a randomly chosen pw' with $\mathsf{pw}' \neq \mathsf{pw}$ and finally checks equality of the output keys of both parties. In the ideal world, where $\mathcal{S}$ issues a TestPwd query to correctly simulate the output of the honest server and learns "wrong guess" due to $\mathsf{pw} \neq \mathsf{pw}'$, $View_{\mathcal{S},\mathcal{F}_{\mathsf{apwKE}}}(\mathcal{Z}_{rand})$ will contain two randomly chosen output keys since $\mathcal{S}$ acts on the (correct) assumption that the passwords do not match. Thus, verification of $(\mathsf{pw}', \mathsf{file})$ succeeds (i.e., file also "works for" pw') only with negligible probability, since otherwise $View_{\mathcal{A},\pi}(\mathcal{Z}_{rand})$ would contain two equal output keys and thus $\mathcal{Z}_{rand}$ contradicts the UC-security of π. Altogether, with overwhelming probability, $\mathcal{Z}$ outputs b in the real execution.

We now analyze the output of $\mathcal{Z}$ in the ideal world. Since $\mathcal{Z}$ issues only StorePwdFile and StealPwdFile queries (which both do not produce any output in this case), no (offline, ...) or (ssid, ...) records are ever created within $\mathcal{F}_{\mathsf{apwKE}}$. Due to the absence of these records, it can be easily checked that none of the interfaces of $\mathcal{F}_{\mathsf{apwKE}}$ provided to $\mathcal{S}$ produce any output. Thus, $\mathcal{S}'s$ view is completely independent of b, and $\mathcal{Z}$ outputs b in the ideal world with probability 1/2, which contradicts the UC-security of π. □

Remark 2.3. *A simple fact in the UC model is that every UC-functionality realizes itself via a trivial protocol $\pi_\emptyset$ that just lets parties relay their input to and obtain their output from the functionality as in the ideal world. Phrased differently, every ideal functionality $\mathcal{F}'$ is trivially realizable in the $\mathcal{F}'$-hybrid model. This immediately raises the question why $\pi_\emptyset$ with $\mathcal{F} = \{\mathcal{F}_{apwKE}\}$ does not contradict Theorem 2.2. The answer is that $\pi_\emptyset$ is not adversarially verifiable: $\mathcal{F}_{apwKE}$ does not assign any value to a file entry (it answers* StealPwdFile *queries with only a bit indicating success) and thus the real-world adversary is not able to check validity of a tuple* (pw, file) *by only interacting with $\mathcal{F}_{apwKE}$.*

From Theorem 2.2, it becomes apparent that the simulator needs more leverage regarding off-line dictionary attacks. Indeed, involving $\mathcal{Z}$ in OfflineTestPwd queries keeps the simulator from using this interface and prevents successful simulation. Moreover, we conjecture that providing $\mathcal{Z}$ with an OfflineTestPwd interface is not meaningful, since this "attack" only provides $\mathcal{Z}$ with information it can already compute herself. It is thus not surprising that the best strategy for $\mathcal{Z}$ is to not use this interface, as the proof of Theorem 2.2 indicates.

Toward an improved modeling, let us first reflect the intuition of security against server compromise:

> *After compromising the server, the best strategy of the adversary to figure out the password is to run off-line dictionary attacks on the password file.*

Note that this resembles the security requirement for standard PAKE regarding on-line dictionary attacks, the only differences being that the latter can be asked at an arbitrary time but only once per protocol run. It thus seems reasonable to let the simulator issue OfflineTestPwd queries similarly to TestPwd queries, namely oblivious to $\mathcal{Z}$[11].

[10] Probabilities are taken over the random coins of all involved entities.

[11] We remark that $\mathcal{F}_{\mathsf{apwKE}}$ as stated in [GMR06] and in Figure 8 still allows for off-line pre-computation attacks. In a nutshell, this attack models the fact that an adversary can produce hashes of password candidates already before stealing the password file, and find out the password quickly by going through the hash list after seeing the file. In a recent work, [JKX18] strengthens the security notion accordingly and gives protocols that protect against these attacks.

We change the model by letting OfflineTestPwd constitute an interface provided to the adversary by $\mathcal{F}_{\mathsf{apwKE}}$ *without* requiring instructions from $\mathcal{Z}$. Consequently, OfflineTestPwd queries by $\mathcal{Z}$ do not have any effect nor produce any output in the real world. Of course, we now need means to bound usage of this interface.

Bounding the simulator's computation. Unlimited access to the OfflineTestPwd interface lets $\mathcal{S}$ find out the server's password eventually, within polynomial time if we assumem passwords that are human memorable. Knowing the password, simulation of the server becomes trivial and even protocols that we would consider insecure can be simulated (e.g., a protocol with $\mathsf{file} = \mathsf{pw}$).

One possible countermeasure is to require the simulator have runtime similar to the real-world adversary. Letting $\mathcal{S}$ only issue OfflineTestPwd when being instructed by $\mathcal{Z}$ enforces this, and was probably the reason for this limitation in [GMR06] in the first place. However, as shown in Theorem 2.2, this restriction on $\mathcal{S}$ is too heavy. Instead, we propose to lift this restriction, as formalized above, and instead require that the simulator's runtime is determined by the length of its overall input. Intuitively, each input bit can be seen as a ticket to provide an input bit to another machine. Since input to the simulator comes from the environment, this restriction lets $\mathcal{Z}$ decide about how many OfflineTestPwd queries $\mathcal{S}$ is allowed to make[12]. To formalize this, we require a simulator for a protocol realizing our revisited functionality $\mathcal{F}_{\mathsf{aPAKE}}$ given in Figure **??** to be *locally T-bounded* (cf. Section A).

3 The Separation Result

The non-programmable random oracle model. In his seminal paper, Nielsen [Nie02] formalizes the non-programmable random oracle model (NPRO) as a variant of the UC framework where all entities (including $\mathcal{Z}$) are granted direct acces to an oracle $\mathcal{O}$. This oracle answers fresh values with fresh randomness, and maintains state to consistently answer queries that were asked before. We recall the formalism from [Nie02] to integrate such a random oracle in the UC framework.

In the NPRO model, all ITMs that exist in the UC framework are equipped with additional oracle tapes, namely an oracle query tape and an oracle input tape. To denote an ITM $\mathcal{Z}$ communicating with oracle $\mathcal{O}$ via these tapes, we write $\mathcal{Z}^{\mathcal{O}}$. $\mathcal{Z}$ can write on his oracle query tape, while the oracle input tape is read-only. As soon as $\mathcal{Z}$ enters a special oracle query state, the content of the oracle query tape is sent to $\mathcal{O}$. The output of $\mathcal{O}$ is then written on the oracle input tape of $\mathcal{Z}$. A random oracle can be implemented by letting $\mathcal{O}$ denote an ITM defining a uniformly random function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\lambda$. We now say that a protocol π *UC-realizes a functionality $\mathcal{F}$ in the NPRO model* if

$$\forall \mathcal{A}^{\mathcal{O}} \; \exists \mathcal{S}^{\mathcal{O}} \text{ s.t. } \forall \mathcal{Z}^{\mathcal{O}} : \quad View_{\pi^{\mathcal{O}},\mathcal{A}^{\mathcal{O}}}(\mathcal{Z}^{\mathcal{O}}) \stackrel{c}{\approx} View_{\mathcal{F}^{\mathcal{O}},\mathcal{S}^{\mathcal{O}}}(\mathcal{Z}^{\mathcal{O}})$$

Theorem 3.1. *The functionality $\mathcal{F}_{\mathsf{aPAKE}}$ as depicted in Fig. 2 is not realizable in the NPRO model by any protocol that is adversarially verifiable.*

More detailed, for every adversarially verifiable protocol π and every polynomial T, there exists an attacker $\mathcal{A}$ and an environment $\mathcal{Z}$ which only issues static corruption requests, such that there is no locally T-bounded simulator $\mathcal{S}$ such that π UC-realizes $\mathcal{F}_{\mathsf{aPAKE}}$ in the NPRO model.

Proof. Let π be an adversarially verifiable protocol that UC-realizes $\mathcal{F}_{\mathsf{aPAKE}}$ in the NPRO model. Consider the following environment $\mathcal{Z}$ running either with an adversary or the simulator.

- $\mathcal{Z}^{\mathcal{O}}$ starts the protocol with (StorePwdFile, sid, $\mathcal{P}_C$, pw) as input to $\mathcal{P}_S$, where $\mathsf{pw} \stackrel{\$}{\leftarrow} \{0,1\}^{\lceil\sqrt{2\lambda}\rceil}$. All parties remain uncorrupted.
- $\mathcal{Z}^{\mathcal{O}}$ sends (StealPwdFile, sid) to the adversary to mount a server compromise attack. It obtains file as answer from the adversary.
- $\mathcal{Z}^{\mathcal{O}}$ verifies $(\mathsf{pw}, \mathsf{file}')$ by running the code of $\mathcal{A}^{\mathcal{O}}$ to verify a password file.

[12] This argument requires that $\mathcal{S}$ cannot use the ideal functionality to augment its input bits. However, all PAKE functionalities give only answers that are shorter than the corresponding queries. Thus, $\mathcal{S}$ can obtain additional "input tickets" only from $\mathcal{Z}$ and each query to the ideal functionality results in losing tickets.

The functionality $\mathcal{F}_{\mathsf{aPAKE}}$ is parameterized with a security parameter λ. It interacts with an adversary $\mathcal{S}$ and a client and a server $\mathcal{P} \in \{\mathcal{P}_C, \mathcal{P}_S\}$ via the following queries:

Password Registration

- On $(\text{StorePwdFile}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw})$ from $\mathcal{P}_S$, if this is the first StorePwdFile message, record $(\text{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$.

Stealing Password Data

- On $\boxed{(\text{StealPwdFile}, \mathsf{sid})}$ from $\mathcal{S}$, if there is no record $(\text{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$, return "no password file" to $\mathcal{S}$. Otherwise, mark $\mathcal{P}_S$ as `compromised`; regardless,
  - ▷ If there is a record $(\text{offline}, \mathsf{pw})$, send pw to $\mathcal{S}$.
  - ▷ Else, return "password file stolen" to $\mathcal{S}$.
- On $(\text{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}')$ from $\mathcal{S}$, do:
  - ▷ If there is a record $(\text{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ and $\mathcal{P}_S$ is compromised, do: if $\mathsf{pw} = \mathsf{pw}'$, return "correct guess" to $\mathcal{S}$; else, return "wrong guess".
  - ▷ Else, record $(\text{offline}, \mathsf{pw}')$

Password Authentication

- On $(\text{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}')$ from $\mathcal{P}_C$, send $(\text{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S)$ to $\mathcal{S}$. Also, if this is the first UsrSession message for ssid, record $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$ and mark it `fresh`.
- On $(\text{SrvSession}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{P}_S$, ignore the query if there is no record $(\text{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$. Else send $(\text{SrvSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S)$ to $\mathcal{S}$ and, if this is the first SrvSession message for ssid, record $(\mathsf{ssid}, \mathcal{P}_S, \mathcal{P}_C, \mathsf{pw})$ and mark it `fresh`.

Active Session Attacks

- On $(\text{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{pw})$ from $\mathcal{S}$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw}')$ marked `fresh`, do: if $\mathsf{pw}' = \mathsf{pw}$, mark it `compromised` and return "correct guess" to $\mathcal{S}$; else, mark it `interrupted` and return "wrong guess" to $\mathcal{S}$.
- On $(\text{Impersonate}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{S}$, if there is a record $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$ marked `fresh`, do: if there is a record $(\text{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$, $\mathcal{P}_S$ is marked `compromised` and $\mathsf{pw}' = \mathsf{pw}$, mark $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$ `compromised` and return "correct guess" to $\mathcal{S}$; else, mark it `interrupted` and return "wrong guess" to $\mathcal{S}$.

Key Generation and Authentication

- On $(\text{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{K})$ from $\mathcal{S}$ where $|key| = \lambda$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ not `completed`, do:
  - ▷ If the record is `compromised`, or either $\mathcal{P}$ or $\mathcal{P}'$ is corrupted, or $\mathsf{K} = \perp$, then send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K})$ to $\mathcal{P}$.
  - ▷ If the record is `fresh`, $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}')$ was sent to $\mathcal{P}'$, and at that time there was a record $(\mathsf{ssid}, \mathcal{P}', \mathcal{P}, \mathsf{pw})$ marked `fresh`, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}')$ to $\mathcal{P}$.
  - ▷ If the record is `interrupted` or if it is `fresh` and there is a record $(\mathsf{sid}, \mathcal{P}', \mathcal{P}, \mathsf{pw}')$ with $\mathsf{pw} \neq \mathsf{pw}'$, then send $(\mathsf{ssid}, \mathsf{ssid}, \perp)$ to $\mathcal{P}$.
  - ▷ Else, pick $\mathsf{K}'' \xleftarrow{\$} \{0,1\}^\lambda$ and send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}'')$ to $\mathcal{P}$.

  Finally, mark $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ as `completed`.

Figure 2: Our revisited ideal functionality $\mathcal{F}_{\mathsf{aPAKE}}$ for asymmetric PAKE, with explicit authentication (cf. Section 4.1). Framed queries can only be asked upon getting instructions from $\mathcal{Z}$. Gray boxes indicate queries that required instructions from $\mathcal{Z}$ in [GMR06], but not in $\mathcal{F}_{\mathsf{aPAKE}}$. To be consistent with the writing conventions for ideal functionalities, $\mathcal{F}_{\mathsf{aPAKE}}$ marks $\mathcal{P}_S$ as `compromised` instead of the file record.

- $\mathcal{Z}^{\mathcal{O}}$ outputs 0 if verification succeeds, else it outputs 1.

Note that step 3 can be performed by $\mathcal{Z}^{\mathcal{O}}$ since it has the same oracle access as $\mathcal{A}^{\mathcal{O}}$. Due to the adversarial verifiability of π, $\mathcal{Z}^{\mathcal{O}}$ always outputs 0 in the real execution. It remains to compute the probability that $\mathcal{S}^{\mathcal{O}}$ outputs $\mathsf{file}_{\mathcal{S}}$ such that $\mathcal{Z}^{\mathcal{O}}$ outputs 0 in the ideal execution. As in the proof of Theorem 2.2, since $\mathcal{Z}^{\mathcal{O}}$ issues only StorePwdFile and StealPwdFile queries (which both do not produce any output in this case), no (offline, ...) or (ssid, ...) records are ever created within $\mathcal{F}_{\mathsf{aPAKE}}$. Due to the absence of these records, the only interface of $\mathcal{F}_{\mathsf{aPAKE}}$ provided to $\mathcal{S}^{\mathcal{O}}$ that produces any pw-depending output is OfflineTestPwd. This interface provides $\mathcal{S}^{\mathcal{O}}$ with a bit, depending on whether the submitted password was equal to pw or not.

Since $\mathcal{S}^{\mathcal{O}}$ is locally T-bounded (see Section A for details), it has runtime $T(n)$ with $n = n_I - n_e$, where T is some function, n_I is the number of bits written to $\mathcal{S}^{\mathcal{O}}$'s input tapes and n_e the number of bits $\mathcal{S}^{\mathcal{O}}$ writes to other input tapes. Since $\mathcal{S}^{\mathcal{O}}$ cannot have a negative runtime, the maximum number of password bits that he can submit to OfflineTestPwd is n_I. In the above attack, n_I consists of the minimal input 1^{λ}, (StealPwdFile, sid) from $\mathcal{Z}^{\mathcal{O}}$ as well as a bit as answer to each StealPwdFile and OfflineTestPwd query that $\mathcal{S}^{\mathcal{O}}$ issues. We now upper bound the number of total password guesses that $\mathcal{S}^{\mathcal{O}}$ can submit. We simplify the analysis by ignoring names and session IDs in queries, and by assuming that (StealPwdFile, sid) has the same bitsize as $\mathcal{S}^{\mathcal{O}}$'s answer $\mathsf{file}_{\mathcal{S}}$. Additionally, we let $\mathcal{S}^{\mathcal{O}}$ know k, i.e., the length of the password of the server. The simplifications yield $n_I = \lambda + m$, where m is the number of OfflineTestPwd queries of $\mathcal{S}^{\mathcal{O}}$. Since $|pw| \geq 1$, we have $m < \lambda$, and thus the maximum number of k-bit long passwords that $\mathcal{S}^{\mathcal{O}}$ can write in queries is $2\lambda/k$. Setting $k = log_2(\lambda) + 2$, and using the fact that $\mathcal{Z}^{\mathcal{O}}$ draws pw at random, the probability that $\mathcal{S}^{\mathcal{O}}$ obtains 1 from OfflineTestPwd is at most $1/2$. It follows that

$$\begin{aligned}\Pr[\mathcal{Z} \to 1|\text{ ideal }] &= \Pr[\mathcal{A}^{\mathcal{O}}(\mathsf{pw}) = \mathsf{file}_{\mathcal{S}}] \\ &\leq 1/2 + 1/2^{k-1},\end{aligned}$$

contradicting the UC-security of π. □

Remark 3.2. *In the above proof, it would suffice to choose k bigger than the overall input. However, it is important to choose k roughly the size as real passwords ($\approx \log_2(\lambda)$), to not make the attack artificial.*

Remark 3.3. *Since in the above attack the oracle $\mathcal{O}$ is not queried before $\mathcal{S}^{\mathcal{O}}$ provides his output, any flavor of observability can be added without invalidating Theorem 3.1. That is, even observing random oracle queries from parties and the environment does not help the simulator to prevent the described attack. Contrarily, our result does not apply with respect to an oracle offering limited programability such as random or weak programmability [FLR+10]. In a nutshell, these oracles give the adversary the freedom to assign images that are chosen by the oracle to inputs of his choice. This seems enough to circumvent our impossibility result since the simulator is now able to solve its commitment issue, while it does not rely on choosing the images itself (e.g., taking a DDH challenge as image). This claim is supported by [JR16] who construct a UC-secure asymmetric PAKE in a limited programmability random oracle model. We leave it as an open question to broaden or invalidate our result for more notions of random oracles, espcially different flavors of* global *random oracles [CDG+18].*

Possibility of PAKE in the NPRO-model. As opposed to asymmetric PAKE, for standard PAKE it is possible to achieve static UC-security without relying on programmability.

A widely used PAKE protocol is DH-EKE [BM92b], which is inspired by the Diffie-Hellman Key Exchange [DH76]. The two flows of the protocol are encrypted using the password as the encryption key with an appropriate symmetric encryption scheme. The EKE protocol has been further formalized by Bellare et al. [BPR00] under the name EKE2. Its UC security is formally analyzed in [ACCP08] and [DHP+18]. By looking at their simulators for static security, it is apparent that the internally simulated oracle is not programmed in any way. Indeed, it is only observability of random oracle queries of $\mathcal{Z}$ that is required for the security proof. Thus, EKE2 is statically secure in the NPRO-model.

A PAKE protocol that is often referred to as the "KOY protocol" was originally introduced by [KOY01]. To enable UC security, it was later slightly modified [CHK+05] and proven to be UC-secure in a CRS-hybrid

model. The protocol makes use of a Hash Proof System which can be obtained from, e.g., standard discrete-log based assumptions. Security of the protocol thus does not rely on any idealized assumption. The same holds for the round-optimal PAKE from [KV11] and the protocol from [BBC+13].

4 UC-Security of the Ω-Method

In their paper, [GMR06] propose a generic method for obtaining an asymmetric PAKE protocol from a symmetric PAKE protocol. Their protocol called Ω-method is a modification of the so-called "Z-method" [Mac02] for turning a standard PAKE into an asymmetric PAKE. In this section, we analyze the UC-security of the Ω-method. Let us first recall the protocol in Figure 3.

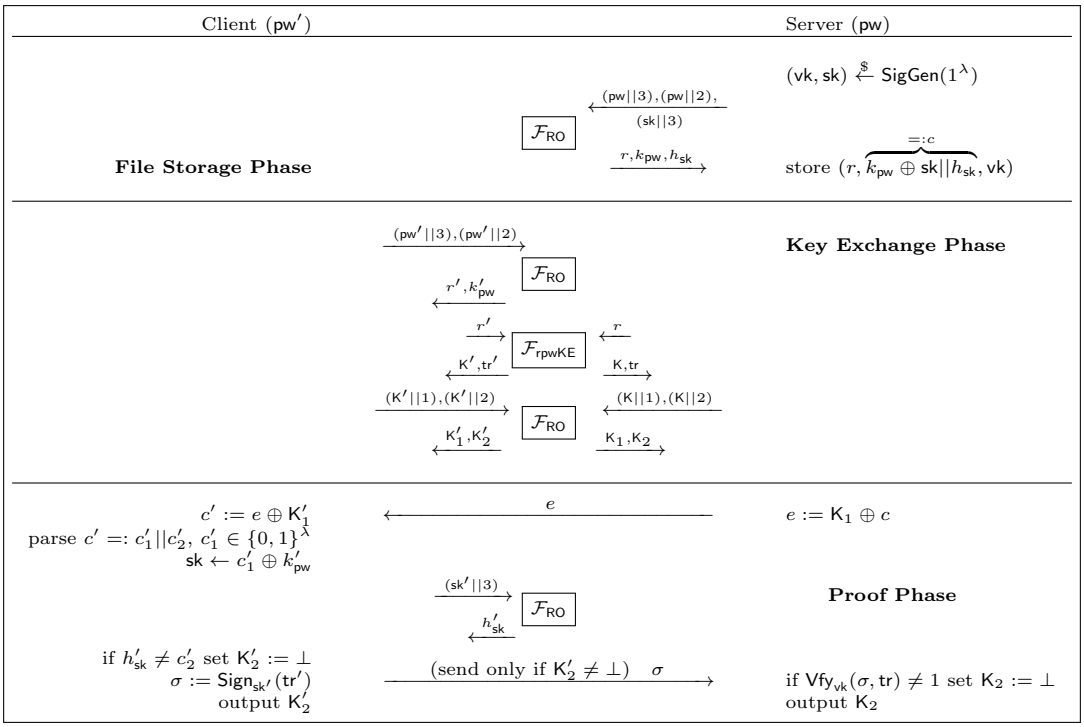

Figure 3: The Ω protocol from [GMR06]. Slightly abusing notation, we assume $\mathcal{F}_{\mathsf{rpwKE}}$ outputs transcripts together with the keys. To avoid using different instantiations of $\mathcal{F}_{\mathsf{RO}}$, we implicitly assume that $\mathcal{F}_{\mathsf{RO}}$ outputs random values of length 2λ for inputs ending with 1, and random values of length λ for inputs ending with 2 or 3. $\oplus$ binds stronger than $||$.

In a nutshell, the Ω-method consists of three phases: file storage, the key exchange phase and the proof phase. We now describe the three phases in more detail.

- **File Storage:** The server stores a hash of a password together with a signing key pair. This file is then used for all further sessions with a specific client.
- **Key Exchange Phase:** Client and server run a symmetric PAKE protocol using the hashes of their passwords as input.
- **Proof Phase:** In this phase, the client has to prove that he actually knows the password[13]. Using the (hash of the) password as an encryption key, the stored signing key is encrypted and sent to the

[13]This phase is necessary since otherwise a server compromise would enable the attacker to impersonate a client using only the hash of the password. This is clearly undesireable and reflected in the functionality that, upon server compromise, only allows to impersonate the server.

client, who decrypts it and signs the transcript together with the session identifier. Besides proving knowledge of the password, this step also informs both users whether their key exchange was successful or not. In case of success, a user outputs the session key that was computed in the key exchange phase.

We formally prove what was claimed in [GMR06], namely, UC security of the Ω protocol, executed in the UC model (cf. Appendix **??**), with respect to our revisited functionality, but using the NewKey and TestAbort interfaces used by [GMR06].

Theorem 4.1. *Protocol Ω depicted in Figure 4 securely realizes $\mathcal{F}_{aPAKE}$ with* NewKey *and* TestAbort *interface as in Figure 8 in the $\{\mathcal{F}_{RO}, \mathcal{F}_{rpwKE}\}$-hybrid model with respect to static (byzantine) corruptions and adaptive server compromise.*

Outline of the proof The proof of the theorem is mainly divided in four steps: simulate on-line dictionary attacks, simulate man-in-the-middle attacks, simulate server compromise attacks and simulate transcript and, if necessary, outputs of honest parties without their passwords and random coins.

- Towards simulating on-line dictionary attacks, the simulator is allowed to use the TestPwd interface of $\mathcal{F}_{\mathsf{aPAKE}}$ once per ssid. However, note that a dictionary attack is mounted by $\mathcal{Z}$ through a corrupted party, meaning that $\mathcal{Z}$ internally chooses a password and runs the party's code internally and then sends messages on behalf of the corrupted party. The simulator has to extract somehow the password that $\mathcal{Z}$ is using in the attack. However, since $\mathcal{Z}$ needs to hash the password, $\mathcal{S}$ can derive it from inputs to the random oracle and the inputs that $\mathcal{Z}$ sends to the hybrid $\mathcal{F}_{\mathsf{rpwKE}}$. If $\mathcal{Z}$ does not issue any of these queries, it is straightforward that the dictionary attack fails with overwhelming probability.
- Since the protocol uses a standard PAKE protocol as a building block, man-in-the-middle attacks against this part of the protocol are already excluded via the UC security of the PAKE scheme. This means that there are only two messages that the environment can attack: the first is a one-time-pad and the second a signature. $\mathcal{Z}$ can suppress or modify these messages, and we show that these denial-of-service type attacks can be handled by the simulator by calling the appropriate interfaces of $\mathcal{F}_{\mathsf{aPAKE}}$.
- Server compromise attacks mounted by the environment require the simulator to provide information that is indistinguishable of what $\mathcal{Z}$ sees in the real world. In the previous section, we detailed via a new corruption model that, in the real world, $\mathcal{Z}$ obtains a part of the servers internal state, which we call the password file. To argue security, we have to show how this file can be simulated. However, since the file contains a password hash, $\mathcal{S}$ can just output a randomly chosen value. Now $\mathcal{Z}$ can "check" this value by hashing the password and comparing it with the file. Our $\mathcal{S}$ will crucially rely on reprogramming the random oracle to match password and file. To learn what he has to program, $\mathcal{S}$ will input all random oracle queries of $\mathcal{Z}$ as OfflineTestPwd guesses to $\mathcal{F}_{\mathsf{aPAKE}}$. Crucially, he relies on having unlimited access to this interface as soon as he has to simulate a password file. Lastly, if $\mathcal{Z}$ uses the password file to mount a network attack on the session (by issueing a TestPwd query to $\mathcal{F}_{\mathsf{rpwKE}}$ using the file), $\mathcal{S}$ asks an Impersonate query. If $\mathcal{Z}$ uses a wrong file, $\mathcal{S}$ can make sure that the key exchange fails by sending $\perp$ via the NewKey interface.
- When simulating honest parties, $\mathcal{S}$ has to use simulated random coins and passwords. For showing indistinguishability of runs with simulated and real honest parties, we heavily rely on the usage of ideal building blocks that output truly random values, and in the case of $\mathcal{F}_{\mathsf{rpwKE}}$ especially outputting truly random values that are different with overwhelming probability in case of mismatching passwords.

The full proof can be found in Appendix C.

4.1 Explicit authentication

A protocol is said to have *explicit authentication* if the parties learn whether the key agreement was successful (in which case they might opt for, e.g., outputting a failure symbol). When designing their asymmetric PAKE functionality, [GMR06] introduced the TestAbort interface to the functionality to allow to analyze the security of protocols either with or without explicit authentication.

TestAbort is too weak. While it is in fact desireable to analyze both types of schemes since both are used in practice, the TestAbort interface weakens $\mathcal{F}_{\mathsf{aPAKE}}$ by not clearly distinguishing which type of protocol is analyzed. This results in a functionality with very weak security guarantees regarding authentication. An adversary can abuse this interface by letting a party in a protocol with explicit authentication output whatever key it computed by *not* making use of the TestAbort interface. This is particularly troubling since, intuitively, a protocol featuring explicit authentication should reliably inform participants whether the key exchange was successful or not. On top of that, no security guarantee about session keys is granted whatsoever in a session under attack. This means that a party under attack would faithfully use even an adversarially determined key to, e.g., encrypt her secrets.

We disclose this weakness of $\mathcal{F}_{\mathsf{aPAKE}}$ with TestAbort by demonstrating with Theorem 4.1 that the Ω-method is securely realizing it *even if the signature scheme is insecure*. By looking at the proof of the theorem, it becomes apparent that we do not rely on the signature scheme used for explicit authentication to fullfill any security notion. Let us explain on a high level why the Ω-method can be shown to securely realize $\mathcal{F}_{\mathsf{aPAKE}}$ even when signatures are easily forgeable. The purpose of the signature is to convince the server that the client holds the correct password. If an attacker manages to inject a forgery as last message, it can convince the server to output a session key in the real execution *even if the client holds a different password*. However, this can be easily simulated in the ideal world by letting the simulator *not* issue a TestAbort query for the server (as he would do for a protocol not featuring explicit authentication). Using this simulation strategy in the proof, we do not have to rely on forgeries only occuring with negligible probability.

Let us emphasize this. Inspired by the above simulation strategy, we make the following claim: a modified Ω-method where the client sends a text message "accept"/"do not accept" to the server also realizes $\mathcal{F}_{\mathsf{aPAKE}}$ in the $(\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{rpwKE}})$-hybrid model. Of course, we do not recommend to use this modified version of the protocol, nor to use it with a flawed signature scheme. This is just a thought experiment to demonstrate the severe weaknesses of the security notion that are introduced by the TestAbort interface.

Thus, $\mathcal{F}_{\mathsf{aPAKE}}$ with TestAbort is not a good abstraction of the security properties of the Ω-method. We now correct this and give a stronger functionality that captures them better.

Strong explicit authentication guarantees. For proving security of a protocol featuring explicit authentication such as the Ω-method, we show how to modify $\mathcal{F}_{\mathsf{aPAKE}}$ to provide strong authentication guarantees. Our modifications closely follow the recommendations for explicit authentication from [CHK+05]. Namely, we let the functionality send a special failure $\perp$ as output to parties with mismatching passwords. However, the functionality hides this information from the adversary, which models the fact that a passive adversary should not be able to tell whether a PAKE between two parties resulted in the same session key or not. Similarly, sessions with "wrong guess" can only get $\perp$ as output from the functionality.

Considering network attacks, an adversary can always mount a DoS attack on the protocol by, e.g., injecting messages of wrong format to make the receiving party believe authentication was unsuccessful. We incorporate this attack into the functionality by letting the adversary propose failure in his NewKey response. Formally, $\mathcal{F}_{\mathsf{aPAKE}}$ enforcing explicit authentication is depicted in Figure 2[14].

We are now ready to state a stronger version of Theorem 4.1, which captures security of the Ω-method more precisely.

Theorem 4.2. *If the signature scheme is EUF-CMA secure, protocol Ω depicted in Figure 4 securely realizes $\mathcal{F}_{aPAKE}$ in the $\{\mathcal{F}_{RO}, \mathcal{F}_{rpwKE}\}$-hybrid model with respect to static (byzantine) corruptions and adaptive server compromise.*

The proof can be found in Appendix D.

5 Multi-Session Security

In practice, asymmetric PAKE schemes are implemented in scenarios comprising many clients accessing various servers. In this section, we analyze how two-party UC security can be leveraged to obtain UC-secure multi-party asymmetric PAKE schemes.

[14] A protocol without explicit authentication, on the other hand, can be proven to securely realize $\mathcal{F}_{\mathsf{aPAKE}}$ with the NewKey interface of the symmetric PAKE functionality $\mathcal{F}_{\mathsf{rpwKE}}$ from Figure 7.

Exploiting the modularity of the UC framework, we can design a multi-party aPAKE protocol by running $\mathcal{F}_{\mathsf{aPAKE}}$ between each client-server pair. The composition theorem of the framework then allows us to subsequently replace all the ideal protocols with their realizations. However, with each such replacement, a new set of hybrid functionalities is created that is used only by one instance of the protocol. For example, in case of the Ω-protocol, this would result in as many $\mathcal{F}_{\mathsf{RO}}$ functionalities as there are disjoint client-server pairs. Clearly, instantiating all these random oracles with different hash functions does not yield a practical scheme.

UC WITH JOINT STATE. Toward a more realistic multi-party protocol, we want all two-party aPAKE protocols to jointly use their hybrid functionalities. For this, the UC with joint state (JUC) framework [CR03] can be used. In a nutshell, this framework provides a tool, the so-called *multi-session extension* $\hat{\mathcal{F}}$ of a functionality $\mathcal{F}$ to replace many hybrid functionality invokations with a single protocol. $\hat{\mathcal{F}}$ is a single functionality comprising arbitrarily many instances of $\mathcal{F}$, distributing calls to and from them consistently. Replacement is handled via the JUC composition theorem:

Theorem 5.1 (Universal composition with joint state [CR03])**.** *Let $\mathcal{F}, \mathcal{G}$ be ideal functionalities. Let π be a protocol in the $\mathcal{F}$-hybrid model, and let $\hat{\rho}$ be a protocol that UC-emulates $\hat{\mathcal{F}}$, the multi-session extension of $\mathcal{F}$, in the $\mathcal{G}$-hybrid model. Then the composed protocol $\pi^{\mathcal{F}\to\hat{\rho}}$ in the $\mathcal{G}$-hybrid model emulates protocol π in the $\mathcal{F}$-hybrid model. Here, $\pi^{\mathcal{F}\to\hat{\rho}}$ denotes the protocol π where each invokation of $\mathcal{F}$ is replaced by a call to $\hat{\rho}$.*

Let us showcase the workflow for the Ω-protocol.

(1) Find a protocol that UC-emulates the multi-session extension of $\mathcal{F}_{\mathsf{RO}}$

(2) Modify the Ω method from Figure 4 to use this protocol instead of $\mathcal{F}_{\mathsf{RO}}$

(3) Apply the JUC composition theorem to replace multiple instances of $\mathcal{F}_{\mathsf{aPAKE}}$ (in an arbitrary application protocol) by the modified Ω-protocol

More detailed, the multi-session extension $\hat{\mathcal{F}}_{\mathsf{RO}}$ of $\mathcal{F}_{\mathsf{RO}}$ is a wrapper around many instances of $\mathcal{F}_{\mathsf{RO}}$. $\hat{\mathcal{F}}_{\mathsf{RO}}$ is called with inputs of the form $(\mathsf{sid}, \mathsf{ssid}, m)$. It distributes the queries to the corresponding inner $\mathcal{F}_{\mathsf{RO}}$ functionality with session ID ssid. Vice versa, sid is added to all outputs of inner functionalities before they leave $\hat{\mathcal{F}}_{\mathsf{RO}}$.

The main difference between $\hat{\mathcal{F}}_{\mathsf{RO}}$ and $\mathcal{F}_{\mathsf{RO}}$ is that $\hat{\mathcal{F}}_{\mathsf{RO}}$ maintains a number of random oracle lists which we can refer to as $L[\mathsf{ssid}]$. Therefore, sending a value m to $\hat{\mathcal{F}}_{\mathsf{RO}}$ can result in different outputs, depending on the ssid of the input. It can be easily argued that a variant of $\mathcal{F}_{\mathsf{RO}}$ that includes the ssid in the hash list UC-emulates $\hat{\mathcal{F}}_{\mathsf{RO}}$. We call the resulting functionality $\mathcal{F}_{\mathsf{sRO}}$ and refer to it as a *shared random oracle*. See Figure 6 for a formal definition.

Lemma 5.2. *The ideal protocol* $\text{IDEAL}_{\mathcal{F}_{sRO}}$ *UC-emulates* $\hat{\mathcal{F}}_{RO}$.

We now modify the Ω-protocol execution in UC to work with $\mathcal{F}_{\mathsf{sRO}}$. Whenever a party issues a query (sid, m) to $\mathcal{F}_{\mathsf{RO}}$, this query is rewritten as (s, sid, m) and send to $\mathcal{F}_{\mathsf{sRO}}$. Here, s is a session ID hard-coded in the protocol, i.e., the same s is used by all instances of the protocol. We call the resulting protocol Ω_s. The effect of the modification is that all instances of the Ω_s protocol in a UC execution will call the same random oracle functionality $\mathcal{F}_{\mathsf{sRO}}$ having session ID s. Let us stress that Ω_s is still a two-party protocol.

We are now ready to investigate security of a protocol π invoking several instances of the Ω_s protocol (π can be thought of being the context of a two-party aPAKE protocol, in the easiest case just a protocol invoking multiple users running aPAKEs among them). As desired, the following theorem lets us replace multiple instances of the ideal functionality $\mathcal{F}_{\mathsf{aPAKE}}$ by the Ω_s-protocol.

Theorem 5.3. *Let $\pi^{\mathcal{F}_{aPAKE}}$ be a protocol in the $\mathcal{F}_{aPAKE}$-hybrid world and $\pi^{\Omega\to\Omega_s}$ be the protocol $\pi^{\mathcal{F}_{aPAKE}}$ where each invokation of $\mathcal{F}_{aPAKE}$ is replaced by an invokation of the Ω_s protocol making calls to $\mathcal{F}_{sRO}$. Then*

$$\pi^{\Omega\to\Omega_s} \text{ UC-emulates } \pi^{\mathcal{F}_{aPAKE}}$$

Proof. The theorem follows directly from Theorem 4.2, Lemma 5.2 and the JUC composition Theorem 5.1. □

We thus obtain strongly secure multi-party asymmetric PAKE schemes using a joint setup from protocols realizing $\mathcal{F}_{\mathsf{aPAKE}}$. Namely, we are guaranteed that all single instances within the multi-party protocol behave in an ideal way. To our knowledge, this is the first statement of UC security of a multi-party aPAKE protocol. The same technique can be applied to the protocols from [JKX18][15].

Remark 5.4. *As done already in [CK02], we could apply the JUC routine also "at the next level" and consider the multi-session extension of $\mathcal{F}_{aPAKE}$. The resulting functionality $\hat{\mathcal{F}}_{aPAKE}$ would be a single setup that can be called by muliple parties in an application protocol. However, while this does not give any new insights regarding security (that is, security would still hold only under the assumption introduced by the first application of JUC, e.g., a shared random oracle), it hinders our original goal to modularly analyze multi-party protocols which use aPAKE bilaterally. For this, we believe that the two-user $\mathcal{F}_{aPAKE}$ is the simplest and best option.*

Relying on a shared random oracle naturally comes at a cost: instantiating $\mathcal{F}_{\mathsf{sRO}}$ requires us to *hash session-specific information* along with the password, e.g., use $H(\mathsf{pw}, \mathsf{ssid})$ as password file in the Ω-protocol. We will elaborate on the effects of a shared random oracle in the remainder of this section. First, we detail a specific class of attacks that are important only in a multi-user scenario.

5.1 Impersonation Attacks

Impersonation attacks are attacks where the data leaked by a compromised server is used by an adversary to impersonate the server in a session with a client. It is well known that the unauthenticated communication channel between a client and a server running an aPAKE protocol makes it prone to such attacks. For the multi-party scenario, things are a bit more complicated. Put simply, multiple parties maintain multiple communication channels, and we have to analyze which password files "work" for which channels. Namely, we will distinguish between two types of impersonation attacks, which we call *local* and *cross-session* attacks.

Local Impersonation Attack. This attack involves only two parties, namely a client $\mathcal{P}_C$ and a server $\mathcal{P}_S$. First, $\mathcal{P}_S$ gets compromised and the adversary learns the password file of $\mathcal{P}_C$. The attacker then uses his ability to control the communication between $\mathcal{P}_C$ and $\mathcal{P}_S$ to mount a man-in-the-middle attack and succesfully exchange a key with $\mathcal{P}_C$.

Cross-session impersonation attack. Besides $\mathcal{P}_C$ and $\mathcal{P}_S$, this attack involves another party $\mathcal{P}'_S$. First, $\mathcal{P}_S$ gets compromised and the adversary learns the password file of $\mathcal{P}_C$. Then the attacker corrupts $\mathcal{P}'_S$, engages in a session with $\mathcal{P}_C$, installs the stolen password file and follows the rest of the protocol honestly.

Looking at the Impersonate interface of $\mathcal{F}_{\mathsf{aPAKE}}$, it can be easily seen that protocols realizing $\mathcal{F}_{\mathsf{aPAKE}}$ (and thus also protocols invoking multiple instances of it) are not supposed to protect against local impersonation attacks. The rationale is that knowing the password file and being able to hijack the connection between server and client already puts the attacker in the position of the server. He should thus be able to authenticate to the client.

On the other hand, and perhaps a bit surprisingly, protocols that are composed from two-party $\mathcal{F}_{\mathsf{aPAKE}}$ instances *protect against cross-session impersonation attacks*. In particular, this holds for the protocols constructed via Theorem5.3. The reason is that the password file "works" only for a specific ssid, and the ssid of the session $(\mathcal{P}_C, \mathcal{P}_S)$ is different from the one of $(\mathcal{P}_C, \mathcal{P}'_S)$. While this might at first glance be a benefit due to stronger security guarantees, protection against cross-session impersonation attacks actually hinders practicality of a multi-party aPAKE protocol.

Session Identifiers. In a UC execution, session identifiers are agreed upon by the parties before a protocol is run. They represent the minimal amount of coordination between parties that is needed in order to distinguish between multiple executions (called *instances*) of protocols. Agreement on sid can happen bilaterally (e.g., each party sending a randomly chosen string to one another and then XORing them) or by the initiating party proposing a sid. The UC model allows reusing a sid among multiple instances of

[15] In their protocols, switching to the shared random oracle $\mathcal{F}_{\mathsf{sRO}}$ lets the server compute session-specific OPRF keys K_{sid} and the password file becomes $PRF_{K_{\mathsf{sid}}}(\mathsf{pw})$.

protocols. However, to avoid ambuigity which instance a message belongs to, a user is not allowed to run *the same* protocol twice under the same sid[16].

In multi-user aPAKE protocols composed from $\mathcal{F}_{\mathsf{aPAKE}}$, a user Alice is thus not allowed to use the same sid in two aPAKE sessions running on her personal computer. Similarly, when Alice uses her computer at work to run aPAKE protocols, she will use different session identifiers since she is not supposed to remember anything besides her password. This makes the protocol impractical since the server needs to store multiple files for Alice, and cannot even erase the password in case she wishes to use a new device.

6 Conclusion

We identify and fix issues of the UC functionality for asymmetric PAKE. We demonstrate usefulness of our revised functionality by proving that the classical Ω protocol securely realizes it. We then show that strong assumptions such as a programmable random oracle are required to achieve UC security of 2-party asymmetric PAKE protocols. When such strongly secure protocols are composed in larger context protocols with many users, we find that the resulting schemes have such strong security guarantees that they even protect against "attacks" that are actually considered relevant use cases of multi-session aPAKE protocols. We conclude that, currently, only in the 2-party case the composable security notion introduced in this work is a useful notion. For composable multi-party aPAKE protocols, on the other hand, a new and relaxed notion of security in the UC framework is required. We provide a formalism, namely cross-session impersonation, that can guide the design of a more realistic composable security notion for multi-party aPAKE protocols.

References

[ACCP08] Michel Abdalla, Dario Catalano, Céline Chevalier, and David Pointcheval. Efficient two-party password-based key exchange protocols in the UC framework. In Tal Malkin, editor, *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, volume 4964 of *Lecture Notes in Computer Science*, pages 335–351. Springer, 2008.

[BBC+13] Fabrice Ben Hamouda, Olivier Blazy, Céline Chevalier, David Pointcheval, and Damien Vergnaud. Efficient UC-secure authenticated key-exchange for algebraic languages. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 272–291. Springer, Heidelberg, February / March 2013.

[BCL+11] Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. *J. Cryptology*, 24(4):720–760, 2011.

[BM92a] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *1992 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 4-6, 1992*, pages 72–84. IEEE Computer Society, 1992.

[BM92b] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, May 1992.

[BP13] Fabrice Benhamouda and David Pointcheval. Verifier-based password-authenticated key exchange: New models and constructions. *IACR Cryptology ePrint Archive*, 2013:833, 2013.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000.

[16] Formally, the UC model enforces this by demanding uniqueness of extended identities, i.e., tuples $(code, \mathsf{sid}, \mathsf{pid})$, where code is the code of the protocol and pid is a party identifier.

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

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CDG+18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. LNCS, pages 280–312. Springer, Heidelberg, 2018.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (preliminary version). In *30th ACM STOC*, pages 209–218. ACM Press, May 1998.

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 404–421. Springer, Heidelberg, May 2005.

[CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 337–351. Springer, Heidelberg, April / May 2002.

[CR03] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, Heidelberg, August 2003.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[DHP+18] Pierre-Alain Dupont, Julia Hesse, David Pointcheval, Leonid Reyzin, and Sophia Yakoubov. Fuzzy password-authenticated key exchange. LNCS, pages 393–424. Springer, Heidelberg, 2018.

[FJ00] Warwick Ford and Burton S. Kaliski Jr. Server-assisted generation of a strong secret from a password. In *9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000), 4-16 June 2000, Gaithersburg, MD, USA*, pages 176–180. IEEE Computer Society, 2000.

[FLR+10] Marc Fischlin, Anja Lehmann, Thomas Ristenpart, Thomas Shrimpton, Martijn Stam, and Stefano Tessaro. Random oracles with(out) programmability. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 303–320. Springer, Heidelberg, December 2010.

[GL01] Oded Goldreich and Yehuda Lindell. Session-key generation using human passwords only. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 408–432. Springer, Heidelberg, August 2001.

[GMR06] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 142–159. Springer, Heidelberg, August 2006.

[HL18] Björn Haase and Benoît Labrique. Aucpace: Efficient verifier-based PAKE protocol tailored for the iiot. *IACR Cryptology ePrint Archive*, 2018:286, 2018.

[HM04] Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 58–76. Springer, Heidelberg, February 2004.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. LNCS, pages 456–486. Springer, Heidelberg, 2018.

[JR16] Charanjit S. Jutla and Arnab Roy. Smooth NIZK arguments with applications to asymmetric UC-PAKE. *IACR Cryptology ePrint Archive*, 2016:233, 2016.

[KOY01] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient password-authenticated key exchange using human-memorable passwords. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 475–494. Springer, Heidelberg, May 2001.

[KV11] Jonathan Katz and Vinod Vaikuntanathan. Round-optimal password-based authenticated key exchange. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 293–310. Springer, Heidelberg, March 2011.

[Mac02] Philip Mackenzie. The pak suite: Protocols for password-authenticated key exchange. 12 2002.

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 111–126. Springer, Heidelberg, August 2002.

[PW17] David Pointcheval and Guilin Wang. VTBPEKE: verifier-based two-basis password exponential key exchange. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 301–312. ACM, 2017.

A Some Details on Universal Composability

In this section we will briefly recall parts of the Universal Composability (UC) framework of [Can01] that are needed in our proofs.

First, let us introduce a convention for the textual descriptions of the UC framework in this work. We will use the term *simulator* to emphasize that we talk about the adversary in the ideal world and the term *real-world adversary* if we talk about the adversary interacting with the real protocol. If we talk about both entities, we use the term *adversary*.

The UC framework uses interactive Turing machines (ITM) to describe a protocol execution, which are basically Turing machines that can send messages to each other. Each entity in the system (parties, adversaries, ideal functionalities) is model as such an ITM and is connected to various other entities. Technically, sending messages works via writing to the tape of an ITM. There are different types of tapes, e.g., for providing initial input or receiving a protocol message from another party, but we can ignore this since it is not important for our purposes and just talk about *input tapes* in general.

Before proceeding, let us quickly recall the notion of a protocol *UC-realizing* an ideal functionality $\mathcal{F}$.

Definition A.1 (UC realization, informally)**.** *A protocol π* UC-realizes *an ideal functionality $\mathcal{F}$ if*

$$\forall \mathcal{A}\, \exists \mathcal{S} \text{ s.t. } \forall \mathcal{Z}: \quad View_{\pi,\mathcal{A}}(\mathcal{Z}) \stackrel{c}{\approx} View_{\mathcal{F},\mathcal{S}}(\mathcal{Z})$$

where $\mathcal{A}, \mathcal{S}, \mathcal{Z}$ are ITMs each having private random coins available.

Of course, for the above definition to make sense one needs to bound the resources of each ITM, in particular its runtime. Bounding the runtime of a Turing machine is usually done by saying that it has to halt within $T(n)$ steps, where T is some function $T : \mathbb{N} \to \mathbb{N}$ and n is the number of overall input bits. Care has to be taken when formulating such a restriction for an interactive Turing machine, since such a machine can create new input bits for other machines. Consider for example two ITMs with runtime bounded by the constant function $T : \mathbb{N} \to 2$. If the only thing that the machines do is writing one bit on the other ITM's input tape upon each activation, the system of these two ITMs will never halt. To avoid infinite runs, input bits are interpreted as "runtime tokens" that can either be consumed by the ITM itself for local computations, or *given away* to other ITMs. This leads to the notion of *locally T-bounded ITMs*.

Definition A.2 (Locally T-bounded ITM, Def. 2 of [Can01])**.** *Let $T : \mathbb{N} \to \mathbb{N}$. An ITM M is* locally T-bounded *if, at any point during an execution of M (namely, in any configuration of M), the overall number of computational steps taken by M so far is at most $T(n)$, where $n = n_I - n_O$, n_I is the overall number of bits written so far on M's input tape, and n_O is the number of bits written by M so far on input tapes of ITM instances.*

Finally, we recall execution of the Ω protocol in the UC model. Most of Figure 4 is taken verbatim from [GMR06] with only adjustments to our notation.

Setup: The protocol uses a random oracle functionality $\mathcal{F}_{\mathsf{RO}}$ and a PAKE functionality $\mathcal{F}_{\mathsf{rpwKE}}$, as well as a signature scheme $(\mathsf{SigGen}, \mathsf{Sign}, \mathsf{Vfy})$. It is executed by a client $\mathcal{P}_C$ and a server $\mathcal{P}_S$.

Password Storage: When $\mathcal{P}_S$ is activated with $(\textsc{StorePwdFile}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw})$ for the first time, he first sends $(\mathsf{sid}, \mathsf{pw}||3), (\mathsf{sid}, \mathsf{pw}||2)$ to $\mathcal{F}_{\mathsf{RO}}$ and receives responses (sid, r) and $(\mathsf{sid}, k_{\mathsf{pw}})$. He generates a signature key pair $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{SigGen}(1^\lambda)$ and sends $(\mathsf{sid}, \mathsf{sk}||3)$ to $\mathcal{F}_{\mathsf{RO}}$, obtaining an answer $(\mathsf{sid}, h_{\mathsf{sk}})$. He computes $c \leftarrow k_{\mathsf{pw}} \oplus \mathsf{sk}||h_{\mathsf{sk}}$, where $\oplus$ binds stronger than $||$, and sets $\mathsf{file}[\mathsf{sid}] := (r, c, \mathsf{vk})$.

Protocol Steps:

- When $\mathcal{P}_C$ receives input $(\textsc{SrvSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C)$ he obtains r from $\mathsf{file}[\mathsf{sid}]$ (aborting if this value is not properly defined), sends $(\textsc{NewSession}, \mathsf{sid}||\mathsf{ssid}, \mathcal{P}_S, \mathcal{P}_C, r, \texttt{server})$ to $\mathcal{F}_{\mathsf{rpwKE}}$ and awaits a response.
- When $\mathcal{P}_C$ receives an input $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw})$ he sends $(\mathsf{sid}, \mathsf{pw}||3)$ to $\mathcal{F}_{\mathsf{RO}}$ and obtains a response r. He then sends $(\textsc{NewSession}, \mathsf{sid}||\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \texttt{client})$ to $\mathcal{F}_{\mathsf{rpwKE}}$ and awaits a response.
- When $\mathcal{P}_S$ is awaiting a response from $\mathcal{F}_{\mathsf{rpwKE}}$ and receives $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K})$ and $(\textsc{transcript}, \mathsf{sid}||\mathsf{ssid}, \mathsf{tr})$ from it, he sends $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K}||1)$ and $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K}||2)$ to $\mathcal{F}_{\mathsf{RO}}$ and receives responses $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K}_1)$ and $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K}_2)$. He retrieves c from $\mathsf{file}[\mathsf{sid}]$ and computes $e \leftarrow \mathsf{K}_1 \oplus c$ and sends the message $(\texttt{flow-zero}, \mathsf{sid}, \mathsf{ssid}, e)$ to $\mathcal{P}_C$.
- When $\mathcal{P}_C$ is awaiting a response from $\mathcal{F}_{\mathsf{rpwKE}}$ and receives $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K})$ and $(\textsc{transcript}, \mathsf{sid}||\mathsf{ssid}, \mathsf{tr})$ from it, he sends $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K}||1)$ and $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K}||2)$ to $\mathcal{F}_{\mathsf{RO}}$ and receives responses $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K}_1)$ and $(\mathsf{sid}||\mathsf{ssid}, \mathsf{K}_2)$.
- When $\mathcal{P}_C$ receives a message $(\texttt{flow-zero}, \mathsf{sid}, \mathsf{ssid}, e)$ he computes $c \leftarrow \mathsf{K}_1 \oplus e$ and parses $c =: c_1||c_2$ with $c_1 \in \{0,1\}^\lambda$. He sends $(\mathsf{sid}, pw||2)$ to $\mathcal{F}_{\mathsf{RO}}$ and receives response k_{pw}. He computes $\mathsf{sk} := c_1 \oplus k_{\mathsf{pw}}$ and sends $(\mathsf{sid}, \mathsf{sk}||3)$ to $\mathcal{F}_{\mathsf{RO}}$, receives response h_{sk} and verifies that $h_s k = c_2$. If not, he outputs $(\textsc{abort}, \mathsf{sid}, \mathsf{ssid})$ and terminates the session. Otherwise, he computes $\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(\mathsf{sid}||\mathsf{ssid}||\mathsf{tr})$, sends $(\texttt{flow-one}, \mathsf{sid}, \mathsf{ssid}, \sigma)$ to $\mathcal{P}_S$, outputs $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}_2)$ and terminates the session.
- When $\mathcal{P}_S$ receives a message $(\texttt{flow-one}, \mathsf{sid}, \mathsf{ssid}, \sigma)$, he checks that $\mathsf{Vfy}_{\mathsf{vk}}(\sigma, \mathsf{sid}||\mathsf{ssid}||\mathsf{tr}) = 1$. If not, he outputs $(\textsc{abort}, \mathsf{sid}, \mathsf{ssid})$ and terminates the session. Otherwise, he outputs $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}_2)$ and terminates the session.

Figure 4: Execution of the Ω protocol from Figure 3 in the UC model.

The functionality $\mathcal{F}_{\mathsf{RO}}$ proceeds as follows, running on security parameter λ, with a set of parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and an adversary $\mathcal{S}$:

- $\mathcal{F}_{\mathsf{RO}}$ keeps a list L (which is initially empty) of pairs of bit strings.
- Upon receiving a value (sid, m) (with $m \in \{0,1\}^*$) from some party $\mathcal{P}_i$ or from $\mathcal{S}$, do:
  - ▷ If there is a pair $(m, \tilde{h})$ for some $\tilde{h} \in \{0,1\}^\lambda$ in the list L, set $h := \tilde{h}$.
  - ▷ If there is no such pair, choose uniformly $h \in \{0,1\}^\lambda$ and store the pair $(m, h) \in L$.

  Once h is set, reply to the activating machine (i.e., either $\mathcal{P}_i$ or $\mathcal{S}$) with (sid, h).

Figure 5: Functionality $\mathcal{F}_{\mathsf{RO}}$

The functionality $\mathcal{F}_{\mathsf{sRO}}$ proceeds as follows, running on security parameter λ, with a set of parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and an adversary $\mathcal{S}$:

- $\mathcal{F}_{\mathsf{sRO}}$ keeps a list L (which is initially empty) of pairs of bit strings.
- Upon receiving a value $(\mathsf{sid}, \mathsf{ssid}, m)$ (with $m \in \{0,1\}^*$) from some party $\mathcal{P}_i$ or from $\mathcal{S}$, do:
  - ▷ If there is a tuple $(\mathsf{ssid}, m, \tilde{h})$ for some $\tilde{h} \in \{0,1\}^\lambda$ in the list L, set $h := \tilde{h}$.
  - ▷ If there is no such pair, choose uniformly $h \in \{0,1\}^\lambda$ and store the pair $(\mathsf{ssid}, m, h) \in L$.

  Once h is set, reply to the activating machine (i.e., either $\mathcal{P}_i$ or $\mathcal{S}$) with (sid, h).

Figure 6: Functionality $\mathcal{F}_{\mathsf{sRO}}$

B Ideal functionalities

We recall the Random Oracle (RO) functionality $\mathcal{F}_{\mathsf{RO}}$ as defined by Hofheinz and Müller-Quade in [HM04] in Figure 5, the revisited ideal functionality for symmetric PAKE from [GMR06], called $\mathcal{F}_{\mathsf{rpwKE}}$ in Figure 7 and the ideal functionality $\mathcal{F}_{\mathsf{apwKE}}$ for asymmetric PAKE also from [GMR06] in Figure 8.

C Proof of Theorem 4.1

Proof. We call a message *adversarially generated* if it was not output by any of the honest parties, neither within the real execution nor the simulation. We refer to a query $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{K})$ from the adversary $\mathcal{S}$ with an honest party $\mathcal{P}_i$ as *due* if

- there is a fresh record of the form $(\mathcal{P}_i, \mathsf{pw})$
- there is a record $(\mathsf{ssid}, \mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw}')$ with $\mathsf{pw} = \mathsf{pw}'$ and $\mathcal{P}_{1-i}$ is honest
- a key K' was sent to the other party while $(\mathsf{ssid}, \mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw}')$ was fresh at the time.

We denote parties with $\mathcal{P}_S, \mathcal{P}_C$ whenever we want to specify their role in the protocol, and with $\mathcal{P}_i, \mathcal{P}_{1-i}$ whenever the role does not matter.

Game $\mathbf{G}_0$: The real protocol execution. The real protocol execution is depicted in Figure 3. For a description of how to execute the protocol in the UC model, we refer the reader to Figure 6 in [GMR06].

Game $\mathbf{G}_1$: Introducing the ideal functionality. In this game we just create new and regroup existing entities in the system which, in the UC framework, are modeled as interactive Turing machines (ITM). Specifically, we create two dummy ITMs, one for each party, who just relay all the messages and are connected to the real parties and the environment. Between dummy and real parties, we create a new ITM that we call $\mathcal{F}$. This ITM will be gradually changed among the upcoming games and in the end

The functionality $\mathcal{F}_{\mathsf{rpwKE}}$ is parameterized by a security parameter λ. It interacts with an adversary $\mathcal{S}$ and two parties $\mathcal{P}_i, \mathcal{P}_{1-i}$ via the following queries:

Upon receiving $(\textbf{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \mathsf{pw}, \texttt{role})$ **from party** $\mathcal{P}_i$**:**

Send $(\text{NEWSESSION}, \mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \texttt{role})$ to $\mathcal{S}$. Also, if this is the first NEWSESSION query, or if this is the second NEWSESSION query and there is a record $(\mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw}')$, then record $(\mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw})$ and mark this record `fresh`.

Upon receiving $(\textbf{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}')$ **from** $\mathcal{S}$**:**

If there is a record $(\mathcal{P}_i, \mathcal{P}_{1-i}, \mathsf{pw})$ which is `fresh`, then do:

- if $\mathsf{pw} = \mathsf{pw}'$, mark the record `compromised` and reply to $\mathcal{S}$ with "correct guess".
- if $\mathsf{pw} \neq \mathsf{pw}'$, mark the record `interrupted` and reply to $\mathcal{S}$ with "wrong guess".

Upon receiving $(\textbf{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{K})$ **from** $\mathcal{S}$ **where** $|\mathsf{K}| = \lambda$**:**

If there is a record of the form $(\mathcal{P}_i, \mathcal{P}_{1-i}, \mathsf{pw})$ that is not marked `completed`, then:

- If this record is `compromised`, or either $\mathcal{P}_i$ or $\mathcal{P}_{1-i}$ is corrupted, then output $(\mathsf{sid}, \mathsf{K})$ to $\mathcal{P}_i$.
- If this record is `fresh`, and there is a record $(\mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw}')$ with $\mathsf{pw}' = \mathsf{pw}$, and a key K' was sent to $\mathcal{P}_{1-i}$, and $(\mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw})$ was `fresh` at that time, then output $(\mathsf{sid}, \mathsf{K}')$ to $\mathcal{P}_i$.
- In any other case, pick a new random key K' of length λ and send $(\mathsf{sid}, \mathsf{K}')$ to $\mathcal{P}_i$.

Either way, mark the record $(\mathcal{P}_i, \mathcal{P}_{1-i}, \mathsf{pw})$ as `completed`.

Upon receiving $(\textbf{NewTranscript}, \mathsf{sid}, \mathcal{P}_i, \mathsf{tr})$ **from** $\mathcal{S}$**:**

If there is a record of the form $(\mathcal{P}_i, \mathcal{P}_{1-i}, \mathsf{pw})$ that is marked `completed`, then:

- If (1) there is a record $(\mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw}')$ for which a tuple $(\text{TRANSCRIPT}, \mathsf{sid}, \mathsf{tr}'')$ was sent to $\mathcal{P}_{1-i}$, (2) either $(\mathcal{P}_i, \mathcal{P}_{1-i}, \mathsf{pw})$ or $(\mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw}')$ was ever `compromised` or `interrupted`, and (3) $\mathsf{tr} = \mathsf{tr}''$, ignore this query.
- In any other case, send $(\text{TRANSCRIPT}, \mathsf{sid}, \mathsf{tr})$ to $\mathcal{P}_i$.

Figure 7: Functionality $\mathcal{F}_{\mathsf{rpwKE}}$ for symmetric PAKE from [GMR06]. It was obtained by adapting the original symmetric PAKE functionality from [CHK+05] to the possiblity of letting the parties obtain a transcript of the protocol.

The functionality $\mathcal{F}_{\mathsf{apwKE}}$ is parameterized with a security parameter λ. It interacts with an adversary $\mathcal{S}$, a client $\mathcal{P}_C$ and a server $\mathcal{P}_S$ via the following queries:

Password Registration

- On $(\textsc{StorePwdFile}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw})$ from $\mathcal{P}_S$, if this is the first StorePwdFile message, record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ and mark it `uncompromised`.

Stealing Password Data

- On $\boxed{(\textsc{StealPwdFile}, \mathsf{sid})}$ from $\mathcal{S}$, if there is no record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$, return "no password file" to $\mathcal{S}$. Otherwise, if the record is marked `uncompromised`, mark it `compromised`; regardless,
  - ▷ If there is a record $(\textsc{offline}, \mathsf{pw})$, send pw to $\mathcal{S}$.
  - ▷ Else, return "password file stolen" to $\mathcal{S}$.
- On $\boxed{(\textsc{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}')}$ from $\mathcal{S}$, do:
  - ▷ If there is a record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ marked `compromised`, do: if $\mathsf{pw} = \mathsf{pw}'$, return "correct guess" to $\mathcal{S}$; else, return "wrong guess".
  - ▷ Else, record $(\textsc{offline}, \mathsf{pw}')$

Password Authentication

- On $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}')$ from $\mathcal{P}_C$, send $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S)$ to $\mathcal{S}$. Also, if this is the first UsrSession message for ssid, record $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$ and mark it `fresh`.
- On $(\textsc{SrvSession}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{P}_S$, ignore the query if there is no record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$. Else send $(\textsc{SrvSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S)$ to $\mathcal{S}$ and, if this is the first SrvSession message for ssid, record $(\mathsf{ssid}, \mathcal{P}_S, \mathcal{P}_C, \mathsf{pw})$ and mark it `fresh`.

Active Session Attacks

- On $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{pw}')$ from $\mathcal{S}$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ marked `fresh`, do: if $\mathsf{pw}' = \mathsf{pw}$, mark it `compromised` and return "correct guess" to $\mathcal{S}$; else, mark it `interrupted` and return "wrong guess" to $\mathcal{S}$.
- On $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{S}$, if there is a record $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$ marked `fresh`, do: if there is a record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ marked `compromised` and $\mathsf{pw}' = \mathsf{pw}$, mark $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$ `compromised` and return "correct guess" to $\mathcal{S}$; else, mark it `interrupted` and return "wrong guess" to $\mathcal{S}$.

Key Generation and Authentication

- On $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{K})$ from $\mathcal{S}$ where $|key| = \lambda$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ not marked `completed`, do:
  - ▷ If the record is marked `compromised`, or either $\mathcal{P}$ or $\mathcal{P}'$ is corrupted, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K})$ to $\mathcal{P}$.
  - ▷ If the record is marked `fresh`, $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}')$ was sent to $\mathcal{P}'$, and at that time there was a record $(\mathsf{ssid}, \mathcal{P}', \mathcal{P}, \mathsf{pw}')$ marked `fresh`, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}')$ to $\mathcal{P}$.
  - ▷ Else, pick $\mathsf{K}'' \xleftarrow{\$} \{0,1\}^\lambda$ and send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}'')$ to $\mathcal{P}$.

  Finally, mark $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ `completed`.
- On $(\textsc{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P})$ from $\mathcal{S}$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ not marked `completed`, do:
  - ▷ If it is marked `fresh` and record $(\mathsf{ssid}, \mathcal{P}', \mathcal{P}, \mathsf{pw})$ exists, send "success" to $\mathcal{S}$.
  - ▷ Else, send "fail" to $\mathcal{S}$ and $(\textsc{abort}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{P}$, and mark $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ `completed`.

Figure 8: Ideal functionality $\mathcal{F}_{\mathsf{apwKE}}$ for asymmetric PAKE from [GMR06], but phrased as in [JKX18] with slight notational changes to avoid confusion between the adversary ($\mathcal{S}$) and server ($\mathcal{P}_S$). $\boxed{\text{Framed queries}}$ can only be asked upon getting instructions from $\mathcal{Z}$.

be equivalent to $\mathcal{F}_{\mathsf{aPAKE}}$. For now, $\mathcal{F}$ connects dummy and real parties by relaying messages between each real party and its dummy party. Lastly, we merge the real parties, the hybrid functionalities and the real world adversary into a single ITM and call it the simulator $\mathcal{S}$. The random tape of $\mathcal{S}$ is used to provide the random tapes of all the ITMs that $\mathcal{S}$ comprises. The changes are depicted in Figure 1 and since none of them impact any outputs or messages, the current and previous game are perfectly indistinguishable.

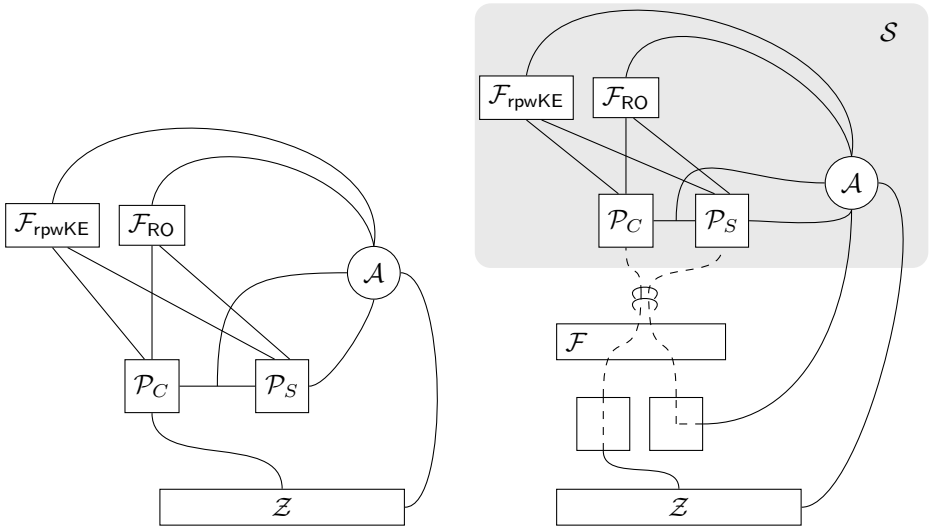

Figure 9: Transition from Game game $\mathbf{G}_0$ (left) to Game game $\mathbf{G}_1$ (right), showing a setting where $\mathcal{P}_S$ is corrupted.

Game $\mathbf{G}_2$: $\mathcal{F}$ keeps records. We now let $\mathcal{F}$ maintain records. On receiving (StorePwdFile, $\mathsf{sid}, \mathcal{P}_C, \mathsf{pw}$ from $\mathcal{P}_S$, if this is the first StorePwdFile query, then $\mathcal{F}$ records $(\mathsf{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$. Upon receiving a message (UsrSession, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}'$) from $\mathcal{P}_C$ and this is the first UsrSession message for ssid, $\mathcal{F}$ records $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$. Similarly, we record SrvSession messages as done in $\mathcal{F}_{\mathsf{aPAKE}}$.

Since the changes do not influence any inputs or outputs of $\mathcal{F}$, game $\mathbf{G}_2$ and game $\mathbf{G}_1$ are perfectly indistinguishable.

Game $\mathbf{G}_3$: adding interfaces to $\mathcal{F}$. We now add all missing interfaces except NewKey to the code of $\mathcal{F}$, namely StealPwdFile, OfflineTestPwd, TestPwd, Impersonate and TestAbort exactly as in $\mathcal{F}_{\mathsf{aPAKE}}$. Since $\mathcal{S}$ so far does not make use of any of these interfaces, the current and last game are perfectly indistinguishable.

Game $\mathbf{G}_4$: random key for interrupted sessions. We now change the simulation and let $\mathcal{S}$ add a NewKey tag and party name to the output of parties. At the same time, we change $\mathcal{F}$ and partly add the NewKey interface as follows:

> On (NewKey, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{K}$) from $\mathcal{S}$ where $|key| = \lambda$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw}')$ not marked `completed`, do:
>
> - If the record is marked `compromised`, or either $\mathcal{P}$ or $\mathcal{P}'$ is corrupted, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K})$ to $\mathcal{P}$.
> - If the record is marked `interrupted`, pick $\mathsf{K}'' \xleftarrow{\$} \{0,1\}^\lambda$ and send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}'')$ to $\mathcal{P}$.
> - Else, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K})$ to $\mathcal{P}$.
>
> Finally, mark $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw}')$ `completed`.

Note that this only changes outputs towards honest parties in case of records that are marked `interrupted`. Since this will not happen in our current simulation ($\mathcal{S}$ does not make use of any interfaces that mark records as `interrupted`), the current and previous game are perfectly indistinguishable.

Game $\mathbf{G}_5$: ask TestPwd query upon dictionary attack. We now change the simulation. If $\mathcal{P}_i$ is honest and $\mathcal{P}_{1-i}$ corrupted and $\mathcal{P}_{1-i}$ provides an input r to $\mathcal{F}_{\mathsf{rpwKE}}$, then $\mathcal{S}$ looks for an entry $(\mathsf{pw}||3, r)$ in L. If there is such an entry, $\mathcal{S}$ submits $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, \mathsf{pw})$ to $\mathcal{F}$. If there is no such entry, $\mathcal{S}$ submits $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, \perp)$ to $\mathcal{F}$[17]. Whenever $\mathcal{S}$ receives "correct guess", he sets the password of the simulated $\mathcal{P}_i$ to be pw.

Since $\mathcal{S}$ still uses the real passwords, overwriting them upon a "correct guess" will not change them. Further, the output of $\mathcal{P}_i$ remains unchanged since, due to the changes, $\mathcal{F}$ keeps relaying K_2 or K'_2, respectively, from the simulation. Note that even `interrupted` records do not obtain session keys determined by $\mathcal{F}$ as stated in game $\mathbf{G}_4$, since the corruption status is more relevant in the NewKey interface of $\mathcal{F}$.

The changes are quite trivial since $\mathcal{S}$ still knows all real passwords. We will change this in the end of the proof, using the current game as a preparation step.

Game $\mathbf{G}_6$: attacks against inner PAKE. If $\mathcal{Z}$ instructs $\mathcal{S}$ to send a $(\textsc{TestPwd}, \langle \mathsf{sid}, \mathsf{ssid}\rangle, \mathcal{P}_i, r)$ query to a specific instance $\langle \mathsf{sid}, \mathsf{ssid}\rangle$ of $\mathcal{F}_{\mathsf{rpwKE}}$, the simulation is changed as follows:

- *(Impersonation attack:)* If sid is an uncorrupted session, $\mathcal{P}_i$ is the client, $\mathcal{Z}$ already issued a StealPwdFile query and $\mathcal{S}$ replied with a value $\mathsf{file} = (r, \cdot, \cdot)$, $\mathcal{S}$ now sends $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{F}$. If the answer is "correct guess", $\mathcal{S}$ continues the simulation of the client with r.
- *(Dictionary attack on PAKE:)* Else, if $\mathcal{Z}$ received r as response of some $\mathcal{F}_{\mathsf{RO}}$ query $(\mathsf{pw}||3)$, $\mathcal{S}$ sends $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, pw)$ to $\mathcal{F}$. If the answer is "correct guess", $\mathcal{S}$ continues the internal simulation of $\mathcal{P}_i$ with pw.

In any case, if the answer is "wrong guess", $\mathcal{S}$ sends $(\textsc{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i)$ to $\mathcal{F}_{\mathsf{aPAKE}}$ and forwards the answer of his own Impersonate query as reply to $\mathcal{Z}$'s TestPwd query.

The output of $\mathcal{P}_i$ is now $\perp$ in case of "wrong guess". Since in game $\mathbf{G}_5$ the outputs of honest parties with non-matching passwords (in case of an impersonation attack) or an interrupted $\mathcal{F}_{\mathsf{rpwKE}}$ session (in case of a dictionary attack against PAKE) were drawn uniformly random upon simulating $\mathcal{F}_{\mathsf{rpwKE}}$ and $\mathcal{F}_{\mathsf{RO}}$, the output of $\mathcal{P}_i$ was already $\perp$ except in case of colliding $\mathcal{F}_{\mathsf{rpwKE}}$ outputs, which happens only with negligible probability.

Game $\mathbf{G}_7$: man-in-the-middle against client. If $\mathcal{Z}$ injects an adversarially generated $e_{\mathcal{Z}}$ as message to the client in an honest session that is different from the corresponding message computed in the internal simulation, $\mathcal{S}$ issues $(\textsc{TestPwd}, \mathsf{ssid}, \mathcal{P}_C, \perp)$ and then $(\textsc{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C)$ to $\mathcal{F}_{\mathsf{aPAKE}}$.

In this game and considered case, $\mathcal{P}_C$ will produce $\perp$ as output due to the combination of TestPwd and TestAbort queries, while in game $\mathbf{G}_6$ an adversarially generated message could have made $\mathcal{P}_C$ output something else. Namely, $\mathcal{P}_C$ outputs $\mathsf{K}'_2 \neq \perp$ in game $\mathbf{G}_6$ only if $e_{\mathcal{Z}} \oplus k'_{\mathsf{pw}}$ parses as some sk', h' such that $H(\mathsf{sk}') = h'$. Since k'_{pw} is drawn uniformly random and h' is of length λ, the probability that this happens is upper bounded by $1/2^{\lambda}$.

Game $\mathbf{G}_8$: man-in-the-middle against server. We change the simulation as follows:

- $\mathcal{S}$ issues a $(\textsc{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C)$ query at the point where the simulated $\mathcal{P}_C$ checks whether $h'_{\mathsf{sk}} \neq c'_2$. If $\mathcal{S}$ gets back "fail", it aborts the simulation.
- If $\mathcal{Z}$ injects $\sigma_{\mathcal{Z}}$ as last message where $\mathsf{Vfy}_{\mathsf{vk}}(\sigma_{\mathcal{Z}}, \mathsf{tr}) = 1$ then nothing is changed.
- If $\mathcal{Z}$ injects $\sigma_{\mathcal{Z}}$ as last message where $\mathsf{Vfy}_{\mathsf{vk}}(\sigma_{\mathcal{Z}}, \mathsf{tr}) = 0$ then $\mathcal{S}$ first issues $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \perp)$ and then $(\textsc{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S)$.
- If $\mathcal{Z}$ does not inject the last message, $\mathcal{S}$ issues $(\textsc{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S)$ when the simulated $\mathcal{P}_S$ runs Vfy.

[17] We assume this query to result in $\mathcal{F}$ marking the corresponding record as `interrupted`.

We now analyze indistinguishability from game $\mathbf{G}_7$. If $\mathcal{Z}$ does not inject any messages, in the "success" case outputs are not modified. In the "fail" case, where client and server hold different passwords, it holds that $k_{\mathsf{pw}} \neq k'_{\mathsf{pw}}$ which ensures $\mathsf{sk} \neq \mathsf{sk}'$ and $h'_{\mathsf{sk}} \neq h'$. This means that $\mathcal{P}_C$ and $\mathcal{P}_S$ computed $\mathsf{K}_2 = \mathsf{K}'_2 = \perp$ already in game $\mathbf{G}_7$[18].

In case of an adversarially generated $\sigma_{\mathcal{Z}}$ with $\mathsf{Vfy}_{\mathsf{vk}}(\sigma, \mathsf{tr}) = 0$, the changes in the current game will let $\mathcal{P}_S$ output $\perp$. However, $\mathcal{P}_S$ already outputs $\mathsf{K}_2 = \perp$ in game $\mathbf{G}_7$ in case of a non-verifying signature.

Game $\mathbf{G}_9$: align session keys. We change $\mathcal{F}$ and add the second bullet to the NewKey query: If the record is marked `fresh`, $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}')$ was sent to $\mathcal{P}'$, and at that time there was a record $(\mathsf{ssid}, \mathcal{P}', \mathcal{P}, \mathsf{pw}')$ marked `fresh`, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}')$ to $\mathcal{P}$.

Indistinguishability from the previous game follows by correctness of the protocol: for an honest session, users holding the same passwords will input the same value into $\mathcal{F}_{\mathsf{rpwKE}}$ and thus $\mathsf{K}_2 = \mathsf{K}'_2$.

Game $\mathbf{G}_{10}$: random session key for honest session. We now change $\mathcal{F}$ and add the last bullet to the NewKey query: Else, pick $\mathsf{K}'' \xleftarrow{\$} \{0,1\}^\lambda$ and send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}'')$ to $\mathcal{P}_i$. Note that this instruction will now comprise the change made in game $\mathbf{G}_4$ concerning interrupted records[19].

For analyzing indistinguishability, we first observe that opposed to game $\mathbf{G}_9$ $\mathcal{F}$ now hands out random session keys for fresh records of honest sessions where the NewKey query is not due. Split up even more, $\mathcal{F}$ now hands out random session keys for a fresh record $(\mathsf{ssid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \mathsf{pw})$ of an honest session where

- no key was sent to $\mathcal{P}_{1-i}$ yet
- a key was sent to $\mathcal{P}_{1-i}$ but at that time there was no $(\mathsf{ssid}, \mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw})$ record that was fresh

In the first case, note that the key output by $\mathcal{P}_i$ in game $\mathbf{G}_9$ was randomly chosen upon simulation of $\mathcal{F}_{\mathsf{rpwKE}}$ and thus the ouput of $\mathcal{P}_i$ is equally distributed. In the second case, (1) either there was a record $(\mathsf{ssid}, \mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw})$ but it was compromised or interrupted, which happens in our simulation in honest sessions only if $\mathcal{Z}$ injects $e_{\mathcal{Z}}$ or a non-verifying $\sigma_{\mathcal{Z}}$ (cf. game $\mathbf{G}_8$). Since the simulation is aborted upon an adversarially generated $e_{\mathcal{Z}}$ the second case will never happen for $\mathcal{P}_i$ being the client. If on the other hand $\mathcal{P}_i$ is the server, then in game $\mathbf{G}_9$ $\mathcal{P}_i$ will output K'_2 that he obtained from $\mathcal{F}_{\mathsf{rpwKE}}$ which is uniformly random just as in game $\mathbf{G}_{10}$. The only other possibility for the second case is (2) mismatching passwords (i.e., there is a fresh record $(\mathsf{ssid}, \mathcal{P}_{1-i}, \mathcal{P}_i, \mathsf{pw}')$ with $\mathsf{pw} \neq \mathsf{pw}'$), in which case indistinguishability holds since both parties obtained freshly chosen random keys from $\mathcal{F}_{\mathsf{rpwKE}}$ in the last game, just like in the current game. Thus, both games are indistinguishable.

We now observe that the NewKey interface handles all cases before the last instruction added in game $\mathbf{G}_3$ can trigger. We thus can remove it without any effect, resulting in a NewKey interface as in $\mathcal{F}_{\mathsf{aPAKE}}$. Note that now $\mathcal{F}$ already resembles $\mathcal{F}_{\mathsf{aPAKE}}$, the only difference being that it still relays the passwords of the parties and informs the simulator about a StorePwdFile input. We will show in the remaining games how to simulate without these.

Game $\mathbf{G}_{11}$: store file when needed. In this game we let $\mathcal{S}$ refrain from executing the file storage part of the server simulation upon receiving a StorePwdFile query from $\mathcal{F}$. Instead, $\mathcal{S}$ only remembers pw from that query and executes the file storage code for the server when receiving either a StealPwdFile or SrvSession query.

Since this game only constitutes a change of the order of execution in the simulated server's code only inbetween two outputs and without affecting them, game $\mathbf{G}_{11}$ and game $\mathbf{G}_{10}$ are indistinguishable.

Game $\mathbf{G}_{12}$: simulate honest server without password. For an honest session, we modify $\mathcal{F}$ to not relay $(\text{StorePwdFile}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw})$ to $\mathcal{S}$ anymore. This means we have to change simulation of an honest server to use a simulated password, which will in case of a successful off-line dictionary attack be overwritten with the real password. In this case, $\mathcal{S}$ can hide usage of the "wrong" password in

[18] We assume that the signature scheme has unique signing keys.

[19] We chose to decouple generating random session keys for interrupted sessions since these changes mainly concern dictionary attacks, which we analyzed already in games $\mathbf{G}_5$ and $\mathbf{G}_6$.

the beginning of the simulation by programming the random oracle accordingly. The changes in the simulation are as follows:

- Instead of submitting a password to $\mathcal{F}_{\mathsf{RO}}$, $\mathcal{S}$ directly chooses $r_{\mathcal{S}} \leftarrow \{0,1\}^{\lambda}, k_{\mathsf{pw}_{\mathcal{S}}} \leftarrow \{0,1\}^{2\lambda}$.
- Upon receiving $(\mathsf{sid}, \mathsf{pw})$ from $\mathcal{Z}$ (intended to reach the simulated $\mathcal{F}_{\mathsf{RO}}$), $\mathcal{S}$ submits $(\textsc{OfflineTestPwd}, \mathsf{pw})$ to $\mathcal{F}$. Upon receiving "correct guess", $\mathcal{S}$ stores $(\mathsf{pw}||3, r_{\mathcal{S}}), (\mathsf{pw}||2, k_{\mathsf{pw}_{\mathcal{S}}})$ in L.
- Upon receiving $(\textsc{StealPwdFile}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{S}$ relays the query to $\mathcal{F}$.
  - ▷ Upon "no password file" from $\mathcal{F}$, $\mathcal{S}$ answers $\perp$ to $\mathcal{Z}$.
  - ▷ Upon receiving pw from $\mathcal{F}$, if $\mathcal{S}$ already created a password file in the simulation of the honest server using $r_{\mathcal{S}}, k_{\mathsf{pw}_{\mathcal{S}}}$, $\mathcal{S}$ now stores $(\mathsf{pw}||3, r_{\mathcal{S}}), (\mathsf{pw}||2, k_{\mathsf{pw}_{\mathcal{S}}})$ in L. If $\mathcal{S}$ did not already create a password file, $\mathcal{S}$ starts simulation of the server with $\mathsf{pw}_{\mathcal{S}} := \mathsf{pw}$.
  - ▷ Upon "password file stolen" from $\mathcal{F}$, no further changes are made.

We first analyze indistinguishability if no server compromise happens. In this case, $\mathcal{S}$ simulates the server now with randomly chosen $r_{\mathcal{S}}, k_{\mathsf{pw}_{\mathcal{S}}}$, opposed to obtaining them from $\mathcal{F}_{\mathsf{RO}}$ using the real password in game $\mathbf{G}_{11}$. However, this does not affect the output of the parties since $\mathcal{F}$ determines them. Regarding the transcript, e is equally distributed in both games since K_1 is uniformly random, and σ does not depend on the password which is used in the simulation (it only depends on the passwords stored in $\mathcal{F}$).

In case of a server compromise, the output of the server is not affected since the NewKey instruction neither depends on whether $\mathcal{P}_S$ is marked `compromised` or not, nor on the outcomes of the OfflineTestPwd queries. This means that, as argued before, $\mathcal{F}$ will determine the outputs depending on the passwords provided to $\mathcal{F}$ upon StorePwdFile and UsrSession queries. Regarding the transcript, e is again equally distributed in both games due to K_1 being chosen uniformly random from $\{0,1\}^{2\lambda}$. The only difference is thus simulation of $\mathcal{F}_{\mathsf{RO}}$, but this is only a matter of timing when the entries $\mathsf{pw}||3$ and $\mathsf{pw}||2$ are added to L. However, since they are in any case stored *before* answering queries $(\mathsf{sid}, \mathsf{pw}||3)$ and $(\mathsf{sid}, \mathsf{pw}||2)$ of $\mathcal{Z}$, we conclude that the simulation of $\mathcal{F}_{\mathsf{RO}}$ is indistinguishable in both games.

Game $\mathbf{G}_{13}$: simulate honest client without password. Upon receiving $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw})$ from $\mathcal{P}_C$ via $\mathcal{F}$, if both parties are honest, instead of simulating the first $\mathcal{F}_{\mathsf{RO}}$ query of the client, $\mathcal{S}$ directly chooses $r'_{\mathcal{S}} \leftarrow \{0,1\}^{\lambda}$ and proceeds the simulation of the client with $r'_{\mathcal{S}}$. Since $\mathcal{S}$ does not make use of pw anymore, we can modify $\mathcal{F}$ to send only $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S)$ to $\mathcal{S}$.

With the same argument that was used for the server, the outputs of the honest client in game $\mathbf{G}_{13}$ and game $\mathbf{G}_{12}$ are equally distributed since they do not depend on the password used in the simulation. Regarding the transcript, creation of the signature just depends on whether client and server use the same password (and whether their transcripts from $\mathcal{F}_{\mathsf{rpwKE}}$ were the same). Since $\mathcal{S}$ will issue a TestAbort query (cf. game $\mathbf{G}_8$), it will obtain this information from $\mathcal{F}$ and thus the distribution of $\sigma_{\mathcal{S}}$ is equal to that in game $\mathbf{G}_{12}$.

Game $\mathbf{G}_{14}$: simulate corrupted session without password. We change the simulation of the honest $\mathcal{P}_i$ when $\mathcal{P}_{1-i}$ is corrupted. Namely, we omit the first usage of the random oracle regarding all inputs depending on the password. After receiving r from the environment as input to $\mathcal{F}_{\mathsf{rpwKE}}$ and issueing a $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, \mathsf{pw})$ query (cf. game $\mathbf{G}_5$), if the answer is "correct guess", we catch up on all random oracle queries using pw. If the answer is "wrong guess" (in which case it might be that $\mathsf{pw} = \perp$), we draw $r_{\mathcal{S}} \xleftarrow{\$} \{0,1\}^{\lambda}, k_{\mathsf{pw}_{\mathcal{S}}} \xleftarrow{\$} \{0,1\}^{2\lambda}$ with $r_{\mathcal{S}} \neq r$ and proceed the simulation using these values. Additionally, since $\mathcal{S}$ does not use the passwords relayed by $\mathcal{F}$ anymore, we delete the password from the UsrSession query that $\mathcal{F}$ sends to $\mathcal{S}$, and modify $\mathcal{F}$ to not send any message to $\mathcal{S}$ anymore upon receiving a StorePwdFile query.

Regarding indistinguishability of games $\mathbf{G}_{14}$ and $\mathbf{G}_{13}$, we first consider the "correct guess" case. In this case, simulation of the honest party uses the same password, and the only difference is when $\mathcal{S}$ issues $\mathcal{F}_{\mathsf{RO}}$ queries on behalf of the honest party. However, since pw was found in $\mathcal{F}_{\mathsf{RO}}$, the corresponding entries already exist in L and thus $\mathcal{Z}$ cannot distinguish both games by distinguishing the simulation

of $\mathcal{F}_{\mathsf{RO}}$. In case of "wrong guess", the simulation of $\mathcal{P}_i$ is proceeded using random $r_{\mathcal{S}}, k_{\mathsf{pw}_{\mathcal{S}}}$ as before, but with overwhelming probability there are no entries in L pointing to these values.

Regarding outputs, note that the output key K computed in the simulation will reach $\mathcal{P}_i$. However, the distribution of this key is exactly as in game $\mathbf{G}_{13}$, since it only depends on whether the inputs to $\mathcal{F}_{\mathsf{rpwKE}}$ will match or not, which is the same in both games.

It is now left to argue indistinguishability of the transcript using randomly drawn $r_{\mathcal{S}}, k_{\mathsf{pw}_{\mathcal{S}}}$. Note that indistinguishability even has to hold when $\mathcal{Z}$ queries $\mathcal{F}_{\mathsf{RO}}$ with the password that he used as input to $\mathcal{P}_i$, thus learning the true values of r, k_{pw}, i.e., the values that were used to generate the transcript in game $\mathbf{G}_{13}$. However, note that even knowing these values, $\mathcal{Z}$ can never learn K, namely what the honest party obtains as output from $\mathcal{F}_{\mathsf{rpwKE}}$. Since this value is uniformly random in both games due to interrupted records in $\mathcal{F}_{\mathsf{rpwKE}}$, the transcripts are equally distributed.

By collecting the changes among the games, it is clear that in game $\mathbf{G}_{14}$, $\mathcal{F} = \mathcal{F}_{\mathsf{aPAKE}}$. Since all game hops are only noticeable by $\mathcal{Z}$ with negligible probability, the theorem follows. For clarity, the code of the simulator is collected in Figures 10 and 11.

□

D Proof of Theorem 4.2

Proof. We adopt all notation from the proof of Theorem 4.1 and merely state how the latter has to be adjusted to prove Theorem 4.2.

Games $\mathbf{G}_0$-$\mathbf{G}_4$ are adopted unchanged.

Game $\mathbf{G}_4$ is changed in terms of the NewKey interface that is partly added:

> On $(\text{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{K})$ from $\mathcal{S}$ where $|key| = \lambda$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw}')$ not marked `completed`, do:
>
> - If the record is `compromised`, or either $\mathcal{P}$ or $\mathcal{P}'$ is corrupted, or $\mathsf{K} = \perp$, then send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K})$ to $\mathcal{P}$.
> - If the record is `interrupted`, send $(\mathsf{ssid}, \mathsf{ssid}, \perp)$ to $\mathcal{P}$.
> - Else, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K})$ to $\mathcal{P}$.
>
> Finally, mark $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw}')$ `completed`.

Note that this consitutes no change regarding outputs towards $\mathcal{Z}$ since $\mathcal{F}$ does not mark records `interrupted` so far, and will thus always relay the output keys from the simulation.

Game $\mathbf{G}_5$ is adopted.

Game $\mathbf{G}_6$ is only slightly changed: $\mathcal{S}$ does not have to ask TestAbort queries in case of obtaining "wrong guess", since $\mathcal{F}$ outputs $\perp$ in case of interrupted records right away. Indistinguishability holds with the same arguments as before.

Game $\mathbf{G}_7$ is changed in the same way: $\mathcal{S}$ only asks TestPwd and no TestAbort. Indistinguishability arguments are the same as before.

Game $\mathbf{G}_{8_\mathbf{a}}$: abort upon signature forgery. We change the simulation. Consider an honest session where the server might be compromised, $\mathcal{Z}$ does not send $(\mathsf{sid}, \mathsf{pw}||2)$ to $\mathcal{F}_{\mathsf{RO}}$ but injects an adversarially generated $\sigma_{\mathcal{Z}}$ as last message. In this case, we let $\mathcal{S}$ abort the simulation.

The simulator $\mathcal{S}$ is parametrized with a security parameter λ. It interacts with the environment $\mathcal{Z}$ and a client and a server party $\mathcal{P} \in \{\mathcal{P}_C, \mathcal{P}_S\}$ as described below. $\mathcal{S}$ internally simulates $\mathcal{F}_{\mathsf{RO}}$ as depicted in Figure 5. $\mathcal{S}$ forwards all instructions from $\mathcal{Z}$ to $\mathcal{A}$ and reports all output of $\mathcal{A}$ towards $\mathcal{Z}$. Instructions of corrupting a player are only obeyed if they are received before the protocol started, i.e., before $\mathcal{S}$ received any UsrSession, SrvSession or StorePwdFile query from $\mathcal{F}_{\mathsf{aPAKE}}$.

Messages from $\mathcal{F}_{\mathsf{aPAKE}}$

- **Upon receiving a query** $(\textbf{UsrSession}, \mathsf{sid}, \mathcal{P}_C)$ **from** $\mathcal{F}_{\mathsf{aPAKE}}$**,** $\mathcal{S}$ initializes an ITM $\mathcal{P}_C$ in the internal simulation running the code of the client in the Ω-method (cf. game $\mathbf{G}_1$), but directly starting with $r'_{\mathcal{S}} \leftarrow \{0,1\}^\lambda, k'_{\mathsf{pw}_\mathcal{S}} \xleftarrow{\$} \{0,1\}^{2\lambda}$ in case both parties are honest (cf. game $\mathbf{G}_{13}$). If $\mathcal{P}_S$ is corrupted, then the simulation of $\mathcal{P}_C$ is not started (cf. game $\mathbf{G}_{14}$).
- **Upon receiving the first of the queries** $\{(\textbf{SrvSession}, \mathsf{sid}, \mathsf{ssid}), (\textbf{StealPwdFile}, \mathsf{sid})\}$**,** $\mathcal{S}$ initializes an ITM $\mathcal{P}_S$ in the internal simulation running the code of the server in the Ω-method (cf. game $\mathbf{G}_{11}$), but directly starting with $r_{\mathcal{S}} \xleftarrow{\$} \{0,1\}^\lambda, k_{\mathsf{pw}_\mathcal{S}} \xleftarrow{\$} \{0,1\}^{2\lambda}$ in case both parties are honest (cf. game $\mathbf{G}_{12}$). If $\mathcal{P}_C$ is corrupted, then the simulation of $\mathcal{P}_S$ is not started (cf. game $\mathbf{G}_{14}$).

Actions triggered by internal simulation

- If an internally simulated party $\mathcal{P}_i$ **produces an output** $(\mathsf{sid}, \mathsf{K})$**,** $\mathcal{S}$ sends $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{K})$ to $\mathcal{F}_{\mathsf{aPAKE}}$. (Cf. game $\mathbf{G}_4$.)
- When the internally simulated $\mathcal{P}_C$ checks whether $h'_{\mathsf{sk}} \neq c'_2$, $\mathcal{S}$ issues $(\textsc{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C)$ towards $\mathcal{F}_{\mathsf{aPAKE}}$. If $\mathcal{S}$ receives back "fail", it aborts the simulation. (Cf. game $\mathbf{G}_8$.)
- When the internally simulated $\mathcal{P}_S$ **runs** Vfy **and** $\mathcal{Z}$ **did not inject the last message,** $\mathcal{S}$ sends $(\textsc{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S)$ to $\mathcal{F}_{\mathsf{aPAKE}}$. (Cf. game $\mathbf{G}_8$.)

Messages from $\mathcal{Z}$

- **Upon receiving** r **from** $\mathcal{Z}$ **intended as input of a corrupted** $\mathcal{P}_{1-i}$**,** $\mathcal{S}$ looks for an entry $(\mathsf{pw}||3)$ in L. (Cf. game $\mathbf{G}_5$.)
  - ▷ If there is such an entry, $\mathcal{S}$ submits $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{aPAKE}}$. If $\mathcal{S}$ receives back "correct guess", he catches up on the first two usages of $\mathcal{F}_{\mathsf{RO}}$ in the simulation of the honest $\mathcal{P}_i$ using pw. If $\mathcal{S}$ receives back "wrong guess", he proceeds simulation of the honest $\mathcal{P}_i$ using $r_{\mathcal{S}} \xleftarrow{\$} \{0,1\}^\lambda, k_{\mathsf{pw}_\mathcal{S}} \xleftarrow{\$} \{0,1\}^{2\lambda}$. (Cf. games $\mathbf{G}_5$ and $\mathbf{G}_{14}$.)
  - ▷ If there is no such entry, $\mathcal{S}$ submits $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, \perp)$ to $\mathcal{F}_{\mathsf{aPAKE}}$. Also in this case, $\mathcal{S}$ proceeds the simulation of the honest $\mathcal{P}_i$ using $r_{\mathcal{S}} \xleftarrow{\$} \{0,1\}^\lambda, k_{\mathsf{pw}_\mathcal{S}} \xleftarrow{\$} \{0,1\}^{2\lambda}$. (Cf. games $\mathbf{G}_5$ and $\mathbf{G}_{14}$.)

Figure 10: The simulator $\mathcal{S}$ for the proof of Theorem 4.1.

Messages from $\mathcal{Z}$

- **Upon receiving** (**TestPwd**, $\langle \mathsf{sid}, \mathsf{ssid} \rangle, \mathcal{P}_i, r$) **from** $\mathcal{Z}$**,** if sid is an uncorrupted session, $\mathcal{P}_i$ is the client, $\mathcal{Z}$ already issued a StealPwdFile query and $\mathcal{S}$ replied with a value $\mathsf{file} = (r, \cdot, \cdot)$, $\mathcal{S}$ now sends (Impersonate, $\mathsf{sid}, \mathsf{ssid}$) to $\mathcal{F}$. If the answer is "correct guess", $\mathcal{S}$ continues the internal simulation of the client with r. Else, if $\mathcal{Z}$ received r as response of some $\mathcal{F}_{\mathsf{RO}}$ query ($\mathsf{pw}||3$), $\mathcal{S}$ sends (TestPwd, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, pw$) to $\mathcal{F}$. If the answer is "correct guess", $\mathcal{S}$ continues the internal simulation of $\mathcal{P}_i$ with pw. In any case, $\mathcal{S}$ forwards the answer of his own Impersonate query to $\mathcal{Z}$ as reply to $\mathcal{Z}$'s TestPwd query. (Cf. game $\mathbf{G}_6$.)
- **Upon receiving** $e_{\mathcal{Z}}$ **from** $\mathcal{Z}$ where $e_{\mathcal{Z}}$ is different from the interally simulated value e, $\mathcal{S}$ sends (TestPwd, $\mathsf{ssid}, \mathcal{P}_C, \perp$) to $\mathcal{F}_{\mathsf{aPAKE}}$. (Cf. game $\mathbf{G}_7$.)
- **Upon receiving** $\sigma_{\mathcal{Z}}$ **from** $\mathcal{Z}$ **where** $\mathsf{Vfy}_{\mathsf{vk}}(\sigma_{\mathcal{Z}}, \mathsf{tr}) = 0$**,** $\mathcal{S}$ sends (TestPwd, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \perp$) and afterwards (TestAbort, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S$). (Cf. game $\mathbf{G}_8$.)
- **Upon receiving** ($\mathsf{sid}, \mathsf{pw}$) **from** $\mathcal{Z}$**,** $\mathcal{S}$ submits (OfflineTestPwd, pw) to $\mathcal{F}_{\mathsf{aPAKE}}$. Upon receiving "correct guess", $\mathcal{S}$ stores $(\mathsf{pw}||3, r_{\mathcal{S}}), (\mathsf{pw}||2, k_{\mathsf{pw}_S})$ in L. (Cf. game $\mathbf{G}_{12}$.)
- **Upon receiving** (**StealPwdFile**, sid) **from** $\mathcal{Z}$**,** $\mathcal{S}$ relays the query to $\mathcal{F}_{\mathsf{aPAKE}}$.
  - ▷ In case of receiving back "no password file" from $\mathcal{F}_{\mathsf{aPAKE}}$, $\mathcal{S}$ sends $\perp$ to $\mathcal{Z}$.
  - ▷ In case of receiving back pw from $\mathcal{F}_{\mathsf{aPAKE}}$, if $\mathcal{S}$ already created a password file in the simulation of the honest server using $r_{\mathcal{S}}, k_{\mathsf{pw}_S}$, $\mathcal{S}$ now stores $(\mathsf{pw}||3, r_{\mathcal{S}}), (\mathsf{pw}||2, k_{\mathsf{pw}_S})$ in L.

Figure 11: The simulator $\mathcal{S}$ for the proof of Theorem 4.1, cont'd.

The current and last game are indistinguishable due to the EUF-CMA-security of the signature scheme.

Game $\mathbf{G}_{8_b}$: man-in-the-middle against server. Fortunately, this game gets much simpler with $\mathcal{F}_{\mathsf{aPAKE}}$ with strong explicit authentication. We change the simulation as follows:

- If $\mathcal{Z}$ injects $\sigma_{\mathcal{Z}}$ as last message where $\mathsf{Vfy}_{\mathsf{vk}}(\sigma_{\mathcal{Z}}, \mathsf{tr}) = 0$ then $\mathcal{S}$ issues (TestPwd, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \perp$)

In case of an adversarially generated $\sigma_{\mathcal{Z}}$ with $\mathsf{Vfy}_{\mathsf{vk}}(\sigma, \mathsf{tr}) = 0$, the changes in the current game will let $\mathcal{P}_S$ output $\perp$. However, $\mathcal{P}_S$ already outputs $\mathsf{K}_2 = \perp$ in game $\mathbf{G}_{13}$ in case of a non-verifying signature.

Game $\mathbf{G}_{8_c}$: completing explicit authentication in $\mathcal{F}$. We add the third and fourth bullet to the NewKey interface - comprising the changes made in game $\mathbf{G}_4$.
The fourth bullet just let $\mathcal{F}$ relay $\perp$ from $\mathcal{S}$, which already was the case in the last game. For the case of $\mathcal{F}$ now deciding to output $\perp$ for fresh records with mismatching passwords, we only have to consider honest sessions without man-in-the-middle attacks by $\mathcal{Z}$, since in case of these attacks the output of the attacked party already was $\perp$ in the previous game. For fresh records, $\mathcal{F}$ now outputs $\perp$ in case there is a fresh record of the other party but with a mismatching password. However, in this case parties outputted $\perp$ already in the last game with overwhelming probability in λ, since $\mathcal{F}_{\mathsf{rpwKE}}$ hands out values that are different with probability $1/2\lambda$.

Game $\mathbf{G}_9$ is adopted unchanged.

Game $\mathbf{G}_{10}$: random session key for honest session. The changes are adopted. However, note that the analysis of indistinguishability becomes easier since the only case in which this code of $\mathcal{F}$ is executed is when the other party did not receive a key yet, but there are fresh records with matching passwords. The randomness of the outputs of $\mathcal{F}_{\mathsf{rpwKE}}$ is enough to argue perfect indistinguishability.

Game $\mathbf{G}_{11}$ remains unchanged.

Game $\mathbf{G}_{12}$: simulate honest server without password. The game remains unchanged except that we now need to also change a part of code of $\mathcal{S}$ that we added in game $\mathbf{G}_{8_\mathbf{a}}$. There, $\mathcal{S}$ made use of his knowledge of the true password of the server by comparing it with random oracle queries issued by $\mathcal{Z}$. We thus replace the change made in game $\mathbf{G}_{8_\mathbf{a}}$ by the following instruction:

> Consider an honest session where the server might be compromised, $\mathcal{S}$ never received "correct guess" from OFFLINETESTPWD and $\mathcal{Z}$ injects an adversarially generated $\sigma_{\mathcal{Z}}$ as last message. In this case, we let $\mathcal{S}$ abort the simulation.

Since $\mathcal{S}$ only uses OFFLINETESTPWD upon random oracle inputs of $\mathcal{Z}$, the replacement is unnoticeable: $\mathcal{S}$ obtains "correct guess" if and only if $\mathcal{Z}$ submits the server's password to $\mathcal{F}_{\mathsf{RO}}$.
The rest of the changes and argumentation of indistinguishability can be adopted.

Game $\mathbf{G}_{13}$: simulate honest client without password. The changes in the simulation are adopted. The argument of indistinguishability has to be slightly changed: now, $\mathcal{S}$ will be automatically informed by $\mathcal{F}$ in case of authentication failure at the client's side, which is enough to let him simulate a signature that verifies or not verifies according to the authentication status.

Game $\mathbf{G}_{14}$ is adopted. This concludes the proof of the theorem. □