PAKE Combiners and Efficient Post-Quantum Instantiations

Julia Hesse*
IBM Research Europe – Zurich
juliahesse2@gmail.com

Michael Rosenberg
Cloudflare Research
michael@mrosenberg.pub

Version 1.0, October 10, 2024

Abstract

Much work has been done recently on developing password-authenticated key exchange (PAKE) mechanisms with post-quantum security. However, modern guidance recommends the use of *hybrid* schemes—schemes which rely on the combined hardness of a post-quantum assumption, e.g., Learning with Errors (LWE), and a more traditional assumption, e.g., decisional Diffie-Hellman. To date, there is no known hybrid PAKE construction, let alone a general method for achieving such.

In this paper, we present two efficient PAKE combiners—algorithms that take two PAKEs satisfying mild assumptions, and output a third PAKE with combined security properties—and prove these combiners secure in the Universal Composability (UC) model. Our sequential combiner, instantiated with efficient existing PAKEs such as CPace (built on Diffie-Hellman-type assumptions) and CHIC[ML-KEM] (built on the Module LWE assumption), yields the first known hybrid PAKE.

Keywords: key agreement, password-based cryptography, post-quantum cryptography

1 Introduction

Memorable passwords are the one of most commonly used forms of authentication today, and exist across a wide variety of applications. While some applications, such as website login, have users send their plaintext password, many large-scale applications utilize password-authenticated key exchange (PAKE) and its variants to limit the amount of information that can be intercepted. These include iCloud Keychain escrow, 1Password user authentication, Facebook Messenger chat history sharing, WhatsApp backup encryption, and passport chip access control [fan24].

*The author was supported by the Swiss National Science Foundation (SNSF) under the AMBIZIONE grant "Cryptographic Protocols for Human Authentication and the IoT".

As a result of the widespread push from industry and government to move towards post-quantum cryptography, many efficient lattice- and isogeny-based PAKEs have been developed in recent years [BCP+23, ABJS24, AEK+22, IY23]. However, these constructions are not necessarily the end goal of the post-quantum efforts. The largest rollout of a post-quantum protocol to date is that of TLS 1.3 with *hybrid* key exchange [ABBO24], i.e., key exchange which enjoys the protection of both newer post-quantum constructions and classical constructions. The purpose of the hybrid in this application is two-fold. Firstly, the newer, less tested post-quantum hardness assumptions may turn out to be easier than expected. Secondly, the newer implementations of the post-quantum algorithms may turn out to have exploitable bugs. Perhaps unintuitively, the second is considered by some as the more likely scenario [Wes24], and in fact already has occurred [BBB+24]. For these reasons, it is important to have hybrid protocols where possible. Unfortunately, no hybrid PAKE currently exists.

Non-solutions. The most common way to develop a primitive from a hybrid assumption is to simply *combine* two copies of the primitive, with each copy relying on a different assumption. However, the obvious combiners from other cryptographic primitives do not work for PAKEs. Hybrid KEMs and hybrid digital signature schemes can be constructed from the *concatenation combiner*: separately execute two copies of the primitive, and output the hash or concatenation of the outputs, respectively [BCD+24, BH23]. Let us attempt this for PAKE: let P be the PAKE given by running some PAKE P_1 and another PAKE P_2 in parallel using the same password pw and then outputting the hash $\mathsf{H}(K_1, K_2)$, where K_i is the session key computed in P_i. This straightforward combiner is only guaranteed to be as strong as the *weaker* of $\mathsf{P}_1, \mathsf{P}_2$. Intuitively, the reason is that we made the situation worse by trusting *two* PAKEs with protecting the password. For a concrete example, consider P_1 to be KC-SPAKE2 [Sho20] and P_2 to be $\mathsf{CAKE}[\mathsf{ML\text{-}KEM}]$ [BCP+23]. In the design of KC-SPAKE2, clients send a MAC tag derived from their password-dependent Diffie-Hellman shared secret. An attacker with a discrete-logarithm oracle (e.g., from a quantum computer) can break P by merely observing one transcript of the KC-SPAKE2 subprotocol, and mount an offline attack where guesses are confirmed against the MAC tag. CAKE, on the other hand, sends samples from the Module Learning with Errors (MLWE) distribution, encrypted under the password. An attacker with a decision-MLWE oracle who wishes to break P can observe one transcript of the CAKE subprotocol to mount an offline attack, because the MLWE oracle will reject with overwhelming probability when the password guess is wrong. Hence, *both* assumptions have to hold in order for the concatenation combiner to protect the password.

Another potential method to develop a hybrid PAKE is to use an existing KEM-based PAKE construction, e.g., CAKE, and instantiate it with a hybrid (concatenated) KEM. Similar to the above, this does not necessarily have the combined security of the two underlying KEMs $\mathsf{KEM}_1, \mathsf{KEM}_2$. The key detail is that CAKE encrypts the KEM public key (a concatenation of KEM_1 and KEM_2 public keys) under pw. Thus, if either of the underlying public keys

has structure that makes it distinguishable from uniform, then trial decryption on the CAKE encrypted public key yields an offline attack. Concretely, take KEM_1 to be X25519 and KEM_2 to be ML-KEM (this concatenation KEM essentially the X-Wing construction [BCD+24]). While X25519 public keys are statistically close to uniform in the space $\{0,1\}^{32}$, ML-KEM public keys are only *computationally* indistinguishable from uniform under the MLWE assumption. If a passive attacker possesses an MLWE oracle, they may trial-decrypt the encrypted concatenated public key, ignore the X25519 public key, and check if the decrypted value is an MLWE sample. Thus, CAKE with X-Wing is at most as secure as MLWE. A similar argument shows that the same is true for other recent KEM-based PAKEs such as OCAKE [BCP+23] and CHIC [ABJS24].

Our contributions. In this paper we give a generalizable method for constructing hybrid PAKEs and demonstrate efficient instantiations. We present two PAKE combiners: ParComb, a parallel combiner, executing both PAKEs in parallel and hashing the results; and SeqComb, a sequential combiner, executing the first PAKE and feeding the resulting session key into the second PAKE. We prove the combiners' security in the universal composability (UC) model with random oracles and static corruptions.

For our ParComb combiner, which follows the blueprint of the usual concatenation combiner, we need to circumvent the above described insecurities. To this end, we formalize a property of a PAKE that is sufficient to prove ParComb secure. Intuitively, we demand that the password protection of both PAKEs used in ParComb does not fatally break if any of the underlying computational assumptions break, i.e., their transcripts information-theoretically hide the password. This ensures that none of the PAKEs leak information about the input password. We identify a handful of PAKE protocols from the literature that satisfy this unconditional hiding property. Unfortunately, none of these are post-quantum PAKEs. So ParComb, while having zero round complexity overhead over its building blocks, does not yet yield hybrid PAKE.

This is why we have to look into sequential combiner designs as well. Our combiner SeqComb derives a shared key from a password through the first PAKE, and then uses this shared key as a password in the second PAKE to derive another shared key. The final key is, again, a (hashed) concatenation of both PAKE keys. As in ParComb, the first PAKE consumes the plain password and hence must enjoy the same unconditional hiding property as both PAKEs in ParComb. For the second PAKE, however, we can get away with a weaker hiding property. This is because we used the first PAKE as an entropy amplifier to derive unguessable preshared keys from the passwords, and hence we do not rely on offline guess protection from the second PAKE to protect these keys. We formalize a mild hiding property, namely that the second PAKE's transcript does not reveal whether the first PAKE run was successful, that is sufficient to prove the security of SeqComb. We identify KEM-based PAKEs with this mild hiding property, namely CHIC [ABJS24], CAKE, and OCAKE [BCP+23]. Thus, when instantiated with a post-quantum KEM such as ML-KEM or Saber [DKRV18], we get a hybrid PAKE with a round complexity that is the sum of the underlying PAKEs',

and computational overhead of only two hashes. To the authors' knowledge, this is the first description of a hybrid PAKE.

Finally, we remark that hybrid PAKE implies hybrid versions of other flavors of PAKE. The Ω-method augmented PAKE (aPAKE) [GMR06] takes a generic PAKE and signature scheme. Similarly, the LATKE identity-binding PAKE (iPAKE) [KR24] takes a generic PAKE and identity-based key exchange (IBKE) protocol. Since hybrid signatures [BH23] and IBKEs [KR24] are known, hybrid PAKE implies hybrid aPAKE and iPAKE.

2 Preliminaries

2.1 Notation

We write probabilistic algorithms as $\mathsf{Alg}(x; r)$ where x denotes the input and r denotes the random coins. We write $y \leftarrow \mathsf{Alg}(x)$ to denote sampling uniform r and setting $y := \mathsf{Alg}(x; r)$, and $x \leftarrow\!\!\$\ S$ to denote sampling a uniform value from a set S. For our security proofs we will consider probabilistic polynomial time (PPT) adversaries, and denote them with calligraphic letters $\mathcal{A}$. We use λ to denote the security parameter.

In our pseudocode for ideal functionalities, **assert** checks the given condition, and early-returns with "assertion failed" on failure. **retrieve** denotes retrieval of a record with a specific marking; if no such record is present, this early-returns with "no record." We mark new variables introduced into the scope by a **retrieve** using square brackets. We write $\boxed{\mathsf{P}^{\mathsf{sid}}}$ to denote execution of a PAKE protocol P with session identifier sid.

2.2 Universal composability

The Universal Composability (UC) model [Can01] provides an alternative to game-based security definitions. The model frames cryptographic protocols as idealized *functionalities*, which can be thought of as black boxes with a tightly constrained interface to the outside world. In the security game showing Π *UC-realizes* the functionality $\mathcal{F}$, the goal of an interactive Turing machine called the *environment* $\mathcal{Z}$ is to distinguish between the *ideal world* and the *real world*, which are defined as follows.

In the *real world* all the parties participate in Π, $\mathcal{Z}$ may view parties' outputs, and is permitted to give arbitrary instructions to a separate interactive Turing machine, called the *adversary* $\mathcal{A}$. The environment can arbitrarily ask the adversary to view/modify/delay/drop messages between parties, corrupt parties, and interact with any ideal functionalities $\mathcal{F}_i'$ used to instantiate Π. In the *ideal world* all the parties are *dummies*, speaking directly to $\mathcal{F}$. In addition, the adversary may also only interact with $\mathcal{F}$, though it is still permitted to corrupt parties.

Π *UC-realizes* $\mathcal{F}$ if for any $\mathcal{A}$, there exists a simulator $\mathcal{S}$ of $\mathcal{A}$ such that any

$\mathcal{Z}$ has at most negligible advantage in distinguishing between $\mathcal{A}$ and $\mathcal{S}$. That is,

$$\textsc{Ideal}^{\mathcal{F}}_{\mathcal{S},\mathcal{Z}} \overset{c}{\approx} \textsc{Exec}_{\mathcal{A},\mathcal{Z}}$$

where the LHS refers to the probability ensemble consisting of the view of the environment in the ideal world, and the RHS in the real world.

By [Can01, Theorem 11], it suffices to imagine that the adversary $\mathcal{A}$ is the *dummy adversary* $\mathcal{A}_D$, which simply delivers backdoor messages generated by the environment to the specified recipients, and delivers to the environment all backdoor messages generated by the protocol parties, as well as the sender machine's identity.

There are two models in which the environment can corrupt parties. In the *static corruption model*, the adversary may not corrupt any parties during protocol execution. In the *adaptive corruption model*, the adversary may corrupt parties at any time. In this work, we will consider adversaries who are allowed static corruptions. We note that, since PAKE sessions are so short lived, and static corruptions permit corruptions in between executions, this security model is still quite strong.

To represent different instances of the same scheme, the UC model uses *session identifiers*. Each party is given a session identifier sid on activation and will only interact with other parties with the same sid. For clarity of presentation, we will assume in our protocol definitions that session identifier establishment has already occurred.

2.3 PAKE

A *(balanced) password authenticated key-exchange protocol* (PAKE) is a two-party key-exchange protocol where parties use mutual knowledge of a low-entropy password pw to establish a high-entropy *session key* K.

2.3.1 PAKE protocols

We define a rudimentary notion of a PAKE protocol by specifying its intput-output behavior and correctness. Looking ahead, the following definition captures the minimal guarantees that we can expect from a single PAKE building block of our combiners, even when its security breaks.

Definition 1 (PAKE protocol)**.** *Let* P *be an interactive protocol between two parties,* $\mathcal{P}$ *and* $\mathcal{P}'$*. We say* P *is a* PAKE protocol *if it satisfies the following properties.*

1. P *takes input* pw *from* $\mathcal{P}$ *and* pw' *from* $\mathcal{P}'$
2. P *produces output* K *for* $\mathcal{P}$ *and* K' *for* $\mathcal{P}'$
3. *If* $\mathsf{pw} = \mathsf{pw}'$ *then* $K = K'$

For any interactive protocol, a *protocol round* is the set of all messages that can be sent in parallel from any point in the protocol [Gon93]. Thus, in a one-round protocol, no message depends on another. We use syntax $\mathsf{P} = (\mathsf{Start}, \mathsf{Finish})$ for one-round PAKEs, defined as follows:

- Start takes as input a password pw and outputs a message x and a state st, and
- Finish takes as input st and a message y and outputs a key.

2.3.2 Ideal PAKE functionality

The security goals for a PAKE are (1) to establish a high-entropy shared session key when both parties are honest, (2) to prevent passive adversaries from learning anything about the password, and (3) to limit active adversaries to one (or another small constant) password guess(es) per protocol instance, even if given access to session keys. These guarantees can be captured by the ideal functionality $\mathcal{F}_{\mathsf{PAKE}}$ [CHK+05]. Some widely used protocols use a slight weakening of this functionality, called *lazy extraction PAKE* ($\mathcal{F}_{\mathsf{lePAKE}}$) [ABB+20], which permits adversaries complete active attacks even after a session is already finished. In this work we also make use of a novel functionality, a *relaxed* version of lazy extraction PAKE ($\mathcal{F}_{\mathsf{rlePAKE}}$), where the adversary additionally learns whether the active attack was successful. We state all functionalities in Figure 1, and include the small modifications from [AHH21] regarding adversary-controlled keys.

Isolated parties. We introduce another small modification to our PAKE ideal functionality. In the original definition, it is possible for a party to output a session key before the counterparty has even started its session. In our definition, we prevent these *isolated parties* from getting their session key. This modification is crucial to using $\mathcal{F}_{\mathsf{PAKE}}$ in a combiner, since we rely on timely extraction from a particular sub-PAKE, and cannot deal with output keys that were produced without any interaction.

This new functionality is slightly stronger than the original, but this is not a problem: any PAKE which UC-realizes the original $\mathcal{F}_{\mathsf{PAKE}}$ UC-realizes the new one as well. Consider a UC simulator for a such a PAKE. We will show that NewKey is never called on an isolated party. First, note if the simulator calls NewKey on an isolated party, it is because the environment $\mathcal{Z}$ has triggered the end to the protocol (this is because NewKey gives $\mathcal{Z}$ the session key, so $\mathcal{Z}$ knows precisely when NewKey is called). Further, if $\mathsf{NewSession}$ has not been called for the counterparty, it is because $\mathcal{Z}$ has not activated that party. So the only case in which NewKey can be called on an isolated party is if only that party was activated and the protocol has finished, i.e., the protocol is at most a one-message PAKE. Since such PAKEs are impossible in the UC model,[1] this scenario never occurs.

[1] More specifically, PAKEs with negligible correctness error and at most one message are impossible. An efficient offline attack is as follows. Let $\mathcal{P}$ be the initiator in the execution of a

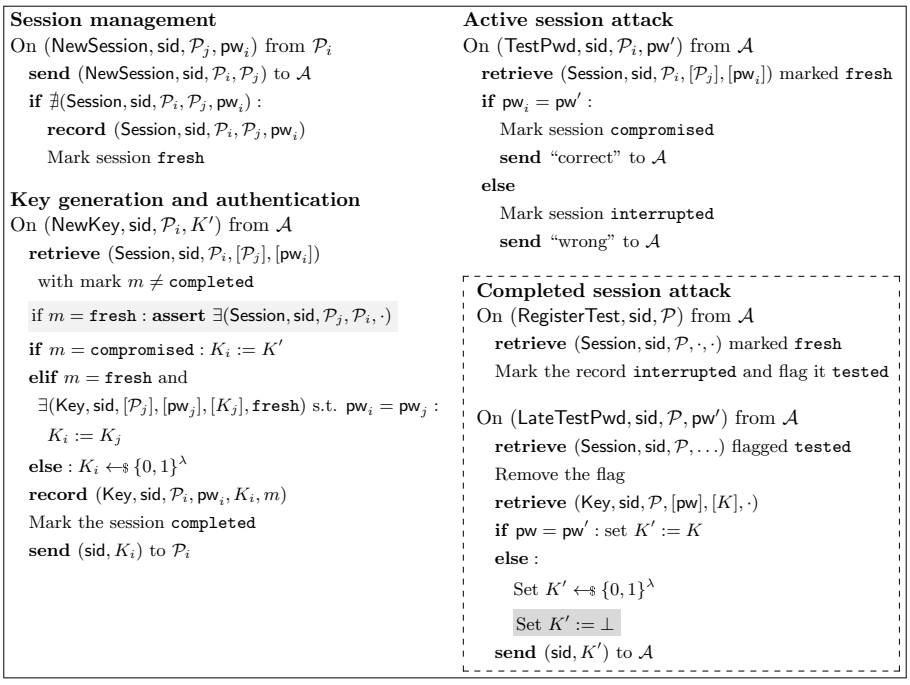


Figure 1: The $\mathcal{F}_{\mathsf{PAKE}}$, $\mathcal{F}_{\mathsf{lePAKE}}$, and $\mathcal{F}_{\mathsf{rlePAKE}}$ ideal functionalities. The original $\mathcal{F}_{\mathsf{PAKE}}$ functionality excludes all gray and dashed parts. Adding the dashed interfaces without the line in dark gray yields the lazy password extraction functionality $\mathcal{F}_{\mathsf{lePAKE}}$. Adding the line in dark gray yields the relaxed lazy extraction functionality $\mathcal{F}_{\mathsf{rlePAKE}}$. In this paper, we include the line in light gray in *all* functionalities, to prevent isolated parties from outputting a key.

2.3.3 Hiding properties of PAKEs

For our combiner constructions, we will require two specific security properties from the underlying PAKEs.

Perfect password hiding. The first property we require is that of *perfect password hiding*—that PAKE execution transcripts perfectly hide the password (as opposed to relying on a computational indistinguishability assumption). Crucially, we demand this property from PAKE protocols *even in situations where its security breaks*, e.g., through the discovery of an algorithm that solves the underlying computational assumption in polynomial time. Hence, we define a standalone property rather than modifying the ideal PAKE functionality. We restrict the definition to one-round PAKEs for simplicity and because the popular

one-message PAKE, sending message m to $\mathcal{P}'$. If $\mathcal{Z}$ permits $\mathcal{P}$ to send m, then $\mathcal{P}'$ immediately outputs its session key K. This K is the output of a function $f(\mathsf{pw}, \mathsf{sid}, m)$ (w.l.o.g., this is deterministic, otherwise we have noticeable correctness error). Since $\mathcal{Z}$ knows sid and m, it can test various pw until it receives an output that matches K.

PAKEs satisfying this property are all one-round anyway.[2]

We formally define perfect password hiding through the existence of a simulator that produces identically distributed protocol messages without access to the password, but is still able to compute keys from simulated messages and passwords, using a simulation trapdoor. We note that the existence of a simulator for producing *computationally* indistinguishable transcripts follows from the fact that a PAKE protocol UC-realizes $\mathcal{F}_{\mathsf{PAKE}}$ or $\mathcal{F}_{\mathsf{lePAKE}}$. Essentially, perfect password hiding demands that this part of the PAKE never breaks.

Definition 2 (Perfectly password-hiding one-round PAKE)**.** *Let* $\mathsf{P} = (\mathsf{Start}, \mathsf{Finish})$ *be a one-round PAKE protocol. We call* P perfectly password hiding *iff there exist PPT algorithms* $\mathsf{Sim} = (\mathsf{Start}, \mathsf{ComputeKey})$ *such that:*

- $\mathsf{Sim.Start}$ *is a probabilistic algorithm that takes as input a security parameter* 1^λ *and outputs a tuple* (x, τ)
- $\mathsf{Sim.ComputeKey}$ *is a deterministic algorithm that takes as input* $(x, y, \tau_y, \mathsf{pw})$ *and outputs a key*
- Perfect hiding. *For any* pw*,* $\mathsf{P.Start}(\mathsf{pw})$ *is identically distributed, where* $|_1$ *denotes the first component of the output. In other words, every message in the honest protocol is equally explained by any password.*
- Trapdoor key computation. *For any* pw*,* $(x, \mathsf{st}) := \mathsf{P.Start}(\mathsf{pw})$*, and* $(y, \tau_y) := \mathsf{Sim.Start}(1^\lambda)$*, it holds that* $\mathsf{Sim.ComputeKey}(x, y, \tau_y, \mathsf{pw}) = \mathsf{P.Finish}(\mathsf{st}, y)$.

EKE, CPace, and SPAKE2 (Figure 2) are perfectly hiding one-round PAKEs. This property follows immediately from the fact that their messages (Diffie-Hellman key shares) are uniformly distributed for any fixed password, i.e., any password equally explains any message.

PSK equality hiding. The second property we require is that of *perfect PSK equality hiding*—that a PAKE, when executed by two parties using high-entropy passwords (a.k.a., a preshared key, or PSK), does not leak whether the passwords match. As before, we require this property even when the PAKE is otherwise insecure. Note that this is a strictly weaker property than perfect password hiding. We provide a proper definition below. Unlike with password hiding, we define this property to include multi-round PAKEs.

Definition 3 (PSK Equality Hiding)**.** *Let* P *be a PAKE protocol. We say that* P *is* PSK equality hiding *iff an active adversary observing two honest parties interacting cannot distinguish between the case where the parties share a high-entropy password and the case where they have distinct high-entropy passwords.*

Formally, we define the advantage of an adversary $\mathcal{A}$ *as*

$$\mathsf{Adv}^{\mathsf{PEH}}_{\mathsf{P}}(\mathcal{A}) = \left|2 \cdot \Pr[\mathsf{G}^{\mathsf{PEH}}_{b,\mathsf{P}}(\mathcal{A}) = b \mid b \leftarrow\!\$\, \{0,1\}] - 1\right|,$$

[2] In fact, we do not know of a PAKE with this property that is not one round. The failure modes were discussed briefly in the introduction, using KC-SPAKE2 and CAKE as examples.

Alice$(\mathsf{sid}, \mathsf{pw})$		**Bob**$(\mathsf{sid}, \mathsf{pw})$
$G := \mathsf{H}_0(\mathsf{sid}, \mathsf{pw})$		$G := \mathsf{H}_0(\mathsf{sid}, \mathsf{pw})$
$r \leftarrow\!\!\$\ \mathbb{F}$		$s \leftarrow\!\!\$\ \mathbb{F}$
$R := rG$	$\xrightarrow{R}$	$S := sG$
	$\xleftarrow{S}$	
$Z := rS$		$Z := sR$
$K := \mathsf{H}_1(\mathsf{sid}, R, S, Z)$		$K := \mathsf{H}_1(\mathsf{sid}, R, S, Z)$
return K		**return** K

(a) The CPace PAKE [HL19]. The random oracle H_0 outputs values in a prime-order group.

Alice$(\mathsf{sid}, \mathsf{pw})$		**Bob**$(\mathsf{sid}, \mathsf{pw})$
$r \leftarrow\!\!\$\ \mathbb{F}$		$s \leftarrow\!\!\$\ \mathbb{F}$
$R := \mathsf{E}^{\mathsf{sid}}_{\mathsf{pw}}(rG)$	$\xrightarrow{R}$	$S := \mathsf{E}^{\mathsf{sid}}_{\mathsf{pw}}(sG)$
	$\xleftarrow{S}$	
$Z := r\mathsf{D}^{\mathsf{sid}}_{\mathsf{pw}}(S)$		$Z := s\mathsf{D}^{\mathsf{sid}}_{\mathsf{pw}}(R)$
$K := \mathsf{H}_1(\mathsf{sid}, R, S, Z)$		$K := \mathsf{H}_1(\mathsf{sid}, R, S, Z)$
return K		**return** K

(b) The EKE PAKE [BM92]. G is a group generator, and (E, D) are the encryption and decryption algorithms of an ideal cipher.

Alice$(\mathsf{sid}, \mathsf{pw})$		**Bob**$(\mathsf{sid}, \mathsf{pw})$
$r \leftarrow\!\!\$\ \mathbb{F}$		$s \leftarrow\!\!\$\ \mathbb{F}$
$R := rG + \mathsf{pw} \cdot M$	$\xrightarrow{R}$	$S := sG + \mathsf{pw} \cdot N$
	$\xleftarrow{S}$	
$Z := r \cdot (S - \mathsf{pw} \cdot N)$		$Z := s \cdot (R - \mathsf{pw} \cdot M)$
$K := \mathsf{H}_1(\mathsf{sid}, \mathsf{pw}, R, S, Z)$		$K := \mathsf{H}_1(\mathsf{sid}, \mathsf{pw}, R, S, Z)$
return K		**return** K

(c) The SPAKE2 PAKE [AP05]. G, M, and N are unrelated generators of a prime-order group. pw is interpreted as an integer (perhaps via hashing).

Figure 2: Three perfectly password-hiding PAKEs

where $\mathsf{G}^{\mathsf{PEH}}$ *is the PSK equality hiding game defined in Figure 3. Note we permit the adversary to read and modify messages, but not see the final protocol output.*

We say that P *is* statistically PSK equality hiding *if* $\mathsf{Adv}^{\mathsf{PEH}}_{\mathsf{P}}(\mathcal{A}) = \mathsf{negl}(\lambda)$ *for any unbounded* $\mathcal{A}$*.*

As with password hiding, the computational version of PSK equality hiding is guaranteed by the fact that the PAKE UC-realizes one of the PAKE functionalities, but the statistical version is not.[3]

We claim that recent efficient post-quantum universally composable PAKEs (Figure 4) achieve this property. At a high level, all these are statistically PSK equality hiding because at least one of the sides' message is encrypted using the password as a key to an ideal cipher. Since the ideal cipher is a pseudorandom

[3]In fact, this is a strict separation. Take any secure PAKE and add $(\mathsf{pk}, \mathsf{Enc}_{\mathsf{pk}}(\mathsf{pw}))$ to both Alice's and Bob's first messages, where pk is a public key in some asymmetric encryption scheme. Then it is clear that this new PAKE is secure, but at best has PSK equality hiding that depends on the computational hardness of decrypting a ciphertext in the asymmetric encryption scheme. This sort of construction occurs in existing PAKEs. The one-round PAKE from smooth projective hash functions presented in [KV11] is neither perfectly password hiding nor statistically PSK equality hiding for precisely this reason.

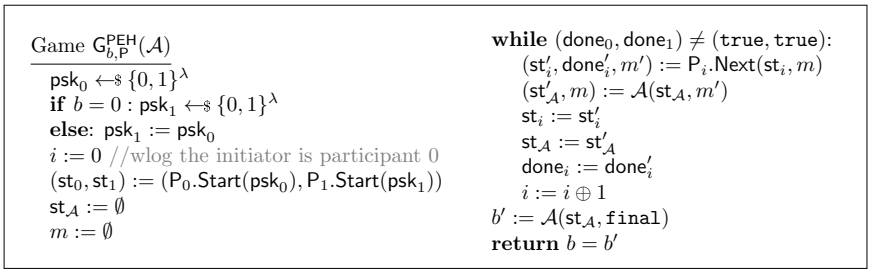


Figure 3: The PSK equality hiding (PEH) security game for the PAKE P.

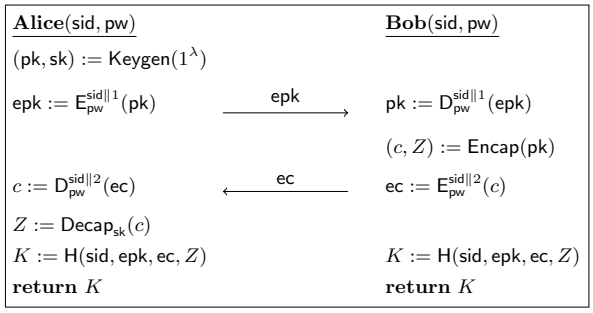
permutation, and the key (here, pw) is high-entropy, the value reveals nothing to the adversary. We provide more detailed proofs below.

OCAKE (Figure 4b). Let game 0 be the PEH game with $b = 0$, i.e., the passwords are different. In game 1, rather than using $\mathsf{pk} := \mathsf{D}^{\mathsf{sid}}_{\mathsf{pw}_B}(\mathsf{epk})$, Bob generates a fresh KEM public key pk and encapsulates to that. Since the ideal cipher is a family of uniformly chosen permutations, advantage of a distinguisher of this hop is bounded by the ability to guess pw_B, which is $2^{-\lambda}$ times the number of ideal cipher queries. Note this holds even when the distinguisher picks epk. In game 2, rather than sending $\mathsf{epk} := \mathsf{E}^{\mathsf{sid}}_{\mathsf{pw}_{\mathcal{A}}}(\mathsf{pk})$, Alice simply sends a uniformly chosen value in the public key space. This is perfectly indistinguishable, again, since the ideal cipher is a family of uniformly random permutations. In game 3, set Alice's password equal to Bob's. This changes nothing. Let games 4 and 5 undo games 2 and 1, respectively, and we are done.

CAKE (Figure 4a). This argument is identical to that of OCAKE. This is possible because the second ideal cipher encryption is made with respect to a distinct and unrelated ideal cipher instance $\mathsf{E}^{\mathsf{sid}\|2}$.

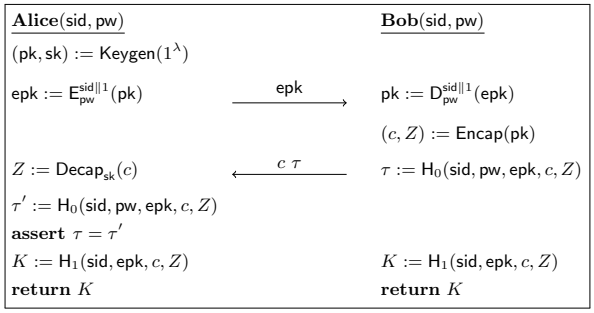
Note epk is uniform because Alice's password is uniform. And ec is uniform an independent of epk because Bob's password is uniform and $\mathsf{E}^{\mathsf{sid}\|2}$ is unrelated to $\mathsf{E}^{\mathsf{sid}\|1}$.

CHIC (Figure 4c). This also bears resemblance to OCAKE, so we will reuse parts of the proof. Let game 0 be the PEH game with $b = 0$, i.e., the passwords are different. In game 1, rather than using $r := \mathsf{D}^{\mathsf{sid}}_t(\mathsf{er})$, Bob uses a uniformly random r. Since the space of split KEM messages is indistinguishable from uniform via its $\mathsf{UNI\text{-}PK}$ property, and because ideal cipher is a family of uniformly chosen permutations, advantage of a distinguisher of this hop is bounded by the ability to guess $t = \mathsf{H}_1(\mathsf{sid}, \mathsf{pw}_B, T)$, which is bounded by the entropy of pw_B. Note this holds even when the distinguisher picks er. In game 2, rather than using $t := \mathsf{H}_1(\mathsf{sid}, \mathsf{pw}, T)$, Bob uses a uniformly random t. The probability of the adversary distinguishing this hop is again bounded by its ability to compute t. In game 3, rather than sending $\mathsf{er} := \mathsf{E}^{\mathsf{sid}}_t(r)$, Alice sends a uniform er. This hop is again indistinguishable via the ideal cipher and its high entropy key. In game 4, rather than using $R := \mathsf{H}_0(\mathsf{sid}, \mathsf{pw}, r)$, Alice uses a uniform R. This is indistinguishable by password entropy. We are now in the scenario where er



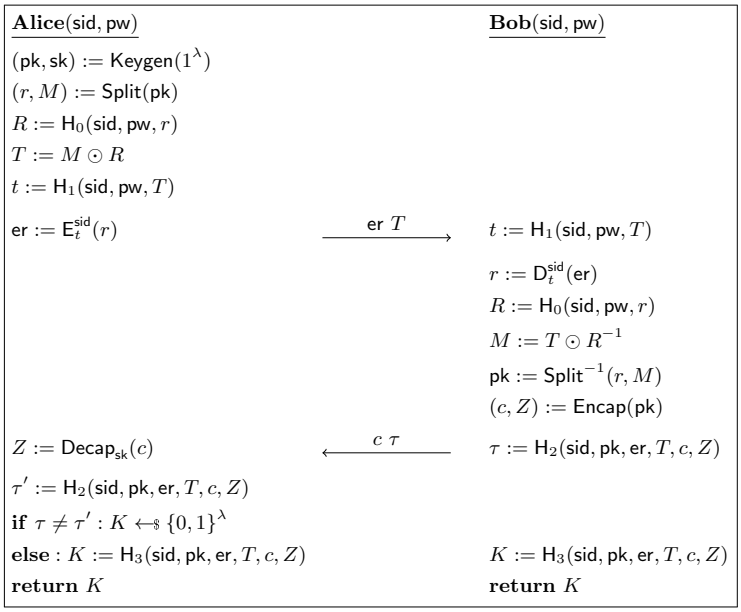
Alice(sid, pw)		**Bob**(sid, pw)
$(\mathsf{pk}, \mathsf{sk}) := \mathsf{Keygen}(1^\lambda)$		
$\mathsf{epk} := \mathsf{E}_{\mathsf{pw}}^{\mathsf{sid}\|\|1}(\mathsf{pk})$	$\xrightarrow{\mathsf{epk}}$	$\mathsf{pk} := \mathsf{D}_{\mathsf{pw}}^{\mathsf{sid}\|\|1}(\mathsf{epk})$
		$(c, Z) := \mathsf{Encap}(\mathsf{pk})$
$c := \mathsf{D}_{\mathsf{pw}}^{\mathsf{sid}\|\|2}(\mathsf{ec})$	$\xleftarrow{\mathsf{ec}}$	$\mathsf{ec} := \mathsf{E}_{\mathsf{pw}}^{\mathsf{sid}\|\|2}(c)$
$Z := \mathsf{Decap}_{\mathsf{sk}}(c)$		
$K := \mathsf{H}(\mathsf{sid}, \mathsf{epk}, \mathsf{ec}, Z)$		$K := \mathsf{H}(\mathsf{sid}, \mathsf{epk}, \mathsf{ec}, Z)$
return K		**return** K

(a) The CAKE PAKE [BCP+23]. (E, D) are the encryption and decryption algorithms of an ideal cipher.



Alice(sid, pw)		**Bob**(sid, pw)
$(\mathsf{pk}, \mathsf{sk}) := \mathsf{Keygen}(1^\lambda)$		
$\mathsf{epk} := \mathsf{E}_{\mathsf{pw}}^{\mathsf{sid}\|\|1}(\mathsf{pk})$	$\xrightarrow{\mathsf{epk}}$	$\mathsf{pk} := \mathsf{D}_{\mathsf{pw}}^{\mathsf{sid}\|\|1}(\mathsf{epk})$
		$(c, Z) := \mathsf{Encap}(\mathsf{pk})$
$Z := \mathsf{Decap}_{\mathsf{sk}}(c)$	$\xleftarrow{c\ \tau}$	$\tau := \mathsf{H}_0(\mathsf{sid}, \mathsf{pw}, \mathsf{epk}, c, Z)$
$\tau' := \mathsf{H}_0(\mathsf{sid}, \mathsf{pw}, \mathsf{epk}, c, Z)$		
assert $\tau = \tau'$		
$K := \mathsf{H}_1(\mathsf{sid}, \mathsf{epk}, c, Z)$		$K := \mathsf{H}_1(\mathsf{sid}, \mathsf{epk}, c, Z)$
return K		**return** K

(b) The OCAKE PAKE [BCP+23]



Alice(sid, pw)		**Bob**(sid, pw)
$(\mathsf{pk}, \mathsf{sk}) := \mathsf{Keygen}(1^\lambda)$		
$(r, M) := \mathsf{Split}(\mathsf{pk})$		
$R := \mathsf{H}_0(\mathsf{sid}, \mathsf{pw}, r)$		
$T := M \odot R$		
$t := \mathsf{H}_1(\mathsf{sid}, \mathsf{pw}, T)$		
$\mathsf{er} := \mathsf{E}_t^{\mathsf{sid}}(r)$	$\xrightarrow{\mathsf{er}\ T}$	$t := \mathsf{H}_1(\mathsf{sid}, \mathsf{pw}, T)$
		$r := \mathsf{D}_t^{\mathsf{sid}}(\mathsf{er})$
		$R := \mathsf{H}_0(\mathsf{sid}, \mathsf{pw}, r)$
		$M := T \odot R^{-1}$
		$\mathsf{pk} := \mathsf{Split}^{-1}(r, M)$
		$(c, Z) := \mathsf{Encap}(\mathsf{pk})$
$Z := \mathsf{Decap}_{\mathsf{sk}}(c)$	$\xleftarrow{c\ \tau}$	$\tau := \mathsf{H}_2(\mathsf{sid}, \mathsf{pk}, \mathsf{er}, T, c, Z)$
$\tau' := \mathsf{H}_2(\mathsf{sid}, \mathsf{pk}, \mathsf{er}, T, c, Z)$		
if $\tau \neq \tau' : K \leftarrow\!\!\$\ \{0,1\}^\lambda$		
else : $K := \mathsf{H}_3(\mathsf{sid}, \mathsf{pk}, \mathsf{er}, T, c, Z)$		$K := \mathsf{H}_3(\mathsf{sid}, \mathsf{pk}, \mathsf{er}, T, c, Z)$
return K		**return** K

(c) The CHIC PAKE [ABJS24]. The KEM used has *splittable* public keys, i.e., public keys have one component which is uniformly random and can be hashed to the space of the other component (in the case of ML-KEM, this is the public matrix seed ρ). The $\odot$ operator is a group operation on the space of (the latter component of) public keys.

Figure 4: Three post-quantum PAKEs with statistical PSK equality hiding. Each uses a generic KEM $(\mathsf{Keygen}, \mathsf{Encap}, \mathsf{Decap})$ satisfying certain properties.

and T are both uniform. In game 5, set Alice's password equal to Bob's. This changes nothing. In the following games, we undo games 1–4. The final game is the PEH game with $b = 1$.

3 Constructions

In this section we present our two PAKE combiners, ParComb (*parallel combiner*) and SeqComb (*sequential combiner*). Unlike the failed attempt at a combiner in our introduction, these combiners will require specific hiding properties of their underlying PAKEs. With these mild assumptions, we show that both combiners achieve security in the UC model.

3.1 The ParComb combiner

We return to the insecure parallel combiner of our introduction. Recall the reason it fails is, if the underlying security assumption were broken, the transcripts of the underlying PAKEs would contain enough auxiliary information to check password guesses. For KC-SPAKE2, this auxiliary information is a MAC whose key is derived from the Diffie-Hellman shared secret, and for CAKE, this is the fact that KEM ciphertexts are MLWE samples.

However, not every PAKE has this auxiliary information. In fact, the most widely deployed PAKEs, such as CPace or SPAKE2, do not. The lack of such auxiliary information is what we formalize in Definition 2 (perfect password hiding). As it turns out, this property is necessary and sufficient in order to make our parallel combiner work. At a high level, ParComb executes two perfectly password-hiding one-round PAKEs P_1 and P_2 in parallel, and hashes the resulting keys. This combiner is highly efficient, requiring only one extra hash, and has round complexity bounded by the maximum round complexity of the underlying PAKEs. We give the full construction in Figure 5.

3.1.1 Security

Before presenting the security theorem, we provide intuition for each assumption and design decision.

Perfect password hiding. We know from our broken combiner example why perfect (or at least statistical) password hiding is necessary for P_1 and P_2. We provide intuition now for why it is sufficient. Consider what happens when P_1 is broken and P_2 remains secure. An active adversary can engage in P_1 with Bob and be able to efficiently compute the guessing function $\mathsf{pw} \mapsto K_1'$ (in EKE, for example, this requires solving computational Diffie-Hellman). However, since the messages are perfectly password-hiding, the only way to check a guess is to compare it to the output key $K := H(\mathsf{sid}, \mathsf{tr}, K_1, K_2)$. So the adversary is in the scenario where it can make guesses of pw and see (a hash of) K_2. This is precisely the UC PAKE game for P_2, which is assumed to be secure.

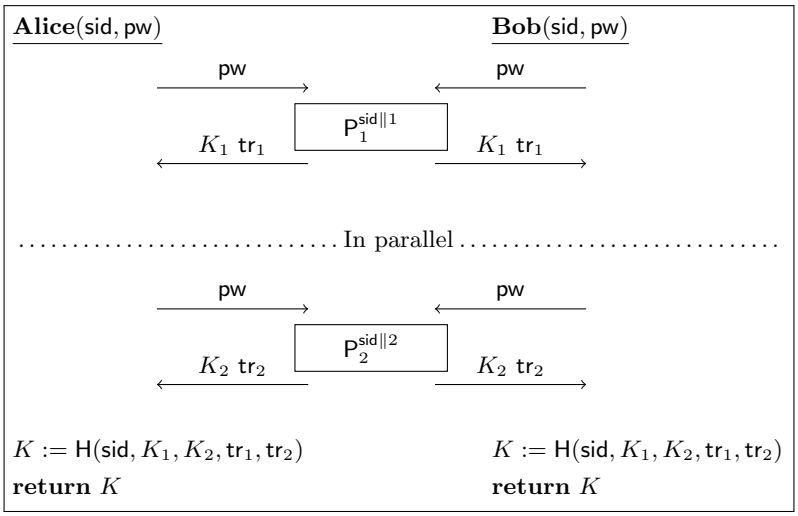


Figure 5: The ParComb combiner for perfectly password-hiding one-round PAKEs P_1 and P_2. The random oracle H outputs values in $\{0,1\}^\lambda$.

The same argument shows that P is secure when P_2 is broken and P_1 remains secure.

Hashing the transcript. We include the full protocol transcript in the final key computation so that the session key can be simulated when messages are mauled. Suppose P_1 is broken, and the adversary has modified one of the P_1 messages. If a simulator does not know either party's password, then it cannot decide whether P_1 should succeed or fail. And since P_1 is broken, the simulator cannot rely on the PAKE UC-realizing $\mathcal{F}_{\mathsf{PAKE}}$. Hashing the transcript resolves the ambiguity, since any mauled messages will cause the protocol to fail (i.e., return different keys) with overwhelming probability.

We now state the security theorem for ParComb. The arguments of the proof are sketched above, and the full proof can be found in Appendix A.1.

Theorem 1. *Let P_1 and P_2 be perfectly password-hiding one-round PAKE protocols, and let $\mathsf{P} := \mathsf{ParComb}[\mathsf{P}_1, \mathsf{P}_2]$. Then the following hold in the static-corruptions setting:*

1. *If P_1 UC-realizes $\mathcal{F}_{\mathsf{PAKE}}$, then so does P (in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model)*
2. *If P_2 UC-realizes $\mathcal{F}_{\mathsf{PAKE}}$, then so does P (in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model).*

In other words, $\mathsf{ParComb}[\mathsf{P}_1, \mathsf{P}_2]$ is at least as strong as the strongest of $\mathsf{P}_1, \mathsf{P}_2$, and is hence a good combiner.

3.1.2 Limitations

While this combiner's efficiency and round complexity are appealing, applicability is currently limited. There are no known one-round post-quantum PAKEs with perfect password hiding. One line of work in one-round post-quantum PAKEs relies on isogeny assumptions [AEK+22, IY23]. However the password hiding property in these depends on computational assumptions about the distribution of random linear combinations of the selected group basis. Another line of work includes the EKE-NIKE construction from [BGHJ24], using the lattice-based NIKE, Swoosh [GdKQ+24]. Again, though, this does not have perfect hiding, since messages are of the form $\mathsf{E}_{\mathsf{pw}}(\mathbf{t})$, where $\mathbf{t}$ is an (M)LWE sample and E is an ideal cipher. This has the same problem as the CAKE construction in the introduction.[4] Thus, we do not yet have a one-round hybrid PAKE construction from a post-quantum assumption.

3.2 The SeqComb combiner

To arrive at the SeqComb combiner, we again consider the minimum possible amount of security necessary for a combined PAKE. Beginning our protocol with a one-round perfectly password hiding PAKE cannot hurt, since even when it is broken, it is difficult to exploit for a password guess. To handle the case that this PAKE P_1 is broken, we may pass pw along with the P_1's session key to a second PAKE P_2, receive a second session key, and output the hash of everything. If P_2 is secure, then including its session key in the final hash will be sufficient to ensure that the overall protocol is secure. If the P_2 is broken then including P_1's session key in the final hash will be sufficient, so long as P_2 is not so broken as to reveal pw outright.

This is almost exactly the definition of SeqComb. That is, SeqComb executes a perfectly password-hiding PAKE P_1, feeds its output into a statistically PSK equality hiding PAKE P_2, and hashes everything at the end. Like ParComb, this combiner is highly efficient, requiring only two extra hashes. Unlike ParComb, its round complexity is the sum of the number of rounds of the underlying PAKEs. We give the full construction in Figure 6.

3.2.1 Security

Again, before presenting the security theorem, we provide intuition for why each component of protocol is necessary.

P_1 perfect password hiding. We require P_1 to be perfectly password hiding for the same reason as in ParComb: it directly deals with pw, and so any leakage here yields a leakage in the combined protocol. More rigorously, if P_1 is broken, an active adversary can efficiently compute the mapping $\mathsf{pw} \mapsto Z'$, but not receive any extra information on whether its guess was correct. So the adversary

[4] While it appears no post-quantum PAKE has this property, we do not believe it is inherent. See Section 4 for potential new directions.

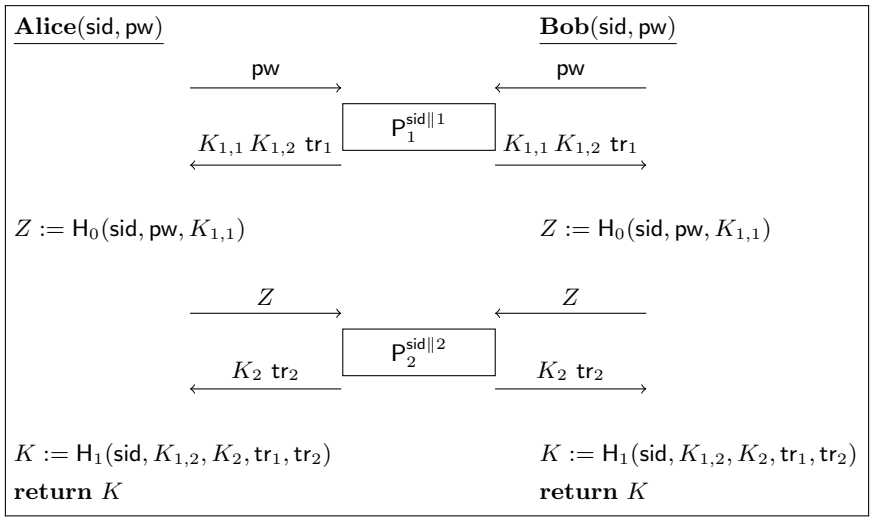


Figure 6: The SeqComb combiner for PAKEs P_1 and P_2. Both random oracles $\mathsf{H}_0, \mathsf{H}_1$ output values in $\{0,1\}^\lambda$. We assume P_1 outputs two keys, each of bitlength λ (if not, it suffices to apply a KDF to stretch its output).

is in the scenario where it has guesses for Z and can see (a hash of) K_2. This is precisely the UC PAKE game for P_2.

P_2 statistical PSK equality hiding. We require P_2 to be statistically PSK equality-hiding in order to permit its transcript to be simulated when P_2 is broken. In the case where both SeqComb parties are honest, and have completed P_1, a simulator must simulate P_2 without knowing whether P_1 succeeded, i.e., without knowing whether the parties agree on the input to P_2. Since this input is high-entropy by assumption (P_1 is a secure PAKE), then the PSK equality hiding property is sufficient to permit the simulator to simulate one case and have it be indistinguishable from the other.

Hashing $K_{1,1}$. We include $K_{1,1}$ in the hash Z because pw alone is insufficient in the case that P_2 is broken. If P_2 were so broken as to leak its password Z in its entirety (technically prevented by PSK equality hiding, but we may assume), then pw would be trivially revealed. We therefore hash $K_{1,1}$, which we assume is high entropy, into Z. Thus, even if Z leaks in its entirety, pw is unaffected.

Hashing pw. We include pw in the computation of Z because $K_{1,1}$ is not sufficient in the case that P_1 is broken. If P_1 is broken, then we must pessimistically assume that both $K_{1,1}$ and $K_{1,2}$ are fully controlled by the adversary. Thus, simply feeding one of these values in P_2 yields no security. Instead, we hash pw with $K_{1,1}$ and feed that into P_2.

Hashing $K_{1,2}$. We include $K_{1,2}$ in the final session key computation because K_2 is not sufficient in the case that P_2 is broken. It suffices to include some yet-unused entropy from the session key of P_1.

We now state the security theorem for $\mathsf{SeqComb}$. The arguments of the proof are sketched above, and the full proof can be found in Appendix A.2.

Theorem 2. *Let P_1 be a one-round perfectly password-hiding PAKE protocol, let P_2 be a statistically PSK equality hiding PAKE protocol, and let $\mathsf{P} := \mathsf{SeqComb}[\mathsf{P}_1, \mathsf{P}_2]$. Then the following hold in the static-corruptions setting:*

1. *If P_1 UC-realizes $\mathcal{F}_{\mathsf{PAKE}}$, then so does P (in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model).*
2. *If P_2 UC-realizes $\mathcal{F}_{\mathsf{PAKE}}$, then so does P (in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model).*

In other words, $\mathsf{SeqComb}[\mathsf{P}_1, \mathsf{P}_2]$ is at least as strong as the strongest of $\mathsf{P}_1, \mathsf{P}_2$, and is hence a good combiner.

3.2.2 Using weaker P_1

Some of the most widely deployed PAKEs such as the Diffie-Hellman-based CPace and SPAKE2 do not enjoy full UC PAKE security but are so-called *lazy extraction* PAKEs. These PAKEs allow adversaries to successfully complete an active attack even after the attacked party finished the protocol. Such prolonged attack completions do not seem to pose any real-world threat, and lazy extraction PAKEs are in widespread use (e.g., for Facebook Messenger chat history transfer [Fac23]). Hence, to make $\mathsf{SeqComb}$ more applicable, we would like to make it work when P_1 UC-realizes $\mathcal{F}_{\mathsf{lePAKE}}$. This functionality is depicted in Figure 1 and is strictly weaker than $\mathcal{F}_{\mathsf{PAKE}}$ due to the additional password test interface on completed sessions. This interface allows the "lazy" attacker to provide a password guess even after an actively attacked party output a session key, and it provides the attacker with that key or a random one depending on whether the guess was correct.

Perhaps surprisingly, $\mathsf{SeqComb}$ is not as tolerant of a lazy-extraction P_1 as one would hope for: $\mathsf{SeqComb}[\mathcal{F}_{\mathsf{lePAKE}}, \mathsf{P}_2]$ does not UC-realize $\mathcal{F}_{\mathsf{lePAKE}}$. This can be made more intuitive by looking at $\mathsf{SeqComb}$'s design. In the case where P_2 cannot be trusted to protect K_2, security is based solely on the output keys $K_{1,1}, K_{1,2}$ of P_1. When these values are revealed to the environment through a correct late password guess on $\mathcal{F}_{\mathsf{lePAKE}}$, the simulator has to explain how they lead to the actual Z and output key K of the honest party. On the other hand, if the guess was wrong, no connection should exist between $K_{1,1}, K_{1,2}$ and Z, K. Since $\mathcal{F}_{\mathsf{lePAKE}}$ does not leak the information of whether the guess was correct or not, the simulator is trapped. We formalize this intuition in Appendix B with a formal distinguisher between $\mathsf{SeqComb}$ with a lazy extraction P_1 and $\mathcal{F}_{\mathsf{lePAKE}}$.

This intuition implies a fix. It would suffice to let the simulator know whether a late password guess succeeded. A PAKE that allows one late password guess with responses of the form "correct" or "wrong" is called a *relaxed* PAKE [ABB+20]. Combining this property with the lazy extraction property yields what we call *relaxed lazy extraction PAKE*, or rlePAKE. We depict the functionality in Figure 1. It has the same guarantees as $\mathcal{F}_{\mathsf{lePAKE}}$, but additionally leaks the information whether or not the late password guess was correct—a weakening

that does not seem to rule out any applications of PAKE protocols. We can prove the following result in Appendix A.3.

Lemma 1. *Let* P_2 *be a statistically PSK equality hiding PAKE protocol, and let* $\mathsf{P} := \mathsf{SeqComb}[\mathsf{P}_1, \mathsf{P}_2]$. *Then the following holds in the static-corruptions setting:*

- *If* P_1 *UC-realizes* $\mathcal{F}_{\mathsf{lePAKE}}$*, then* P *UC-realizes* $\mathcal{F}_{\mathsf{rlePAKE}}$ *(in the* $\mathcal{F}_{\mathsf{RO}}$*-hybrid model).*

Lemma 1 implies that SeqComb is safe to use with a lazy extraction PAKE such as CPace or SPAKE2 as P_1.

3.2.3 Concrete instantiations yielding hybrid PAKE

Unlike ParComb, SeqComb can be instantiated with existing protocols to produce a hybrid PAKE. For P_1, it is possible to use the Diffie-Hellman-based CPace, SPAKE2, or EKE. For P_2, any statistically PSK equality-hiding PAKE will do. While we have only verified this property for the plausibly post-quantum CHIC, CAKE, and OCAKE protocols, it appears to be a natural one, particularly for PAKEs in the random oracle and ideal cipher models. We thus suspect many more plausibly post-quantum PAKEs enjoy this property, e.g., [MRR20, PZ23, BCJ+19, SGJ23]

4 Future work

We identify some interesting problems which we do not address in this work.

Adaptive security So far, we have only defined hybrid PAKE in the static corruptions setting. This may be inherent to the combiner construction. No one-round PAKE can achieve security in the UC model with adaptive corruptions. A simple attack is as follows. The environment $\mathcal{Z}$ initiates a session between an honest Alice and Bob. $\mathcal{Z}$ forwards Alice's message to Bob, allowing Bob to finish the protocol and output K, and withholds Bob's message to Alice. Now, $\mathcal{Z}$ corrupts Alice. There is no way for a simulator to respond to this corruption. This is because the simulator must have used $\mathcal{F}_{\mathsf{PAKE}}$ .NewKey to key Bob, since both parties were honest at the time. Thus, the simulator does not know K, and so cannot choose an appropriate response to $\mathcal{Z}$'s corruption query. We conclude that any PAKE combiner which permits one-round PAKEs is limited to static corruptions.

We thus ask whether there exists a method for constructing adaptively secure PAKEs and, if so, whether it be done generically.

One-round hybrid PAKE In our assessment of ParComb, we concluded that it cannot be built using a post-quantum PAKE, since none has perfect password hiding, to the authors' knowledge. This does not seem inherent, though. We conjecture that it is possible to modify EKE-NIKE[Swoosh] to perfectly hide the password. This would involve sending two pairs of

public keys, replacing public keys of the form $\mathbf{As} + \mathbf{e}$ with ones of the form $\mathbf{As}$ for $\mathbf{A}, \mathbf{s}, \mathbf{e}$ of appropriate dimension. Effectively, this undoes the size improvements of [LP11] in exchange for statistically uniform public keys. Further, since the uniformity result relies on the leftover hash lemma [HILL99], which is not known to apply in the cyclotomic rings of typical module lattice constructions, the NIKE must use unstructured LWE. We estimate public key sizes in the resulting scheme to be 1GB at the 128-bit security level.

We leave as an open question whether such a scheme could be instantiated and improved on to be more practical.

Analysis against quantum adversaries While our combiners can use PAKEs that are secure against quantum adversaries, our reductions are classical and in the random oracle model (ROM). It is known that if a reduction in the ROM is *history-free*, i.e., oracle queries are not recorded and oracle responses do not depend on the values of previous responses, then the reduction holds in the quantum ROM [BDF+11]. We believe our reductions are history-free, but leave the exploration of this path to future work.

Capturing more flavors of PAKE As mentioned earlier, hybrid PAKE implies hybrid aPAKE and iPAKE via the Ω-method [GMR06] and LATKE [KR24]. However, it is not clear how to achieve the *strong* variants of these constructions (called *saPAKE* and *siPAKE*, respectively). A strong aPAKE is an aPAKE which is resistant to precomputation attacks—an adversary must compromise a server before they are able to begin brute-forcing the password. Existing constructions such as OPAQUE [JKX18] use an oblivious pseuodrandom function (OPRF) to achieve strongness. However, no hybrid OPRF construction is currently known, and so it is not clear how to hybridize OPAQUE. More generally, it would be interesting to understand how much strongness relies on an OPRF-like construction, and how much can be achieved using generic combiners. For (s)aPAKE and (s)iPAKE in general, such black-box combiners would further be desirable in order to give developers the option to plug post-quantum protocols into their existing deployments.

Acknowledgements

Thanks to Phillip Gajland for the suggestion of the information-theoretic variant of Swoosh. Julia wants to say: thank you, Michael, for sharing your inspiring ideas with me – this has been an awesome project!

References

[ABB+20] Michel Abdalla, Manuel Barbosa, Tatiana Bradley, Stanislaw Jarecki, Jonathan Katz, and Jiayu Xu. Universally composable relaxed password authenticated key exchange. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 278–307. Springer, Heidelberg, August 2020. doi:10.1007/978-3-030-56784-2_10.

[ABBO24] David Adrian, Bob Beck, David Benjamin, and Devon O'Brien. Advancing Our Amazing Bet on Asymmetric Cryptography, 2024. URL: https://blog.chromium.org/2024/05/advancing-our-amazing-bet-on-asymmetric.html.

[ABJS24] Afonso Arriaga, Manuel Barbosa, Stanislaw Jarecki, and Marjan Skrobot. C'est très chic: A compact password-authenticated key exchange from lattice-based kem. Cryptology ePrint Archive, Paper 2024/308, 2024. https://eprint.iacr.org/2024/308. URL: https://eprint.iacr.org/2024/308.

[AEK+22] Michel Abdalla, Thorsten Eisenhofer, Eike Kiltz, Sabrina Kunzweiler, and Doreen Riepel. Password-authenticated key exchange from group actions. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 699–728. Springer, Heidelberg, August 2022. doi:10.1007/978-3-031-15979-4_24.

[AHH21] Michel Abdalla, Björn Haase, and Julia Hesse. Security analysis of CPace. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part IV*, volume 13093 of *LNCS*, pages 711–741. Springer, Heidelberg, December 2021. doi:10.1007/978-3-030-92068-5_24.

[AP05] Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 191–208. Springer, Heidelberg, February 2005. doi:10.1007/978-3-540-30574-3_14.

[BBB+24] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in kyber implementations. Cryptology ePrint Archive, Paper 2024/1049, 2024. URL: https://eprint.iacr.org/2024/1049.

[BCD+24] Manuel Barbosa, Deirdre Connolly, João Diogo Duarte, Aaron Kaiser, Peter Schwabe, Karolin Varner, and Bas Westerbaan. X-wing. *IACR Communications in Cryptology*, 1(1), 2024. doi:10.62056/a3qj89n4e.

[BCJ+19] Tatiana Bradley, Jan Camenisch, Stanislaw Jarecki, Anja Lehmann, Gregory Neven, and Jiayu Xu. Password-authenticated public-key encryption. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 442–462. Springer, Heidelberg, June 2019. doi:10.1007/978-3-030-21568-2_22.

[BCP+23] Hugo Beguinet, Céline Chevalier, David Pointcheval, Thomas Ricosset, and Mélissa Rossi. GeT a CAKE: Generic transformations from key encaspulation mechanisms to password authenticated key exchanges. In Mehdi Tibouchi and Xiaofeng Wang, editors, *ACNS 23, Part II*, volume 13906 of *LNCS*, pages 516–538. Springer, Heidelberg, June 2023. doi:10.1007/978-3-031-33491-7_19.

[BDF+11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 41–69. Springer, Heidelberg, December 2011. doi:10.1007/978-3-642-25385-0_3.

[BGHJ24] Manuel Barbosa, Kai Gellert, Julia Hesse, and Stanislaw Jarecki. Bare pake: Universally composable key exchange from just passwords. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, pages 183–217, Cham, 2024. Springer Nature Switzerland.

[BH23] Nina Bindel and Britta Hale. A note on hybrid signature schemes. Cryptology ePrint Archive, Paper 2023/423, 2023. https://eprint.iacr.org/2023/423. URL: https://eprint.iacr.org/2023/423.

[BM92] S.M. Bellovin and M. Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, 1992. doi:10.1109/RISP.1992.213269.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. doi:10.1109/SFCS.2001.959888.

[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. doi:10.1007/11426639_24.

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key

exchange, CPA-secure encryption and CCA-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 18*, volume 10831 of *LNCS*, pages 282–305. Springer, Heidelberg, May 2018. doi:10.1007/978-3-319-89339-6_16.

[Fac23] Facebook. The Labyrinth Encrypted Message Storage Protocol. December 2023. URL: https://engineering.fb.com/wp-content/uploads/2023/12/TheLabyrinthEncryptedMessageStorageProtocol_12-6-2023.pdf.

[fan24] Deployments of fancy cryptography, September 2024. URL: https://github.com/fancy-cryptography/fancy-cryptography.

[GdKQ+24] Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta, and Peter Schwabe. SWOOSH: Efficient Lattice-Based Non-Interactive key exchange. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 487–504, Philadelphia, PA, August 2024. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/gajland.

[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. doi:10.1007/11818175_9.

[Gon93] Li Gong. Lower bounds on messages and rounds for network authentication protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 26–37. ACM Press, November 1993. doi:10.1145/168588.168592.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[HL19] Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR TCHES*, 2019(2):1–48, 2019. https://tches.iacr.org/index.php/TCHES/article/view/7384. doi:10.13154/tches.v2019.i2.1-48.

[IY23] Ren Ishibashi and Kazuki Yoneyama. Compact password authenticated key exchange from group actions. In Leonie Simpson and Mir Ali Rezazadeh Baee, editors, *ACISP 23*, volume 13915 of *LNCS*, pages 220–247. Springer, Heidelberg, July 2023. doi:10.1007/978-3-031-35486-1_11.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78372-7_15.

[KR24] Jonathan Katz and Michael Rosenberg. Latke: A framework for constructing identity-binding pakes. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, pages 218–250, Cham, 2024. Springer Nature Switzerland.

[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. doi:10.1007/978-3-642-19571-6_18.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, Heidelberg, February 2011. doi:10.1007/978-3-642-19074-2_21.

[MRR20] Ian McQuoid, Mike Rosulek, and Lawrence Roy. Minimal symmetric PAKE and 1-out-of-N OT from programmable-once public functions. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 425–442. ACM Press, November 2020. doi:10.1145/3372297.3417870.

[PZ23] Jiaxin Pan and Runzhi Zeng. A generic construction of tightly secure password-based authenticated key exchange. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VIII*, volume 14445 of *LNCS*, pages 143–175. Springer, Heidelberg, December 2023. doi:10.1007/978-981-99-8742-9_5.

[RX23] Lawrence Roy and Jiayu Xu. A universally composable PAKE with zero communication cost - (and why it shouldn't be considered UC-secure). In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 714–743. Springer, Heidelberg, May 2023. doi:10.1007/978-3-031-31368-4_25.

[SGJ23] Bruno Freitas Dos Santos, Yanqi Gu, and Stanislaw Jarecki. Randomized half-ideal cipher on groups with applications to UC (a)PAKE. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 128–156. Springer, Heidelberg, April 2023. doi:10.1007/978-3-031-30589-4_5.

[Sho20] Victor Shoup. Security analysis of itSPAKE2+. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of

LNCS, pages 31–60. Springer, Heidelberg, November 2020. doi:10.1007/978-3-030-64381-2_2.

[Wes24] Bas Westerbaan. The state of the post-quantum Internet, March 2024. The Cloudflare Blog. URL: https://blog.cloudflare.com/pq-2024.

A Proofs

A.1 Proof of Theorem 1

We show the theorem for an intact P_1 PAKE protocol, i.e., for the protocol $\mathsf{ParComb}[\mathcal{F}_{\mathsf{PAKE}}^{(1)}, \mathsf{P}_2]$. Because the UC framework follows a sequential model of execution where only one machine is active at any given point in time, we cannot execute both $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ and P_2 in parallel. We hence analyze a protocol version where parties first receive their output from $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ and only then send the first message in P_2. Because the execution of P_2 does not depend on $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$, when instantiating $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ with an interactive protocol, its messages can be sent together with the P_2 messages.

Proof. **Game $\mathbf{G}_0$: The real execution.** This is the real execution as in Figure 9, with a dummy adversary $\mathcal{A}$. W.l.o.g. we replace $\mathsf{P}_1^{\mathsf{sid}||1}$ by a hybrid functionality $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$.

Game $\mathbf{G}_1$: Change layout. We change the previous game as follows:

- We move the whole execution into a single machine and call it the simulator Sim.
- We add all the record-keeping of the simulator as in Figures 7 and 8.
- In between $\mathcal{Z}$ and Sim we add one dummy party for each real party.
- In between the dummy parties and Sim, we add the ideal functionality $\mathcal{F}_{\mathsf{PAKE}}$ as in Figure 1, but relaying passwords of honest parties to Sim and relaying outputs keys provided by Sim to the dummy parties. We call that functionality $\mathcal{F}$.

The changes are only syntactical since the real execution runs on same inputs and produces outputs the same way as in the previous game. We hence have

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

Game $\mathbf{G}_2$: Simulate the P_2 message of honest parties. In this game we change the simulation to compute P_2 messages using $(x, \tau_x) := \mathsf{Sim}^{(2)}.\mathsf{Start}()$, and compute the P_2 output keys as $\mathsf{Sim}^{(2)}.\mathsf{ComputeKey}(y, x, \tau_x)$ for incoming message y. Because P_2 is perfectly password-hiding, messages are distributed equally and the P_2 output keys are equal to $\mathbf{G}_1$. We hence have

On $(\mathsf{NewSession}, \mathsf{sid}, \mathcal{P}, \mathcal{P}')$ from $\mathcal{F}_{\mathsf{PAKE}}$
- ($\mathbf{G}_{10}$) Record $(\mathsf{P1Session}, \mathcal{P}, \mathcal{P}', \bot)$ and mark it `fresh`
- ($\mathbf{G}_{10}$) Send $(\mathsf{NewSession}, \mathsf{sid}||1, \mathcal{P}, \mathcal{P}')$ to $\mathcal{Z}$ from $\mathcal{F}^{(1)}_{\mathsf{PAKE}}$

On $(\mathsf{NewSession}, \mathsf{sid}||1, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ from corrupt $\mathcal{P}$ to $\mathcal{F}^{(1)}_{\mathsf{PAKE}}$
- ($\mathbf{G}_1$) Record $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ and it mark it `fresh`

On $(\mathsf{TestPwd}, \mathsf{sid}||1, \mathcal{P}, \mathsf{pw})$ from $\mathcal{Z}$ to $\mathcal{F}^{(1)}_{\mathsf{PAKE}}$ or as internal call // w.l.o.g. $\mathcal{P}$ is honest
- ($\mathbf{G}_6$) Retrieve $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}, \cdot, \bot)$ marked `fresh`
- ($\mathbf{G}_6$) Send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{PAKE}}$
- ($\mathbf{G}_6$)Upon response "correct":
  - ($\mathbf{G}_6$) Mark $\mathsf{P1Session}$ as `compromised`
  - ($\mathbf{G}_6$) Record $(\mathsf{CorrectTestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$
  - ($\mathbf{G}_6$) Return "correct"
- ($\mathbf{G}_6$) Upon response "wrong":
  - ($\mathbf{G}_6$) Mark $\mathsf{P1Session}$ as `interrupted`
  - ($\mathbf{G}_6$) Return "wrong"

On $(\mathsf{NewKey}, \mathsf{sid}||1, \mathcal{P}, K'_1)$ from $\mathcal{Z}$ to $\mathcal{F}^{(1)}_{\mathsf{PAKE}}$
- ($\mathbf{G}_1$) Retrieve $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}, [\mathcal{P}'], [\mathsf{pw}])$ marked $m \neq$ `completed`
- ($\mathbf{G}_1$) If $m =$ `compromised`: $K_1 := K'_1$
- ($\mathbf{G}_1$) Elif $m =$ `fresh` and $\exists(\mathsf{P1Key}, \mathsf{sid}, \mathcal{P}', [K''_1], \texttt{fresh})$: // $\mathcal{P}$ finishes last
  - ($\mathbf{G}_7$) If $\exists(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}', \mathsf{pw}, \texttt{correct})$: $K_1 := K''_1$ // $Z \neq \bot \Rightarrow \mathcal{P}$ corrupt $\Rightarrow$ w.l.o.g, $\mathcal{P}'$ is honest, and we already ran TestPwd when $\mathcal{P}'$ finished.
  - ($\mathbf{G}_{10}$) Elif $\mathcal{P}$ and $\mathcal{P}'$ are honest: $K_1 := \bot$ // $\mathcal{F}_{\mathsf{PAKE}}$ controls the final K anyway
  - ($\mathbf{G}_1$) Else: $K_1 \leftarrow\!\$\ \{0,1\}^\lambda$ // $\mathcal{P}$ corrupt, wrong password
- ($\mathbf{G}_1$) Elif $\exists(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, \cdot)$ then set $K_1 \leftarrow\!\$\ \{0,1\}^\lambda$ // $\mathcal{P}$ finishes first or interrupted, or $\mathcal{P}'$ finished non-fresh.
- ($\mathbf{G}_1$) Else ignore the query // Isolated $\mathcal{P}$ does not output a key
- ($\mathbf{G}_7$) If $m =$ `fresh`:
  - ($\mathbf{G}_7$) If $\mathcal{P}$ is honest and $\mathcal{P}'$ is corrupt: // Now that TestPwd can no longer be called on this session, we can finally call it
    - ($\mathbf{G}_7$) Retrieve $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, [\mathsf{pw}'])$
    - ($\mathbf{G}_7$) Run code of $(\mathsf{TestPwd}, \mathsf{sid}||1, \mathcal{P}, \mathsf{pw}')$
- ($\mathbf{G}_1$) Record $(\mathsf{P1Key}, \mathsf{sid}, \mathcal{P}, K_1, m)$
- ($\mathbf{G}_1$) Mark $\mathsf{P1Session}$ as `completed`
- ($\mathbf{G}_1$) If $\mathcal{P}$ is corrupt:
  - ($\mathbf{G}_1$) Send (sid, K_1) to $\mathcal{P}$
  - ($\mathbf{G}_1$) Return
- ($\mathbf{G}_2$) Compute $x, \tau_x \leftarrow \mathsf{Sim}^{(2)}.\mathsf{Gen}(1^\lambda)$
- ($\mathbf{G}_2$) Record $(\mathsf{P2State}, \mathsf{sid}, \mathcal{P}, x, \tau_x)$
- ($\mathbf{G}_1$) Record $(\mathsf{Sent}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', x)$
- ($\mathbf{G}_1$) Send $(\mathcal{P} \to \mathcal{P}', \mathsf{sid}, x)$ to $\mathcal{Z}$

Figure 7: Simulator for $\mathsf{ParComb}[\mathcal{F}^{(1)}_{\mathsf{PAKE}}, \mathsf{P}_2]$ realizing $\mathcal{F}_{\mathsf{PAKE}}$. $\mathsf{Sim}^{(2)}$ represents the simulator for the perfectly password-hiding property of PAKE P_2. Random oracle H is simulated as in the real execution.

On $(\mathcal{P} \to \mathcal{P}', \mathsf{sid}, y)$ from $\mathcal{Z}$ // P2 message delivered to $\mathcal{P}'$

- ($\mathbf{G}_1$) Record $(\mathsf{Recvd}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', y)$
- ($\mathbf{G}_1$) If $\mathcal{P}$ is corrupted: record $(\mathsf{Sent}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', y)$
- ($\mathbf{G}_9$) If $\mathcal{P}'$ is honest AND $\exists(\mathsf{CorrectTestPwd}, \mathsf{sid}, \mathcal{P}', [\mathsf{pw}])$:// Compute K_2 for attacked $\mathcal{P}'$
  - ($\mathbf{G}_9$) Retrieve $(\mathsf{P2State}, \mathsf{sid}, \mathcal{P}', [x], [\tau])$
  - ($\mathbf{G}_9$) Let $K_2 := \mathsf{Sim}^{(2)}.\mathsf{ComputeKey}(\mathsf{sid}, x, y, \tau, \mathsf{pw})$
  - ($\mathbf{G}_9$) Record $(\mathsf{P2Key}, \mathsf{sid}, \mathcal{P}', K_2)$
  - ($\mathbf{G}_2$) Call $\mathsf{FinalizeHonest}(\mathsf{sid}, \mathcal{P}')$
- ($\mathbf{G}_2$) If $\mathcal{P}'$ is honest AND $\exists(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, \cdot)$ marked `completed`:// P1 is done and the P2 message was received - $\mathcal{P}'$ outputs a key now.
  - ($\mathbf{G}_2$) Call $\mathsf{FinalizeHonest}(\mathsf{sid}, \mathcal{P}')$

// This procedure calls NewKey on an honest party. For an honest party finishing, it holds that (1) there is always a P1Key record, and (2) there is a P2Key record if the party is actively attacked. The final key is generated from these records if both exist.

Procedure $\mathsf{FinalizeHonest}(\mathsf{sid}, \mathcal{P})$

- ($\mathbf{G}_{10}$) Set $K := \bot$
- ($\mathbf{G}_1$) If $\exists(\mathsf{P1Key}, \mathsf{sid}, \mathcal{P}, K_1, \mathtt{fresh})$ and $(\mathsf{P2Key}, \mathsf{sid}, \mathcal{P}, K_2)$: // $\mathcal{P}$ is running a session with the adversary
  - ($\mathbf{G}_1$) Retrieve $(\mathsf{Sent}, \mathsf{sid}, [\mathcal{P}'], \mathcal{P}, [x])$ and $(\mathsf{Recvd}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, [y])$
  - ($\mathbf{G}_1$) Set $K := \mathsf{H}(\mathsf{sid}, K_1, K_2, x, y)$
- ($\mathbf{G}_5$) Else: // $\mathcal{P}$ is running a session with an honest party
  - ($\mathbf{G}_5$) Retrieve $(\mathsf{Sent}, \mathsf{sid}, [\mathcal{P}'], \mathcal{P}, [x'])$, $(\mathsf{Sent}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', [y])$, $(\mathsf{Recvd}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, [x])$ and $(\mathsf{Recvd}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, [y'])$
  - ($\mathbf{G}_5$) If $x \neq x'$ or $y \neq y'$ send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \bot)$ to $\mathcal{F}_{\mathsf{PAKE}}$: // If the transcripts of two honest parties aren't equal, tank the session
- ($\mathbf{G}_1$) Send $(\mathsf{NewKey}, \mathsf{sid}, \mathcal{P}, K)$ to $\mathcal{F}_{\mathsf{PAKE}}$

Figure 8: Simulator (cont.)

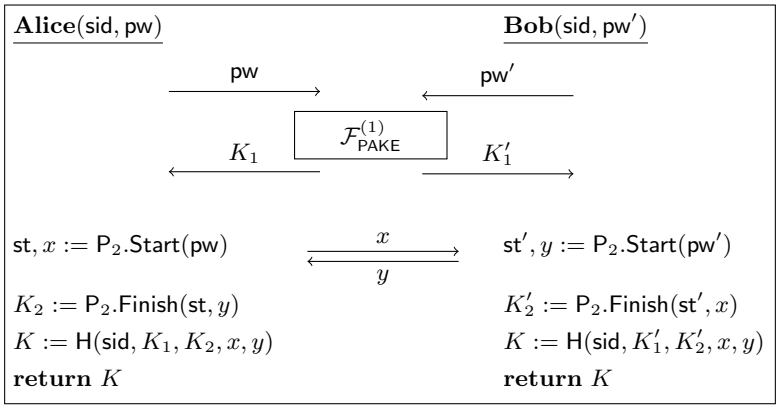


Figure 9: UC execution of the ParComb combiner with a hybrid functionality $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ and P_2 being a perfectly password-hiding one-round PAKE. When instantiating $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$, messages can be sent in parallel with x, y. The random oracle H outputs values in $\{0,1\}^\lambda$.

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_2].$$

Game $\mathbf{G}_3$: Functionality aligns keys in honest sessions. We change the functionality to ignore the simulator's output key for honest parties finishing last in honest sessions, with matching passwords, and where both records were `fresh` at the time of output generation (i.e., no $\mathsf{TestPwd}$ was queried by the simulator on any of the two parties). In this case, the functionality also outputs the session key of the first party to the second party.

This and the previous game are not distinguishable by the (trivial) correctness of the protocol, and hence we have

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

Game $\mathbf{G}_4$: Randomize output key of parties in an honest interaction. We change the functionality $\mathcal{F}$ to ignore the output keys from the simulation in case an honest party finishes in an unattacked session, and instead uses the output key from its internal Key record.

Note that the change (a) randomizes the output key of the honest user who finish first in an unattacked session, (b) randomizes the output key of the honest user who finish last in an unattacked session, and (c) aligns keys of the honest session in case of matching passwords. In this game, parties with tampered P_2 transcripts still receive their output key from the simulation, which is uncorrelated to their honest partner's key (which is randomized in this game) due to the P_2 transcript that is included in the hash.

Because $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ draws K_1 uniformly at random and the adversary does not query $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ in any way that it generates output to the adversary that depends on K_1, $\mathcal{Z}$ only sees a difference in this and the previous game if it queries the random oracle with $H(\mathsf{sid}, x, y, K_1, K_2)$. We hence have

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_3]| \leq \frac{q_{\mathsf{H}}}{2^\lambda}.$$

Game $\mathbf{G}_5$: Randomize output key upon message tampering. We change the simulator to send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \perp)$ to $\mathcal{F}$ (guaranteeing the session is `interrupted`) when $\mathcal{Z}$ tampers with the P_2 message sent to $\mathcal{P}$ in an otherwise honest interaction (in particular, when there are no $\mathsf{TestPwd}$ queries against $\mathcal{P}$ to $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$). At the same time, we let the functionality ignore the output keys from the simulator of the session of $\mathcal{P}$ in case such a $\mathsf{TestPwd}$ happened, and instead output the key from its Key record.

The argument is very similar to $\mathbf{G}_4$: $\mathcal{Z}$ can only notice a difference it it queries the final hash of $\mathcal{P}$ to the random oracle. We again have

$$|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| \leq \frac{q_{\mathsf{H}}}{2^\lambda}.$$

As of this game, all `fresh` records in $\mathcal{F}$ produce output keys.

Game $\mathbf{G}_6$: Relay TestPwd against P_1. We change the code of the simulator upon $\mathcal{Z}$ sending $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ for an honest $\mathcal{P}$. Sim now also sends this query to $\mathcal{F}$ and uses that response as its own response to $\mathcal{Z}$.

Because both functionality instances work on the same inputs and compromised or interrupted markings in $\mathcal{F}$ do not yet affect the outputs, the changes are only syntactical and we have

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

Game $\mathbf{G}_7$: Extract a password from a corrupt party. We change the simulator to send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}$ upon $(\mathsf{NewKey}, \mathsf{sid}, \mathcal{P}, \cdot)$ from $\mathcal{Z}$ for an honest $\mathcal{P}$ that is in a session with a corrupt $\mathcal{P}'$ that had previously send input $(\mathsf{NewSession}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$. The simulator then sends the output key of $\mathcal{P}$ to $\mathcal{F}$ via NewKey, and $\mathcal{F}$ uses that key for compromised sessions.

While in $\mathbf{G}_6$ $\mathcal{F}$ just outputted all keys from the simulation, in this game it still does the same but receives the ones for compromised sessions through the simulator's NewKey queries. Hence, the changes are only syntactical and we have

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

Game $\mathbf{G}_8$: Randomize output keys of attacked users with wrong password guess. We change the functionality to ignore the simulator's key for parties who finish on `interrupted` sessions, and instead use the output key from the Key record.

Again, the argument is very similar to the one in $\mathbf{G}_4$: in $\mathbf{G}_7$, upon a wrong password guess, $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ issues a random K_1 to $\mathcal{P}$ of which $\mathcal{Z}$ only sees the output $H(\mathsf{sid}, K_1, K_2, x, y)$ of $\mathcal{P}$. We can now randomize this output unnoticed by $\mathcal{Z}$ except with the negligible probability that $\mathcal{Z}$ guesses K_1, i.e.,

$$|\Pr[\mathbf{G}_8] - \Pr[\mathbf{G}_7]| \leq \frac{q_{\mathsf{H}}}{2^\lambda}.$$

As of this game, all `interrupted` records in $\mathcal{F}$ produce output keys.

Game $\mathbf{G}_9$: Simulate compromised parties using $\mathsf{TestPwd}$ guesses. We change the simulator upon sending $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}$ for an honest $\mathcal{P}$, as detailed in $\mathbf{G}_6$ and $\mathbf{G}_7$. If $\mathcal{F}$ replies with "correct", we now let the simulator use pw in the computation of $\mathsf{Sim}^{(2)}.\mathsf{ComputeKey}$.

The change is only syntactical because the passwords used in $\mathsf{ComputeKey}$ are identical in this and the previous game. We have

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

Game $\mathbf{G}_{10}$: Remove the passwords from the simulation. We modify the simulator to use $\perp$ as password in all $\mathsf{NewSession}$ inputs to the internally simulated $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ for honest parties. At the same time, we change the functionality not to forward input passwords of honest parties to the simulator, but relay $(\mathsf{NewSession}, \mathsf{sid}, \mathcal{P}, \mathcal{P}')$ from $\mathcal{F}$ to $\mathcal{Z}$ as coming from $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$.

We need to argue that outputs of $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ that depend on input passwords of honest parties, i.e., any K_1 that was output to an honest party and chosen by $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$, are not relevant to the execution anymore. This concerns `fresh` and `interrupted` records (`compromised` records get an adversarially chosen key instead of a secure one picked by $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$). This is immediate to see in $\mathbf{G}_9$: $\mathcal{F}$ computes all output keys as of $\mathbf{G}_9$, and the P_2 execution does not depend on K_1.

Because forwarding of input passwords and relaying of simulator output keys was the only difference when we introduced $\mathcal{F}$ in $\mathbf{G}_1$, the functionality $\mathcal{F}$ in $\mathbf{G}_{10}$ is equal to $\mathcal{F}_{\mathsf{PAKE}}$ and we have

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

The theorem thus follows with the simulator depicted in Figures 7 and 8. □

A.2 Proof of Theorem 2

The proof of this theorem is split into two cases, namely analyzing P relying on the UC security of P_1, and P relying on the UC security of P_2. The corresponding protocol relying on a secure P_1 is depicted in Figure 13. The protocol relying on a secure P_1 is depicted in Figure 10, relying on a P_2 with arbitrary communication pattern.

Theorem 2 then follows from Lemmas 2 and 3. Since the proofs each require the transcript of either the first or the second PAKE to be included in the final hash, for the combiner to be secure, the full protocol transcript needs to be included in the final hash.

Lemma 2. *Let P_1 be a one-round perfectly password-hiding PAKE protocol. Then protocol $\mathsf{SeqComb}[\mathsf{P}_1, \mathcal{F}_{\mathsf{PAKE}}^{(2)}]$ (Figure 10) UC-realizes $\mathcal{F}_{\mathsf{PAKE}}$ in the $(\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{PAKE}})$-hybrid model, where $\mathsf{H}_0, \mathsf{H}_1$ are modeled as random oracles (i.e., as calls to $\mathcal{F}_{\mathsf{RO}}$), and $\mathcal{F}_{\mathsf{PAKE}}^{(2)} = \mathcal{F}_{\mathsf{PAKE}}$ (i.e., the superscript is only added to differentiate between the two instances of $\mathcal{F}_{\mathsf{PAKE}}$ in the statement).*

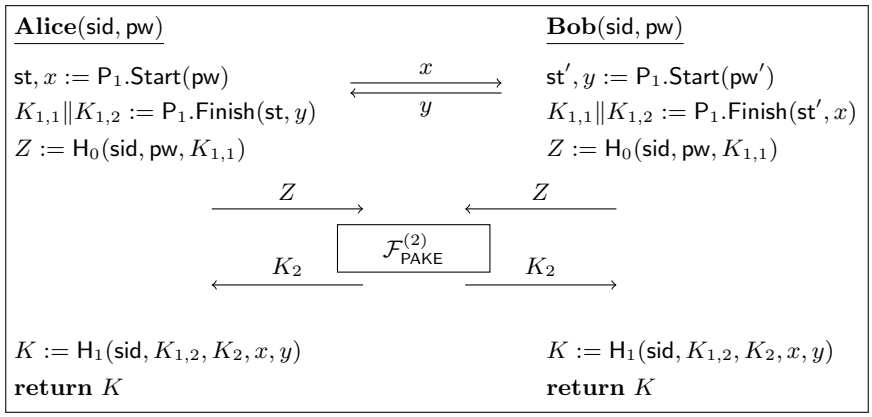


Figure 10: Protocol $\mathsf{SeqComb}[\mathsf{P}_1, \mathcal{F}_{\mathsf{PAKE}}^{(2)}]$ relying on an intact second PAKE protocol.

For proving the lemma we make the following simplifying assumptions.

1. $\mathcal{Z}$ corrupts at most one party in a session
2. $\mathcal{Z}$ never issues a TestPwd query against a corrupt party
3. $\mathcal{Z}$ never issues a TestPwd query against an honest party if the other party is corrupt

These simplifications are without loss of generality, i.e., our proof still implies that there is no distinguishing environment $\mathcal{Z}$. This is because (1) in the fully corrupt setting the simulator knows all secrets and hence the simulation is straightforward, (2) a TestPwd query against a corrupt party compares two passwords that were both given by $\mathcal{Z}$, and (3) the password guess can be issued via a corrupt NewSession instead of TestPwd.

Game $\mathbf{G}_0$: The real execution. This is the real execution as in Figure 10, with a dummy adversary $\mathcal{A}$ and a hybrid functionality $\mathcal{F}_{\mathsf{PAKE}}^{(2)}$ in place of P_2.

Game $\mathbf{G}_1$: Change layout. We change the previous game as follows:

- We move the whole execution into a single machine and call it the simulator Sim.
- We add all the record-keeping of the simulator as in Figures 11 and 12.
- In between $\mathcal{Z}$ and Sim we add one dummy party for each real party.
- In between the dummy parties and Sim, we add the ideal functionality $\mathcal{F}_{\mathsf{PAKE}}$ as in Figure 1, but relaying passwords of honest parties to Sim and relaying outputs keys provided by Sim to the dummy parties. We call that functionality $\mathcal{F}$.

On $(\mathsf{NewSession}, \mathsf{sid}, \mathcal{P}, \mathcal{P}')$ from $\mathcal{F}_{\mathsf{PAKE}}$ // Simulate the P1 message

- ($\mathbf{G}_2$) Compute $x, \tau_x \leftarrow \mathsf{Sim}^{(1)}$
- ($\mathbf{G}_1$) Record $(\mathsf{P1State}, \mathsf{sid}, \mathcal{P}, x, \tau_x)$
- ($\mathbf{G}_1$) Record $(\mathsf{Sent}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', x)$
- ($\mathbf{G}_1$) Send $(\mathcal{P} \to \mathcal{P}', \mathsf{sid}, x)$ to $\mathcal{Z}$

On $(\mathcal{P} \to \mathcal{P}', \mathsf{sid}, y)$ from $\mathcal{Z}$ // P1 message delivery

- ($\mathbf{G}_1$) If $\exists(\mathsf{Recvd}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \cdot)$: ignore this query
- ($\mathbf{G}_1$) Record $(\mathsf{Recvd}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', y)$
- If $\mathcal{P}$ is corrupted: // Record the sent message because it's adversarially generated
  - ($\mathbf{G}_1$) Record $(\mathsf{Sent}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', y)$
- If $\mathcal{P}'$ is honest:// If the receiver is honest, use this as the time to start its P2 session
  - ($\mathbf{G}_8$) Record $(\mathsf{P2Session}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \bot)$ marked `fresh`
  - ($\mathbf{G}_1$) Send $(\mathsf{NewSession}, \mathsf{sid}\|2, \mathcal{P}, \mathcal{P}')$ to $\mathcal{Z}$

On $(\mathsf{NewSession}, \mathsf{sid}||2, \mathcal{P}', Z)$ from corrupt $\mathcal{P}$ to $\mathcal{F}^{(2)}_{\mathsf{PAKE}}$

- ($\mathbf{G}_1$) Record $(\mathsf{P2Session}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', Z)$
- ($\mathbf{G}_1$) Send $(\mathsf{NewSession}, \mathsf{sid}\|2, \mathcal{P}, \mathcal{P}')$ to $\mathcal{Z}$ //Wait with TestPwd until other party received input

On $(\mathsf{TestPwd}, \mathsf{sid}||2, \mathcal{P}, Z)$ from $\mathcal{Z}$ to $\mathcal{F}^{(2)}_{\mathsf{PAKE}}$ or as internal call // w.l.o.g., $\mathcal{P}$ is honest

- ($\mathbf{G}_1$) Retrieve $(\mathsf{P2Session}, \mathsf{sid}, \mathcal{P}, \cdot, \bot)$ marked `fresh`
- // Z is the correct password iff it is what $\mathcal{P}$ would have computed. This is the case iff both the pw and $K_{1,1}$ used in the computation of Z are correct. So we test those.
- ($\mathbf{G}_4$) Retrieve $(\mathsf{H}_0, \mathsf{sid}, [\mathsf{pw}], [K_1^*], Z)$ or GOTO `wrong`
- ($\mathbf{G}_4$) Retrieve $(\mathsf{P1State}, \mathsf{sid}, \mathcal{P}, [x], [\tau_x])$, $(\mathsf{Recvd}, \mathsf{sid}, \cdot, \mathcal{P}, [y])$
- ($\mathbf{G}_4$) Let $[K_{1,1}]\|[K_{1,2}] := \mathsf{Sim}^{(1)}.\mathsf{ComputeKey}(\mathsf{sid}, x, y, \tau_x, \mathsf{pw})$
- ($\mathbf{G}_4$) Record $(\mathsf{P1Key}, \mathcal{P}, K_{1,1}, K_{1,2})$
- ($\mathbf{G}_4$) If $K_1^* \neq K_{1,1}$ GOTO `wrong` // Early-fail if the P1 key disagrees
- ($\mathbf{G}_4$) Send $(\mathsf{TestPwd}, \mathsf{sid}||2, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{PAKE}}$
- ($\mathbf{G}_1$) If "correct": GOTO `correct`
- ($\mathbf{G}_1$) If "wrong": GOTO `wrong`
- ($\mathbf{G}_1$) Label `wrong`:
  - ($\mathbf{G}_1$) Mark retrieved P2Session `interrupted`
  - ($\mathbf{G}_1$) Record $(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw}, \texttt{wrong})$
  - ($\mathbf{G}_1$) Return "wrong"
- ($\mathbf{G}_1$) Label `correct`:
  - ($\mathbf{G}_1$) Mark retrieved P2Session `compromised`
  - ($\mathbf{G}_1$) Record $(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw}, \texttt{correct})$
  - ($\mathbf{G}_1$) Return "correct"

Figure 11: Simulator for $\mathsf{SeqComb}[\mathsf{P}_1, \mathcal{F}^{(2)}_{\mathsf{PAKE}}]$ realizing $\mathcal{F}_{\mathsf{PAKE}}$, Lemma 2. Random oracles $\mathsf{H}_0, \mathsf{H}_1$ are simulated as in the real execution.

On $(\mathsf{NewKey}, \mathsf{sid}\|2, \mathcal{P}, K_2^*)$ from $\mathcal{Z}$ to $\mathcal{F}_{\mathsf{PAKE}}^{(2)}$

- ($\mathbf{G}_1$) Retrieve $(\mathsf{P2Session}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', [Z])$ with marking $m \neq \texttt{completed}$
- // Pick K_2 the same way a normal PAKE functionality would pick the key
- ($\mathbf{G}_1$) If $m = \texttt{compromised}$: $K_2 := K_2^*$
- ($\mathbf{G}_1$) Else if $m = \texttt{fresh}$ AND $\exists(\mathsf{P2Key}, \mathsf{sid}, \mathcal{P}', [K_2'], \texttt{fresh})$: // $\mathcal{P}$ finishes last
  - ($\mathbf{G}_5$) If $\exists(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}', Z, \texttt{correct})$: $K_2 := K_2'$ // $Z \neq \perp \Rightarrow \mathcal{P}$ corrupt $\Rightarrow$ w.l.o.g, $\mathcal{P}'$ is honest, and we already ran TestPwd when $\mathcal{P}'$ finished.
  - ($\mathbf{G}_8$) Elif $\mathcal{P}$ and $\mathcal{P}'$ are honest: $K_2 := \perp$ // $\mathcal{F}_{\mathsf{PAKE}}$ controls the final K anyway
  - ($\mathbf{G}_1$) Else: $K_2 \leftarrow\!\$\ \{0,1\}^\lambda$ // $\mathcal{P}$ corrupt, wrong password
- ($\mathbf{G}_1$) Elif $\exists(\mathsf{P2Session}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, \cdot)$ then set $K_2 \leftarrow\!\$\ \{0,1\}^\lambda$ // $\mathcal{P}$ finishes first or interrupted, or $\mathcal{P}'$ finished non-fresh.
- ($\mathbf{G}_1$) Else ignore the query // Isolated $\mathcal{P}$ does not output a key
- // Same as PAKE functionality, record the key and return it to the party (if corrupt)
- ($\mathbf{G}_1$) Record $(\mathsf{P2Key}, \mathsf{sid}, \mathcal{P}, K_2, m)$
- ($\mathbf{G}_1$) Mark P2Session $\texttt{completed}$
- ($\mathbf{G}_1$) If $\mathcal{P}$ is corrupt:
  - ($\mathbf{G}_1$) Send (sid, K_2) to $\mathcal{P}$ // No need to call $\mathcal{F}_{\mathsf{PAKE}}$.NewKey on a corrupt party
- // Now we compute the final key $\mathsf{H}_2(\mathsf{sid}, K_{1,2}, K_2, \mathsf{tr})$ for honest $\mathcal{P}$ if we can find K_1.
- ($\mathbf{G}_4$) If $\mathcal{P}'$ is corrupt AND P2Session is $\texttt{fresh}$: // Attacked $\mathcal{P}$ finishes. Opportunistically run TestPwd in the honest-corrupt setting.
  - ($\mathbf{G}_4$) Retrieve $(\mathsf{P2Session}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, [Z'])$
  - ($\mathbf{G}_4$) Run code of $(\mathsf{TestPwd}, \mathsf{sid}\|2, \mathcal{P}, Z')$ // Ensure P1Key is set in attacked session
- ($\mathbf{G}_4$) If $\mathcal{P}'$ is corrupt OR $\exists(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}, \cdot, \texttt{correct})$:
  - ($\mathbf{G}_4$) Retrieve $(\mathsf{P1Key}, \mathsf{sid}, \mathcal{P}, \cdot, [K_{1,1}, K_{1,2}])$ // Must exist in a non-tanked session because, above, Sim always calls TestPwd on an honest-corrupt setting
  - ($\mathbf{G}_4$) $K := \mathsf{H}_2(\mathsf{sid}, K_{1,2}, K_2, \mathsf{tr})$
- Else:
  - ($\mathbf{G}_8$) $K := \perp$ // Honest-honest setting: $\mathcal{F}_{\mathsf{PAKE}}$ decides
  - ($\mathbf{G}_3$) Retrieve $(\mathsf{Sent}, \mathsf{sid}, [\mathcal{P}'], \mathcal{P}, [x'])$, $(\mathsf{Sent}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', [y])$, $(\mathsf{Recvd}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, [x])$ and $(\mathsf{Recvd}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, [y'])$
  - ($\mathbf{G}_3$) If $x \neq x'$ or $y \neq y'$ send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \perp)$ to $\mathcal{F}_{\mathsf{PAKE}}$: // DoS attack on P_1
- Send $(\mathsf{NewKey}, \mathsf{sid}, \mathcal{P}, K)$ to $\mathcal{F}_{\mathsf{PAKE}}$

Figure 12: Cont. simulator.

The changes are only syntactical since the real execution runs on same inputs and produces outputs the same way as in the previous game. We hence have

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

Game $\mathbf{G}_2$: Simulate the P_1 message of honest parties. In this game we change the simulation to compute P_1 messages using $(x, \tau_x) := \mathsf{Sim}^{(1)}.\mathsf{Start}()$, and compute the P_1 output keys as $\mathsf{Sim}^{(1)}.\mathsf{ComputeKey}(y, x, \tau_x, \mathsf{pw})$ for incoming message y. Because P_1 is perfectly password-hiding, messages are distributed equally and the P_1 output keys are equal to $\mathbf{G}_1$. We hence have

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_2].$$

Game $\mathbf{G}_3$: Randomize outputs keys in case of DoS attack in P_1. We modify the simulator to send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \perp)$ for an honest party $\mathcal{P}$ in case $\mathcal{P}'$ is also honest, and there was no $\mathsf{TestPwd}$ query against $\mathcal{P}$ by $\mathcal{Z}$, and $\mathcal{Z}$ did not deliver both x, y untampered. This can only be decided once $\mathcal{Z}$ sends NewKey for $\mathcal{P}$ to $\mathcal{F}_{\mathsf{PAKE}}^{(2)}$, and hence the simulator performs this check at the very end of the NewKey interface.

Because `interrupted` markings in $\mathcal{F}$ do no yet affect output keys, the changes do not affect the output distribution and we have

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

Game $\mathbf{G}_4$: Extract password from corrupt party. We change the simulator for an honest $\mathcal{P}$, either upon $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, Z)$ from $\mathcal{Z}$ where w.l.o.g. $\mathcal{P}'$ is honest, or upon a corrupt $\mathcal{P}'$ receiving input $(\mathsf{NewSession}, \mathsf{sid}, \mathcal{P}, Z)$, where $Z = \mathsf{H}_0(\mathsf{sid}, [\mathsf{pw}], [K_{1,1}^*])$. The simulator computes $[K_{1,1}] \| [K_{1,2}] := \mathsf{Sim}^{(1)}.\mathsf{ComputeKey}(\mathsf{sid}, x, y, \tau_x, \mathsf{pw})$ and if $K_{1,1} = K_{1,1}^*$ sends $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{PAKE}}$. In all other cases, the simulator keeps letting the internally emulated $\mathcal{F}_{\mathsf{PAKE}}^{(2)}$ instance handle the guess. When $\mathcal{P}$ is supposed to generate output, if the guess came back "correct", the simulator sets the key of $\mathcal{P}$ to be $\mathsf{H}_2(\mathsf{sid}, K_{1,2}, K_2^*, \mathsf{tr})$, where K_2^* is taken from the NewKey query from $\mathcal{Z}$ to $\mathcal{F}_{\mathsf{PAKE}}^{(2)}$ for $\mathcal{P}$. The simulator sends this key through NewKey to $\mathcal{F}$, which outputs it for `compromised` records.

The changes of key computation of $\mathcal{P}$ in this game are only syntactical because the correct $\mathsf{TestPwd}$ guess guarantees that pw is the input password of $\mathcal{P}$. We hence have

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

Game $\mathbf{G}_5$: Removing two password equality checks from the simulation. We change the functionality to ignore the simulator's output key for honest parties finishing last in honest sessions, with matching passwords, and where both records were `fresh` at the time of output generation (i.e., no $\mathsf{TestPwd}$ was queried by the simulator on any of the two parties). In this case, the functionality outputs the output key of the first party also to the second party.

At the same time we make a similar change in the simulation of $\mathcal{F}_{\mathsf{PAKE}}^{(2)}$, for a corrupt party finishing last in a session with an honest party, where simulator's $\mathsf{TestPwd}$ query to $\mathcal{F}_{\mathsf{PAKE}}$ returned "correct" for that honest party, for Z input by the corrupt party. Instead of comparing the honest party's password with the corrupt party's password, we let the simulator output the key K_2^* that was previously output to the corrupt party.

This and the previous game are not distinguishable by the (trivial) correctness of the protocol and the correctness of $\mathcal{F}_{\mathsf{PAKE}}$. We note that the

correctness of $\mathcal{F}_{\mathsf{PAKE}}$ can be used as an argument only in case of no adversarial interference, as is the case in this game, but does not hold in general as demonstrated by Roy and Xu [RX23]. Hence we have

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

Game $\mathbf{G}_6$: Randomize output keys of interrupted sessions. We change the functionality to ignore the simulator's output keys for `interrupted` sessions and instead output they key stored in the Key record, i.e., a randomly chosen key.

In this game, `interrupted` sessions got marked as such due to either an incorrect password guess, or $\mathcal{Z}$ tampering with the transcript between two honest users. $\mathcal{Z}$ can only notice a difference if it queries what the honest user with the interrupted session outputs, namely $\mathsf{H}_1(\mathsf{sid}, K_{1,2}, K_2, x, y)$. In case of message tampering without an active attack on $\mathcal{F}^{(2)}_{\mathsf{PAKE}}$, this happens at most with probability $q_{\mathsf{H}}/2^\lambda$ because $\mathcal{F}^{(2)}_{\mathsf{PAKE}}$ chooses K_2 uniformly at random and does not leak any information about it since the other party is also honest and outputs a hash $\mathsf{H}_1(\mathsf{sid}, K_{1,2}, K_2, x', y')$ where $(x, y) \neq (x', y')$. In case of an incorrect password guess, $\mathcal{F}^{(2)}_{\mathsf{PAKE}}$ chooses a uniformly random K_2 for the honest party with the interrupted session and outputs it only to that party. Hence, we have

$$|\Pr[\mathbf{G}_6] - \Pr[\mathbf{G}_5]| \leq \frac{q_{\mathsf{H}}}{2^\lambda}.$$

Game $\mathbf{G}_7$: Randomize keys of an honest party finishing first with a fresh record. We change the functionality to ignore the simulator's output keys for an honest $\mathcal{P}$ finishing on a `fresh` session, and instead output to $\mathcal{P}$ the key from the Key record, i.e., a randomly chosen key.

Note that, as of $\mathbf{G}_5$, the functionality aligns the output key in case of matching passwords, which implies that the randomized key chosen of $\mathcal{P}$ in this game might get repeated to the other party.

Because the simulation of $\mathbf{G}_7$ always asks $\mathsf{TestPwd}$ queries if an honest $\mathcal{P}$ either runs with a corrupt party, or with an honest party but is attacked by either a message tampering adversary or an active adversary through $\mathsf{TestPwd}$ on $\mathcal{F}^{(2)}_{\mathsf{PAKE}}$, $\mathcal{P}$ finishing on a fresh record is not under any attack. In $\mathbf{G}_7$, $\mathcal{F}^{(2)}_{\mathsf{PAKE}}$ chooses a fresh K_2 for $\mathcal{P}$ that is only given to the honest $\mathcal{P}'$ (if passwords match; otherwise $\mathcal{P}'$ receives no information about K_2). We hence have

$$|\Pr[\mathbf{G}_7] - \Pr[\mathbf{G}_6]| \leq \frac{q_{\mathsf{H}}}{2^\lambda}.$$

Game $\mathbf{G}_8$: Remove the passwords from the simulation. We modify the simulator to use $\perp$ as password in all $\mathsf{NewSession}$ inputs to the internally

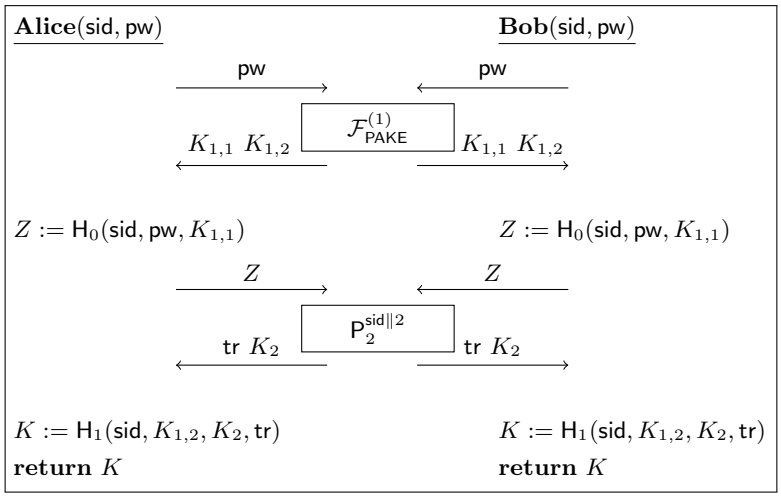


Figure 13: Protocol $\mathsf{SeqComb}[\mathcal{F}_{\mathsf{PAKE}}^{(1)}, \mathsf{P}_2]$ relying on an intact first PAKE protocol.

simulated $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ for honest parties. We let it omit ComputeKey runs on the input passwords of honest parties. At the same time, we change the functionality not to forward input passwords of honest parties to the simulator, but relay $(\mathsf{NewSession}, \mathsf{sid}, \mathcal{P}, \mathcal{P}')$ from $\mathcal{F}$ to $\mathcal{Z}$ as coming from $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$.

As of $\mathbf{G}_7$, outputs are fully determined by $\mathcal{F}$ except for compromised records where $\mathcal{F}$ forwards the key from the simulator. A careful inspection of the simulator code of $\mathbf{G}_8$ shows that it does not use honest parties' input passwords anymore: the protocol transcript is generated without passwords of honest parties as of $\mathbf{G}_2$, with ComputeKey taking extracted passwords as input as of $\mathbf{G}_4$. The internal simulation of $\mathcal{F}_{\mathsf{PAKE}}^{(2)}$ also works without passwords since we replace password equality checks in $\mathbf{G}_5$. We hence have

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

Because forwarding of input passwords and relaying of simulator output keys was the only difference when we introduced $\mathcal{F}$ in $\mathbf{G}_1$, the functionality $\mathcal{F}$ in $\mathbf{G}_8$ is equal to $\mathcal{F}_{\mathsf{PAKE}}$. The theorem thus follows with the simulator depicted in Figures 11 and 12.

Lemma 3. *Let* P_2 *be any PSK equality hiding PAKE protocol. Then protocol* $\mathsf{SeqComb}[\mathcal{F}_{\mathsf{PAKE}}^{(1)}, \mathsf{P}_2]$ *(Figure 13) UC-realizes* $\mathcal{F}_{\mathsf{lePAKE}}$ *in the* $(\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{PAKE}})$*-hybrid model with respect to static party corruptions, where* $\mathsf{H}_0, \mathsf{H}_1$ *are modeled as random oracles (i.e., as calls to* $\mathcal{F}_{\mathsf{RO}}$*), and* $\mathcal{F}_{\mathsf{PAKE}}^{(1)} = \mathcal{F}_{\mathsf{PAKE}}$ *(i.e., the superscript is only added to differentiate between the two instances of* $\mathcal{F}_{\mathsf{PAKE}}$ *in the statement).*

We again make the following simplifying assumptions.

1. $\mathcal{Z}$ corrupts at most one party in a session
2. $\mathcal{Z}$ never issues a $\mathsf{TestPwd}$ query against a corrupt party
3. $\mathcal{Z}$ never issues a $\mathsf{TestPwd}$ query against an honest party if the other party is corrupt

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

Game $\mathbf{G}_1$: Change layout. We change the previous game as follows:

- We move the whole execution into a single machine and call it the simulator Sim.
- We add all the record-keeping of the simulator as in Figure 14.
- In between $\mathcal{Z}$ and Sim we add one dummy party for each real party.
- In between the dummy parties and Sim, we add the ideal functionality $\mathcal{F}_{\mathsf{PAKE}}$ as in Figure 1, but relaying passwords of honest parties to Sim and relaying outputs keys provided by Sim to the dummy parties. We call that functionality $\mathcal{F}$.

The changes are only syntactical since the real execution runs on same inputs and produces outputs the same way as in the previous game. We hence have

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

Game $\mathbf{G}_2$: Extract password from corrupt party We change the simulator upon a corrupt $\mathcal{P}$ sending input $(\mathsf{NewSession}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ to $\mathcal{F}^{(1)}_{\mathsf{PAKE}}$, or upon $\mathcal{Z}$ sending $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}', \mathsf{pw})$. W.l.o.g., $\mathcal{P}'$ from these queries is honest. Upon $\mathsf{TestPwd}$, the simulator immediately forwards the query to $\mathcal{F}$. For the $\mathsf{NewSession}$, the simulator first waits until $\mathcal{P}'$ is supposed to output a key through $\mathcal{F}^{(1)}_{\mathsf{PAKE}}$ (i.e., $\mathcal{Z}$ sends a NewKey query for $\mathcal{P}'$ to $\mathcal{F}^{(1)}_{\mathsf{PAKE}}$), and then sends the password guess to $\mathcal{F}$. If the response from $\mathcal{F}_{\mathsf{PAKE}}$ is "correct", the simulator uses pw in the computation of Z for $\mathcal{P}$.

Because of the input password of $\mathcal{P}$ is the same in the simulation and in $\mathcal{F}$, the switch to pw upon "correct" is only syntactical and we have

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_2].$$

Game $\mathbf{G}_3$: Randomize output key upon message tampering in P_2. We change the simulator to send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \perp)$ to $\mathcal{F}$ in case of $\mathcal{P}, \mathcal{P}'$ both honest and $\mathcal{Z}$ tampering with a message from $\mathcal{P}'$. At the same time, we let the functionality ignore the simulator's output key for $\mathcal{P}$ in case such a $\mathsf{TestPwd}$ happened, and instead take the key from its Key record (i.e., a randomly chosen one).

On $(\mathsf{NewSession}, \mathsf{sid}, \mathcal{P}, \mathcal{P}')$ from $\mathcal{F}_{\mathsf{PAKE}}$
- Record $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \perp)$ and mark it `fresh`
- ($\mathbf{G}_1$) Send $(\mathsf{NewSession}, \mathsf{sid}\|1, \mathcal{P}, \mathcal{P}')$ to $\mathcal{Z}$

On $(\mathsf{NewSession}, \mathsf{sid}\|1, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ from corrupt $\mathcal{P}$ to $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$
- ($\mathbf{G}_1$) Record $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ and mark it `fresh`
- ($\mathbf{G}_1$) Send $(\mathsf{NewSession}, \mathsf{sid}\|1, \mathcal{P}, \mathcal{P}')$ to $\mathcal{Z}$

On $(\mathsf{NewKey}, \mathsf{sid}, \mathcal{P}, K^*)$ from $\mathcal{Z}$ to $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$
// Follow the logic of the $\mathcal{F}_{\mathsf{PAKE}}$ NewKey interface to compute the output key of P_2:
- ($\mathbf{G}_1$) Retrieve $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}, [\mathcal{P}'], [\mathsf{pw}])$ with mark $m \neq$ `completed`
- ($\mathbf{G}_1$) If $m =$ `compromised`: $K_{1,1}, K_{1,2} := K^*$
- ($\mathbf{G}_8$) Elif $m =$ `fresh` AND $\exists(\mathsf{P1Key}, \mathsf{sid}, \mathcal{P}', [K'], \mathtt{fresh})$: // $\mathcal{P}$ finishes last
  - If $\exists(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}', \mathsf{pw}, \mathtt{correct})$: $K_{1,1}, K_{1,2} := K'$ // $Z \neq \perp \Rightarrow \mathcal{P}$ corrupt $\Rightarrow$ w.l.o.g, $\mathcal{P}'$ is honest, and we already ran TestPwd when $\mathcal{P}'$ finished.
  - ($\mathbf{G}_1$) Elif $\mathcal{P}$ and $\mathcal{P}'$ are honest: $K_{1,1}, K_{1,2} := \perp$ // $\mathcal{F}_{\mathsf{PAKE}}$ controls the final K anyway
  - ($\mathbf{G}_1$) Else: $K_{1,1}, K_{1,2} \leftarrow\!\!\$\ \{0,1\}^\lambda$ // $\mathcal{P}$ corrupt, wrong password
- ($\mathbf{G}_1$) Elif $\exists(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, \cdot)$ then set $K_{1,1}, K_{1,2} \leftarrow\!\!\$\ \{0,1\}^\lambda$ // $\mathcal{P}$ finishes first or interrupted, or $\mathcal{P}'$ finished non-fresh.
- ($\mathbf{G}_1$) Else ignore the query // Isolated $\mathcal{P}$ does not output a key
- ($\mathbf{G}_2$) If $\mathsf{P1Session}$ is `fresh`, $\mathcal{P}$ is honest and $\mathcal{P}'$ is corrupt: // Now that TestPwd can no longer be called on this session, we can finally call it
  - ($\mathbf{G}_2$) Retrieve $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}', \mathcal{P}, [\mathsf{pw}'])$
  - ($\mathbf{G}_2$) Run code of $(\mathsf{TestPwd}, \mathsf{sid}\|1, \mathcal{P}, \mathsf{pw}')$
- ($\mathbf{G}_1$) Record $(\mathsf{P1Key}, \mathsf{sid}, \mathcal{P}, K_{1,1}, K_{1,2}, m)$
- ($\mathbf{G}_1$) Mark the $\mathsf{P1Session}$ `completed`
- If $\mathcal{P}$ is corrupt:
  - ($\mathbf{G}_1$) Send $(\mathsf{sid}, K_{1,1}, K_{1,2})$ to $\mathcal{P}$
  - ($\mathbf{G}_1$) Return
- Else: // Now compute Z for honest $\mathcal{P}$ if we can, and run the rest of the protocol
  - ($\mathbf{G}_2$) If $\exists(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}, [\mathsf{pw}'], \text{“correct”})$: $Z := \mathsf{H}_0(\mathsf{sid}, \mathsf{pw}', K_{1,1})$ // $K_{1,1} \neq \perp$ because it's only $\perp$ when both parties are honest and $\mathcal{P}$'s session is uncompromised. But when that's the case, $(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}, \ldots)$ doesn't exist
  - ($\mathbf{G}_9$) & ($\mathbf{G}_{10}$) Else: $Z \leftarrow\!\!\$\ \{0,1\}^\lambda$
  - ($\mathbf{G}_1$) Store $(\mathsf{P2Input}, \mathsf{sid}, \mathcal{P}, Z)$
  - ($\mathbf{G}_1$) Run P_2 on behalf of $\mathcal{P}$ on input Z

On $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ from $\mathcal{Z}$ to $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ or as internal call // W.l.o.g., $\mathcal{P}$ is honest
- ($\mathbf{G}_2$) Retrieve $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}, \cdot, \cdot)$ marked `fresh`
- ($\mathbf{G}_2$) Send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{PAKE}}$, get result b, and mark $\mathsf{P1Session}$ accordingly
- ($\mathbf{G}_2$) Record $(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw}, b)$
- ($\mathbf{G}_2$) Return b

On $\mathcal{Z}$ sending the final message in P_2 to honest $(\mathsf{sid}, \mathcal{P})$
- ($\mathbf{G}_1$) Ignore if simulated $\mathcal{P}$ is not ready to receive the final P2 message
- ($\mathbf{G}_1$) Retrieve $(\mathsf{P1Key}, \mathsf{sid}, \mathcal{P}, \cdot, [K_{1,1}, K_{1,2}], \cdot)$ // Exists because $\mathcal{P}$ starts P2 after P1Key was recorded
- ($\mathbf{G}_1$) If $K_{1,1}, K_{1,2} \neq \perp$: // If we can compute K_2 and K then do so
  - ($\mathbf{G}_1$) Compute P_2 protocol output tr, K_2 on behalf of $\mathcal{P}$
  - ($\mathbf{G}_1$) Record $(\mathsf{P2Key}, \mathsf{sid}, \mathcal{P}, K_2, \mathsf{tr})$
  - ($\mathbf{G}_1$) Compute $K := \mathsf{H}_1(\mathsf{sid}, K_{1,2}, K_2, \mathsf{tr})$
- Else: $K := \perp$ // P1 is uncompromised and interacting with honest party: Sim's key will be ignored.
- ($\mathbf{G}_3$) If $\exists$ record $(\mathsf{P2Key}, \mathsf{sid}, \mathcal{P}', *, [\mathsf{tr}'])$ with $\mathsf{tr} \neq \mathsf{tr}'$, send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \perp)$ to $\mathcal{F}_{\mathsf{PAKE}}$ // DoS on P_2
- ($\mathbf{G}_1$) Send $(\mathsf{NewKey}, \mathsf{sid}, \mathcal{P}, K)$ to $\mathcal{F}_{\mathsf{PAKE}}$

Figure 14: Simulator for $\mathsf{SeqComb}[\mathcal{F}_{\mathsf{PAKE}}^{(1)}, \mathsf{P}_2]$ realizing $\mathcal{F}_{\mathsf{PAKE}}$, Lemma 3. Random oracles $\mathsf{H}_0, \mathsf{H}_1$ are simulated as in the real execution.

In an honest interaction, the only values depending on key $K_{1,2}$ are the outputs of the two honest parties, which because of the inclusion of tr in the final hash are different. Hence, this and the previous game are equally distributed unless $\mathcal{Z}$ queries $(\mathsf{sid}, K_{1,2}, K_2, \mathsf{tr})$ to H_1 which happens only with probability $1/2^\lambda$ for each H_1 query of $\mathcal{Z}$, because $K_{1,2}$ is chosen uniformly at random. Hence we have

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq \frac{q_\mathsf{H}}{2^\lambda}.$$

Game $\mathbf{G}_4$: Abort if $\mathcal{Z}$ queries a secret P_1 output key to H_0. We abort the simulation if $\mathcal{Z}$ queries $\mathsf{H}_0(\mathsf{sid}, \cdot, K_{1,1})$ for a $K_{1,1}$ that $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ generated as output to an honest party $\mathcal{P}$, except when $\mathcal{P}$ is attacked with a correct password guess.

Because $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ generates output keys uniformly at random and, unless $\mathcal{P}$ is subject to a successful active attack, these output keys are not given to anybody else than the honest $\mathcal{P}$, the probability for the abort to happen is $1/2^\lambda$ per hash query, and hence we have

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_3]| \leq \frac{q_\mathsf{H}}{2^\lambda}.$$

Game $\mathbf{G}_5$: Randomize output keys of interrupted sessions. We change $\mathcal{F}$ to ignore the simulation's ouput keys for interrupted sessions and instead output the key from the Key record. This means that honest parties who received wrong password guesses now obtain a fresh random key from $\mathcal{F}$.

The argument is the same as in $\mathbf{G}_4$, this time relying on the negligible probability of $\mathcal{Z}$ querying $K_{1,2}$ to H_1. We again have

$$|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| \leq \frac{q_\mathsf{H}}{2^\lambda}.$$

Game $\mathbf{G}_6$: Functionality aligns keys In this game we let $\mathcal{F}$ output the same key to the party who finishes last in a fresh session, with matching passwords, and thereby ignore the output key provided by the simulator.

Due to the (trivial) correctness of the protocol and of $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ in an unattacked session, the changes are only syntactical and to not affect the output distribution. We have

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

Game $\mathbf{G}_7$: Randomize output keys parties finishing first in fresh sessions We change $\mathcal{F}$ to ignore the simulator's output keys for honest parties finishing first on a fresh session, and instead output the key from the Key record. I.e., these parties get a fresh uniform key generated by $\mathcal{F}$ now.

Because of games $\mathbf{G}_2$ and $\mathbf{G}_3$, the simulator queries $\mathsf{TestPwd}$ for honest parties under attack, and hence the fresh sessions modified in this game are guaranteed to belong to honest parties that are running with another honest party and an untampered transcript. Hence, $K_{1,2}$ chosen uniformly by $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ are only given to the honest parties. $\mathcal{Z}$ can only notice a difference in the output distribution of $\mathbf{G}_7$ and $\mathbf{G}_6$ if it queries $K_{1,2}$ to H_1, which happens with negligible probability

$$|\Pr[\mathbf{G}_7] - \Pr[\mathbf{G}_6]| \leq \frac{q_{\mathsf{H}}}{2^{\lambda}}.$$

Game $\mathbf{G}_8$: Remove password equality check from simulation. We change the simulation to replace the password equality check in the NewKey interface of the internally simulated $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$: instead of comparing the input password of a finishing corrupt $\mathcal{P}$ with its honest counterparty's password, the simulator compares with the correct password guess against the counterparty (if it happened).

Because $\mathsf{TestPwd}$ compares a password guess against an input password, the changes are only syntactical and we have

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

Game $\mathbf{G}_9$: Randomize Z of honest unattacked parties with matching passwords. We change the simulation to use a randomly chosen $Z \leftarrow\!\!\$\ \{0,1\}^{\lambda}$ as a password in P_2 both honest parties if they have matching passwords and are not attacked, i.e., parties that received a session key from $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ while their session at $\mathcal{F}$ was marked `fresh`.

A distinguisher between this and the previous game breaks the pre-shared key equality hiding property of P_2. To see this, in $\mathbf{G}_8$ both parties were running on the same Z, which was randomly chosen by the random oracle. In this game, both parties are running on randomly chosen Z values. Further, $\mathcal{Z}$ does not see the output keys K_2 since we randomized outputs of honest parties in $\mathbf{G}_7$. We hence have

$$|\Pr[\mathbf{G}_9] - \Pr[\mathbf{G}_8]| \leq \mathsf{Adv}_{\mathsf{P}_2}^{\mathsf{PEH}}.$$

Game $\mathbf{G}_{10}$: Randomize Z of honest attacked parties and honest parties with mismatching passwords. We now change the simulation to use a random Z for an honest $\mathcal{P}$ in all other cases except if $\mathcal{P}$'s session in $\mathcal{F}$ is `compromised`, which the simulator can determine via the existence of a $(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}, \cdot, \texttt{correct})$ entry.

In all these cases, $\mathcal{F}_{\mathsf{PAKE}}^{(1)}$ outputs a key $K_{1,1}$ to $\mathcal{P}$ that is not given to anybody else. Hence, $\mathcal{Z}$ can only notice a difference if it queries $K_{1,1}$ to H_0. We hence have

$$|\Pr[\mathbf{G}_{10}] - \Pr[\mathbf{G}_9]| \leq \frac{q_{\mathsf{H}}}{2^\lambda}.$$

We are now in a situation where, in the simulation, honest parties whose sessions are not marked `compromised` in $\mathcal{F}$ no longer use $K_{1,1}$ to compute Z. The output keys of these parties are determined by $\mathcal{F}$ and the corresponding output keys produced by the simulation are ignored by $\mathcal{F}$. Hence, the whole execution does not depend on the input passwords of honest parties anymore, and we can replace them by dummy values in the next game.

Game $\mathbf{G}_{11}$: Remove passwords from simulation. We let the simulator use $\perp$ as input password for honest parties, and change $\mathcal{F}$ to not forward passwords of honest parties to the simulator anymore.

As argued above, the changes go unnoticed by $\mathcal{Z}$ because the distribution does not depend on the honest parties' passwords in the simulation as of $\mathbf{G}_{10}$. We hence have

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

Because forwarding of input passwords and relaying of simulator output keys was the only difference when we introduced $\mathcal{F}$ in $\mathbf{G}_1$, the functionality $\mathcal{F}$ in $\mathbf{G}_{11}$ is equal to $\mathcal{F}_{\mathsf{PAKE}}$. The theorem thus follows with the simulator depicted in Figure 14.

A.3 Proof of Lemma 1

The overall idea of the proof is to let $\mathcal{F}_{\mathsf{rlePAKE}}$ handle $\mathsf{LateTestPwd}$ queries by $\mathcal{Z}$. However, $\mathcal{Z}$ issues those queries against the first sub-PAKE, i.e., $\mathcal{F}^{(1)}_{\mathsf{lePAKE}}$, while the simulator has a $\mathsf{LateTestPwd}$ interface at $\mathcal{F}_{\mathsf{rlePAKE}}$ which handles the overall key exchange. This causes a slight timing issue: if P_1 completed but P_2 is still ongoing, $\mathcal{Z}$ can issue a late password guess against P_1 while $\mathcal{F}_{\mathsf{rlePAKE}}$ would reject such a query. The solution is to let our simulator translate late password guesses against P_1 to online guesses against $\mathcal{F}_{\mathsf{rlePAKE}}$ while the overall key exchange is not yet completed.

The proof is derived from the proof of Lemma 3 by changing all occurrences of $\mathcal{F}^{(1)}_{\mathsf{PAKE}}$ to $\mathcal{F}^{(1)}_{\mathsf{lePAKE}}$, changing all occurrences of $\mathcal{F}_{\mathsf{PAKE}}$ to $\mathcal{F}_{\mathsf{rlePAKE}}$, and modifying the games as follows.

- **Registering lazy extractions at $\mathcal{F}_{\mathsf{rlePAKE}}$.** We add a new game $\mathbf{G}_{2.1}$ right after $\mathbf{G}_2$ where we change the simulation to forward a $(\mathsf{RegisterTest}, \mathsf{sid}, \mathcal{P})$ query to $\mathcal{F}_{\mathsf{rlePAKE}}$, but only at the very last moment before $\mathcal{P}$ is producing an output key. That is, we mark the $\mathsf{P1Session}$ of $\mathcal{P}$ with a `tested` flag and, right before sending $(\mathsf{NewKey}, \mathsf{sid}, \mathcal{P}, \cdot)$ to $\mathcal{F}_{\mathsf{rlePAKE}}$, the simulator sends $(\mathsf{RegisterTest}, \mathsf{sid}, \mathcal{P})$ to $\mathcal{F}_{\mathsf{rlePAKE}}$. Because the effect of $\mathsf{RegisterTest}$ is an `interrupted` record and these markings do not yet affect any output keys in this game, the changes do not affect the output distribution and the new game $\mathbf{G}_{2.1}$ is equally distributed to $\mathbf{G}_2$.

- **Answering "online" LateTestPwd queries without passwords.** We add a new game $\mathbf{G}_{2.2}$ right after $\mathbf{G}_{2.1}$ where we change the simulation in case $\mathcal{Z}$ sends $(\mathsf{LateTestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ for a P1Session that is completed and flagged `tested`, but where $\mathcal{P}$ has not yet produced an output key. The simulator sends $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{rlePAKE}}$. Upon "correct", it returns the P1Key record $K_{1,1}, K_{1,2}$ to $\mathcal{Z}$, otherwise a random key. Because the passwords in both $\mathcal{F}_{\mathsf{rlePAKE}}$ and $\mathcal{F}_{\mathsf{lePAKE}}^{(1)}$ match, the changes are only syntactical and $\mathbf{G}_{2.1}$ and $\mathbf{G}_{2.2}$ are equally distributed.

- **Answering "completed" LateTestPwd queries without passwords.** We add a new game $\mathbf{G}_{2.3}$ right after $\mathbf{G}_{2.2}$ where we change the simulation in case $\mathcal{Z}$ sends $(\mathsf{LateTestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw})$ for a $\mathcal{P}$ that has a completed P1 session flagged `tested`, and that has already produced an output. In case the guess comes back correct with a key $K \neq \perp$, the simulator programs $\mathsf{H}_0[\mathsf{sid}, \mathsf{pw}^*, K_{1,1}] := Z$ and $\mathsf{H}_1[\mathsf{sid}, K_{1,2}, K_2, \mathsf{tr}] := K$ for $K_{1,1}, K_{1,2}, Z, K_2$ from the simulation of $\mathcal{P}$. The changes are only syntactical because the simulator re-programs exactly what $\mathcal{P}$ already computed.

- **Randomize output keys of interrupted sessions.** This is the original $\mathbf{G}_5$, where we additionally let the simulator skip the output key computation of interrupted sessions. Because LateTestPwd queries also result in interrupted records, these output keys are also randomized by this game. The effect is that LateTestPwd responses are now also random keys chosen by $\mathcal{F}_{\mathsf{rlePAKE}}$, and hence the programming of the simulator in $\mathbf{G}_{2.3}$ becomes crucial as it reinstalls consistency between the simulated protocol values for $\mathcal{P}$ and its $\mathcal{F}_{\mathsf{rlePAKE}}$-determined output key. Hence, the switch to different keys goes unnoticed by the environment except with negligible probability as argued in $\mathbf{G}_5$, despite the additional interrupted records through the LateTestPwd queries.

On $\mathcal{Z}$ sending the final message in P_2 to honest $(\mathsf{sid}, \mathcal{P})$

- Ignore if simulated $\mathcal{P}$ is not ready to receive the final P2 message
- Retrieve $(\mathsf{P1Key}, \mathsf{sid}, \mathcal{P}, \cdot, [K_{1,1}, K_{1,2}], \cdot)$ // Exists because $\mathcal{P}$ starts P2 after P1Key was recorded
- If $K_{1,1}, K_{1,2} \neq \perp$: // If we can compute K_2 and K then do so
  - Compute P_2 protocol output tr, K_2 on behalf of $\mathcal{P}$
  - Record $(\mathsf{P2Key}, \mathsf{sid}, \mathcal{P}, K_2, \mathsf{tr})$
  - Compute $K := \mathsf{H}_1(\mathsf{sid}, K_{1,2}, K_2, \mathsf{tr})$
- Else: $K := \perp$ // P1 is uncompromised and interacting with honest party: Sim's key will be ignored.
- If $\exists$ record $(\mathsf{P2Key}, \mathsf{sid}, \mathcal{P}', \cdot, [\mathsf{tr}'])$ with $\mathsf{tr} \neq \mathsf{tr}'$, send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \perp)$ to $\mathcal{F}_{\mathsf{rlePAKE}}$ // DoS on P_2
- ($\mathbf{G}_{2.1}$) If $\mathcal{P}$ is honest AND P1Session is flagged `tested` AND $\nexists(\mathsf{TestedPwd}, \mathsf{sid}, \mathcal{P}, \ldots)$: send $(\mathsf{RegisterTest}, \mathsf{sid}, \mathcal{P})$ to $\mathcal{F}_{\mathsf{rlePAKE}}$ // Forward the RegisterTest at the last possible moment
- Send $(\mathsf{NewKey}, \mathsf{sid}, \mathcal{P}, K)$ to $\mathcal{F}_{\mathsf{rlePAKE}}$

On $(\mathsf{RegisterTest}, \mathsf{sid}\|1, \mathcal{P})$ from $\mathcal{Z}$ to $\mathcal{F}^{(1)}_{\mathsf{lePAKE}}$

- ($\mathbf{G}_{2.1}$) Retrieve $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}, \cdot, \cdot)$ marked `fresh`
- ($\mathbf{G}_{2.1}$) Mark it `interrupted` and flag it `tested`
- // Defer forwarding the RegisterTest until right before we do NewKey

On $(\mathsf{LateTestPwd}, \mathsf{sid}\|1, \mathcal{P}, \mathsf{pw}^*)$ from $\mathcal{Z}$ to $\mathcal{F}^{(1)}_{\mathsf{lePAKE}}$ // W.l.o.g., $\mathcal{P}$ is honest

- ($\mathbf{G}_1$) Retrieve $(\mathsf{P1Session}, \mathsf{sid}, \mathcal{P}, \cdot, \cdot)$ marked `completed` with flag `tested`
- ($\mathbf{G}_1$) Remove the flag `tested`
- ($\mathbf{G}_1$) Retrieve $(\mathsf{P1Key}, \mathsf{sid}, \mathcal{P}, [K_{1,1}, K_{1,2}], [m])$ // $K_{1,1}, K_{1,2}$ can't be $\perp$ because that only happens for untested sessions
- ($\mathbf{G}_{2.2}$) If $m =$ `interrupted`: // TestPwd was not queried on this session yet. It was marked interrupted by RegisterTest, so the normal TestPwd points were never triggered
  - ($\mathbf{G}_{2.2}$) Send $(\mathsf{TestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{rlePAKE}}$, get result b
  - ($\mathbf{G}_{2.2}$) If "correct": return $K_{1,1}, K_{1,2}$
  - ($\mathbf{G}_{2.2}$) Else: return $K'_1 \leftarrow\!\!\$\ \{0,1\}^{2\lambda}$
- Else: // The session ended. Need to make it such that $K_{1,2}$ explains the final K
  - ($\mathbf{G}_{2.3}$) Retrieve $(\mathsf{P2Key}, \mathsf{sid}, \mathcal{P}, K_2, \cdot)$
  - ($\mathbf{G}_{2.3}$) Retrieve $(\mathsf{P2Input}, \mathsf{sid}, \mathcal{P}, Z)$
  - ($\mathbf{G}_{2.3}$) Send $(\mathsf{LateTestPwd}, \mathsf{sid}, \mathcal{P}, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{rlePAKE}}$ and get K
  - ($\mathbf{G}_{2.3}$) If $K = \perp$ return $K'_1 \leftarrow\!\!\$\ \{0,1\}^{2\lambda}$
  - Else:
    - ($\mathbf{G}_{2.3}$) Set $\mathsf{H}_0[\mathsf{sid}, \mathsf{pw}^*, K_{1,1}] := Z$ //Program the P2 input to the correct password.
    - ($\mathbf{G}_{2.3}$) Set $\mathsf{H}_1[\mathsf{sid}, K_{1,2}, K_2, \mathsf{tr}] := K$ // Program the final hash to the key K previously output by $\mathcal{P}$.
    - ($\mathbf{G}_1$) Return $K_{1,1}, K_{1,2}$

Figure 15: Simulator for Lemma 1 in terms of added/modified interfaces from Figure 14.

B Formal distinguisher for SeqComb with lazy extraction

To make $\mathsf{SeqComb}$ work with lazy extraction PAKEs, we would need to prove the following statement: if P_1 UC-realizes $\mathcal{F}_{\mathsf{lePAKE}}$ and P_2 is PSK equality hiding, then P UC-realizes $\mathcal{F}_{\mathsf{lePAKE}}$ (in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model). In this section we give a formal distinguisher for this statement.

The PSK equality hiding property does not rule out offline attacks against P_2. We use that to distinguish the protocol from $\mathcal{F}_{\mathsf{lePAKE}}$.

1. $\mathcal{Z}$ starts honest $\mathcal{P}$ with pw, sends $\mathsf{RegisterTest}$ to $\mathcal{F}^{(1)}_{\mathsf{lePAKE}}$ and completes $\mathcal{P}$'s session to receive output key K.
2. $\mathcal{Z}$ flips a coin b and sends pw_b as $\mathsf{LateTestPwd}$ guess to $\mathcal{F}^{(1)}_{\mathsf{lePAKE}}$, where $\mathsf{pw}_0 = \mathsf{pw}$ and pw_1 is a different password.
3. $\mathcal{Z}$ receives key $K_{1,1}, K_{1,2}$ from $\mathcal{F}^{(1)}_{\mathsf{lePAKE}}$
4. $\mathcal{Z}$ tests whether $H_0(\mathsf{sid}, \mathsf{pw}_b, K_{1,1})$ was used by $\mathcal{P}$ in P_2.
   - If $b = 0$ $\mathcal{Z}$ expects that test to pass
   - If $b = 1$ $\mathcal{Z}$ expects that test to fail.

A simulator Sim can react in the following way to this $\mathcal{Z}$

1. Sim has no information about $\mathcal{P}$'s password, so it chooses a random Z to run P_2
2. Sim submits pw_b via $\mathsf{LateTestPwd}$ to $\mathcal{F}_{\mathsf{lePAKE}}$ and receives back K_b, where $K_b = K$ for $b = 0$ and random otherwise.
3. Sim chooses random $K_{1,1}, K_{1,2}$ to reply to $\mathcal{Z}$.
4. Sim obtains query $H_0(\mathsf{sid}, \mathsf{pw}_b, K_{1,1})$ and can now program it:
   - If Sim programs the hash to Z and $b = 1$, the test passes—$\mathcal{Z}$ can distinguish
   - If Sim programs the hash to a different Z' and $b = 0$, the test fails—$\mathcal{Z}$ can distinguish