Separating Symmetric and Asymmetric Password-Authenticated Key Exchange

Julia Hesse

IBM Research, Zurich, Switzerland
`jhs@zurich.ibm.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 symmetric PAKE since the latter is not supposed to protect the password upon compromise. In this paper, we provide three contributions:

- **Separating symmetric 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 symmetric PAKE, programmability is not required. Our results also rule out the existence of UC-secure asymmetric PAKE in the CRS model.
- **Revising the security definition.** We identify and close some gaps 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-method proposed by Gentry et al. satisfies our new security notion for asymmetric PAKE. To our knowledge, this is the first formal security proof of the Ω-method in the literature.
- **Composable multi-party asymmetric PAKE.** We showcase how our revisited security notion for 2-party asymmetric PAKE can be used to obtain asymmetric PAKE protocols in the multi-user setting and discuss important aspects for implementing such a protocol.

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. An important cryptographic building block 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

Reference	Type	Sec. notion	Model
[BPR00]	symmetric	BPR	non-prog. RO
[GL01, KOY01, BCL$^+$11, KV11]	symmetric	BPR	standard
[BP13]	asymmetric	BPR	non-prog. RO & GGM
[PW17]	asymmetric	BPR	GGM
[CHK$^+$05, KV11, BBC$^+$13]	symmetric	UC	standard
[GMR06]	asymmetric	UC	progr. RO
[JR16]	asymmetric	UC	limited progr. RO
[HL18]	asymmetric	UC	progr. RO
[JKX18]	asymmetric	UC, strong	progr. RO
[BJX19]	asymmetric	UC, strong	non-prog. RO & GGM

Table 1: Comparison of static security of different PAKE and aPAKE schemes, where "strong" denotes schemes that prevent precomputation of password files (e.g., via precomputing hash tables). RO means random oracle and GGM means generic group model.

of PAKE is the TLS-SRP ciphersuite (specified in RFCs 2945 and 5054), which is used by GnuTLS, OpenSSL and Apache.

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* PAKE (aPAKE) (sometimes also *augmented* or *verifier-based* in the literature). To emphasize that we talk about a PAKE protocol without different roles, we write *symmetric* PAKE.

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 symmetric 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 required.

Is asymmetric PAKE harder than symmetric PAKE? Intuitively, asymmetric PAKE seems more challenging than symmetric PAKE, since asymmetric PAKE protocols provide a guarantee on top: they are supposed to protect passwords whenever the storage of a server is leaked to the adversary. This claim has evidence in the literature: there are symmetric 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 when considering 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 or the generic group model (GGM). See Table 1 for a comparison.

To our knowledge, in none of the aforementioned models there exist any formal proof of asymmetric PAKE being harder to achieve than symmetric PAKE. A close look at Table 1 reveals that the "gap" in the assumption is much bigger in case of UC security. Can we make this gap as small as for BPR secure schemes? For sure we would like to answer this question in the affirmative, since for asymmetric PAKE, UC-security has two notable advantages over BPR security: it comes with a composability guarantee, and it considers adversarially-chosen passwords.

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 symmetric 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 asymmetric PAKE is harder to achieve than symmetric PAKE. Interestingly, our impossibility result directly extends to a setting where parties, additionally to the non-programmable random oracle, have access to a common reference string (CRS). Although such a CRS offers a limited form of programmability, we can show that this is not enough to obtain UC-secure asymmetric PAKE.

In preparation of this formal result, and as a separate contribution, we revisit the ideal functionality $\mathcal{F}_{\mathsf{apwKE}}$ for asymmetric PAKE of Gentry, MacKenzie and Ramzan [GMR06]. Our changes summarize 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. This was already proposed but not enforced by Gentry et al. [GMR06].

- We show that $\mathcal{F}_{\mathsf{apwKE}}$ allows attacks on *explicit authentication.* In a PAKE protocol with explicit authentication, parties are informed whether the other party held the same password and thus computed the same session key. However, $\mathcal{F}_{\mathsf{apwKE}}$ allows the adversary to make parties believe in the security of adversarially chosen session keys. This is clearly devastating for applications such as secure channels. In our revisited functionality $\mathcal{F}_{\mathsf{aPAKE}}$ we exclude such attacks by introducing a proper modeling of explicit authentication.

We argue plausibility of our revisited functionality $\mathcal{F}_{\mathsf{aPAKE}}$ by showing that it is realized by an asymmetric PAKE protocol called the Ω-method, introduced by Gentry et al. [GMR06]. To our knowledge, this is the first proof of security of the Ω-method in the literature. Not surprisingly, the original publication of the Ω-method did not include a proof but only a claim of security, which is invalidated with our findings of their functionality being impossible to realize. Finally, we showcase how our 2-party functionality $\mathcal{F}_{\mathsf{aPAKE}}$ can be used to obtain multi-user asymmetric PAKE protocols. We highlight a specific artifact of UC-security that has to be considered when implementing such a scheme. Let us now explain our results in more detail.

Separating symmetric and asymmetric PAKE. As already mentioned, asymmetric PAKE protocols are supposed to provide some protection of the password in case of a server compromise. Essentially, in case of a security breach where account data of users are leaked to the adversary, we want that the adversary does not obtain all user passwords in the clear. Formally, this attack is modeled via a partial leakage of the internal state of the server who already stored the user's account data. 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 ones stated in the UC model, adaptive attacks often impose a problem. Such problems are often referred to as "commitment problem": they require the simulator to explain how, e.g., a transcript that he committed to in the beginning of the protocol matches certain secrets of honest participants that are revealed only later. 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, an NPRO 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.

Inspired by the work of Nielsen, we obtain the following result: UC-secure asymmetric PAKE is impossible to achieve in the NPRO model. The intuition is as follows: 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 contains. The attacker can now test whether some password is contained in the simulated file. Since accessing the external oracle is sufficient to compute this test, the simulator merely learns the tested password but cannot influence the outcome of the test.

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]. For non-committing encryption, their strategy 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 there exists only one password file at the server. And 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 high probability, he will exhaust before finding a good file.

We further investigate how our proof technique extends to more setup assumptions. We find that our impossibility result directly extends to the NPRO model where parties can access an ideal common reference string (CRS). While the CRS can be "programmed" in the simulation, it does not resolve the simulator's commitment issue: neither does usage of the CRS provide the simulator with any information about the server's password prior to simulating the file, nor does determining the CRS let the simulator influence the aforementioned test. We can thus rule out the existence of UC-secure asymmetric PAKE protocols plain CRS as well as the NPRO+CRS model.

Opposed to our findings for asymmetric PAKE, it is known from the literature that UC-secure *symmetric* 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 Canetti et al. [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 symmetric to asymmetric PAKE in the UC model, we are forced to move further away from realistic setup assumptions.

Modeling server compromise. Towards separating symmetric and asymmetric PAKE, we first carefully revisit the ideal functionality $\mathcal{F}_{\mathsf{apwKE}}$ (cf. Figure 1) for asymmetric PAKE of Gentry et al. [GMR06], which adopts the ideal functionality for symmetric PAKE [CHK$^+$05] to the asymmetric case. For the reader not familiar with $\mathcal{F}_{\mathsf{apwKE}}$, we provide the functionality and a thorough introduction to it in Appendix C. For providing an overview of our contributions, we first focus on how server compromise attacks are modeled by $\mathcal{F}_{\mathsf{apwKE}}$. After learning about a server compromise attack, $\mathcal{F}_{\mathsf{apwKE}}$ enables the adversary to (a) make off-line password guesses against the file and (b) impersonate the server using the file. $\mathcal{F}_{\mathsf{apwKE}}$ only allows the adversary to compromise and make password guesses upon getting instructions from the distinguisher $\mathcal{Z}$.

We show that it is necessary to revisit $\mathcal{F}_{\mathsf{apwKE}}$ by proving that restricting the adversary to only submit off-line password guesses upon instructions from $\mathcal{Z}$ results in $\mathcal{F}_{\mathsf{apwKE}}$ being impossible to realize. By this, we invalidate the claimed security of the Ω-method [GMR06]. We also observe that putting restrictions such as "only ask query x if $\mathcal{Z}$ tells you" on the adversary is not conform with the UC framework and invalidates important properties of the framework such as simulation with respect to the dummy adversary.

Towards a better modeling, and towards resolving the now-open question which security guarantees the Ω-method fulfils, we revisit $\mathcal{F}_{\mathsf{apwKE}}$ and propose our own ideal functionality $\mathcal{F}_{\mathsf{aPAKE}}$. The changes are as follows: we lift the aforementioned restriction on the adversary regarding off-line password guesses and argue why the resulting security notion captures what we expect from an asymmetric PAKE. We further propose a UC-conform modeling of server compromise attacks as "partial" corruption queries which, in the real execution of the protocol, partly leak the internal state of an honest party to the adversary (i.e., the password file).

We call our revisited functionality for asymmetric PAKE $\mathcal{F}_{\mathsf{aPAKE}}$. It however differs in another aspect from $\mathcal{F}_{\mathsf{apwKE}}$, 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 or not, 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.*

To demonstrate usefulness of our revisited functionality $\mathcal{F}_{\mathsf{aPAKE}}$, we show that the Ω-method UC-realizes it. To our knowledge, this is the first full proof of security for the Ω-method.

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. But realistic scenarios for PAKE comprise thousands of users and hundreds of servers all using the same protocol to establish secure communication channels or to authenticate clients. 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 its realization. Each client would invoke an instance of $\mathcal{F}_{\mathsf{aPAKE}}$, and a server can participates in arbitrarily many of them. To avoid that all instances use their "own" setup, which would require a server to use, e.g., a different hash function for each client, all functionalities could share their setups. This can be achieved by transforming the setup, e.g., a random oracle $\mathcal{F}_{\mathsf{RO}}$ to a multi-party functionality $\hat{\mathcal{F}}_{\mathsf{RO}}$ that acts as a wrapper for multiple copies of $\mathcal{F}_{\mathsf{RO}}$. This approach is widely used and called UC with joint state (JUC) [CR03].

We showcase this transformation to the multi-user setting for the Ω-method. Interestingly, a client in the multi-user Ω-method is now required to remember which is "her" copy of the random oracle. We demonstrate that this does not hinder practicality of the scheme since we can identify a party's random oracle by information that this party, in any implementation, has to remember anyway (e.g., the server's URL and her own username).

Related work. Canetti et al. [CHK+05] show impossibility of $\mathcal{F}_{\mathsf{apwKE}}$ in the plain model. Gentry et al. [GMR06] show how to transform a UC-secure PAKE into a UC-secure asymmetric PAKE (the opposite direction is trivial). 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 gap between non-programmable and programmable random oracles. Fischlin et al. [FLR+10] introduce 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, Jutla and Roy [JR16] propose an aPAKE that is secure w.r.t a limited programmability random oracle.

Canetti and Krawczyk [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 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 by Canetti and Krawczyk [CK02] can deal with many users from the start. While the goal of their 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-method. 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 aPAKE Security Model, revisited

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 by Canetti et al. [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 1 and refer the reader not familiar to it to Appendix B for a thorough introduction to the functionality. To model a server compromise attack, $\mathcal{F}_{\mathsf{apwKE}}$ provides three interfaces called StealPwdFile, OfflineTestPwd and Impersonate, which we now describe in more detail.

- StealPwdFile 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.

- OfflineTestPwd enables an off-line dictionary attack: the adversary can test whether some password is contained in the password file. The answer is a bit, depending on whether the guess was correct or not. The attack is called "off-line" since it can be mounted by the adversary without interacting with the client. Like StealPwdFile, this attack can only be mounted by the simulator if $\mathcal{Z}$ instructs him to do so. The adversary gets confirmation on a correct guess only after a StealPwdFile query happened.

- Impersonate can be used by the adversary *after* StealPwdFile was issued by $\mathcal{Z}$. This interface enables the adversary to engage in a key exchange session with the client, using the stolen password file as authenticating data. This interface 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.

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 $\mathcal{P}$.
  - ▷ Else, 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 $\mathsf{K}'' \overset{\$}{\leftarrow} \{0,1\}^\lambda$ 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 1: 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 $\mathcal{Z}$.

Definitional issues with StealPwdFile and OfflineTestPwd interfaces of $\mathcal{F}_{\text{APWKE}}$. Let us make two observations about these interfaces. Firstly, the restriction of letting the adversary ask specific queries only upon receiving them from $\mathcal{Z}$ constitutes a change of the UC framework (Gentry et al. [GMR06] 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 the adversary. A real-world adversary $\mathcal{A}$ can easily decode this query and perform the desired attack, while the simulator in the ideal world cannot accomplish the corruption at $\mathcal{F}_{\mathsf{aPAKE}}$ due to the wrong message format. This way, $\mathcal{Z}$ can keep the simulator from using his interfaces, leaving him with no leverage to presume his simulation. Clearly, such an "intelligent" real-world adversary is worse than a dummy adversary, who would just relay the message without any effect. 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 $\mathcal{A}$ obtains StealPwdFile and OfflineTestPwd queries from $\mathcal{Z}$. Gentry et al. [GMR06] 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 [Can20], Section 4.3.1). Let us explain why this underspecification is problematic. In the UC framework, the real-world adversary $\mathcal{A}$ only has influence on the communication channel and corrupted parties, and none of this helps him to steal a password file from the server. For analyzing 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 has to be simulated by $\mathcal{S}$. 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 three types.

2.1 Fix no.1: Defining the Corruption Model

To address the aforementioned issues, we first define server compromise to *be party corruption*. This possibility was already pointed out by Gentry et al. [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 lets us flexibly define the effect of server compromise in the real world w.r.t internal state of the server. For example, 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. We note that this leads to a UC-conform modeling since arbitrary corruption models can be integrated into the UC framework (cf. [Can20], section 7.1).

Formally, besides Byzantine party corruption that is usually modeled via a query $(\text{corrupt}, \mathcal{P}, \mathsf{sid})$ from $\mathcal{Z}$ to $\mathcal{A}$, we allow $\mathcal{Z}$ to issue an additional corruption query $(\text{StealPwdFile}, \mathsf{sid})$. To formalize what happens upon this query in the ideal world, we adopt the conventions for corruption queries from the UC framework [Can20] (cf. Section 7.1) and let dummy parties in the ideal world ignore corruption messages. Instead, these queries are handeled by the ideal functionality, who receives them directly from $\mathcal{S}$. We now detail the effect of the two types of corruption queries that we allow $\mathcal{Z}$ to ask.

Byzantine corruption
Real world: Upon receiving a message $(\textsc{corrupt}, \mathcal{P}, \mathsf{sid})$ 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 $(\textsc{corrupt}, \mathcal{P}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{S}$ delivers the message to $\mathcal{F}_{\mathsf{apwKE}}$, who marks $\mathcal{P}$ as $\texttt{corrupted}$. 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.

Server compromise
Real world: Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ be an efficiently computable function. We denote the internal state of $\mathcal{P}_S$ with state, consisting of all messages received by and sent to the server ITI, its random coins and current program's state. Upon receiving a message $(\textsc{StealPwdFile}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{A}$ delivers the message to $\mathcal{P}_S$, who immediately sends $f(\mathsf{state})$ to $\mathcal{A}$. This definition of corruption resembles what Canetti [Can20] describes as *physical "side channel" attacks* since it results in leaking a function of the internal state of a party. A natural choice is $f(\mathsf{state}) = \mathsf{file}$, with file being the variable in the server's code storing the password file, which we will use in this work.
Ideal world: Upon $(\textsc{StealPwdFile}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{S}$ sends this message to $\mathcal{F}_{\mathsf{apwKE}}$. $\mathcal{F}_{\mathsf{apwKE}}$ marks $\mathcal{P}_S$ as $\texttt{compromised}$. If there are records $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ ("server $\mathcal{P}_S$ used pw to generate password file") and $(\textsc{offline}, \mathsf{pw})$ ("$\mathcal{S}$ guessed that file contains 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 the aPAKE literature [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 results in leakage of an empty file.

2.2 Fix no.2: Bounding Offline Attacks

We now turn our attention to the OfflineTestPwd interface. If the server is compromised, which we formalized via the corruption query StealPwdFile, the adversary can query OfflineTestPwd to figure out the password within the password file. Such an attack on the password file is called "off-line" to emphasize that there is no further interaction with the client required.

Clearly, a meaningful aPAKE security notion should offer means to tell a protocol with $\mathsf{file} = \mathsf{pw}$ apart from a protocol with, e.g., $\mathsf{file} = H(pw)$ 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, which in this example represent computations of the function $H()$, explicit to $\mathcal{Z}$.

To this end, Gentry et al. [GMR06] define $\mathcal{F}_{\mathsf{aPAKE}}$ such that $\mathcal{A}$ is allowed to query OfflineTestPwd only by relaying queries of $\mathcal{Z}$. Clearly, this gives $\mathcal{Z}$ a way to bound the number of OfflineTestPwd guesses and does not allow to prove a protocol with $\mathsf{file} = \mathsf{pw}$ 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 Appendix B for a definition of this assumption). In a nutshell, the reason is that requiring $\mathcal{Z}$'s permission to issue OfflineTestPwd queries keeps the simulator from using this interface and prevents successful simulation of any aPAKE protocol. We formally prove this impossibility result in Appendix D.

To circumvent the impossibility result, we propose to 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 the simulator's usage of this interface. Fortunately, the UC framework provides a technical tool for this, as we will detail now.

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 assume passwords to be 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 Gentry et al. [GMR06] to use this limitation in the first place. However, as we show in Theorem D.1, 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. 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. To enforce this bounded usage of OfflineTestPwd, we require a simulator for a protocol realizing our revisited functionality $\mathcal{F}_{\mathsf{aPAKE}}$ to be *locally T-bounded* (cf. Section A for the formal definition of this runtime bound).

We are now ready to state our revisited functionality $\mathcal{F}_{\mathsf{aPAKE}}$ in Figure 2. All differences to $\mathcal{F}_{\mathsf{apwKE}}$ were already explained above, except for changes regarding the TestAbort interface concerning explicit authentication which we describe in Section 4.1. The reasons for postponing are two-fold: the changes can be best explained when investigating security of the Ω-method, and the interface is not relevant for our separation result in the upcoming section. To summarize, the differences between using $\mathcal{F}_{\mathsf{apwKE}}$ or our revisited $\mathcal{F}_{\mathsf{aPAKE}}$ are as follows:

- StealPwdFile is now a corruption query (change does not show in $\mathcal{F}_{\mathsf{aPAKE}}$ but in the assumed corruption model).
- OfflineTestPwd can be asked without getting instructions from $\mathcal{Z}$, but a simulator interacting with $\mathcal{F}_{\mathsf{aPAKE}}$ can only access OfflineTestPwd as long as its runtime remains locally T-bounded.
- The TestAbort interface is removed and NewKey is adjusted to analyze *either* protocols with *or* without explicit authentication.

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

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, mark $\mathcal{P}_S$ as `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 $(\textsc{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}')$ from $\mathcal{S}$, do:
  - ▷ If there is a record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ and $\mathcal{P}_S$ is 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 $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$, $\mathcal{P}_S$ is `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 `completed`, do:
  - ▷ Else if the record is `compromised`, or $\mathcal{P}$ or $\mathcal{P}'$ is corrupted, or $\mathsf{K} = \perp$, 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 $\mathcal{P}$.
  - ▷ Else 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 $(\mathsf{sid}, \mathsf{ssid}, \perp)$ to $\mathcal{P}$ and $\mathcal{S}$.
  - ▷ Else, pick $\mathsf{K}'' \overset{\$}{\leftarrow} \{0,1\}^\lambda$ 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). $\boxed{\text{Framed queries}}$ can only be asked upon getting instructions from $\mathcal{Z}$. Gray boxes indicate queries that required instructions from $\mathcal{Z}$ according to Gentry et al. [GMR06], but not in our $\mathcal{F}_{\mathsf{aPAKE}}$. 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.

3 The Separation Result

After revisiting the UC security notion for aPAKE and closing some of its definitional gaps in the previous section, we will now turn to our main result. Namely, we give the first formal evidence that UC secure asymmetric PAKE is indeed harder to achieve than symmetric PAKE.

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 access 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 of Nielsen 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

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

Verifiable Password Files. To state our separation result, we need to formally capture what it means for an aPAKE protocol formulated in the UC framework to have a *verifiable password file.* In a nutshell, the definition captures whether for a password file file leaked by a compromised server, the environment $\mathcal{Z}$ is provided with all necessary information and interfaces at hybrid functionalities to determine whether any given pw is contained in file. We do not specify how $\mathcal{Z}$ determines this, e.g., whether $\mathcal{Z}$ tries to recompute file from pw or whether $\mathcal{Z}$ runs the aPAKE protocol internally on inputs pw and file. For example, hashing the password results in a verifiable file $H(pw)$ if $\mathcal{Z}$ posesses the description of $H()$ or can access a functionality computing $H()$[1].

Definition 3.1 (Verifiable Password File)**.** *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 has a* verifiable password file file *if, for a given* pw*, $\mathcal{Z}$ can efficiently determine whether* file *was created upon input* pw*, by only interacting with the ideal functionalities $\mathcal{F}$ (via $\mathcal{A}$).*

We emphasize that only adversarial interfaces at hybrid functionalities may help $\mathcal{Z}$ in verifying correctness of the password file, and no further inputs to other ITMs are required. This will become crucial in proving our impossibility result. Let us emphasize that Definition 3.1 does not actually restrict the class of asymmetric PAKE protocols. In fact, the only asymmetric PAKE protocol formulated in the UC framework that does not have a verifiable password file that we are aware of is the ideal protocol $\mathcal{F}_{\mathsf{aPAKE}}$ itself. This protocol has a trivial password file ("no password file"/"password file stolen" are possible outputs upon server compromise), requires no hybrid functionalities and thus the password file is not verifiable. And indeed, the following Theorem would not hold w.r.t the ideal protocol $\mathcal{F}_{\mathsf{aPAKE}}$, due to the well-known fact that every UC functionality realizes itself. Let us stress though that all practical aPAKE protocols proposed in the literature do

[1]While it seems contradictory for an asymmetric PAKE protocol to not have a verifiable file, one could indeed build such strong protocols by shielding information from $\mathcal{Z}$. One example would be involving interaction with a third party. If this party is involved in the registration phase but is not accessible by $\mathcal{Z}$, off-line attacks can be prevented. To put an example, sharing the role of the server among several parties yields a protocol that does *not* have verifiable files. Such protocols are called *threshold PAKE* [FJ00]. While $\mathcal{F}_{\mathsf{apwKE}}$ allows to consider such protocols, we are not interested in analyzing them in this paper.

have verifiable password files, simply because a password at the client side and a file at the server side need to be enough to complete the key exchange. Definition 3.1 should thus be viewed as a necessary formalism to prove our separation result and not as a restriction of it.

With the NPRO model and the property of verifiable password files we now have all tools to prove our separation result in the following theorem.

Theorem 3.2. *The functionality $\mathcal{F}_{\mathsf{aPAKE}}$ as depicted in Fig. 2 is not realizable in the NPRO model by any protocol with verifiable password file. More detailed, for every such protocol π and every polynomial T, there exists an attacker $\mathcal{A}$ and an environment $\mathcal{Z}$ restricted to static Byzantine corruptions and adaptive server compromise, 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 $(\textsc{StorePwdFile}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw})$ as input to $\mathcal{P}_S$, where $\mathsf{pw} \overset{\$}{\leftarrow} \{0,1\}^{\lceil\sqrt{2\lambda}\rceil}$. All parties remain uncorrupted.
- $\mathcal{Z}^{\mathcal{O}}$ sends $(\textsc{StealPwdFile}, \mathsf{sid})$ to the adversary to compromise the server. $\mathcal{Z}$ obtains file as answer from the adversary.
- $\mathcal{Z}^{\mathcal{O}}$ verifies $(\mathsf{pw}, \mathsf{file})$.
- $\mathcal{Z}^{\mathcal{O}}$ outputs 0 if verification succeeds, else it outputs 1.

Note that step 3 can be performed by $\mathcal{Z}^{\mathcal{O}}$ without any interaction with $\mathcal{A}$, simply since it has the same oracle access as $\mathcal{A}^{\mathcal{O}}$. To verify the file, $\mathcal{Z}$ runs π internally, using pw as input to the client and file for the server. It outputs 0 if both client and server compute the same session key. Due to the file 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. Since $\mathcal{Z}^{\mathcal{O}}$ issues only $\textsc{StorePwdFile}$ and $\textsc{StealPwdFile}$ queries (which both do not produce any output in this case), no $(\textsc{offline}, ...)$ or $(\mathsf{ssid}, ...)$ 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 $\textsc{OfflineTestPwd}$. 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 $\textsc{OfflineTestPwd}$ is n_I. In the above attack, n_I consists of the minimal input 1^λ, $(\textsc{StealPwdFile}, \mathsf{sid})$ from $\mathcal{Z}^{\mathcal{O}}$ as well as a bit as answer to each $\textsc{StealPwdFile}$ and $\textsc{OfflineTestPwd}$ 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 $(\textsc{StealPwdFile}, \mathsf{sid})$ 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 $\textsc{OfflineTestPwd}$ queries of $\mathcal{S}^{\mathcal{O}}$. Since $|\mathsf{pw}| \geq 1$, 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 $\textsc{OfflineTestPwd}$ is at most 1/2. It follows that $\Pr[\mathcal{Z} \rightarrow 1|\text{ ideal }] = \Pr[\mathcal{A}^{\mathcal{O}}(\mathsf{pw}) = \mathsf{file}_{\mathcal{S}}] \leq 1/2 + 1/2^{k-1}$, contradicting the UC-security of π. □

Theorem 3.2 indicates that some form of programmability is required to realize $\mathcal{F}_{\mathsf{aPAKE}}$. However, our proof technique differs significantly from other NPRO impossibility results such as the one from Nielsen [Nie02] for non-committing encryption. Essentially, in non-committing encryption, simulation of *arbitrarily many* ciphertexts are necessary, while for asymmetric PAKE as specified by $\mathcal{F}_{\mathsf{aPAKE}}$ *just one* password file needs to be simulated. An interesting question is thus whether a setup assumption such as a common reference string (CRS), which offers a limited form of programmability, is enough to realize $\mathcal{F}_{\mathsf{aPAKE}}$. Unfortunately, we answer this question negatively by extending the impossibility result of Theorem 3.2 to a setting where a common reference string (CRS) is used as additional setup assumption.

Theorem 3.3. *Theorem 3.2 holds also in a setting where all entities are additionally granted access to an ideal common reference string (CRS).*

Proof Sketch. The argumentation is the same as before, but additionally showing that the CRS does not help the simulator since he *does not know* which password to program the CRS for.

We modify the proof of Thm 3.2 as follows: as a new first step, $\mathcal{Z}$ asks for the CRS value. Since this completely "consumes" the CRS functionality, meaning that no further interaction with $\mathcal{A}$ is necessary to verify the file, $\mathcal{Z}$'s verification result in the real world is again always 0. The analysis of the probability of file depending on pw in the ideal world remains the same as for Theorem 3.2 since $\mathcal{Z}$ does not provide $\mathcal{S}$ with any information about pw when querying the CRS. $\square$

Remark 3.4. ***Extending Thm. 3.2 to variants of random oracles.*** *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 3.2. 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 Jutla and Roy [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].*

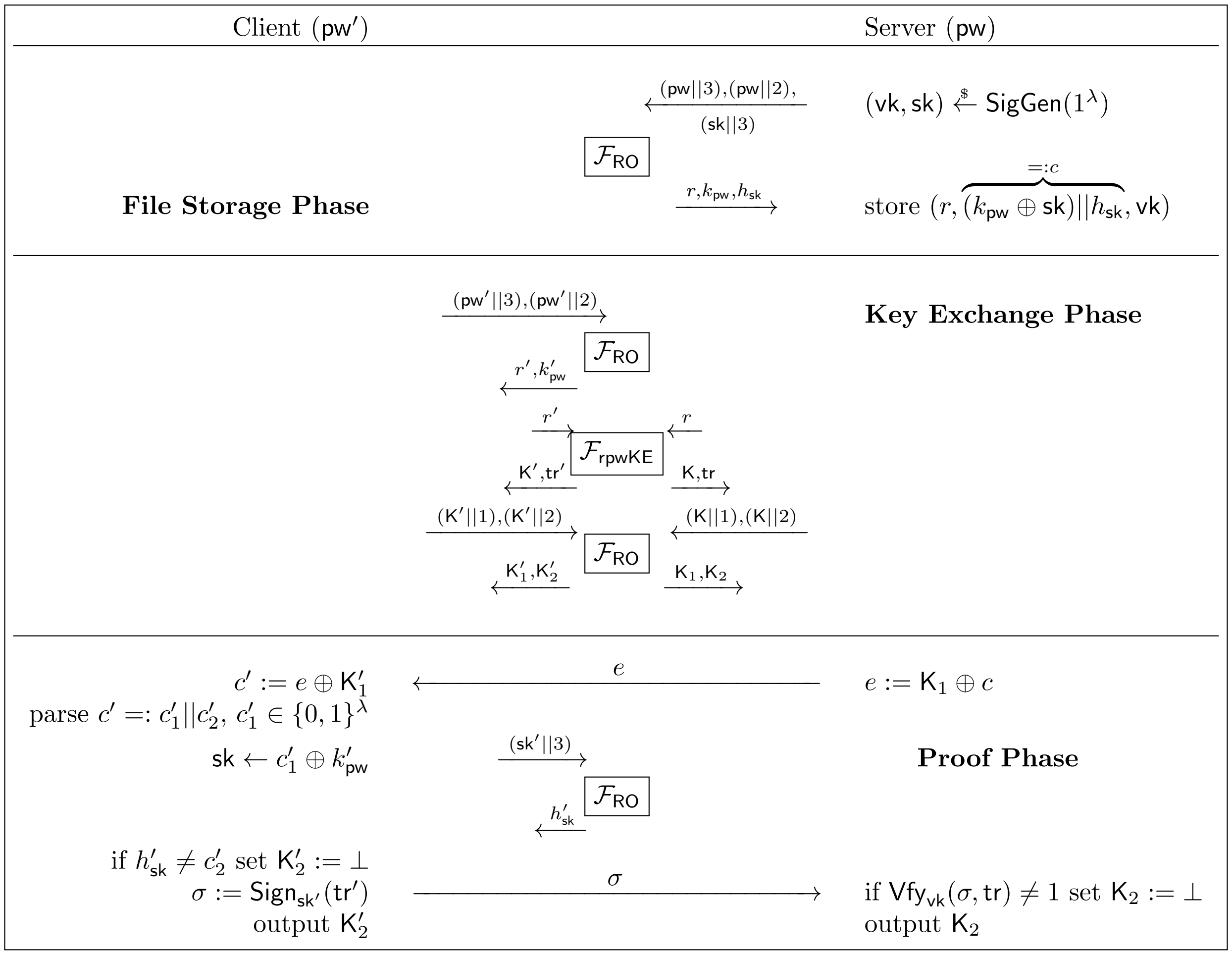
Possibility of PAKE in the NPRO-model. As opposed to asymmetric PAKE, for symmetric PAKE it is possible to achieve static UC-security without relying on programmability. A widely used PAKE protocol is DH-EKE [BM92b], which is inspired by the Diffie-Hellman Key Exchange [DH76]. The two flows of the protocol 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 and proven to be UC secure [ACCP08, DHP+18]. By looking at the 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 and proven to be UC-secure in a CRS-hybrid model [CHK+05]. 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 of Katz and Vaikuntanathan [KV11] and the protocol of Benhamouda et al. [BBC+13].

Altogether, our results in this section formally demonstrate that UC-secure asymmetric PAKE cannot be obtained from the same minimal assumptions as UC-secure PAKE.

4 UC-Security of the Ω-Method

Gentry et al. [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 symmetric 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 and describe its phases.



Client (pw')
Server (pw)
$(\mathsf{pw}||3),(\mathsf{pw}||2),(\mathsf{sk}||3)$
$(\mathsf{vk},\mathsf{sk}) \xleftarrow{\$} \mathsf{SigGen}(1^\lambda)$
$\mathcal{F}_{\mathsf{RO}}$
$r,k_{\mathsf{pw}},h_{\mathsf{sk}}$
File Storage Phase
store $(r, \overbrace{(k_{\mathsf{pw}} \oplus \mathsf{sk})||h_{\mathsf{sk}}}^{=:c}, \mathsf{vk})$
$(\mathsf{pw}'||3),(\mathsf{pw}'||2)$
Key Exchange Phase
r',k'_{pw}
r'
r
$\mathcal{F}_{\mathsf{rpwKE}}$
K',tr'
K,tr
$(\mathsf{K}'||1),(\mathsf{K}'||2)$
$(\mathsf{K}||1),(\mathsf{K}||2)$
$\mathsf{K}'_1,\mathsf{K}'_2$
$\mathsf{K}_1,\mathsf{K}_2$
$c' := e \oplus \mathsf{K}'_1$
e
$e := \mathsf{K}_1 \oplus c$
parse $c' =: c'_1||c'_2,\ c'_1 \in \{0,1\}^\lambda$
$\mathsf{sk} \leftarrow c'_1 \oplus k'_{\mathsf{pw}}$
$(\mathsf{sk}'||3)$
Proof Phase
h'_{sk}
if $h'_{\mathsf{sk}} \neq c'_2$ set $\mathsf{K}'_2 := \perp$
$\sigma := \mathsf{Sign}_{\mathsf{sk}'}(\mathsf{tr}')$
σ
if $\mathsf{Vfy}_{\mathsf{vk}}(\sigma,\mathsf{tr}) \neq 1$ set $\mathsf{K}_2 := \perp$
output K'_2
output K_2

Figure 3: The Ω-method [GMR06]. Slightly abusing notation, we assume $\mathcal{F}_{\mathsf{rpwKE}}$ outputs transcripts together with keys. To merge 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.

- **File Storage Phase:** 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 password hashes as input. To obtain more than one session key, this phase can be repeated.
- **Proof Phase:** For each key exchange phase, the client has to prove that he actually knows the password. 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. Using the (hash of the) password as an encryption key, the stored signing key is encrypted and sent to the 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-method, executed in the UC model (cf. Appendix A), with respect to our revisited functionality and corruption model, but using the NewKey and TestAbort interfaces as in $\mathcal{F}_{\mathsf{apwKE}}$ (cf. Figure 1). The full proof can be found in Appendix E.

Theorem 4.1. *The protocol depicted in Figure 5 securely realizes $\mathcal{F}_{aPAKE}$ with* NewKey *and* TestAbort *interface as in Figure 1 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 divided in four steps: simulate on-line dictionary attacks, simulate man-in-the-middle attacks, simulate server compromise attacks and transcripts 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 symmetric 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 $\mathcal{S}$ relies 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, $\mathcal{S}$ 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.

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). The asymmetric PAKE functionality $\mathcal{F}_{\mathsf{apwKE}}$ (cf. Fig. 1) features a TestAbort interface to allow analyzing protocols either with or without explicit authentication. Essentially, an adversary querying this interface for an ongoing key exchange session (1) learns whether the passwords matched or not and (2) triggers output of a failure symbol to the parties in case of mismatching passwords.

TestAbort is too weak. While it is desireable to analyze both types of schemes since both are used in practice, the TestAbort interface weakens an aPAKE functionality by not clearly distinguishing which type of protocol is analyzed. This results in very weak security guarantees regarding authentication: adversary can force a party in a protocol with explicit authentication to 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 parties would even faithfully use adversarially determined keys to, e.g., encrypt their 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 $\mathcal{F}_{\mathsf{aPAKE}}$ with TestAbort *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 fullfil any security notion. Let us explain on a high level why the Ω-method can be shown to securely realize $\mathcal{F}_{\mathsf{aPAKE}}$ with TestAbort 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, or only the file and no password at all.* 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 being unlikely.

Since it often leads to confusion, let us also shortly mention corrupted sessions. For example, in the Ω-method a corrupted client using a stolen password file has to prove knowledge of the plain password by recovering the signing key. An insecure signing scheme would let such a client authenticate to the server although he does not know the password. However, this can be simulated: upon the client sending a verifiying signature, the simulator computes the session key of the client from the simulated password file and lets the honest server output the same session key (by using the NewKey interface).

Let us emphasize. Inspired by the above simulation strategies, 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 $\mathcal{F}_{\mathsf{aPAKE}}$ with TestAbort 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 introduced by the TestAbort interface.

We correct this weakness of $\mathcal{F}_{\mathsf{apwKE}}$ by equipping $\mathcal{F}_{\mathsf{aPAKE}}$ with a NewKey interface enforcing explicit authentication, following informal recommendations of Canetti et al. [CHK+05]. Strong

EXPLICIT AUTHENTICATION GUARANTEES. For proving security of a protocol featuring explicit authentication such as the Ω-method, we show how to modify the functionality's key generation and authentication interfaces to provide strong authentication guarantees. We let the functionality send a special failure symbol $\perp$ to parties with mismatching passwords or in case of a failed online attack. The adversary gets informed about such failure. While $\mathcal{F}_{\mathsf{aPAKE}}$ would be stronger without this leakage, the Ω-method requires it. Namely, in a session where none of the parties is corrupted but the server is compromised, the verification key used by the server is leaked to the adversary via the password file. The adversary can thus learn whether key exchange succeeded by checking validity of the signature sent by the client.

An adversary can always mount a DoS attack by, e.g., injecting messages of wrong format to make the receiving party output failure. We incorporate this attack into $\mathcal{F}_{\mathsf{aPAKE}}$ by letting the adversary propose failure in his NEWKEY response. $\mathcal{F}_{\mathsf{aPAKE}}$ enforcing explicit authentication is depicted in Figure 2[2]. We are now ready to state a stronger version of Theorem 4.1, which captures security of the Ω-method more precisely. The proof can be found in Appendix F.

Theorem 4.2. *If the signature scheme is EUF-CMA secure, the protocol in Figure 5 securely realizes $\mathcal{F}_{\mathsf{aPAKE}}$ in the $\{\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{rpwKE}}\}$-hybrid model with respect to static Byzantine corruptions and adaptive server compromise.*

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 a protocol realizing $\mathcal{F}_{\mathsf{aPAKE}}$ can be leveraged to obtain a UC-secure multi-party asymmetric PAKE scheme.

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 $\mathcal{F}_{\mathsf{aPAKE}}$ protocols with their realizations, say, some protocol π_{aPAKE} in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model (cf. Figure 5). However, with each such replacement, a new set of hybrid functionalities is created that is used only by one instance of the protocol. This results 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.

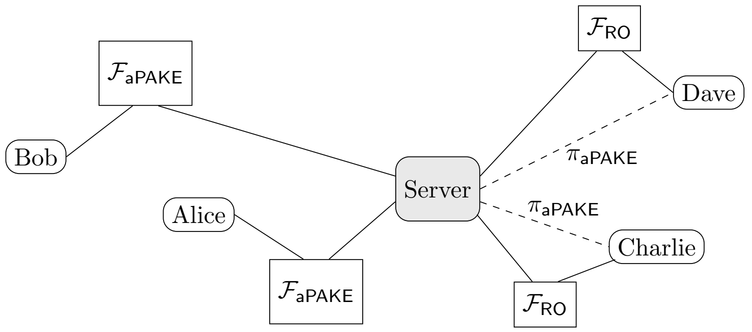

$\mathcal{F}_{\mathsf{aPAKE}}$
$\mathcal{F}_{\mathsf{RO}}$
Dave
Bob
π_{aPAKE}
Server
Alice
π_{aPAKE}
Charlie
$\mathcal{F}_{\mathsf{aPAKE}}$
$\mathcal{F}_{\mathsf{RO}}$

Figure 4: A multi-user aPAKE scenario where instances of $\mathcal{F}_{\mathsf{aPAKE}}$ are subsequently replaced by their realization π_{aPAKE} in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model.

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

[2]A protocol without explicit authentication 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 8.

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 protocol π where each invokation of $\mathcal{F}$ is replaced by a call to $\hat{\rho}$.*

Let us showcase the workflow for the Ω-method.

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

(2) Modify the Ω method from Figure 5 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-method

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 7 for a formal definition.

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

We now modify the Ω-method 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-method. It follows directly from Theorem 4.2, Lemma 5.2 and the JUC composition Theorem 5.1.

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

$$\pi^{\Omega\to\Omega_s} \textit{ UC-realizes } \pi^{\mathcal{F}_{\mathsf{aPAKE}}}$$

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][3].

[3]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 $PRF_{K_{\mathsf{sid}}}(\mathsf{pw})$.

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 $\mathcal{F}_{\mathsf{aPAKE}}$. The resulting functionality $\hat{\mathcal{F}}_{\mathsf{aPAKE}}$ 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 $\mathcal{F}_{\mathsf{aPAKE}}$ is the simplest and best option.*

Implementation considerations. 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 $H(\mathsf{pw}, \mathsf{sid})$ as password file in the Ω-method. Such a file "works" only for a specific sid, and the sid of the session $(\mathcal{P}_C, \mathcal{P}_S)$ is different from the one of $(\mathcal{P}_C, \mathcal{P}'_S)$. This has to be kept in mind: when running key exchange session from different machines, not only does a party need to remember its password, it also needs to remember its session identifier sid at a specific server. While this at first glance violates our expectations of a PAKE protocol, namely, that the user has to only remember its password to successfully exchange a key, it does not seem to be an obstacle in practice: the sid can be chosen as simple as $\mathsf{sid} = (\texttt{username}, \texttt{server-URL})$, since this combination is unique per server.

Acknowledgements

The author would like to thank Jiayu Xu, Dennis Hofheinz, David Pointcheval and Victor Shoup for helpful discussions. Discussion with Victor on how to resolve issues with session identifiers for the multi-user setting were particularly instructing.

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.

[BJX19] Tatiana Bradley, Stanislaw Jarecki, and Jiayu Xu. Strong asymmetric PAKE based on trapdoor CKEM. LNCS, pages 798–825. Springer, Heidelberg, 2019.

[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.

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

[Can20] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, revised version 2020-02-11, 2020. `https://eprint.iacr.org/2000/067`.

[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 Canetti [Can01] that are needed in our proofs. We refer to the most recent revision of the framework as [Can20]. 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 who interacts 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 modeled 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*

$$\forall \mathcal{A}\, \exists \mathcal{S} \text{ s.t. } \forall \mathcal{Z} : \quad \mathit{View}_{\pi,\mathcal{A}}(\mathcal{Z}) \overset{c}{\approx} \mathit{View}_{\mathcal{F},\mathcal{S}}(\mathcal{Z})$$

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. 5 of [Can20])**.** *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-method in the UC model. Most of Figure 5 is taken verbatim from [GMR06] with only adjustments to our notation.

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 6, the ideal functionality for symmetric PAKE, called $\mathcal{F}_{\mathsf{rpwKE}}$ in Figure 8, of Gentry et al. [GMR06].

C Introduction to UC-secure asymmetric PAKE

We now explain the functionality $\mathcal{F}_{\mathsf{apwKE}}$ [GMR06] depicted in Figure 1. $\mathcal{F}_{\mathsf{apwKE}}$ can be seen as a trusted third party that can be accessed by two users. One of the users takes the role of a client, the other takes the role of the server. To "start" the trusted party $\mathcal{F}_{\mathsf{apwKE}}$, the server inputs a password pw via the StorePwdFile interface. This password is persistent and cannot be changed later. The client can now initiate a key exchange session with the server by providing a password via the UsrSession interface. As a trusted party, $\mathcal{F}_{\mathsf{apwKE}}$ guarantees that, if both passwords match and both client and server follow the protocol, they will both obtain a uniformly random session key (cf.

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 5: Execution of the Ω-method 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:
  - $\triangleright$ If there is a pair $(m, \tilde{h})$ for some $\tilde{h} \in \{0,1\}^\lambda$ in the list L, set $h := \tilde{h}$.
  - $\triangleright$ 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 6: 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 7: Functionality $\mathcal{F}_{\mathsf{sRO}}$

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

- Send (NEWSESSION, $\mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \mathtt{role}$) 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 (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 (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 $\mathcal{P}_i$.
  - ▷ Else, 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 (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 (NEWTRANSCRIPT, $\mathsf{sid}, \mathcal{P}_{1-i}, \mathsf{tr}''$) 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 (TRANSCRIPT, $\mathsf{sid}, \mathsf{tr}$) to $\mathcal{P}_i$.

Figure 8: 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.

second case in NewKey interface). Thus, $\mathcal{F}_{\mathsf{apwKE}}$ enables a password-authenticated key exchange. Moreover, $\mathcal{F}_{\mathsf{apwKE}}$ can be used again by client and server to exchange more keys. Each such sub-session is equipped with a sub-session identifier ssid. While the client can use a different password every time (reflecting the fact that he might not fully remember his password or mistype), the server always uses the initially registered password (reflecting the fact that he sticks to the password that the user initially registered with). For this reason, the SrvSession interface which lets the server participate in a key exchange session does not require a password.

Let us now look at the guarantees $\mathcal{F}_{\mathsf{apwKE}}$ provides when passwords mismatch or parties misbehave. If client and server use different passwords in one of their key exchange sessions but behave honestly (are not *corrupted*), $\mathcal{F}_{\mathsf{apwKE}}$ provides both with randomly chosen session keys (cf. third case of NewKey interface) or even lets them report failure (cf. second case of TestAbort interface). The choice here is up to the adversary, which enables analysis of protocols with explicit authentication (where parties get informed of a failed key exchange by receiving a failure symbol) as well as protocols with implicit authentication (where a party cannot say whether the other party computed the same key). When one of the parties misbehaves (is *corrupted*), then the key that is provided to the honest party is determined by the adversary (cf. first case of NewKey). This is motivated by the fact that a misbehaving party *becomes* the adversary and might learn the session key computed by the honest party.

Distribution of passwords. By not choosing passwords of client and server itself, $\mathcal{F}_{\mathsf{apwKE}}$ intentionally does not make any assumption about their length or distribution. Indeed, $\mathcal{F}_{\mathsf{apwKE}}$ can be used with arbitrary inputs such as $\mathsf{pw} = 1$ or pw =abc123+-!. Moreover, client and server can use arbitrarily related passwords, modeling a client forgetting his password or mistyping. By allowing a distinguishing environment to observe runs of $\mathcal{F}_{\mathsf{apwKE}}$ started with all kinds of passwords, the security guarantees provided by $\mathcal{F}_{\mathsf{apwKE}}$ are preserved even when passwords come from arbitrary sources or extraneous protocol runs (a property which is usually referred to as *universal composability*, hence the name of the framework).

This concludes $\mathcal{F}_{\mathsf{apwKE}}$'s basic functionality and description of inputs and outputs of client and server. We now turn to attacks against asymmetric PAKE protocols that are unavoidable even when interacting with an ideal version of the protocol. All these attacks have to be reflected by $\mathcal{F}_{\mathsf{apwKE}}$, and we will now consider them one by one.

Guessing a password. An essential guarantee of $\mathcal{F}_{\mathsf{apwKE}}$ is that knowledge of the right password is enough to successfully exchange a session key with the other party. Since the password is the only mean of authentication, this guarantee holds with respect to interactions of honest clients and servers as well as with the adversary. The ability of the adversary to simply engage in a protocol run with an honest client or server using a guessed password (and eventually learn whether the guess was correct) is hard-coded into $\mathcal{F}_{\mathsf{apwKE}}$ via the TestPwd interface. This interface can be used only by the adversary. Per key exchange session, TestPwd allows the adversary to issue *at most one* password guess. The adversary gets informed about the outcome and, depending on whether the guess was correct or not, the honest party obtains either a random or adversarially determined session key. Limiting the interface to one usage per key exchange session ensures that the adversary cannot exclude more than one password guess per run of the key exchange protocol.

Deviating from the protocol. Misbehaving clients or servers are captured via Byzantine corruption of said party. Detailing the corruption model, i.e., what happens upon party corruption in the real and in the ideal world, is an essential component of the UC framework. Deviation from the protocol is usually called *Byzantine corruption.* Since this is a widely used corruption model and effects in both real protocol execution and ideal functionalities can be stated generically for all protocols and functionalities, $\mathcal{F}_{\mathsf{apwKE}}$ can be implicitly assumed to provide an appropriate interface for such corruption. As detailed in the literature [Can01], upon the adversary $\mathcal{S}$ corrupting a party,

$\mathcal{F}_{\mathsf{apwKE}}$ is informed about the corruption. All messages that were already sent through this party are handed to $\mathcal{S}$, and from now on $\mathcal{S}$ is put in place to receive and send messages on behalf of the corrupted party.

Stealing a password file. Besides maliciously behaving parties, a quite realistic attack scenario for asymmetric PAKE is that the server's passsword file gets leaked to the adversary. Such an attack is called a *server compromise*. Essentially, it enables the adversary to (a) extract the password of the client from the password file and (b) impersonate the server towards the honest client by using the password file in an honest protocol run. Regarding (a), a particular goal of modeling security of *asymmetric* PAKE is that the password file stored at the server should not directly reveal the password. However, the adversary holding the file can surely test whether a specific password was used to generate the file, e.g., by hashing a password guess and comparing the result against the file.

$\mathcal{F}_{\mathsf{apwKE}}$ provides the adversary with three interfaces to account for the attacks described above. Initially, to indicate that a run of the protocol is executed where the password file is stolen by the adversary, the functionality expects to receive a (StealPwdFile, sid) query. Since $\mathcal{F}_{\mathsf{apwKE}}$ does not maintain or hand out an actual password file, it now needs to provide the adversary with an interface to impersonate the server. This is the Impersonate interface, which lets the adversary engage in a protocol run with the client, resulting in exchange of an (adversarially-determined) session key in case the password used by the honest client is equal to the one used by the server. Not handing any password-dependent file to the adversary also enables $\mathcal{F}_{\mathsf{apwKE}}$ to *control* the effort of the adversary to extract the password from the file. Namely, $\mathcal{F}_{\mathsf{apwKE}}$ enforces the adversary $\mathcal{S}$ to ask one OfflineTestPwd query per password guess against the file. Note that OfflineTestPwd can even be asked before compromising the server via StealPwdFile. However, the adversary is only provided with an accumulated answer on all his precomputed guesses *upon the compromise*, modeling the fact that the adversary can precompute, e.g., hash tables and directly learn the password upon seeing the file.

D Impossibility result for $\mathcal{F}_{\mathsf{apwKE}}$

In this section we formally state and prove that the aPAKE functionality of Gentry et al. [GMR06] is inherently unrealizable due to its restriction to allow OfflineTestPwd queries only upon getting instructions from $\mathcal{Z}$. The Theorem uses Definition 3.1 of an aPAKE protocol with verifiable password file.

Theorem D.1. *Let $\mathcal{F}$ be a set of ideal functionalities. Then $\mathcal{F}_{\mathsf{apwKE}}$ as depicted in Fig. 1 is not realizable in the $\mathcal{F}$-hybrid model by any protocol that has a verifiable password files.*

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

- $\mathcal{Z}$ starts the protocol with (StorePwdFile, $\mathsf{sid}, \mathcal{P}_C, \mathsf{pw}$) as input to $\mathcal{P}_S$. All parties remain uncorrupted.
- $\mathcal{Z}$ sends (StealPwdFile, sid) to the adversary to compromise the server. $\mathcal{Z}$ 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 verifies $(\mathsf{pw}', \mathsf{file})$. $\mathcal{Z}$ outputs 0 if verification succeeds, else it outputs 1.

We first show that $\mathcal{Z}$ will output b in the real execution with overwhelming probability. 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 over the random coins of all involved entities. 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 $\mathsf{pw} \neq \mathsf{pw}'$, $\mathcal{Z}_{rand}$ will see an honest server outputting a uniformly random session key, since $\mathcal{S}$ acts on the (correct) assumption that the passwords do not match. In the real world, a verifying file would let an honest server output the same session key as computed by the corrupted client. Since $\mathcal{Z}_{rand}$ can tell the difference, this would contradict the UC-security of π and we conclude that file verifies only with negligible probability, leading to $\mathcal{Z}$ outputting 1 with overwhelming probability in case of $b = 1$. 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. □

From Theorem D.1, it becomes apparent that the simulator needs more leverage regarding offline 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 D.1 indicates.

E 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 (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 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.

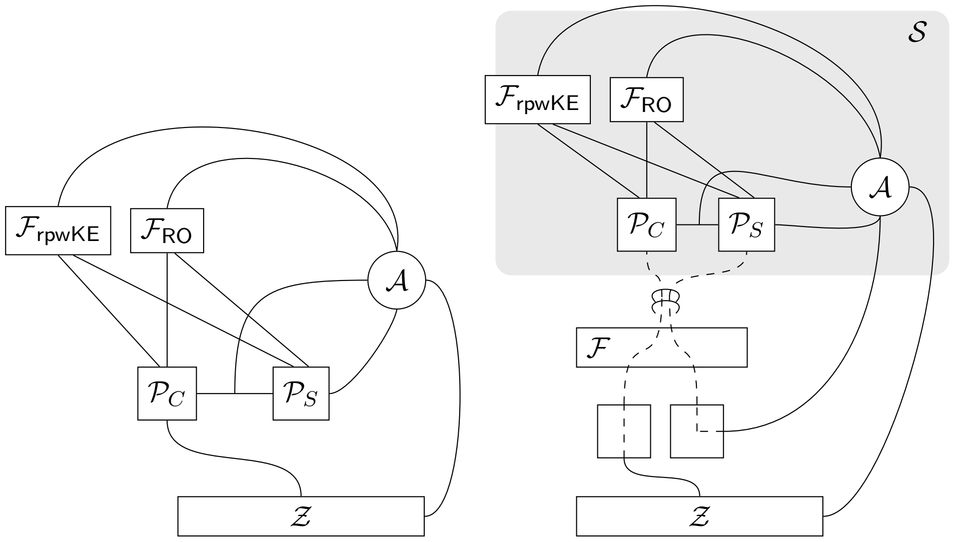


Figure 9: Transition from Game game $\mathbf{G}_0$ (left) to Game game $\mathbf{G}_1$ (right), showing a setting where $\mathcal{P}_S$ is corrupted.

Game $\mathbf{G}_2$: $\mathcal{F}$ keeps records. We now let $\mathcal{F}$ maintain records. On receiving $(\textsc{StorePwdFile}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw}$ from $\mathcal{P}_S$, if this is the first $\textsc{StorePwdFile}$ query, then $\mathcal{F}$ records $(\mathsf{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$. Upon receiving a message $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}')$ from $\mathcal{P}_C$ and this is the first $\textsc{UsrSession}$ message for ssid, $\mathcal{F}$ records $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$. Similarly, we record $\textsc{SrvSession}$ 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 $\textsc{NewKey}$ to the code of $\mathcal{F}$, namely $\textsc{StealPwdFile}$, $\textsc{OfflineTestPwd}$, $\textsc{TestPwd}$, $\textsc{Impersonate}$ and $\textsc{TestAbort}$ 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 $\textsc{NewKey}$ tag and party name to the output of parties. At the same time, we change $\mathcal{F}$ and partly add the $\textsc{NewKey}$ interface as follows:

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 `interrupted`, pick $\mathsf{K}'' \overset{\$}{\leftarrow} \{0,1\}^\lambda$ 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 $\mathcal{F}$ (we assume this query to result in $\mathcal{F}$ marking the corresponding record as `interrupted`). 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.

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 $\mathbf{G}_7$. Note that here, we assume that the signature scheme has unique signing keys.

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 $\textsc{NewKey}$ 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 $\textsc{NewKey}$ 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. 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 $\mathbf{G}_6$.

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 $\textsc{NewKey}$ 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 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 $(\text{OfflineTestPwd}, \mathsf{pw})$ 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 $(\text{StealPwdFile}, \mathsf{sid})$ 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 $\textsc{TestAbort}$ 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 $r_{\mathcal{S}} \overset{\$}{\leftarrow} \{0,1\}^\lambda, k_{\mathsf{pw}_{\mathcal{S}}} \overset{\$}{\leftarrow} \{0,1\}^{2\lambda}$ 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 $\textsc{UsrSession}$ query that $\mathcal{F}$ sends to $\mathcal{S}$, and modify $\mathcal{F}$ to not send any message to $\mathcal{S}$ anymore upon receiving a $\textsc{StorePwdFile}$ 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, in game $\mathbf{G}_{14}$ we have $\mathcal{F} = \mathcal{F}_{\mathsf{aPAKE}}$. 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.

□

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 6. $\mathcal{S}$ 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** $(\mathbf{UsrSession}, \mathsf{sid}, \mathcal{P}_C)$ **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 $r'_{\mathcal{S}} \leftarrow \{0,1\}^{\lambda}, k'_{\mathsf{pw}_{\mathcal{S}}} \overset{\$}{\leftarrow} \{0,1\}^{2\lambda}$ 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** $\{(\mathbf{SrvSession}, \mathsf{sid}, \mathsf{ssid}), (\mathbf{StealPwdFile}, \mathsf{sid})\}$**,** $\mathcal{S}$ 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 $r_{\mathcal{S}} \overset{\$}{\leftarrow} \{0,1\}^{\lambda}, k_{\mathsf{pw}_{\mathcal{S}}} \overset{\$}{\leftarrow} \{0,1\}^{2\lambda}$ 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 (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 (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 (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 $(\mathsf{pw}||3, r)$ in L. (Cf. game $\mathbf{G}_5$.)
  - ▷ If there is such an entry, $\mathcal{S}$ submits (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 $r_{\mathcal{S}} \overset{\$}{\leftarrow} \{0,1\}^{\lambda}, k_{\mathsf{pw}_{\mathcal{S}}} \overset{\$}{\leftarrow} \{0,1\}^{2\lambda}$. (Cf. games $\mathbf{G}_5$ and $\mathbf{G}_{14}$.)
  - ▷ If there is no such entry, $\mathcal{S}$ submits (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 $r_{\mathcal{S}} \overset{\$}{\leftarrow} \{0,1\}^{\lambda}, k_{\mathsf{pw}_{\mathcal{S}}} \overset{\$}{\leftarrow} \{0,1\}^{2\lambda}$. (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** ($\mathbf{TestPwd}, \langle \mathsf{sid}, \mathsf{ssid} \rangle, \mathcal{P}_i, r$) **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 $(\mathsf{pw}||3, r_{\mathcal{S}}), (\mathsf{pw}||2, k_{\mathsf{pw}_{\mathcal{S}}})$ in L. (Cf. game $\mathbf{G}_{12}$.)
- **Upon receiving** ($\mathbf{StealPwdFile}, \mathsf{sid}$) **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 $r_{\mathcal{S}}, k_{\mathsf{pw}_{\mathcal{S}}}$, $\mathcal{S}$ now stores $(\mathsf{pw}||3, r_{\mathcal{S}}), (\mathsf{pw}||2, k_{\mathsf{pw}_{\mathcal{S}}})$ in L.

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

F 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 $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{K})$ 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 `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 $\mathbf{G}_{8_\mathsf{a}}$: 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 current and last game are indistinguishable due to the EUF-CMA-security of the signature scheme.

Game $\mathbf{G}_{8_\mathsf{b}}$: man-in-the-middle against server. This game now 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 $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \perp)$

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 $\mathbf{G}_{8_\mathsf{c}}$: 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 $\mathbf{G}_{8_\mathsf{a}}$. 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 $\mathbf{G}_{8_\mathsf{a}}$ by the following instruction:

> Consider an honest session where the server might be compromised, $\underline{\mathcal{S}\text{ never received “correct guess” from }\textsc{OfflineTestPwd}}$ 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. $\square$