More Salt, Please: A Simple Compiler for Post-Quantum Strong aPAKE

William Duro[⋆,1,2][0009-0008-5313-3743], Sebastian Faller[1,2][0009-0005-4126-3098], Julia Hesse[⋆,1][0000-0002-2875-6198] and Máté Horváth[⋆⋆,3,4][0000-0001-9512-5196]

[1] IBM Research Europe, [2] ETH Zurich, [3] Bergische Universität Wuppertal, [4] Budapest University of Technology and Economics
william.duro@ibm.com, sebastian.faller@ibm.com, jhs@zurich.ibm.com, mhorvath@crysys.hu

Abstract. Resistance to precomputation attacks is a central security goal in password-based cryptography, as it drastically increases the runtime of brute-force attacks on breached password hashes. Strong asymmetric password-authenticated key exchange (saPAKE), which enforces such precomputation resistance, is standardized by the IRTF and deployed today at large scale. Unfortunately, no practical and easy-to-deploy saPAKE exists yet that withstands quantum attackers. Current constructions use rate-limiting tools such as oblivious pseudo-random functions (OPRFs), of which post-quantum versions are significantly more costly and complex than their discrete-log based counterparts.
In this paper, we take a new approach to precomputation resistance. Starting from the compilers from PAKE to aPAKE by Gentry-MacKenzie-Ramzan (Crypto 2006) and Liu-Lyu-Han (Asiacrypt 2024), we propose a *password file ratchet* that adds precomputation resistance to these compilers through continuously updating the password file. The update is a simple salted hash. With a variety of practical post-quantum PAKEs at hand, this yields a ready-to-deploy post-quantum saPAKE at the trade-off of a changing server state.

Keywords: Password-Authenticated Key Exchange · Post-Quantum Cryptography · Universal Composability

1 Introduction

Password-authenticated key exchange (PAKE) has been studied for decades to enable two parties to establish a session key and mutually authenticate using

⋆ The author was supported by the Swiss National Science Foundation (SNSF) under the Starting grant "Low-Entropy Cryptography for the Quantum Era"
⋆⋆ The author was supported by the German Research Foundation (DFG), project JA2445/6-1.

human-memorable secrets over an insecure channel [10,9,15,17,35]. In the client-server communication setting, asymmetric PAKE (aPAKE) [30] protects client passwords against server compromise by requiring that the server does not store cleartext passwords, but an image of them under some one-way function. We usually refer to this as *password file*. Later, Jarecki et al. [40] identified a gap in the aPAKE security definition [30,37] that permits precomputation of the password file already before a server breach (or *compromise*). For weak passwords, such an attack can lead to almost immediate password leakage after the compromise, eliminating the additional protection that aPAKE is intended to provide compared to PAKE. Therefore, [40] proposed a new notion, called strong aPAKE (saPAKE) that also protects against precomputation attacks and which has become the prevailing security notion for PAKE in the client-server setting.

Interestingly, all saPAKE constructions in the literature are either built from oblivious pseudorandom functions (OPRFs) [18], or they rely on assumptions that imply OPRFs.[1] Intuitively, OPRFs turn the computation of $H(\mathsf{pw})$ into an interactive computation, such that for every password guess an interaction with the server is required. This rate-limiting is what gives us precomputation protection.

The challenge of post-quantum security. Providing password-authenticated keys used for data encryption and user authentication, saPAKEs must protect against quantum attackers. While OPRFs, especially the so-called 2HashDH OPRF [39,21], enable very efficient saPAKE realizations, the common building block turns into a bottleneck when aiming for post-quantum (PQ) security. This has multiple reasons. PQ-secure OPRF proposals [2,26,3,36,7,28,1,11,56] are currently inefficient compared to the classical ones. While there are efficiency improvements for achieving better amortized costs for applications like private set intersection [56,25], in case of PAKE these are not applicable because one key exchange typically involves only one OPRF evaluation. Currently, this takes at least roughly 185 ms and 750 KB of communication [11] in the PQ setting, but only $0,5$ms and 64 bytes classically with the 2HashDH OPRF [39]. Maybe an even bigger problem is that PQ-secure OPRFs are fairly complex and often rely on less established assumptions, lowering confidence in their security. Indeed, the isogeny-based OPRF of [13] was affected by SIDH attacks [19,44,50]. Finally, hybridization of OPRF-based protocols is challenging, since black-box OPRF combiners are impossible [27]. Hence, a faulty post-quantum OPRF combined with 2HashDH could actually void the security of 2HashDH.

Altogether, the road towards PQ saPAKE via OPRFs appears to be long and rocky. We hence ask the following question:

Is there a different approach to constructing saPAKE that avoids OPRFs and uses only primitives that we know how to instantiate post-quantum?

[1] To the best of our knowledge, the only saPAKE not relying on OPRFs is built from group actions [45] which can also be used to construct threshold OPRFs [49].

Our contributions. In this work, we answer this question in the affirmative. We build on a line of work by Gentry et al. [30] and Lyu et al [43] who construct aPAKE by first running PAKE on the hashed password, and then letting the client prove password knowledge over the PAKE channel. We generalize and augment their works by proposing a black-box compiler that turns *a PAKE into a strong aPAKE*. First, we abstract the notion of an *interactive proof of password knowledge* (wPoPWK) and investigate what properties such a proof must have for the compiler to be secure. Second, we augment the PAKE+wPoPWK framework with *only symmetric primitives* to protect against precomputation attacks. With practical and simple post-quantum PAKEs available [5,6] and a wPoPWK notion that can be instantiated with ML-KEM [47], we obtain a post-quantum secure saPAKE.[2]

At first sight, since we use only symmetric primitives to add strongness to the PAKE-to-aPAKE blueprint, one might think that the gap between aPAKE and saPAKE is smaller than what post-quantum migration attempts for saPAKE hint at. However, we do enter some trade-offs in order to achieve a simple solution. First, our cheap precomputation protection requires *updating* the password file stored at the server, which might be a roadblock in some deployments. Second, in order to allow users to exchange keys from updated files, information from each update must be sent to the client, which results in a growing communication complexity. Third, precomputation protection does not hold *during* a protocol run; it only holds as soon as the file is updated, which happens only after the key exchange is completed.

In order to mitigate the effects of these trade-offs, we provide a mechanism for upper bounding the message and server state size by the number of consecutive wrong password guesses, and we argue that standard techniques can be applied to further compress them.

1.1 Technical overview of our saPAKE compiler

Motivated by the recent advances in the design of PQ-secure PAKE [4], our goal is to turn any symmetric PAKE into a strong asymmetric PAKE in a modular way, while only relying on primitives that have easy to deploy, PQ-secure instantiations. We take inspiration from existing PAKE-to-aPAKE transformations [30,38,43,32]. We modularize their joint approach of sending a proof of password knowledge protected by a PAKE key, and extend it such that the resulting protocols have built-in resistance against precomputation attacks, a.k.a., they are strong aPAKEs.

[2] We note that our security analysis considers classical adversaries and the classical random oracle model (ROM). We use the term post-quantum to refer to the underlying computational assumptions, which are currently not known to be efficiently solvable by either classical or quantum algorithms. We do not claim post-quantum security of the protocols themselves, and leave a formal analysis against quantum adversaries, including in the quantum random oracle model (QROM), for future work.

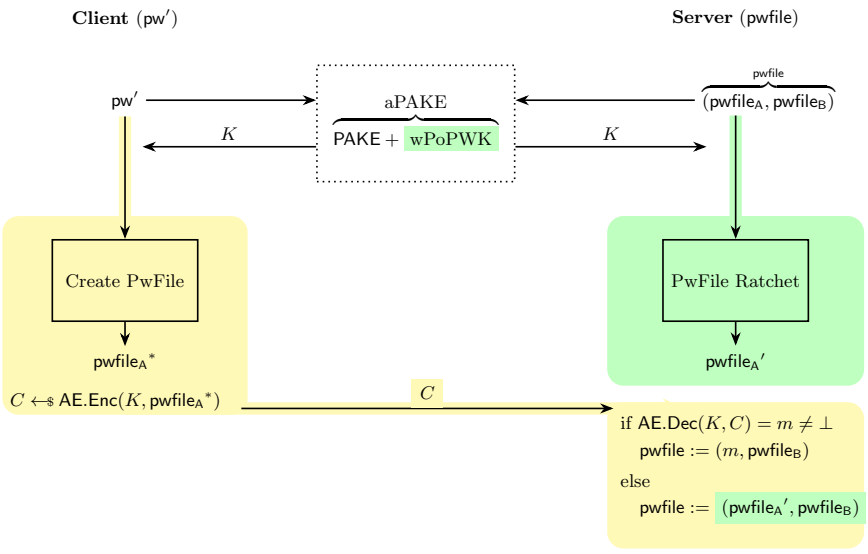

Fig. 1. The high-level idea of our saPAKE construction. Yellow background indicates techniques from updatable PAKE by Dayanikli et al. [24], while green highlights the ratchet we wrap it with. The aPAKE appearing in the illustration is implicit in our PAKE-to-saPAKE transformation.

First, we observe that the existing PAKE-to-aPAKE methods all follow the same approach. On the server side, a so-called password file pwfile is stored instead of the plaintext password pw. This is some one-way function of the password preventing immediate leakage upon server compromise. If pwfile is a password hash $H(\mathsf{pw})$, the client can simply hash their password and run a symmetric PAKE on "password" $H(\mathsf{pw})$ with the server. However, this is not yet an aPAKE because of the following challenge of the asymmetric setting. At this point, it is still possible that an adversary can successfully exchange a key without knowing the right password, if earlier it has obtained leaked password files from a compromised server – since these now contain the PAKE "passwords". To prevent this attack, previous works forced the client to compute some function of the password attempt that the server can verify using some check value added to the password file, which we denote with $\mathsf{pwfile}_\mathsf{B}$. Intuitively, this requires the usage of asymmetric cryptography, since $\mathsf{pwfile}_\mathsf{B}$ should not enable a client to actually demonstrate knowledge of a password, and indeed, [30,43] build it from signatures and KEMs, respectively. Our isolation of the proof component enables modular replacement and analysis, which makes the transition to post-quantum easier. For example, with ML-KEM being more mature than post-quantum PAKEs where standards are still in the drafting stage, one could already deploy a post-quantum proof while still going with a classical PAKE that is replaced later.

Our second goal is to upgrade the compiler's output to be a *strong* aPAKE, and we depict the high level idea in Figure 1. To achieve strongness, we have to ensure that all password dependent values, stored by the server also have to depend on high-entropy values known only to the server. The first crucial observation is that the $\mathsf{pwfile}_\mathsf{B}$ part can be left unprotected. This is because the wPoPWK transcript is protected by the PAKE key. Therefore, any information leaked from the server through the interaction is either learned by someone who knows the password (which is fine) or who already compromised the server and got through password authentication with the help of a leaked pwfile, i.e. the time for precomputation is already over.

The more challenging part is hence to protect $\mathsf{pwfile}_\mathsf{PAKE}$ of an aPAKE from precomputation attacks. A naive attempt is to use a salted password hash in $\mathsf{pwfile}_\mathsf{PAKE}$. Dayanikli and Lehmann [23] observed that in this case, aPAKE security extends with precomputation resistance until the first key exchange, when the server has to reveal the salt value (to enable the client to prepare its PAKE input). Once the adversary learns the salt, $\mathsf{pwfile}_\mathsf{PAKE}$ becomes precomputable. Indeed, existing saPAKE constructions do not reveal the server entropy, such as a secret OPRF key, during a key exchange.

We take a different approach and relax the "all time" protection to "protection with interruptions". Concretely, our goal is to limit the security degradation caused by salt disclosure *to the duration of a session*, that is short enough to make any precomputation meaningless. For this, we need an update mechanism that compensates the information disclosure and restores precomputation resistance after the session is finished. We propose a *password file ratchet* (see the green part of Figure 1) that takes a password file and fresh randomness, and returns a new password file for the same password. Crucially, this mapping is one-way, so knowing the ratchet's output, it remains hard to find out the input password file. One-wayness guarantees that even if protocol messages enable precomputation of some password-related value stored on the server, if the server gets compromised after the session, the updated password file already hides the previously precomputable values.

The idea of our ratchet is surprisingly simple. Let us define $\mathsf{pwfile}_\mathsf{A} = (H(\mathsf{pw}, s), s)$. We can re-randomize $\mathsf{pwfile}_\mathsf{A}$ using a fresh salt s' by computing the salted hash of the first file element and concatenating s' to the second file element, which can be thought of now as a salt list, i.e., $\mathsf{pwfile}_\mathsf{A}{}' := (H(\mathsf{pwfile}_1, s'), \mathsf{pwfile}_2 \| s')$. There are two things to check:

1. Security: the ratchet has to be one-way;
2. Efficiency: the iterative usage of the ratchet has to be space and time efficient;

It is easy to see that the above ratchet achieves its security goal if we assume that the used hash function is preimage resistant (that is implied by the random oracle model (ROM), that we use). Also note that having access to pwfile', it is not easier to learn pwfile_1 than successfully offline guessing pw.

Turning towards efficiency, our ratchet causes every file update, which we need in all protocol runs, to increase the file and server-to-client message sizes because of the growing salt list. Fortunately, we can leverage the interaction with

honest clients to circumvent ever growing salt lists. Dayanikli and Lehmann [24] observed that since successful authentication implies correct password knowledge, the client can recompute its pwfile_A with fresh salt and the same password. Sending the new file over the freshly established secure channel enables the server to update the client's password file without using the ratchet. Most importantly, such updates can reset the salt list to its initial state achieving that file size only increases with unsuccessful authentication attempts (see the yellow part of Figure 1). In practice, this is sufficient because password-based protocols usually use rate limiting to prevent online password guessing or denial of service (DoS) attacks. The unsuccessful attempt threshold upper bounds our protocol's transcript size and server storage. For more details about our compiler, we refer to Section 3.

Efficiency. In Table 1, we compare current saPAKE constructions and our (instantiated) compiler. We highlight that some of them are proven secure in relaxed security models, we do not rely on any security relaxations. However, ours is the first saPAKE construction where message length and the size of server state are not constant (regarding the storage and communication costs, we explore various trade-offs in Table 2). We consider this and the one or two extra communication rounds, compared to other solutions, as the price of being able to bridge the gap between aPAKE and saPAKE with only using symmetric primitives. Compared to other solutions, we also do not rely on any additional idealized models besides ROM and we do not need to use any exotic assumptions.

1.2 Related work

Our work builds upon the PAKE-to-aPAKE compilers of Gentry-MacKenzie-Ramzan [30] and Lyu-Liu-Han [43], which both follow the blueprint of sending a password-proof over a PAKE channel. Additionally, our proof of password knowledge is extracted from [43]'s KEM-based password confirmation, slightly modified to enable black-box usage. Both compilers add two extra messages to the communication after PAKE that was reduced to a single message by [38].

Dayanikli and Lehmann [23] characterized the gap between aPAKE and strong aPAKE and provided intermediate precomputation resistance levels through salting the password hash. They observed that protocols such as SRP-6a [55]) achieve mild levels of precomputation resistance. In a subsequent work, the same authors [24] proposed *updatable aPAKE* that lies between PAKE and aPAKE, and that serves as an inspiration to our password file ratchet. They use updates to "reset" precomputation attacks upon every successful key exchange. We extend their ideas by reducing the timeframe almost to zero rendering precomputation practically impossible, improving resistance by a lot in particular for scenarios where key exchange is only run seldomly. Dayanikli and Lehmann apply their update technique to OKAPE-HMQV [51] and SRP-6a [55], i.e., they do not provide a generic compiler.

OPAQUE [40], a strong aPAKE from OPRFs and authenticated key exchange, was selected in IRTF's standardization initiative [52,14]. Instantiated

Name	Ref.	Security			Efficiency		
		Sec.Def.	Model (+ROM)	Assumptions[3]	PQ	Rounds (1st)	State (S)/Msg. size
OPAQUE-HMQV	[40]	Relaxed	-	OMGDH	✗	2 (C)	static
OPRF+aPAKE	[40]	Relaxed	-	OMGDH	✗	3 (C)	static
Strong AuCPace	[34,33]	Relaxed*		SDH, OMGDH	✗	3	static
CKEM-saPAKE	[16]	Full	GGM	2-SDH,DDH	✗	2 (C)	static
	[45]	Full	GGM,AGM	CDH	✗	2 (S)	static
	[45]	Full	GGAM,AGAM	GACDH	✓	2 (S)	static
CRISP	[20]	Full*	GBGM	CDH	✗	3	static
OneTwoPAKE	[42]	Full	-	DDH, BB-EUF	✗	2 (C)	static
Ours (generic)	Section 3	Full	-	PAKE+KEM (OW-PCA)	✓	PAKE+2 (S)	changing
Ours (NoIC+LLH)	Section 3	Full	-	MLWE	✓	4 (S)	changing

Table 1. Comparison of saPAKE constructions. In case of generic constructions, we consider the instantiations that were proposed by the authors. Star denotes a stronger model than the respective relaxed/full definitions, where full refers to the (revised) saPAKE ideal functionality of [41]. In the security model column we do not list the ROM since it is used in all of the listed constructions' security arguments. We use the following abbreviations:
OMGDH – One-More Gap Diffie–Hellman assumption [39]
DSLS – Decision Shifted Legendre Symbol assumption [31]
(GA)CDH – (Group Action) Computational DH assumption [46]
q-SDH – q-Strong Diffie–Hellman assumption [12]
G[B]G(A)M – Generic [Bilinear] Group (Action) Model [53,46]
AG(A)M – Algebraic Group (Action) Model [29,46]
BB-EUF – Boneh–Boyen existential unforgeability assumption [42]

with classical 2HashDH OPRF [39] and Triple Diffie-Hellman (3DH), it is deployed, e.g., by WhatsApp's end to end encrypted backup storage [54,22]. As noted in [14], a post-quantum, forward-secret AKE can protect OPAQUE's past session traffic even with a classical OPRF. However, quantum attacks on the OPRF can enable offline password guessing from harvested credential material, so this does not provide full post-quantum saPAKE security.

E.g., the WhatsApp backup protocol becomes vulnerable to HNDL attacks on breached password files and transcripts. To the best of our knowledge, the saPAKE constructions of [45] are the first and so-far only ones that do not rely on OPRFs. Moreover, their group action-based proposal also provides post-quantum security. From the applicability perspective evaluating group actions is still relatively inefficient but most importantly their security is not yet stable enough, similarly to the corresponding idealized models whose validity is not yet clear [57].

2 Preliminaries

Definition 1 (OW-PCA security for KEMs [48]). *Let* $\mathsf{KEM} = (\mathsf{KGen}, \mathsf{Encap}, \mathsf{Decap})$ *denote the algorithms of a key encapsulation mechanism* KEM. *We then say* KEM *is one-way under plaintext checking attacks (OW-PCA) if for any PPT adversary there exists a function* η *negligible in security parameter* λ

such that

$$\mathsf{Adv}^{\mathsf{OW-PCA}}_{\mathcal{A},\mathsf{KEM}}(\lambda) := \Pr[K^* = K | K^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Check}}(\mathsf{sk},*,*)}(\mathsf{pk}, c)] \leq \eta,$$

where $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}$*,* $(c, K) \leftarrow \mathsf{Encap}(\mathsf{pk})$*, and* $\mathcal{O}_{\mathsf{Check}}(\mathsf{sk}, c, K)$ *returns* 1 *iff* $\mathsf{Decap}(\mathsf{sk}, c) = K$*.*

We next define a notion of *unpredictability of decapsulation under random public keys*, capturing that decapsulated keys are unpredictable for randomly sampled KEM key pairs, on even adversarial ciphertexts but where the adversary does not see the public key.

Definition 2 (UNP-DEC security for KEMs). *Let* $\mathsf{KEM} = (\mathsf{KGen}, \mathsf{Encap}, \mathsf{Decap})$ *denote the algorithms of a key encapsulation mechanism* KEM*. We then say* KEM *has unpredictable decapsulation (UNP-DEC) if for any PPT adversary there exists a function* η *negligible in security parameter* λ *such that*

$$\mathsf{Adv}^{\mathsf{UNP-DEC}}_{\mathcal{A},\mathsf{KEM}}(\lambda) := \Pr[\mathsf{KEM.Decap}(sk, c) = K^* | (c, K^* \leftarrow \mathcal{A}(1^\lambda)] \leq \eta,$$

where $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$*.*

2.1 Authenticated Encryption

Our saPAKE construction uses an authenticated encryption scheme. We start by giving the definition of ciphertext integrity for symmetric encryption schemes.

Definition 3 (Ciphertext integrity for SE [8]). *A symmetric encryption scheme* $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *has integrity of ciphertexts (INT-CTXT) if for all* $\lambda \in \mathbb{N}$ *and all PPT* $\mathcal{A}$*, we have*

$$\mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathcal{A},\mathsf{SE}}(\lambda) \coloneqq \Pr\left[\mathsf{Exp}^{\mathrm{int\text{-}ctxt}}_{\mathcal{A},\mathsf{SE}}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{negl}(\lambda)$ *is a negligible function in* λ *and* $\mathsf{Exp}^{\mathrm{int\text{-}ctxt}}_{\mathcal{A},\mathsf{SE}}(\lambda)$ *is defined below.*

$\mathsf{Exp}^{\mathrm{int\text{-}ctxt}}_{\mathcal{A},\mathsf{SE}}(\lambda)$	$\mathsf{Dec}^*_k(c)$
$k \leftarrow \mathsf{Gen}(1^\lambda)$	**if** $\perp \neq \mathsf{Dec}_k(c)$: **return** 1
if $\mathcal{A}^{\mathsf{Enc}_k(\cdot),\mathsf{Dec}^*_k(\cdot)}$ makes a query c	**else return** 0
to $\mathsf{Dec}^*(\cdot)$ s.t.	
– $\mathsf{Dec}^*_k(c) = 1$ and	
– c was no response of $\mathsf{Enc}_k(\cdot)$	
return 1	
return 0	

Definition 4 (Authenticated Encryption). *An* authenticated encryption scheme AE *is a symmetric encryption scheme that is IND-CPA secure and has INT-CTXT security.*

On input $(\mathsf{NewSession}, \mathsf{sid}, \mathsf{P}, \mathsf{P}', \mathsf{pw}, \mathtt{role})$ from P:
- Send $(\mathsf{NewSession}, \mathsf{sid}, \mathsf{P}, \mathsf{P}', \mathtt{role})$ to $\mathcal{A}$.
- Furthermore, if this is the first NewSession message for sid, or this is the second NewSession message for sid and there is a record $\langle \mathsf{P}', \mathsf{P}, * \rangle$, then record $\langle \mathsf{P}, \mathsf{P}', \mathsf{pw} \rangle$ and mark it `fresh`.

On input $(\mathsf{TestPwd}, \mathsf{sid}, \mathsf{P}, \mathsf{pw}')$ from $\mathcal{A}$:
- If there is a record $\langle \mathsf{P}, \mathsf{P}', \mathsf{pw} \rangle$ `fresh`, do
  - If $\mathsf{pw} = \mathsf{pw}'$, mark the record `compromised` and return CorrectGuess to $\mathcal{A}$.
  - Else, mark the record `interrupted` and return WrongGuess to $\mathcal{A}$.

On input $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{P}, \mathsf{K}^*, \mathsf{tr}^*)$ from $\mathcal{A}$, where $|\mathsf{K}| = \lambda$:
- Ignore if there is no $\langle \mathsf{P}, \mathsf{P}', \mathsf{pw} \rangle$, or $\langle \mathsf{P}, \mathsf{P}', \mathsf{pw} \rangle$ is `fresh` and there is no record $\langle \mathsf{P}', \mathsf{P}, * \rangle$.
- Else define K, tr as follows:
  - If $\langle \mathsf{P}, \mathsf{P}', \mathsf{pw} \rangle$ or $\langle \mathsf{P}', \mathsf{P}, * \rangle$ have been marked `compromised` or `interrupted` and tr^* was sent to P', exit the query. Else set $\mathsf{tr} \leftarrow \mathsf{tr}^*$
  - If the record is marked `compromised`, set $\mathsf{K} \leftarrow \mathsf{K}^*$.
  - Else, if the record is marked `fresh`, $(\mathsf{sid}, \mathsf{K}', \mathsf{tr}')$ was sent to P', and at that time, there was a record $\langle \mathsf{P}', \mathsf{P}, \mathsf{pw} \rangle$ marked `fresh`, then set $\mathsf{K} \leftarrow \mathsf{K}'$ and $\mathsf{tr} \leftarrow \mathsf{tr}'$.
  - Else, sample $\mathsf{K}'' \xleftarrow{\$} \{0,1\}^\lambda$ and set $\mathsf{K} \leftarrow \mathsf{K}''$.
- Output $(\mathsf{sid}, \mathsf{K}, \mathsf{tr})$ to P and mark $\langle \mathsf{P}, \mathsf{P}', \mathsf{pw} \rangle$ as `completed`.

Fig. 2. Ideal functionality for symmetric PAKE.

2.2 Password-Authenticated Key Exchange

PAKE Functionality. We present the PAKE functionality $\mathcal{F}_{\mathsf{PAKE}}$ in Figure 2. Following Gentry, MacKenzie, and Ramzan [30], we extend $\mathcal{F}_{\mathsf{PAKE}}$ to output a transcript value tr alongside the session key. The transcript is adversarially chosen but crucially, if either party's session has been compromised or interrupted, the two parties cannot share the same transcript. This is useful when the password based key exchange is used to establish a secure channel between parties, even if an adversary successfully attacks against both PAKE sessions the parties can identify transcript mismatch and stop the execution. A related formulation, in which session identifiers are protocol outputs, is used by Lyu, Liu, and Han [43]. Moreover, $\mathcal{F}_{\mathsf{PAKE}}$ ignores NewKey for a fresh record of P unless the corresponding record for P' exists. This requires both parties' participation before an unattacked session outputs a key. Note that this restriction is wlog: an interactive protocol where a party can terminate its execution immediately after sending a message would allow an eavesdropper to derive a candidate session key for each password guess from the transcript, contradicting session-key indistinguishability.

Strong aPAKE Functionality. Our transformation realizes the saPAKE functionality $\mathcal{F}_{\mathsf{saPAKE}}$ depicted in Figure 3. Note that we include a TestAbort interface as in [30,40]. The adversary can query TestAbort before key delivery: a failure response forces the tested party to abort, whereas a success response leaves its session fresh.

Registration

On input $(\mathsf{StorePwdFile}, \mathsf{sid}, \mathsf{C}, \mathsf{pw})$ from S:
- record $\langle \mathsf{File}, \mathsf{C}, \mathsf{S}, \mathsf{sid}, \mathsf{pw} \rangle$ and send $(\mathsf{StorePwdFile}, \mathsf{C}, \mathsf{sid})$ to $\mathcal{A}$.

Stealing Password Data

On input $\boxed{(\mathsf{StealPwdFile}, \mathsf{sid})}$ from $\mathcal{A}$, on instruction from $\mathcal{Z}$:
- If there is no record $\langle \mathsf{File}, \mathsf{C}, \mathsf{S}, \mathsf{sid}, [\mathsf{pw}] \rangle$, return $\mathsf{NoPwdFile}$ to $\mathcal{A}$.
- Else, mark S as COMPROMISED and return $\mathsf{PwFileStolen}$ to $\mathcal{A}$.

On input $\boxed{(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw})}$ from $\mathcal{A}$:
- If there is a record $\langle \mathsf{File}, \mathsf{C}, \mathsf{S}, \mathsf{sid}, \mathsf{pw} \rangle$ and S is marked COMPROMISED, return $\mathsf{CorrectGuess}$ to $\mathcal{A}$
- Else, return $\mathsf{WrongGuess}$ to $\mathcal{A}$.

Password Authentication

On input $(\mathsf{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \mathsf{pw}')$ from C:
- If this is the first $\mathsf{UsrSession}$ message for ssid, record $\langle \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \mathsf{pw}' \rangle$ and mark it `fresh`.
- Send $(\mathsf{UsrSession}, \mathsf{ssid}, \mathsf{C}, \mathsf{S})$ to $\mathcal{A}$.

On input $(\mathsf{SrvSession}, \mathsf{sid}, \mathsf{ssid})$ from S:
- Ignore the query if there is no record $\langle \mathsf{File}, \mathsf{C}, \mathsf{S}, \mathsf{sid}, [\mathsf{pw}] \rangle$
- If this is the first $\mathsf{SrvSession}$ message for ssid, record $\langle \mathsf{ssid}, \mathsf{S}, \mathsf{C}, \mathsf{pw} \rangle$ and mark it `fresh`.
- Send $(\mathsf{SrvSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S})$ to $\mathcal{A}$.

Active Session Attacks

On input $(\mathsf{TestPwd}, \mathsf{ssid}, \mathsf{P}, \mathsf{pw}')$ from $\mathcal{A}$:
- If there is a record $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', [\mathsf{pw}] \rangle$ marked `fresh`, do
  - If $\mathsf{pw} = \mathsf{pw}'$, mark the record `compromised` and return $\mathsf{CorrectGuess}$ to $\mathcal{A}$.
  - Else, mark the record `interrupted` and return $\mathsf{WrongGuess}$ to $\mathcal{A}$.

On input $(\mathsf{Impersonate}, \mathsf{ssid})$ from $\mathcal{A}$:
- If there is a record $\langle \mathsf{ssid}, \mathsf{C}, \mathsf{S}, [\mathsf{pw}'] \rangle$ marked `fresh`, do
  - If there is a record $\langle \mathsf{File}, \mathsf{C}, \mathsf{S}, \mathsf{sid}, [\mathsf{pw}] \rangle$, S is marked COMPROMISED, and $\mathsf{pw} \stackrel{?}{=} \mathsf{pw}'$, mark $\langle \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \mathsf{pw}' \rangle$ as `compromised` and return $\mathsf{CorrectGuess}$ to $\mathcal{A}$.
  - Else, mark the record $\langle \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \mathsf{pw}' \rangle$ as `interrupted` and return $\mathsf{WrongGuess}$ to $\mathcal{A}$.

Key Generation and Authentication

On input $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{K})$ from $\mathcal{A}$, where $|\mathsf{K}| = \lambda$ or $\mathsf{K} = \perp$:
- If there is a record $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \mathsf{pw} \rangle$ not marked `completed`, do
  - If the record is marked `compromised` output $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K})$ to P.
  - Else, if the record is marked `fresh`, $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}')$ was sent to P', and at that time, there was a record $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \mathsf{pw} \rangle$ marked `fresh`, give output $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}')$ to P.
  - Else, if the record is `interrupted` or if it is `fresh` and there is a record $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \mathsf{pw}' \rangle$ with $\mathsf{pw} \neq \mathsf{pw}'$, then give output $(\mathsf{sid}, \mathsf{ssid}, \perp)$ to P and $\mathcal{A}$.
  - Else, pick $\mathsf{K}'' \stackrel{R}{\leftarrow} \{0,1\}^\lambda$ and give output $(\mathsf{sid}, \mathsf{ssid}, \mathsf{K}'')$ to P.
- Finally, mark $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \mathsf{pw} \rangle$ as `completed`.

On input $(\mathsf{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P})$ from $\mathcal{A}$:
- Ignore if no record $\langle \mathsf{ssid}, \mathsf{P}, *, * \rangle$ marked `fresh` exists.
- Else if there is a record $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ marked `fresh` and there is a counter record $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \mathsf{pw}' \rangle$ marked `fresh` or `completed`, send $\mathsf{CorrectGuess}$
- Else send Abort to $\mathcal{A}$, output $(\mathsf{sid}, \mathsf{ssid}, \perp)$ to P and mark the record `completed`

Fig. 3. Ideal functionality $\mathcal{F}_{\mathsf{saPAKE}}$ [41]. $\boxed{\text{Framed queries}}$ can only be sent with permission of $\mathcal{Z}$.

3 PAKE to saPAKE transformation

We start by explaining our transformation. The full description is presented in Figure 5.

The standard approach to thwart precomputation attacks against password files is letting the server store a salted hash of the password. That is, the server stores a salt s and $\mathsf{H}(\mathsf{pw}, s)$, where $s \xleftarrow{\$} \{0,1\}^\lambda$. Therefore, a naive first attempt to lift PAKE to strong aPAKE would be to let the server store $(s, \mathsf{H}(\mathsf{pw}, s))$ as the password file and then let both parties provide $\mathsf{H}(\mathsf{pw}, s)$ as their PAKE-input. This does not yield an saPAKE because the client must know s in order to compute the PAKE input $\mathsf{H}(\mathsf{pw}, s)$. If the server sends s to the client, all precomputation protection is lost because a network adversary observing s can start computing $\mathsf{H}(\mathsf{pw}^*, s)$ for password guesses pw^* from that point on. To avoid this, our transformation "ratchets" the password file forward. By that we mean that after sending s to the client, the server picks a new salt $s' \xleftarrow{\$} \{0,1\}^\lambda$, computes $h' = \mathsf{H}(s', \mathsf{H}(\mathsf{pw}, s))$, and updates its password file to (s, s', h'). The network adversary who observed s cannot start precomputing against this new updated file because it has not yet seen s'. The downside of the file update is that the next time server and client want to exchange a key, the server must send both salts, s and s' to the client so that the client can compute the PAKE input $\mathsf{H}(s', \mathsf{H}(\mathsf{pw}, s))$. This can be repeated arbitrarily often with a growing salt list $s_1, \ldots, s_n$ that has to be sent from server to client. This procedure is sufficient to thwart precomputation attacks[4] but comes with two other problems that need to be addressed.

1. The protocol described above does not check that the client actually knows pw. In fact, the hash h_n from the server's file is sufficient to make the PAKE successful (in the n-th key exchange session). Put differently, if the server's file gets compromised, a malicious client could learn h_n and make the PAKE succeed without any password guess at all. Note that this is weaker than what aPAKE security demands. In aPAKE, a compromised file allows the adversary to impersonate the server but it must not allow it to impersonate the client. Even with a compromised file, a secure aPAKE must require an offline guessing attack in order to authenticate as the client.
2. The indefinite growth of the protocol transcript and server state might become a serious efficiency concern if the protocol is used regularly, say for daily login to some service. It is desirable to limit the growth of the salt list.

We also assume that the server does not start an execution if another is still ongoing. In the following we describe how our transformation approaches the two problems.

[4] Precomputation attacks are only thwarted if we are willing to assume that the adversary does not compromise the server's file in the narrow time window between the server sending s_i and overwriting $(s_1, \ldots, s_i, \mathsf{H}(h_{i-1}, s_i))$ with $(s_1, \ldots, s_i, s_{i+1}, \mathsf{H}(h_i, s_{i+1}))$. We believe this assumption to be reasonable given that this timeframe is likely only a few milliseconds while the file might be stored for years and could be compromised at any point in that time.

Proof-of-Password-Knowledge. The PAKE with a salted hash of the password as input does not prove that the client knows the preimage of that hash. Existing PAKE-to-aPAKE compilers from the literature also had to solve this problem. For instance, in the Ω-method [30], the server uses the already established PAKE channel to send a password-encrypted signing key to the client. If the client can decrypt the signing key and sign the protocol transcript, this proves to the server that the client knows the password because merely having the hash of the password is not sufficient to decrypt the signing key. We make the following key observations about the proof our compiler requires:

- A proof transcript is only revealed to somebody who managed to establish a PAKE connection, i.e., who knows the encoded password stored at the server (e.g., $H(\mathsf{pw}, s)$)
- Since the encoding already permits offline-guessing pw simply by recomputing it, a proof with an *offline attackable* transcript is sufficient for our compiler
- Since offline attacks are stronger than precomputation attacks, precomputations on the verifier's state can be permitted once the prover sent their first message[5].

Instead of directly using a mechanism like the encrypted signing key from the Ω-method, we abstract a *weak Proof of Password Knowledge (wPoPWK)* as a building block of PAKE-to-aPAKE compilers. Our goal is to formalize the *minimal guarantees needed* from such a proof for our PAKE-to-saPAKE compiler, following the observations made above. The ideal wPoPWK functionality $\mathcal{F}_{\text{wPoPWK}}$ is depicted in Figure 4. It can be used by a prover (client) to demonstrate password knowledge multiple times, but always w.r.t the same *statement* POK-gs.2 stored at the verifier (server). One can imagine that the statement roughly corresponds to the password-encrypted signing key in the Ω-method. Then any prover and the verifier can interact with each other such that the prover inputs a password and the verifier learns a single bit as output which indicates whether the input password is equal to the password that generated the statement. Because we want to use this proof in our saPAKE protocol, we have to allow for the statement to get compromised by the adversary (via the LeakStatement command POK-ls.2). This is the analog of an aPAKE's password file being compromised. A leaked statement allows an adversary to offline-guess the password POK-otp.1, but it must not directly allow the adversary to prove password knowledge. Before compromise and also before the first interaction with the verifier, $\mathcal{F}_{\text{wPoPWK}}$ disallows precomputation attacks. To capture that the proof's transcript may allow for offline attacks, $\mathcal{F}_{\text{wPoPWK}}$ allows offline-guessing passwords against both a prover's POK-otp.2 and a verifier's POK-otp.1 transcript. While the proof is called *weak* to capture these offline attack vulnerabilities, $\mathcal{F}_{\text{wPoPWK}}$ does enforce *soundness*, i.e., the verifier can only be convinced by a prover who knows the password POK-fin.1.

[5] This notion of precomputation is what Dayanikli and Lehmann [24] call *level 2* protection for an aPAKE.

Statement generation

On input (GenStatement, sid, pw) from $\mathcal{P}$:

POK-gs.1 If this is the first GenStatement message, send (GenStatement, sid) to $\mathcal{A}$ and receive back b

POK-gs.2 If $b = 1$, record $\langle$Statement, sid, pw$\rangle$ and send a delayed output (GenStatement) to $\mathcal{V}$.

Attacks

On input (LeakStatement, sid) from $\mathcal{A}$, on instruction from $\mathcal{Z}$:

POK-ls.1 If there is no record $\langle$Statement, sid, [pw]$\rangle$, return NoStatement to $\mathcal{A}$.

POK-ls.2 Else, mark $\mathcal{V}$ as Compromised and if there is record (sid, pw) then return (StatementLeaked, pw) to $\mathcal{A}$. If there is no such record, return StatementLeaked to $\mathcal{A}$.

On input (OfflineTestPwd, sid, ssid, P, pw) from $\mathcal{A}$: //Offline-attack a prover session or a leaked statement

POK-otp.1 If P $= \mathcal{V}$ and there is a record $\langle$Statement, sid, pw$\rangle$:
- If $\mathcal{V}$ is Compromised or there was already an input (Verify, ...) by $\mathcal{V}$, return CorrectGuess to $\mathcal{A}$ //Offline attack on stolen statement or Verifier's transcript

POK-otp.2 Else if P $= \mathcal{P}$ and there exists a record $\langle$ssid, P, $*$, pw$\rangle$, return CorrectGuess to $\mathcal{A}$ //Offline attack on prover's transcript

Proof of password knowledge

On input (Prove, sid, ssid, pw$'$) from $\mathcal{P}$ or $\mathcal{A}$:

POK-prv.1 If this is not the first Prove message for ssid from the sender, ignore the query.

POK-prv.2 If this is $\mathcal{P}$'s first Prove input for ssid, or it comes from $\mathcal{A}$, record $\langle$ssid, $\mathcal{P}$, $\mathcal{V}$, pw$'\rangle$

POK-prv.3 Set $b = \perp$; if there is a record $\langle$ssid, $\mathcal{V}$, $\mathcal{P}$, pw$\rangle$, set $b = (\text{pw} \stackrel{?}{=} \text{pw}')$ //Leaking proof outcome to the adversary

POK-prv.4 Send (Prove, sid, ssid, b) to $\mathcal{A}$

On input (Verify, sid, ssid, pw) from $\mathcal{V}$ or $\mathcal{A}$:

POK-ver.1 If this is not the first Verify message for ssid from the sender, ignore the query.

POK-ver.2 Retrieve $\langle$Statement, sid, [pw$'$]$\rangle$. If $\langle$Statement$\rangle$ does not exist and $\mathcal{V}$ is honest ignore the query; else if $\mathcal{V}$ is honest or ($\mathcal{V}$ is marked Compromised and pw $= \perp$), set pw $\leftarrow$ pw$'$ //pw $= \perp$: impersonation attack

POK-ver.3 If this is $\mathcal{V}$'s first Verify input for ssid, or it comes from $\mathcal{A}$, record $\langle$ssid, $\mathcal{V}$, $\mathcal{P}$, pw$\rangle$

POK-ver.4 Set $b = \perp$; if there is a record $\langle$ssid, $\mathcal{P}$, $\mathcal{V}$, pw$'\rangle$, set $b = (\text{pw} \stackrel{?}{=} \text{pw}')$ //Leaking proof outcome to the adversary

POK-ver.5 Send (Verify, sid, ssid, b) to $\mathcal{A}$

On input (Finalize, sid, ssid) from $\mathcal{A}$:

POK-fin.1 If there is a record $\langle$ssid, $\mathcal{V}$, $\mathcal{P}$, pw$\rangle$ not marked `completed` and $\mathcal{V}$ sent a Verify for ssid, do
- If $\mathcal{A}$ never sent Verifyfor ssid and there exists a record $\langle$ssid, $\mathcal{P}$, $\mathcal{V}$, pw$\rangle$ then output (Verified, sid, ssid, 1) to $\mathcal{V}$
- Else output (Verified, sid, ssid, 0) to $\mathcal{V}$

POK-fin.2 Mark $\langle$ssid, $\mathcal{V}$, $\mathcal{P}$, pw$\rangle$ as `completed`.

Fig. 4. Ideal functionality $\mathcal{F}_{\text{wPoPWK}}$ formalizing weakly secure proofs of password knowledge.

For our PAKE-to-saPAKE transformation, this means that the server's file now consists of the salted hash part $(s_1, \ldots, s_n, h_n)$ plus a statement x_{pw} for the wPoPWK. Note that it is necessary for the wPoPWK to be executed over the PAKE protected channel to prevent offline attacks on the wPoPWK transcript.[6]

We note, however, that even though we took inspiration from the proof-of-password-style parts of [30] and [43], they are not directly applicable to our setting. This is because, unlike [30] and [43], we are aiming for *strong* aPAKE security. Therefore, the wPoPWK must also have some security against precomputation attacks. We discuss instantiations of wPoPWK that protect against precomputation attacks in Section 3.1.

Resetting the Password File. To limit the growth of the password file, we use the intuition that a new registration of the client after a successful authentication would result in a fresh server file with only a single salt value s. But in fact, our protocol does not re-register the full password file. Concretely, we use that at the end of a successful key exchange, the two parties have a password-authenticated channel that allows them to securely communicate. This allows the client to reset the salted-hash part of the server's file by picking a fresh s'_1 and computing a fresh $\mathsf{H}(\mathsf{pw}, s'_1)$. The two new values can be sent to the server using symmetric encryption under the PAKE key and the server can replace the potentially long salt list $(s_1, \ldots, s_n, h_n)$ with $(s'_1, \mathsf{H}(\mathsf{pw}, s'_1))$.

As a result, the salt list and the server state no longer grow with the number of key exchange sessions per user but only with the number of unsuccessful key exchange sessions per user. This is particularly useful in practice, as many services will apply rate-limits on the number of wrong login-attempts anyway. If a server blocks a user after 10 wrong password attempts, the salt list and server file will never outgrow 10 salts. Note that the wPoPWK part of the file is not updated. This ensures that a malicious client cannot change the registered password upon a successful authentication.

In summary, our transformation combines the three discussed parts in order. First, a growing salt list is sent to the client. Then, server and client use a hashed version of the password as input to a PAKE. The resulting PAKE key then protects the wPoPWK. Finally, if the wPoPWK was successful, the client and the server reset the salt list, or, if the wPoPWK was not successful, the server ratchets the password file forward.

Salt update options. The ratchet admits several variants that trade client and server computation, communication, and storage without affecting the security argument. We compare these alternatives in Appendix B. In the remainder, we use the simple unbounded variant in which the server appends one fresh salt after each unsuccessful authentication.

[6] This corresponds to the case in the Ω-method where the server sends the password-encrypted signing key to the client. If the server sent this value in the clear, a network adversary could precompute candidate signing keys and signatures and verify the guess upon learning the verification key from a compromised file.

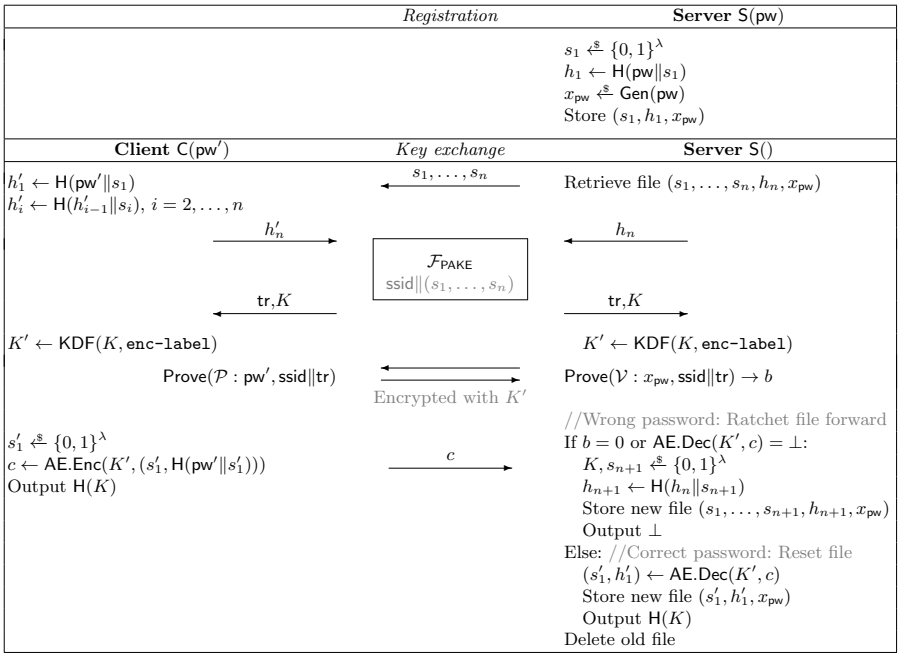

Fig. 5. Transformation from $\mathcal{F}_{\mathsf{PAKE}}$ to $\mathcal{F}_{\mathsf{saPAKE}}$, using a PAKE, an authenticated-encryption scheme AE, and a proof of password knowledge $(\mathsf{Gen}, \mathsf{Prove})$. A fresh $\mathcal{F}_{\mathsf{PAKE}}$ instance is invoked for every key exchange session, with its identifier additionally containing the salt list $(s_1, \ldots, s_n)$. Throughout the whole protocol, if $\mathsf{AE.Dec}$ fails under K', the party outputs $\perp$. The server also ratchets the file forward if the execution fails. We assume wlog. that the PoPWK requires one message in each direction and that PoPWK has fixed length. All encrypted messages include step identifiers; receiving a message with an unexpected identifier makes the party output $\perp$.

Theorem 1. *The protocol in Figure 5 UC-realizes* $\mathcal{F}_{\mathsf{saPAKE}}$ *in the* $(\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{PAKE}})$*-hybrid model, assuming* $(\mathsf{Gen}, \mathsf{Prove})$ *UC-realizes* $\mathcal{F}_{wPoPWK}$*,* AE *is INT-CTXT and IND-CPA secure.*

Proof (Sketch). The simulation of the PAKE part is straightforward, as the PAKE key is never revealed to the adversary for an honest party. Attack queries are answered by extracting the password and forwarding the attack to $\mathcal{F}_{\mathsf{saPAKE}}$. For the wPoPWK part, all the messages from honest parties that do not share a key with the adversary are substituted with random ciphertext. When the session is not attacked, $\mathsf{TestAbort}$ is used to decide whether the decryption of the first wPoPWK message is successful or not. Moreover, this query also determines the success of the wPoPWK protocol, as the honest party exchanged a key successfully only if the correct password has been used during the key exchange. To simulate the messages of an attacked party, we switch to $\text{SIM}_{\text{wPoPWK}}$, i.e. the simulator of wPoPWK, interacting with $\mathcal{F}_{\text{wPoPWK}}$. The code is executed exactly until $\mathcal{F}_{\text{wPoPWK}}$ needs to compare password in an adversarial PROVE or VERIFY query with the honest one. When such a query arrives the simulator confronts the submitted password with the one used to successfully attack the honest session and computes the bit b accordingly. The final update sent by the client is also randomized in the honest session, and computed with adversarial password if the client was successfully attacked. Finally, when the adversary successfully guesses the password and sends the update message to the server, we must verify the correctness of the update. To this end, it is sufficient to verify if the password used to compute the new hash was the same that successfully attacked the server record via $\mathsf{TestPwd}$. If that is not the case we mark the file `invalid`. Note that honest parties will never be able to successfully exchange a key on an invalid file due to the mismatch between the PAKE and the wPoPWK part. Nonetheless, we still require password extraction on the malicious update: on the next honest interaction the parties will not abort on the first received wPoPWK message only if the honest client's password coincides with the one used to update the file, which can be learned via a $\mathsf{TestPwd}$ query.

The full proof is given in Appendix A.

3.1 Instantiating the Proof of Password Knowledge

In Figure 6 we give an instantiation of a weak proof of password knowledge from IND-CPA secure key encapsulation methods (KEM). The proof is extracted from the PAKE-to-aPAKE compiler by Lyu, Liu and Han [43]. The idea is to deterministically generate a KEM key pair using the (hashed) password as randomness, and store the public key at the verifier. For proving, a verifier then encapsulates a KEM key and sends the ciphertext to the prover. The prover computes a KEM key pair from their password, decapsulates the KEM key, and sends a key confirmation message back to the verifier. Replay attacks are prevented by adding a nonce ssid.

We augment this proof by adding a salt to the KEM key derivation that is sent along with the first message from the verifier. This achieves only a very

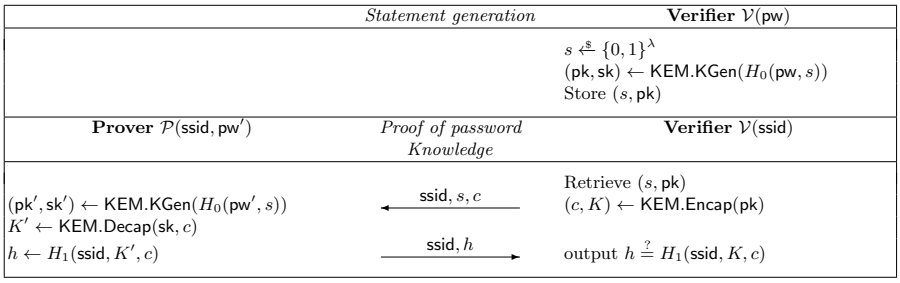

Fig. 6. wPoPWK from an OW-PCA and UNP-DEC secure key encapsulation mechanism $\mathsf{KEM} := (\mathsf{KGen}, \mathsf{Encap}, \mathsf{Decap})$, inspired by [43] and augmented with a salt for weak precomputation resistance (resembling level-1 security of aPAKE files in [24]).

weak form of precomputation protection, namely only up until the verifier first speaks, but which we have shown to be sufficient for our compiler to achieve strong precomputation resistance.

Theorem 2. *If* KEM *is OW-PCA and UNP-DEC secure, the wPoPWK protocol in Figure 6 UC-realizes* $\mathcal{F}_{wPoPWK}$*, with* H_0, H_1 *modeled as random oracles.*

Proof (Sketch). We depict the simulator in Figure 10. Indistinguishability is relatively easy to argue since the simulation runs on the correct password in many cases. These passwords are found via $\mathsf{OfflineTestPwd}$ queries to $\mathcal{F}_{\text{wPoPWK}}$ that the simulator submits against the verifier's statement and *every* prover session upon each password hash submitted to H_0. If passwords are not yet known, the simulator uses a randomly sampled KEM key pair for the verifier and randomly sampled key confirmation messages for the prover. Both are equivocated via the random oracle as soon as the environment reveals (via a H_0 query) which passwords were used.

The interesting part of the proof is due to the fact that the prover is deterministic and implements a "plaintext checking oracle" (PCA) in the following sense: upon receiving a KEM ciphertext c, the prover uses a randomly sampled KEM secret key to decapsulate c. The resulting key K is then hashed as $h = H_1(\mathsf{ssid}, K, c)$ and the prover reveals h. This enables an attacker holding a ciphertext c and "plaintext" K to check (by recomputing the hash) whether c decapsulates to K under the secret key used by the prover. This could leak information about sk used by the prover, which is derived from the password and hence the leakage could accumulate over multiple sessions where the prover runs on the same input password. We hence have to rule out that an attacker can *predict* a decapsulated key computed by a prover who runs on a randomly sampled KEM key pair $(\mathsf{sk}, \mathsf{pk})$ that is not known to the environment (if it is known, the prover password is known and simulation becomes trivial). As a first step, we make the execution independent of sk used by provers who run on mismatching passwords by randomizing their h messages. The reduction to UNP-DEC is straightforward, and it is applied per such prover session and with a guess on which random oracle query by the environment contains the key prediction.

The main purpose of the KEM is that it protects the verifier, i.e., ensures soundness. It must hence be hard to compute KEM keys without knowing the statement password stored at the verifier. In the second step, as depicted in Figure 10, the execution aborts in case of an honest verifier receiving a key confirmation message h from the adversary that was computed from verifier's KEM key K but where the environment never actually hashed the password from the statement. Intuitively, this means that the environment computed K in a different way. We exploit this by letting a reduction, running on an OW-PCA challenge (pk, c), insert pk as the public key of the statement, and use c as the KEM ciphertext in the i-th session of the honest verifier. The reduction produces the KEM keys in all other attacked sessions from pk. Since honest clients implement a PCA oracle when running on the same password as the honest verifier, the reduction (knowing input passwords!) uses its own PCA oracle on every $H_1(\mathsf{ssid}, K', c)$ query of the environment that corresponds to an honest prover's session ssid. Upon receiving back 1, the reduction produces the prover message from h, otherwise it samples a random h. Finally, the reduction randomly selects one of the environment's hash queries $H_1(\mathsf{ssid}_i, K^*, c)$ and submits it as response to the OW-PCA challenger.

With a hybrid argument, the theorem follows with a loss dominated by $q_p q_h \cdot \mathsf{Adv}^{\mathsf{OW-PCA}}_{\mathcal{A},\mathsf{KEM}}(\lambda)\mathsf{Adv}^{\mathsf{UNP-DEC}}_{\mathcal{A},\mathsf{KEM}}(\lambda)$, where q_p corresponds to the number of proofs carried out by the system and q_h to the number of hash queries.

The KEM of our wPoPWK can be instantiated with Kyber/ML-KEM. OW-PCA security follows from IND-CCA security. UNP-DEC holds thanks to the Fujisaki-Okamoto (FO) transform and implicit rejection of ML-KEM.

Lemma 1. *If* `SHAKE256` *is modeled as a random oracle, ML-KEM is statistically UNP-DEC-secure.*

Proof. Recall the decapsulation algorithm of ML-KEM, where $(\mathsf{Enc}, \mathsf{Dec})$ is a CPA-secure encryption scheme and G a random oracle:

$$m' \leftarrow \mathsf{Dec}(\mathsf{sk}, c); \quad (K', r') \leftarrow G(H(pk)||m'); \quad c' \leftarrow \mathsf{Enc}(\mathsf{pk}, m', r')$$

$$\mathsf{Decap}(\mathsf{sk}, c') = \begin{cases} K' & \text{if } c = c' \\ \texttt{SHAKE256}(z||c) & \text{otherwise} \end{cases}$$

z is a uniform 32-byte string sampled at key generation, i.e., it is stored together with sk. In case of rejection ($c \neq c'$), learning the output key requires guessing z since `SHAKE256` is a random oracle. Due to the decapsulation key pair being randomly generated after c is fixed, guessing z is successful only with probability $\frac{1}{2^{256}}$, yielding a $q_S/2^{256}$ bound for the adversary knowing the key in the rejection case, where q_S is the number of $\mathsf{SHAKE256}$ queries by the adversary. Otherwise ($c = c'$), a similar argument over pk that contains a 32-byte seed and the random oracles G, H yields a bound of $(q_G + q_H)/2^{256}$. Overall, we have

$$\Pr[\text{Adversary wins}] \leq \frac{q_G + q_H + q_S}{2^{256}}.$$

4 Future Work

We discuss a few avenues for future work that arise from our results. First, our PAKE-to-saPAKE compiler recovers existing compilers from the literature (up to small details). Hence our result that the KEM-based weak proof of password knowledge taken from Lyu, Liu and Han [43] can be plugged into our compiler. We expect the same to hold for the signature-based proof used by Gentry, MacKenzie and Ramzan [30], but given that post-quantum KEMs are more efficient than signatures, it is maybe of less interest to formally verify that. What is more interesting is the following: given the very weak (offline attackable) nature of the wPoPWK notion, can it be realized by a purely symmetric construction?

Finally, a strong aPAKE that is simple, instantiable from only post-quantum assumptions, and with a constant message size and server state is still elusive. In this regard, a theoretical study of the gap between the strong aPAKE and aPAKE UC notions could shed more light onto whether or not a *compact* yet OPRF-free saPAKE is feasible.

Acknowledgments

We would like to thank Tobias Handirk, Michael Rosenberg, Tibor Jager and Marc Westenberger for the insightful and stimulating discussions on our work.

References

1. Alamati, N., Policharla, G.V., Raghuraman, S., Rindal, P.: Improved alternating-moduli PRFs and post-quantum signatures. In: Reyzin, L., Stebila, D. (eds.) CRYPTO 2024, Part VIII. LNCS, vol. 14927, pp. 274–308. Springer, Cham (Aug 2024). https://doi.org/10.1007/978-3-031-68397-8_9
2. Albrecht, M.R., Davidson, A., Deo, A., Smart, N.P.: Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In: Garay, J. (ed.) PKC 2021, Part II. LNCS, vol. 12711, pp. 261–289. Springer, Cham (May 2021). https://doi.org/10.1007/978-3-030-75248-4_10
3. Albrecht, M.R., Gür, K.D.: Verifiable oblivious pseudorandom functions from lattices: Practical-ish and thresholdisable. In: Chung, K.M., Sasaki, Y. (eds.) ASIACRYPT 2024, Part IV. LNCS, vol. 15487, pp. 205–237. Springer, Singapore (Dec 2024). https://doi.org/10.1007/978-981-96-0894-2_7
4. Alnahawi, N., Haas, D., Mauß, E., Wiesmaier, A.: Sok: PQC PAKEs design, security and performance. IACR Communications in Cryptology **3**(2) (2026). https://doi.org/10.62056/a6n5txl7s
5. Arriaga, A., Barbosa, M., Jarecki, S.: NoIC: PAKE from KEM without ideal ciphers. Cryptology ePrint Archive, Paper 2025/231 (2025), https://eprint.iacr.org/2025/231
6. Arriaga, A., Barbosa, M., Jarecki, S.: Tempo: ML-KEM to PAKE compiler resilient to timing attacks. IACR Cryptol. ePrint Arch. **2025**, 1399 (2025), https://eprint.iacr.org/2025/1399

7. Basso, A.: A post-quantum round-optimal oblivious PRF from isogenies. In: Carlet, C., Mandal, K., Rijmen, V. (eds.) SAC 2023. LNCS, vol. 14201, pp. 147–168. Springer, Cham (Aug 2024). https://doi.org/10.1007/978-3-031-53368-6_8
8. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Journal of Cryptology **21**(4), 469–491 (Oct 2008). https://doi.org/10.1007/s00145-008-9026-x
9. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 139–155. Springer, Berlin, Heidelberg (May 2000). https://doi.org/10.1007/3-540-45539-6_11
10. Bellovin, S.M., Merritt, M.: Encrypted key exchange: Password-based protocols secure against dictionary attacks. In: 1992 IEEE Symposium on Security and Privacy. pp. 72–84. IEEE Computer Society Press (May 1992). https://doi.org/10.1109/RISP.1992.213269
11. Beullens, W., Dodgson, L., Faller, S.H., Hesse, J.: The 2Hash OPRF framework and efficient post-quantum instantiations. In: Fehr, S., Fouque, P.A. (eds.) EUROCRYPT 2025, Part VIII. LNCS, vol. 15608, pp. 332–362. Springer, Cham (May 2025). https://doi.org/10.1007/978-3-031-91101-9_12
12. Boneh, D., Boyen, X.: Short signatures without random oracles. In: Cachin, C., Camenisch, J. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 56–73. Springer, Berlin, Heidelberg (May 2004). https://doi.org/10.1007/978-3-540-24676-3_4
13. Boneh, D., Kogan, D., Woo, K.: Oblivious pseudorandom functions from isogenies. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part II. LNCS, vol. 12492, pp. 520–550. Springer, Cham (Dec 2020). https://doi.org/10.1007/978-3-030-64834-3_18
14. Bourdrez, D., Krawczyk, H., Lewi, K., Wood, C.A.: The OPAQUE Augmented Password-Authenticated Key Exchange (aPAKE) Protocol. RFC 9807 (Jul 2025). https://doi.org/10.17487/RFC9807, https://www.rfc-editor.org/info/rfc9807
15. Boyko, V., MacKenzie, P.D., Patel, S.: Provably secure password-authenticated key exchange using Diffie-Hellman. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 156–171. Springer, Berlin, Heidelberg (May 2000). https://doi.org/10.1007/3-540-45539-6_12
16. Bradley, T., Jarecki, S., Xu, J.: Strong asymmetric PAKE based on trapdoor CKEM. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 798–825. Springer, Cham (Aug 2019). https://doi.org/10.1007/978-3-030-26954-8_26
17. Canetti, R., Halevi, S., Katz, J., Lindell, Y., MacKenzie, P.D.: Universally composable password-based key exchange. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 404–421. Springer, Berlin, Heidelberg (May 2005). https://doi.org/10.1007/11426639_24
18. Casacuberta, S., Hesse, J., Lehmann, A.: SoK: Oblivious pseudorandom functions. In: 2022 IEEE European Symposium on Security and Privacy. pp. 625–646. IEEE Computer Society Press (Jun 2022). https://doi.org/10.1109/EuroSP53844.2022.00045
19. Castryck, W., Decru, T.: An efficient key recovery attack on SIDH. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part V. LNCS, vol. 14008, pp. 423–447. Springer, Cham (Apr 2023). https://doi.org/10.1007/978-3-031-30589-4_15
20. Cremers, C., Naor, M., Paz, S., Ronen, E.: CHIP and CRISP: Protecting all parties against compromise through identity-binding PAKEs. In: Dodis, Y., Shrimpton,

T. (eds.) CRYPTO 2022, Part II. LNCS, vol. 13508, pp. 668–698. Springer, Cham (Aug 2022). https://doi.org/10.1007/978-3-031-15979-4_23
21. Davidson, A., Faz-Hernandez, A., Sullivan, N., Wood, C.A.: Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups. RFC 9497 (Dec 2023). https://doi.org/10.17487/RFC9497, https://www.rfc-editor.org/info/rfc9497
22. Davies, G.T., Faller, S.H., Gellert, K., Handirk, T., Hesse, J., Horváth, M., Jager, T.: Security analysis of the WhatsApp end-to-end encrypted backup protocol. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part IV. LNCS, vol. 14084, pp. 330–361. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38551-3_11
23. Dayanikli, D., Lehmann, A.: (Strong) aPAKE revisited: Capturing multi-user security and salting. In: 2024 IEEE European Symposium on Security and Privacy. pp. 415–439. IEEE Computer Society Press (Jul 2024). https://doi.org/10.1109/EuroSP60621.2024.00030
24. Dayanikli, D., Lehmann, A.: Updatable aPAKE: Security against bulk precomputation attacks. In: Huang, C.Y., Chen, J.C., Shieh, S.P., Lie, D., Cortier, V. (eds.) ACM CCS 2025. pp. 1158–1172. ACM Press (Oct 2025). https://doi.org/10.1145/3719027.3765155
25. Doerner, J., Haitner, I., Ishai, Y., Makriyannis, N.: From OT to OLE with subquadratic communication. In: Huang, C.Y., Chen, J.C., Shieh, S.P., Lie, D., Cortier, V. (eds.) ACM CCS 2025. pp. 2249–2263. ACM Press (Oct 2025). https://doi.org/10.1145/3719027.3765225
26. Esgin, M.F., Steinfeld, R., Tairi, E., Xu, J.: LeOPaRd: Towards practical post-quantum oblivious PRFs via interactive lattice problems. Cryptology ePrint Archive, Report 2024/1615 (2024), https://eprint.iacr.org/2024/1615
27. Faller, S.H., Fischlin, M., Hardt, J., Hesse, J.: Combining oblivious pseudorandom functions. In: Daemen, J., Thomé, E. (eds.) Advances in Cryptology - EUROCRYPT 2026 - 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rome, Italy, May 10-14, 2026, Proceedings, Part II. Lecture Notes in Computer Science, vol. 16542, pp. 97–127. Springer (2026). https://doi.org/10.1007/978-3-032-25317-0_4, https://doi.org/10.1007/978-3-032-25317-0_4
28. Faller, S.H., Ottenhues, A., Ottenhues, J.: Composable oblivious pseudo-random functions via garbled circuits. In: Aly, A., Tibouchi, M. (eds.) LATINCRYPT 2023. LNCS, vol. 14168, pp. 249–270. Springer, Cham (Oct 2023). https://doi.org/10.1007/978-3-031-44469-2_13
29. Fuchsbauer, G., Kiltz, E., Loss, J.: The algebraic group model and its applications. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 33–62. Springer, Cham (Aug 2018). https://doi.org/10.1007/978-3-319-96881-0_2
30. Gentry, C., MacKenzie, P., Ramzan, Z.: A method for making password-based key exchange resilient to server compromise. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 142–159. Springer, Berlin, Heidelberg (Aug 2006). https://doi.org/10.1007/11818175_9
31. Grassi, L., Rechberger, C., Rotaru, D., Scholl, P., Smart, N.P.: MPC-friendly symmetric key primitives. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 430–443. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978332
32. Gu, Y.: New Paradigms For Efficient Password Authentication Protocols. Ph.D. thesis, University of California, Irvine, USA (2024), https://www.escholarship.org/uc/item/7qm0220s

33. Haase, B., Labrique, B.: AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. Cryptology ePrint Archive, Report 2018/286 (2018), https://eprint.iacr.org/2018/286
34. Haase, B., Labrique, B.: AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. IACR TCHES **2019**(2), 1–48 (2019). https://doi.org/10.13154/tches.v2019.i2.1-48, https://tches.iacr.org/index.php/TCHES/article/view/7384
35. Hao, F., van Oorschot, P.C.: SoK: Password-authenticated key exchange - theory, practice, standardization and real-world lessons. In: Suga, Y., Sakurai, K., Ding, X., Sako, K. (eds.) ASIACCS 22. pp. 697–711. ACM Press (May / Jun 2022). https://doi.org/10.1145/3488932.3523256
36. Heimberger, L., Kales, D., Lolato, R., Mir, O., Ramacher, S., Rechberger, C.: Leap: A fast, lattice-based OPRF with application to private set intersection. In: Fehr, S., Fouque, P.A. (eds.) EUROCRYPT 2025, Part VII. LNCS, vol. 15607, pp. 254–283. Springer, Cham (May 2025). https://doi.org/10.1007/978-3-031-91098-2_10
37. Hesse, J.: Separating symmetric and asymmetric password-authenticated key exchange. In: Galdi, C., Kolesnikov, V. (eds.) SCN 20. LNCS, vol. 12238, pp. 579–599. Springer, Cham (Sep 2020). https://doi.org/10.1007/978-3-030-57990-6_29
38. Hwang, J.Y., Jarecki, S., Kwon, T., Lee, J., Shin, J.S., Xu, J.: Round-reduced modular construction of asymmetric password-authenticated key exchange. In: Catalano, D., De Prisco, R. (eds.) SCN 18. LNCS, vol. 11035, pp. 485–504. Springer, Cham (Sep 2018). https://doi.org/10.1007/978-3-319-98113-0_26
39. Jarecki, S., Kiayias, A., Krawczyk, H.: Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 233–253. Springer, Berlin, Heidelberg (Dec 2014). https://doi.org/10.1007/978-3-662-45608-8_13
40. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 456–486. Springer, Cham (Apr / May 2018). https://doi.org/10.1007/978-3-319-78372-7_15
41. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. Cryptology ePrint Archive, Report 2018/163 (2018), https://eprint.iacr.org/2018/163
42. Kondi, Y., McQuoid, I., Melissaris, K., Orlandi, C., Roy, L., Tyner, L.: Onetwopake: Two-round strong asymmetric PAKE with ideal security. In: Daemen, J., Thomé, E. (eds.) Advances in Cryptology - EUROCRYPT 2026 - 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rome, Italy, May 10-14, 2026, Proceedings, Part II. Lecture Notes in Computer Science, vol. 16542, pp. 284–313. Springer (2026). https://doi.org/10.1007/978-3-032-25317-0_10, https://doi.org/10.1007/978-3-032-25317-0_10
43. Lyu, Y., Liu, S., Han, S.: Efficient asymmetric PAKE compiler from KEM and AE. In: Chung, K.M., Sasaki, Y. (eds.) ASIACRYPT 2024, Part V. LNCS, vol. 15488, pp. 34–65. Springer, Singapore (Dec 2024). https://doi.org/10.1007/978-981-96-0935-2_2
44. Maino, L., Martindale, C., Panny, L., Pope, G., Wesolowski, B.: A direct key recovery attack on SIDH. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part V. LNCS, vol. 14008, pp. 448–471. Springer, Cham (Apr 2023). https://doi.org/10.1007/978-3-031-30589-4_16
45. McQuoid, I., Xu, J.: An efficient strong asymmetric PAKE compiler instantiable from group actions. In: Guo, J., Steinfeld, R. (eds.) ASIACRYPT 2023, Part VIII.

LNCS, vol. 14445, pp. 176–207. Springer, Singapore (Dec 2023). https://doi.org/10.1007/978-981-99-8742-9_6
46. Montgomery, H., Zhandry, M.: Full quantum equivalence of group action DLog and CDH, and more. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part I. LNCS, vol. 13791, pp. 3–32. Springer, Cham (Dec 2022). https://doi.org/10.1007/978-3-031-22963-3_1
47. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard (2024). https://doi.org/10.6028/NIST.FIPS.203
48. Okamoto, T., Pointcheval, D.: REACT: Rapid Enhanced-security Asymmetric Cryptosystem Transform. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 159–175. Springer, Berlin, Heidelberg (Apr 2001). https://doi.org/10.1007/3-540-45353-9_13
49. Pedersen, R.: Threshold oblivious pseudorandom functions from isogeny group actions. In: Bai, S., Persichetti, E. (eds.) Public-Key Cryptography - PKC 2026 - 29th IACR International Conference on Practice and Theory of Public-Key Cryptography, West Palm Beach, FL, USA, May 25-28, 2026, Proceedings, Part III. Lecture Notes in Computer Science, vol. 16553, pp. 121–156. Springer (2026). https://doi.org/10.1007/978-3-032-26737-5_5, https://doi.org/10.1007/978-3-032-26737-5_5
50. Robert, D.: Breaking SIDH in polynomial time. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part V. LNCS, vol. 14008, pp. 472–503. Springer, Cham (Apr 2023). https://doi.org/10.1007/978-3-031-30589-4_17
51. Santos, B.F.D., Gu, Y., Jarecki, S., Krawczyk, H.: Asymmetric PAKE with low computation and communication. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 127–156. Springer, Cham (May / Jun 2022). https://doi.org/10.1007/978-3-031-07085-3_5
52. Schmidt, J.: Requirements for Password-Authenticated Key Agreement (PAKE) Schemes. RFC 8125 (Apr 2017). https://doi.org/10.17487/RFC8125, https://www.rfc-editor.org/info/rfc8125/
53. Shoup, V.: Lower bounds for discrete logarithms and related problems. In: Fumy, W. (ed.) EUROCRYPT'97. LNCS, vol. 1233, pp. 256–266. Springer, Berlin, Heidelberg (May 1997). https://doi.org/10.1007/3-540-69053-0_18
54. WhatsApp: Security of End-to-End Encrypted Backups. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf (September 2021)
55. Xu, J., Zhao, Z.: Just how secure is srp, really? In: Heninger, N., Rosulek, M. (eds.) Advances in Cryptology - CRYPTO 2026 - 46th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-20, 2026, Proceedings, Part X. Lecture Notes in Computer Science, vol. 16809, pp. 331–362. Springer (2026). https://doi.org/10.1007/978-3-032-35428-0_11, https://doi.org/10.1007/978-3-032-35428-0_11
56. Yang, Y., Benhamouda, F., Halevi, S., Krawczyk, H., Rabin, T.: Gold OPRF: Post-quantum oblivious power-residue PRF. In: Blanton, M., Enck, W., Nita-Rotaru, C. (eds.) 2025 IEEE Symposium on Security and Privacy. pp. 259–278. IEEE Computer Society Press (May 2025). https://doi.org/10.1109/SP61157.2025.00116
57. Zhang, C., Zhou, H.S., Katz, J.: An analysis of the algebraic group model. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part IV. LNCS, vol. 13794, pp. 310–322. Springer, Cham (Dec 2022). https://doi.org/10.1007/978-3-031-22972-5_11

A Proof of the saPAKE theorem

We prove Theorem 1 for the protocol in Figure 5.

Proof. **Game $\mathbf{G}_0$:** This is the real protocol execution involving the environment and a dummy adversary. The environment can instruct parties to execute the protocol, choose their inputs and see their outputs. It can also instruct the adversary to insert messages, tamper with messages or compromise parties.

Game $\mathbf{G}_1$: Here we introduce the simulator and the functionality. The functionality $\mathcal{F}$ is $\mathcal{F}_{\mathsf{saPAKE}}$ as in Figure 3 except that it also forwards all the inputs it receives to the simulator Sim. Sim executes the protocol exactly as in Figure 5. In particular, it produces output for the parties. The only difference is that Sim receives the initial input from the functionality. Moreover, Sim answers all the hash queries via lazy sampling, and aborts whenever a collision is found. Finally, the simulator also does the following: when the honest server receives a value c, it decrypts it to (s', h') and uses it to update the file; Sim checks that the password used during registration (sent by $\mathcal{F}$) and s' were used to compute the hash value h'. If that is not the case, it marks the file `invalid`. Note that this marking is only internal and does not modify the execution in any way. Thus we have

$$|\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_0]| \leq \frac{q_{\mathsf{H}}^2}{2^\lambda}$$

Game $\mathbf{G}_2$: In this game, we make Sim simulate the PAKE part without using the honest parties' input for a file not marked `invalid`. When the adversary delivers $s_1^*, \ldots, s_{n'}^*$ to the honest client, Sim simulates a fresh PAKE session for the C by sending $(\mathsf{NewSession}, \mathsf{ssid} = \mathsf{sid}\|\mathsf{PAKE}\|s_1^*, \ldots, s_{n'}^*, \mathsf{C}, \mathsf{S})$ from $\mathcal{F}_{\mathsf{PAKE}}$ to $\mathcal{A}$. If the adversary queries $(\mathsf{TestPwd}, \ldots, h')$ against the C then three possible situations can occur: 1) the adversary computed the password guess via the chain of salted hashes, 2) it used the value from a stolen password file or an honest update received during a successful impersonation in a prior session, or 3) it derived the guess without using either of the two. In case 1), Sim queries $\mathsf{TestPwd}$ to $\mathcal{F}_{\mathsf{saPAKE}}$ with the extracted password guess. Similarly, in case 3), Sim queries $\mathsf{TestPwd}$ with $\emptyset$. Moreover, it forwards the $\mathcal{F}_{\mathsf{saPAKE}}$ replies to the adversary. For case 2), Sim checks that either: $h', s_1, \ldots, s_n$ is contained in the leaked file and the received salt list $s_1, \ldots, s_n$; h', s' was delivered by an honest client to the adversary during a successful impersonation attack and the received salt list is s'; there exists $h, s_1, \ldots, s_n$ (resp. h, s') such that the received hash list is $s_1, \ldots, s_n, s_1^*, \ldots, s_m^*$ (resp. $s', s_1^*, \ldots, s_m^*$) and $h' = \mathsf{H}(\ldots(\mathsf{H}(h\|s_1^*)\|s_2^*)\ldots)\|s_m^*)$. If any of the conditions holds Sim queries $\mathsf{Impersonate}$. Otherwise it sends $\mathsf{TestPwd}$ on $\emptyset$. In any case it delivers the output of the query to the adversary.

For the honest server, after sending to the client the salt list $s_1, \ldots, s_n$, Sim simulates a fresh PAKE session by sending $(\mathsf{NewSession}, \mathsf{ssid} = \|\mathsf{PAKE}\|s_1, \ldots,$

$s_n, \mathsf{S}, \mathsf{C}$) from $\mathcal{F}_{\mathsf{PAKE}}$ to $\mathcal{A}$. On TestPwd it acts as before: if a password can be extracted from the RO queries using the sent salts, Sim sends TestPwd to $\mathcal{F}_{\mathsf{saPAKE}}$ and forwards the reply, if the value from the leaked file is used, it outputs CorrectGuess. Finally, if no password could be extracted, Sim sends TestPwd with $\emptyset$.

When a party is corrupt and Sim receives a NewSession query for $\mathcal{F}_{\mathsf{PAKE}}$, it creates the record as $\mathcal{F}_{\mathsf{PAKE}}$ would and answers TestPwd queries as described above. After the simulation of any of the above TestPwd, Sim marks the attacked PAKE record either interrupted or compromised, according to the result of the attack. On NewKey, we have to consider multiple situations:

1. For a compromised or interrupted record, Sim acts as $\mathcal{F}_{\mathsf{PAKE}}$.
2. For fresh records where the other party did not receive a PAKE key for the same ssid, Sim samples a random key.
3. If no record was attacked, the server is corrupt, and one party has already received its PAKE key for the same ssid: Sim checks whether the server's NewSession input is the hash value from a leaked file or from a previous update following a successful impersonation, possibly extended by further salted hashes, and whether the client received that file's salt list followed by exactly the extension salts. If so, Sim queries Impersonate to $\mathcal{F}_{\mathsf{saPAKE}}$. Otherwise, Sim extracts a password pw from the RO queries corresponding to the server's NewSession input and the salts delivered to the client, and queries TestPwd to $\mathcal{F}_{\mathsf{saPAKE}}$ with pw, using $\perp$ if extraction fails. In either case, Sim delivers the same key to the other party on CorrectGuess and samples a fresh key otherwise.
4. If no record was attacked, the client is corrupt, and the other party computed the PAKE key for ssid: if in NewSession the client used the value from a leaked file, it outputs the same key. Otherwise, Sim extracts the password guess from the hash record and queries TestPwd against the Server with the extracted value, setting it to $\perp$ if no extraction was possible. It outputs the same key on CorrectGuess and samples a fresh one otherwise.
5. If no record was attacked, parties are honest, and the other party computed the PAKE key for ssid: Sim delivers the same key. Moreover, if the same key was delivered, before simulating the first wPoPWK message, Sim verifies whether the client password in the UsrSession query was the correct one, and if it was not, Sim samples a fresh key to use instead of the PAKE one. Note that this last change is only temporary, as Sim will not have access to the honest client password in the final simulation.

Whenever Sim is able to extract a password guess from TestPwd, or the values in the leaked file are used, Sim outputs the guess result and the keys exactly as in $\mathbf{G}_1$. The only difference occurs when the adversary uses a fresh value (either as corrupt party's NewSession or as input to TestPwd) such that no password can be extracted from the hash record and in $\mathbf{G}_1$ the guess would nonetheless be correct. In $\mathbf{G}_2$ Sim will instead return WrongGuess or output different password to the parties. Note that at most one such guess per honest party is possible per session, one against the current client

password guess and one against the password file. Over q_s sessions, we have

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq \frac{2q_s}{2^\lambda}$$

Before moving on, we explain why this hybrid imposes the restriction that the file was not maliciously updated. When that happens, the adversary might not follow the protocol and inject a different password. Hence, relying on the $\mathcal{F}_{\mathsf{saPAKE}}$ attack interfaces would no longer work. Instead, after receiving a maliciously computed file update that would be accepted by the honest S, Sim updates the password file and from that moment on simulates the execution as in $\mathbf{G}_1$.

Game $\mathbf{G}_3$: In this game we change how Sim executes the wPoPWK and the final message when both parties are honest and the PAKE execution was not attacked. To avoid clutter of the proof, we mention here that whenever Sim makes the server abort, Sim updates the password file as in the code for $b = 0$. We divide this game into a series of incremental steps:

1. If any message received by the party was not computed by the simulator, Sim sends $(\mathsf{TestPwd}, \mathsf{ssid}, \mathsf{P}, \emptyset)$ and $\mathsf{NewKey}, \mathsf{ssid}, \mathsf{P}, \perp$. Note that the party always fails in $\mathbf{G}_3$ upon receiving an adversarial message, whereas in $\mathbf{G}_2$ it fails if the message does not decrypt with the PAKE key. For the two games to differ, the adversary would have to compute a valid ciphertext for an unknown key, which would break the INT-CTXT game. Assume that there exists $\mathcal{Z}'$ that can distinguish between $\mathbf{G}_3$ and $\mathbf{G}_2$ by causing the different output behaviour, then we can construct an adversary $\mathcal{B}_0$ against the INT-CTXT as follows: $\mathcal{B}_0$ chooses $i^* \in \{1, \ldots, q_s\}$. If any party is corrupt, the PAKE execution was attacked or there is an adversarial query $\mathsf{KDF}(K, \texttt{enc-label})$ $\mathcal{B}_0$ aborts. Else, instead of computing K' it uses the encryption oracle of the INT-CTXT experiment to compute the honest messages and retrieves the plaintext from the queries instead of decrypting it explicitly. If during this recovery the output behaviour w.r.t. $\mathbf{G}_2$ differs then $\mathcal{B}_0$ wins the INT-CTXT using the message msg^* that causes the mismatch. Note that at the beginning, the adversary might have sent a different set of salts to the client causing the parties to exchange a different key, in such occasions $\mathcal{B}_0$ will compute only one party's messages using the encryption oracle.
2. Sim does not test the password after the saPAKE and does not switch to a different key in case of mismatch. If the honest parties do not share the same key, Sim computes the first message as an encryption of a random plaintext, if the message is delivered Sim queries $\mathsf{TestPwd}$ with $\emptyset$ and makes the party output $\perp$ via NewKey. If the parties share the same key, Sim replaces every encrypted wPoPWK message with an encryption of an independent random plaintext of the same length. When the adversary delivers the first honest message to the intended recipient, Sim queries $(\mathsf{TestAbort}, \mathsf{ssid}, \mathsf{P})$ to verify whether the parties exchange the same key. Note that upon an unsuccessful execution $\mathcal{F}$ makes P abort.

If it does not receive Abort it continues the execution. Upon computing the client final message, SIM encrypts a random plaintext but keeps the intended plaintext in the record, it then sends NewKey to $\mathcal{F}_{\mathsf{saPAKE}}$. When SIM receives the final client message it updates the file as for the server code, except that it uses the recorded plaintext instead of decrypting the update message and queries NewKey to $\mathcal{F}_{\mathsf{saPAKE}}$ to produce the output. Note that the distribution of the final output is the same as $\mathbf{G}_2$ in the passive case. The remaining difference concerns all encrypted wPoPWK messages and the final update. Let m be the maximum number of encrypted messages per session. Starting from the experiment after step 1, denoted $\mathbf{G}_2^{0,m}$, consider hybrids $\mathbf{G}_2^{i,j}$ for $i \in \{1,\ldots,q_s\}$ and $j \in \{0,\ldots,m\}$, with $\mathbf{G}_2^{i,0} = \mathbf{G}_2^{i-1,m}$. In $\mathbf{G}_2^{i,j}$, all encrypted messages in the first $i-1$ sessions and the first j encrypted messages in session i encrypt independent random plaintexts of the corresponding lengths; the remaining messages encrypt their original plaintexts. If there is $\mathcal{Z}'$ that distinguishes between $\mathbf{G}_2^{0,m}$ and $\mathbf{G}_2^{q_s,m} = \mathbf{G}_3$ there exist two adjacent games described above that $\mathcal{Z}'$ can distinguish with non negligible advantage. Then, we construct an adversary against the IND-CPA game. $\mathcal{B}_1$ samples $i^* \xleftarrow{\$} \{1,\ldots,q_s\}$ and $j^* \xleftarrow{\$} \{1,\ldots,m\}$ and simulates $\mathbf{G}_2^{i^*,j^*-1}$, it aborts on the same condition described before for $\mathcal{B}_0$ and otherwise uses the IND-CPA challenge key in place of the encryption key for the selected message. At the j^*-th encrypted message of session i^*, it submits the original plaintext and an independent random plaintext of the same length to the challenger and uses the returned ciphertext c^*. It generates all other ciphertexts under the challenge key via the encryption oracle. If c^* encrypts the original plaintext, the execution is distributed as $\mathbf{G}_2^{i^*,j^*-1}$; if it encrypts the random plaintext, it is distributed as $\mathbf{G}_2^{i^*,j^*}$. $\mathcal{B}_1$ uses $\mathcal{Z}'$'s output to distinguish the two cases.

Combining the two steps, we obtain

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq 2q_s\mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathcal{B}_0,\mathsf{AE}}(\lambda) + mq_s\mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathcal{B}_1,\mathsf{AE}}(\lambda) + \frac{q_h}{2^\lambda}.$$

Game $\mathbf{G}_4$: In this game we change how SIM executes the first message in wPoPWK when both parties are honest, at least one side was attacked and all attacks were unsuccessful. SIM encrypts the first wPoPWK message with a random plaintext. If the attacked party receives a message after the PAKE, SIM queries NewKey; otherwise, if the first message after the PAKE is received by an unattacked party, SIM sends TestPwd with $\emptyset$ and then NewKey. In both cases, it then stops the execution. For the server, SIM also updates the file as in the code for $b = 0$. The two differences are again the distribution of the transcript and a correctly decrypted adversarial message, and a distinguisher can be reduced to a distinguisher for the IND-CPA or INT-CTXT game similarly to what is described in $\mathbf{G}_3$. Thus we have

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_3]| \leq 2q_s\mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathcal{B}_0,\mathsf{AE}}(\lambda) + q_s\mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathcal{B}_1,\mathsf{AE}}(\lambda).$$

Game $\mathbf{G}_5$: In this game we change how SIM simulates the wPoPWK protocol when the parties are not honest or at least one party has been attacked. Note that wPoPWK UC-realizes $\mathcal{F}_{\text{wPoPWK}}$, hence there exists a simulator $\text{SIM}_{\text{wPoPWK}}$ such that for any environment $\mathcal{Z}''$ the simulated interaction is indistinguishable from the real one. SIM will internally run $\mathcal{F}_{\text{wPoPWK}}$ and $\text{SIM}_{\text{wPoPWK}}$ and relay messages between the two. In particular, SIM will query GENSTATEMENT to $\mathcal{F}_{\text{wPoPWK}}$ when receiving a StorePwdFile message. During a session SIM generates the parties' messages in wPoPWK by either giving (PROVE, ...) or (VERIFY, ...) inputs to $\mathcal{F}_{\text{wPoPWK}}$ or forwarding adversarial messages to $\text{SIM}_{\text{wPoPWK}}$. SIM then forwards the messages output by $\text{SIM}_{\text{wPoPWK}}$ to $\mathcal{A}$. We have that

$$|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| \leq \mathbf{Dist}_{\mathcal{Z}'}^{\text{wPoPWK},\{\mathcal{F}_{\text{wPoPWK}},\text{SIM}_{\text{wPoPWK}}\}}(\lambda)$$

Game $\mathbf{G}_6$: In this game we change how SIM executes the wPoPWK and the final message when both parties are honest, at least one side was attacked and all attacks were successful. For the unattacked party, it does not run PROVE (resp. VERIFY) anymore and instead computes a ciphertext from a random plaintext, and upon receiving a message it sends TestPwd with $\emptyset$ and NewKey. Note that this reduces again to IND-CPA and INT-CTXT as before. If instead the attacked party receives a message that does not decrypt, SIM queries (NewKey, ..., $\perp$). For the server SIM also updates the file as in the code for $b = 0$. If instead the party that was attacked only receives messages that successfully decrypt, SIM does the following: if the attacked party is the server (resp. client), SIM continues executing $\text{SIM}_{\text{wPoPWK}}$ for the honest party; whenever it queries PROVE (resp. VERIFY) on behalf of the adversary if SIM was able to extract a password guess from TestPwd query to $\mathcal{F}_{\text{PAKE}}$, SIM verifies that the extracted password was the same used in PROVE (resp. VERIFY) and sets $b = 1$ if that's the case and $b = 0$ otherwise. If no password could be extracted from the PAKE's TestPwd query, SIM proceeds as follows: if the attacked party is the server, SIM queries TestPwd to $\mathcal{F}_{\text{saPAKE}}$ with the value used in PROVE and sets $b = 1$ on CorrectGuess and $b = 0$ otherwise; if the attacked party is the client, SIM sets $b = 1$ if the last entry of VERIFY is $\perp$ or if it is $\text{pw} \neq \perp$ but $\text{SIM}_{\text{wPoPWK}}$ sent OfflineTestPwd to $\mathcal{F}_{\text{wPoPWK}}$, which returned CorrectGuess, and $b = 0$ otherwise. If $\text{SIM}_{\text{wPoPWK}}$ queries OfflineTestPwd and $\mathcal{F}_{\text{wPoPWK}}$ would test the guess against the attacked party transcript, SIM tests the guess with the value used in TestPwd against that same party and replies accordingly. If the leaked password file was used, SIM queries OfflineTestPwd to $\mathcal{F}$ and forwards the reply. Finally, SIM computes the client's ciphertext c using the adversarial password if a successful TestPwd was sent, or the honest password if the adversary used Impersonate, and makes $\mathcal{F}$ output the appropriate key via NewKey. For the attacked server, it updates the file as in the code if the message decrypts and then makes $\mathcal{F}$ output the correct key by setting NewKey appropriately. If $\text{SIM}_{\text{wPoPWK}}$ sends FINALIZE and no PROVE query was sent, SIM makes the server update the file for $b = 0$ and output $\perp$ via NewKey (and possibly

TestPwd with $\emptyset$). The outputs are distributed as in $\mathbf{G}_5$, and the only difference is again in the transcript. For the same reason described in the previous game, we have

$$|\Pr[\mathbf{G}_6] - \Pr[\mathbf{G}_5]| \leq 2q_s \mathsf{Adv}^{\mathrm{int-ctxt}}_{\mathcal{B}_0,\mathsf{AE}}(\lambda) + q_s \mathsf{Adv}^{\mathrm{ind-cpa}}_{\mathcal{B}_1,\mathsf{AE}}(\lambda).$$

Note that SIM can follow the above description in the case of one party successfully attacked and one party unsuccessfully attacked. That is because the unsuccessful attack makes that party fail regardless of the message it receives, exactly as in the unattacked case when the other party was successfully attacked.

Game $\mathbf{G}_7$: We now change how SIM simulates wPoPWK when one party is corrupt. If the parties do not share the same PAKE key SIM can simulate the honest party code and messages exactly as described above, including the occasion where an adversarial TestPwd against the honest party was sent to $\mathcal{F}_{\mathsf{PAKE}}$. OfflineTestPwd query and simulation before PROVE (resp. VERIFY) query on behalf of the malicious client (resp. server) are handled as above. Otherwise, if the C is malicious and the parties do share the same key, SIM sets b after receiving the PROVE queries as follows: if a corrupt password guess was extracted from the PAKE execution, SIM sets $b = 1$ if the same value is used, and $b = 0$ otherwise. If no password guess was extracted, SIM queries TestPwd with pw' from the PROVE query and sets $b = 1$ on CorrectGuess and $b = 0$ otherwise. For the corrupt server interacting with the client, when receiving VERIFY, SIM sets $b = 1$ if either the same password was used for both VERIFY and TestPwd, or no password was used in VERIFY and SIM queried Impersonate, or SIM queried Impersonate and the password used in VERIFY makes $\mathcal{F}_{\mathsf{saPAKE}}$ return CorrectGuess when the value is used as a guess in an OfflineTestPwd query. Note that these changes do not modify the view of the adversary and thus we have again

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

Game $\mathbf{G}_8$: We now make SIM simulate the password file creation and update without using the secret values. To that end, when SIM receives StorePwdFile, sid from $\mathcal{F}_{\mathsf{saPAKE}}$, SIM samples fresh s_1, h_1 and x_{pw} using $\mathrm{SIM}_{\mathrm{wPoPWK}}$ and records them as the password file. When SIM computes c for the honest client and the parties are not attacked, it samples random h', s' and then makes the honest server update the file according to the result of TestAbort. SIM computes the client message in the same way if the client was attacked and the adversary used the leaked password file. To simulate server's file ratcheting in case of unsuccessful execution, SIM samples random s', h' and updates the file with such values. When a valid $c = \mathsf{AE.Enc}(K', (s', h'))$ is maliciously computed and sent to the server, and SIM simulated a correct wPoPWK with the adversary, SIM does the following: it looks at the random oracle query to find $h' = \mathsf{H}([\mathsf{pw}^*] \| s')$. If no query is found, let $\mathsf{pw}^* = \perp$ and in any case if the value differs from the password that SIM extracted during wPoPWK, it marks the file $(\texttt{invalid}, \mathsf{pw}^*)$. On (StealPwdFile, sid),

SIM outputs the simulated file. Upon receiving $\mathsf{H}([pw^*]\|s_1)$, with s_1 the first salt in a leaked file not marked invalid, SIM queries $(\mathsf{OfflineTestPwd}, \mathsf{sid}, pw^*)$ and on CorrectGuess does the following: let n be the number of salts present in the last not marked leaked file $(s_1, \ldots, [s_n], [h_n], *)$ such that h_n was not programmed as the output of a hash query, then for every $i \in \{1, \ldots, n\}$ SIM looks for leaked not invalid file of the form $(s_1, \ldots, s_i, h_i, *)$. If that is the case it programs $h_i = \mathsf{H}(h_{i-1}\|s_i)$, and samples random h_i otherwise. Otherwise, on an adversarial query $\mathsf{H}([h^*]\|s^*)$ the simulator checks if the adversary compromised a password file not marked $(\texttt{invalid}, *)$ of the form $(s_1, \ldots, s_n, h_n, *)$ with $s_n = s^*$ at some point. If that is the case, the simulator traces the RO queries back to the password. Concretely, it checks if there exists a sequence of H queries $(\mathsf{H}, h_{i-1}\|s_i, [h_i])$ for $s_1, \ldots, s_{n-1}$ and if $h_{n-1} = h^*$. If all the checks pass, then h_0 is the password and SIM sends $(\mathsf{OfflineTestPwd}, \mathsf{sid}, h_0)$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and programs the output to h_n if it receives CorrectGuess. In particular, $\mathcal{A}$ can now consistently verify the hash chain leading to the compromised h_n. If the records $(\mathsf{H}, h_{i-1}\|s_i, [h_i])$ exist up to $i > 1$ or $h^* = h_{n-1}$ and the file $(s_1, \ldots, s_{i-1}, h_{i-1}, *)$ (resp. $(s_1, \ldots, s_{n-1}, h_{n-1}, *)$) was leaked SIM outputs h_n. Otherwise, SIM checks if the client ever sent an update of the form $(\bar{h}, s^*)$ following an adversarial impersonation, and in that case queries $\mathsf{OfflineTestPwd}, \mathsf{sid}, h^*$ to $\mathcal{F}$ and programs the query to $\bar{h}$ if the functionality returns CorrectGuess. The only difference to $\mathbf{G}_7$ happens if the adversary can guess the right s_n to compute h_n before the file is leaked. [7] Among q_h hash queries, we have

$$|\Pr[\mathbf{G}_8] - \Pr[\mathbf{G}_7]| \leq \frac{q_h}{2^\lambda}$$

Game $\mathbf{G}_9$: In this game we simulate an execution with an invalid file. First note that in the session in which the file became `invalid`, the adversary must have input the correct password during the proof of password knowledge to convince the server, and SIM extracted it and programmed x_{pw} to it. In particular this means that SIM has the same file as a real Server would and can simulate perfectly the interaction with the adversary. If the adversary remains passive SIM simulates the execution of PAKE as before. To decide whether the party will successfully decrypt the first simulated wPoPWK message, SIM queries TestPwd against the client record using the malicious password recorded during the adversarial update (and using $\emptyset$ if no password was extracted). On WrongGuess it makes the party fail querying NewKey (and dummy TestPwd for the server). Otherwise, it sets $b = 0$ in the wPoPWK part. In that case, SIM ensures that the server outputs $\perp$ via a dummy TestPwd query. Note that this works because the file is `invalid` and the password used for the saPAKE part will not work in the wPoPWK part (or vice versa). Finally, SIM executes the client interaction with an adversary exactly as described before. Note that an honest server always rejects if it

[7] As discussed in Footnote 4, we exclude the case of $\mathcal{A}$ stealing the file during a running execution.

has an invalid file, the interaction of the adversary with the client follows the same argument as above and remains indistinguishable and the interaction with the server follows exactly the one in the previous game. Thus we have

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

Note that an adversary might subsequently make the file back to a valid state by updating the PAKE part with the original password. This can be detected by SIM via lookup on the hash query and in that case the `invalid` marking is removed.

Game $\mathbf{G}_{10}$: Note that SIM does not make use of the secret values forwarded by $\mathcal{F}$ and makes the functionality produce the output for the parties. We can therefore remove the changes made in $\mathbf{G}_1$ and use $\mathcal{F} = \mathcal{F}_{\mathsf{saPAKE}}$. This concludes the proof.

B Salt Update

There are multiple ways of updating salts, which essentially lead to the same security but with different trade-offs between storage, communication, and computational complexity of the participants. Possible options depending on the application:

1. with upper bound B on unsuccessful authentication attempts:
   (a) The client can prepare B hash, salt pairs and send them over to S during the first initialization and in any following update.
   (b) As 1a, but instead of sampling B salt values, the client only samples one salt $s_B \leftarrow \{0,1\}^\lambda$ and sets the $B-i$th salt for $i = 1, \ldots, B-1$ to be the ith iterated hash of s_B.
   (c) The client only sends a single hash-salt pair per initialization or update. The server stores B salts together with the file; after leaking s_i, the server deletes the corresponding hash
   (d) At initialization and each successful reset, the client samples a seed and hashes it B times to obtain the initial salt, and sends the seed and salt to the server. After each failed attempt, the server moves backwards along the chain and updates the password hash as usual. It sends only the current salt and its index, from which the client reconstructs all previous salts.
2. for unbounded number of authentication attempts:
   (a) initially only a single Hash-salt pair is stored, the hash gets updated and the salt list is extended after unsuccessful authorization.

When SIM appends a value to list, if such list does not exist, SIM initializes it. SIM retrieves the latest $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1), h_1, x, n, j, * \rangle$, that is, in order: the biggest j and the biggest n for which the file record exists. Once $\langle \mathsf{pwfile} \rangle$ is marked LEAKED all the subsequent $\langle \mathsf{pwfile} \rangle$ not marked `invalid` are marked LEAKED. An `invalid` $\langle \mathsf{pwfile} \rangle$ is never marked LEAKED. No concurrent session can be started.

On input $(\mathsf{StorePwdFile}, \mathsf{C}, \mathsf{sid})$ **from** $\mathcal{F}_{\mathsf{saPAKE}}$**:**

SPD.1 Sample $s_1, h_1, \xleftarrow{\$} \{0,1\}^\lambda$.

SPD.2 Simulate $\text{SIM}_{\text{wPoPWK}}$ on (GENSTATEMENT, $\mathsf{sid} \| \text{wPoPWK}$) and call the simulated statement x.

SPD.3 Save $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1), h_1, x, 1, 1, \perp \rangle$ and send $(\mathsf{StorePwdFile}, \mathsf{C}, \mathsf{sid})$ to $\mathcal{A}$.

On input $(\mathsf{StealPwdFile}, \mathsf{sid})$ **from** $\mathcal{A}$**:**

SPF.1 If $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), h_n, x, [n], [j], * \rangle$ is marked $\mathsf{CorrectGuess}, [\mathsf{pw}]$ or `invalid`, $*$ send $\langle (s_1, \ldots, s_n), h_n, x \rangle$ to $\mathcal{A}$.

SPF.2 Retrieve $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), h_n, x, [n], [j], * \rangle$. Forward the query to $\mathcal{F}_{\mathsf{saPAKE}}$.

SPF.3 On $\mathsf{NoPwdFile}$, forward it to $\mathcal{A}$ and exit the query.

SPF.4 On $\mathsf{PwFileStolen}$ send LEAKSTATEMENT to $\mathcal{S}_{\text{wPoPWK}}$ and mark $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), h_n, x, n, j, * \rangle$ LEAKED and send $\langle (s_1, \ldots, s_n), h_n, x \rangle$ to $\mathcal{A}$. Moreover, if $fL[\mathsf{sid}, \mathsf{C}, j]$ is undefined, set $fL[\mathsf{sid}, \mathsf{C}, j] \leftarrow n$.

On input $(\mathsf{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S})$ **from** $\mathcal{F}_{\mathsf{saPAKE}}$**:**

US.1 If this is the first $\mathsf{UsrSession}$ for ssid record $\langle \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \perp, \ldots \rangle$; wait for message $(\mathsf{ssid}, [s_1], \ldots)$ from $\mathcal{A}$.

US.2 Else ignore the query.

On input $(\mathsf{SrvSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S})$ **from** $\mathcal{F}_{\mathsf{saPAKE}}$**:**

SS.1 If this is not the first $\mathsf{SrvSession}$ message for ssid ignore the query.

SS.2 Else retrieve $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), * \rangle$, record $\langle \mathsf{ssid}, \mathsf{S}, \mathsf{C}, s_1, \ldots, s_n, \perp, \ldots \rangle$ and send $(\mathsf{ssid}, s_1, \ldots, s_n)$ to C.

SS.3 Send $(\mathsf{NewSession}, \mathsf{ssid} \| (s_1, \ldots, s_n), \mathsf{S}, \mathsf{C}, \mathsf{server})$ to $\mathcal{A}$ and record $\langle \mathsf{ssid}^*, \mathsf{S}, \mathsf{C}, \perp \rangle$.

On message $(\mathsf{ssid}, s_1, \ldots, s_n)$ **from** $\mathcal{A}$ **to an honest** C**:**

S.1 Retrieve $\langle \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \perp, \ldots \rangle$. If no such record exists wait for $(\mathsf{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S})$ message to create the record, then continue.

S.2 Record $\langle \mathsf{ssid}, \mathsf{C}, \mathsf{S}, s_1, \ldots, s_n, \perp, \ldots \rangle$

S.3 Let $\mathsf{ssid}^* = \mathsf{ssid} \| (s_1, \ldots, s_n)$ and send $(\mathsf{NewSession}, \mathsf{ssid}^*, \mathsf{C}, \mathsf{S}, \mathsf{client})$ to $\mathcal{A}$ and record $\langle \mathsf{ssid}^*, \mathsf{C}, \mathsf{S}, \perp \rangle$.

On input $(\mathsf{NewSession}, \mathsf{ssid}^*, \mathsf{P}, \mathsf{P}', \mathsf{pw}^*, \mathtt{role})$ **from** $\mathcal{A}$ **for corrupt** P**:**

NS.1 If this is the first $\mathsf{NewSession}$ message for P and ssid^*, record $\langle \mathsf{ssid}^*, \mathsf{P}, \mathsf{P}', \mathsf{pw}^* \rangle$.

NS.2 If $\mathsf{P}' = \mathsf{S}$ and $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), h_n, x, n, j, [\mathsf{pw}'] \rangle$ is marked `invalid`, $*$ execute the rest of the protocol exactly, initializing $\langle \mathsf{ssid}^*, \mathsf{S}, \mathsf{C}, h_n \rangle$ when needed and using x from $\langle \mathsf{pwfile} \rangle$ for the wPoPWK part. Moreover, when $\mathsf{SrvSession}$ is sent, or if it has already been sent: query $(\mathsf{TestPwd}, *, \mathsf{S}, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and use NewKey to produce the right output.

NS.3 Send $(\mathsf{NewSession}, \mathsf{ssid}^*, \mathsf{P}, \mathsf{P}', \mathtt{role})$ to $\mathcal{A}$.

On input $(\mathsf{TestPwd}, \mathsf{ssid}^*, \mathsf{P}, \mathsf{pw}^*)$ **from** $\mathcal{A}$ **to** $\mathcal{F}_{\mathsf{PAKE}}$**:**

TP.1 Retrieve $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], [\mathsf{pw}] \rangle$, if no record exists or the record is not marked `fresh` ignore the query.

TP.2 If P is corrupt, check if $\mathsf{pw} = \mathsf{pw}^*$. Return $\mathsf{CorrectGuess}$ and mark the record `compromised` if that's the case, return $\mathsf{WrongGuess}$ and mark it `interrupted` otherwise.

TP.3 if $\mathsf{P} = \mathsf{S}$ and $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), h_n, x, n, j, [\mathsf{pw}'] \rangle$ is marked `invalid`, $*$ overwrite $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], h_n \rangle$ and execute the rest of the server code exactly, using x from $\langle file \rangle$ for the wPoPWK part. Moreover, send $(\mathsf{TestPwd}, *, \mathsf{S}, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and use NewKey to produce the right output.

TP.4 If $\mathsf{P} = \mathsf{S}$ and $\langle \mathsf{pwfile} \rangle$ is marked `invalid`, pw' do the following: if $\mathsf{pw}' \neq \mathsf{pw}^*$ or $\mathsf{pw}' = \perp$ return $\mathsf{WrongGuess}$ and mark the record `interrupted`. Else return $\mathsf{CorrectGuess}$ and mark the record `compromised`.

TP.5 Otherwise retrieve $\langle \mathsf{ssid}, \mathsf{P}, *, s_1, \ldots, s_n, \ldots \rangle$

TP.6 If there exists a sequence of records $\langle \mathsf{H}, h_{i-1} \| s_i, h_i \rangle$ with $i = 1, \ldots, n$ and $h_n = \mathsf{pw}^*$ send $(\mathsf{TestPwd}, \mathsf{ssid}, \mathsf{P}, h_0)$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and forward the reply to $\mathcal{A}$. Moreover, on $\mathsf{CorrectGuess}$ mark the record $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], \perp \rangle$ `compromised`, otherwise mark it `interrupted`.

TP.7 Else, if $\mathsf{P} = \mathsf{C}$, parse $\mathsf{ssid}^* = \mathsf{ssid} \| (s_1, \ldots, s_n)$ and look for a leaked record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_t), [h_t], *, t, *, * \rangle$ with $1 \leq t \leq n$ and hash records $\langle \mathsf{H}, h_{i-1} \| s_i, [h_i] \rangle$ for $i = t+1, \ldots, n$, ending at $h_n = \mathsf{pw}^*$. If such records exist, send $(\mathsf{Impersonate}, \mathsf{ssid})$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and forward the reply to $\mathcal{A}$. Moreover, on $\mathsf{CorrectGuess}$ mark the record $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], \perp \rangle$ `compromised`, otherwise mark it `interrupted`.

TP.8 Else, if $\mathsf{P} = \mathsf{C}$ and there exists $(s_1, [h_1]) \in L$ such that: $n = 1$ and $h_1 = \mathsf{pw}^*$ or $\langle \mathsf{H}, h_{i-1} \| s_i, [h_i] \rangle$ for $i = 2, \ldots, n$ with $h_n = \mathsf{pw}^*$. If so, send $(\mathsf{Impersonate}, \mathsf{ssid})$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and forward the reply to $\mathcal{A}$. Moreover, on $\mathsf{CorrectGuess}$ mark the record $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], \perp \rangle$ `compromised`, otherwise mark it `interrupted`.

TP.9 Else if $\mathsf{P} = \mathsf{S}$, $\langle \mathsf{pwfile} \rangle$ is marked LEAKED and $\mathsf{pw}^* = h_n$ output $\mathsf{CorrectGuess}$ to $\mathcal{A}$ and mark the record $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], \perp \rangle$ `compromised`.

TP.10 Else send $(\mathsf{TestPwd}, \mathsf{ssid}, \mathsf{P}, \emptyset)$ to $\mathcal{F}_{\mathsf{saPAKE}}$, mark the record $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], \perp \rangle$ `interrupted` and output $\mathsf{WrongGuess}$ to $\mathcal{A}$.

On input $(\mathsf{NewKey}, \mathsf{ssid}^*, \mathsf{P}, K^*, \mathsf{tr}^*)$ **from** $\mathcal{A}$ **to** $\mathcal{F}_{\mathsf{PAKE}}$**:** //key selection

NK.1 If there is no record $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], [\mathsf{pw}] \rangle$, the record is marked `completed`, or the record is marked `fresh` and there is no record $\langle \mathsf{ssid}^*, \mathsf{P}', \mathsf{P}, [\mathsf{pw}'] \rangle$ ignore the query.

NK.2 If tr^* was given as output to P' and either record was ever marked `compromised` or `interrupted`, ignore the query. Else set $\mathsf{tr} = \mathsf{tr}^*$.

NK.3 If both P and P' are corrupted execute $\mathcal{F}_{\mathsf{PAKE}}$ code for NewKey on records $\langle \mathsf{ssid}^*, \mathsf{P}, \mathsf{P}', \mathsf{pw} \rangle$ and $\langle \mathsf{ssid}^*, \mathsf{P}', \mathsf{P}, \mathsf{pw}' \rangle$, then exit the query.

NK.4 If $\langle \mathsf{ssid}^*, \mathsf{P}, \mathsf{P}', [\mathsf{pw}] \rangle$ is marked `compromised`, set $K = K^*$

Fig. 7. PAKE-to-saPAKE simulator 1

NK.5 Else if $\langle \mathsf{ssid}^*, \mathsf{P}, \mathsf{P}', [\mathsf{pw}] \rangle$ is marked `interrupted`, sample $K \xleftarrow{\$} \{0,1\}^\lambda$.

NK.6 Else if $\langle \mathsf{ssid}^*, \mathsf{P}, \mathsf{P}', [\mathsf{pw}] \rangle$ is marked `fresh` and $\langle \mathsf{ssid}^*, \mathsf{P}', \mathsf{P}, * \rangle$ was marked `interrupted` or `compromised` sample $K \xleftarrow{\$} \{0,1\}^\lambda$.

NK.7 Else if P is corrupted: if $\langle \mathsf{ssid}^*, \mathsf{P}', \mathsf{P}, * \rangle$ was marked CorrectGuess retrieve K', tr' delivered to P' and set $K, \mathsf{tr} = K', \mathsf{tr}'$, else sample $K \xleftarrow{\$} \{0,1\}^\lambda$.

NK.8 Else if $\langle \mathsf{ssid}^*, \mathsf{P}', \mathsf{P}, * \rangle$ is not marked `completed` and P' is honest sample $K \xleftarrow{\$} \{0,1\}^\lambda$.

NK.9 Else if P and P' are honest, $\langle \mathsf{ssid}^*, \mathsf{P}', \mathsf{P}, * \rangle$ is marked `completed` and P' was never attacked do the following: retrieve $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, s_1, \ldots, s_n, K_{\mathsf{P}'}, \mathsf{tr} = \mathsf{tr}_{\mathsf{P}'}, \ldots \rangle$ and set $K = K_{\mathsf{P}'}$ and $\mathsf{tr} = \mathsf{tr}_{\mathsf{P}'}$

NK.10 Else, if P' is corrupted and $\langle \mathsf{ssid}^*, \mathsf{P}', \mathsf{P}, [\mathsf{pw}^*] \rangle$ is marked `fresh` or `completed` execute steps TP.4–TP.10 on the honest party excluding record marking and without forwarding the result to $\mathcal{A}$. Mark $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], [\mathsf{pw}] \rangle$ CorrectGuess or WrongGuess according to the result. Moreover, on WrongGuess or if $\langle \mathsf{ssid}^*, \mathsf{P}', \mathsf{P}, * \rangle$ is marked `fresh` sample $K \xleftarrow{\$} \{0,1\}^\lambda$. Else retrieve (K', tr') sent to P' and set $K, \mathsf{tr} = K', \mathsf{tr}'$.

NK.11 Mark $\langle \mathsf{ssid}^*, \mathsf{P}, [\mathsf{P}'], [\mathsf{pw}] \rangle$ `completed`.

NK.12 If P is corrupt output $(\mathsf{ssid}^*, K, \mathsf{tr})$ to $\mathcal{A}$. Else continue.

NK.13 If the first message of wPoPWK is sent by P', exit the query.

NK.14 Else if $\langle \mathsf{ssid}^*, \mathsf{P}, \mathsf{P}', * \rangle$ was not marked `compromised` or CorrectGuess sample $K'' \leftarrow \mathsf{KDF}(K, \texttt{enc-label})$, encrypt a random plaintext with it and send it to P'.

NK.15 Else if $\langle \mathsf{ssid}^*, \mathsf{P}, \mathsf{P}', * \rangle$ is marked `compromised` or CorrectGuess execute $\text{SIM}_{\text{wPoPWK}}$ and $\mathcal{F}_{\text{wPoPWK}}$ on $(\mathit{Prove}, *, \perp)$ if $\mathsf{P} = \mathsf{C}$ and $(\text{VERIFY}, *, \perp)$ if $\mathsf{P} = \mathsf{S}$. If an adversarial VERIFY (resp. PROVE) input has already been submitted for this proof execution, compute b as in step PrVe.1, cases (a)–(e). Otherwise, continue with $b = \perp$. Moreover, let $K'' \leftarrow \mathsf{KDF}(K, \texttt{enc-label})$ and encrypt all $\text{SIM}_{\text{wPoPWK}}$ with it before sending them to P'.

On the first wPoPWK message m from $\mathcal{A}$ to the recipient P:

POK.1 If there is no record $\langle \mathsf{ssid}, \mathsf{P}, [\mathsf{P}'], \perp \rangle$ ignore the query.

POK.2 Check if any of the following conditions hold: 1) the record was not marked `completed`, 2) either the record or $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \perp \rangle$ was marked `compromised` or `interrupted` and the message was computed by SIM on behalf of P', 3) $\langle \mathsf{pwfile} \rangle$ is `invalid`, $\perp$ and the m was computed by SIM on behalf of P', 4) the record was never marked `compromised` or CorrectGuess and the message was not computed by the simulator, 5) the record was marked `compromised` or CorrectGuess and the decryption of m under $K'' \leftarrow \mathsf{KDF}(K_{\mathsf{P}}, \texttt{enc-label})$ fails. If any of the conditions holds do the following:

- If TestPwd or Impersonate was never sent against P, send TestPwd with $\emptyset$ and NewKey to $\mathcal{F}_{\mathsf{saPAKE}}$
- Otherwise send NewKey, $*, \perp$ to $\mathcal{F}_{\mathsf{saPAKE}}$
- If $\mathsf{P} = \mathsf{S}$, sample $h', s' \xleftarrow{\$} \{0,1\}^\lambda$ and record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n, s'), h', x, n+1, j, \mathsf{pw} \rangle$. Moreover, if $\langle \mathsf{pwfile} \rangle$ is marked LEAKED or `invalid` record $\langle \mathsf{H}, h_n \| s', h' \rangle$.

POK.3 Else if the record is marked `completed` and it was marked `compromised` or CorrectGuess decrypt m with $K'' \leftarrow \mathsf{KDF}(K_{\mathsf{P}}, \texttt{enc-label})$ and if the decryption does not fail do the following: execute $\text{SIM}_{\text{wPoPWK}}$ and $\mathcal{F}_{\text{wPoPWK}}$ on $(\text{PROVE}, *, \perp)$ if $\mathsf{P} = \mathsf{C}$ and $(\text{VERIFY}, *, \perp)$ if $\mathsf{P} = \mathsf{S}$. If an adversarial VERIFY (resp. PROVE) input has already been submitted for this proof execution, compute b as in step PrVe.1, cases (a)–(e). Otherwise, continue with $b = \perp$. Encrypt all $\text{SIM}_{\text{wPoPWK}}$ messages with K''. If $\text{SIM}_{\text{wPoPWK}}$ would make the party stop the execution or decryption fails send TestPwd with $\emptyset$ and NewKey to $\mathcal{F}_{\mathsf{saPAKE}}$ and if the failing party is S sample $h', s' \xleftarrow{\$} \{0,1\}^\lambda$ and record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n, s'), h', x, n+1, j, \mathsf{pw} \rangle$. Moreover, if $\langle \mathsf{pwfile} \rangle$ is marked LEAKED or `invalid` record $\langle \mathsf{H}, h_n \| s', h' \rangle$.

POK.4 If P and P' are honest and both $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \perp \rangle$ and $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \perp \rangle$ are marked `completed` and m was computed by the simulator do the following: if $\langle \mathsf{pwfile} \rangle$ is not marked `invalid` send TestAbort to $\mathcal{F}_{\mathsf{saPAKE}}$. On CorrectGuess set $b = 1$ and go to the next step, otherwise on output Abort if $\mathsf{P} = \mathsf{S}$ sample $h', s' \xleftarrow{\$} \{0,1\}^\lambda$ and record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n, s'), h', x, n+1, j, \mathsf{pw} \rangle$. Moreover, if $\langle \mathsf{pwfile} \rangle$ is marked LEAKED record $\langle \mathsf{H}, h_n \| s', h' \rangle$.

POK.5 If $\langle \mathsf{pwfile} \rangle$ is `invalid`, $[\mathsf{pw}^*]$ and m was computed by SIM on behalf of P' do the following: send TestPwd, $*$, C, pw^* and on CorrectGuess set $b = 0$, retrieve K'' used to encrypt the message and send the next wPoPWK message as an encryption of a random plaintext. On WrongGuess: if $\mathsf{P} = \mathsf{C}$ send NewKey, $*, \perp$, otherwise send TestPwd with $\emptyset$ against S and NewKey to $\mathcal{F}_{\mathsf{saPAKE}}$ and sample $h', s' \xleftarrow{\$} \{0,1\}^\lambda$, record $\langle \mathsf{H}, h_n \| s', h' \rangle$ and record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n, s'), h', x, n+1, j, \mathsf{pw} \rangle$.

POK.6 Otherwise simulate all wPoPWK messages with random plaintexts. If $\mathcal{A}$ ever delivers a message that was not computed by SIM, or replays a previous message, send TestPwd with $\emptyset$ and NewKey to $\mathcal{F}_{\mathsf{saPAKE}}$. If this happens for S additionally sample $h', s' \xleftarrow{\$} \{0,1\}^\lambda$ and record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n, s'), h', x, n+1, j, \mathsf{pw} \rangle$. Moreover, if $\langle \mathsf{pwfile} \rangle$ is marked LEAKED record $\langle \mathsf{H}, h_n \| s', h' \rangle$.

POK.7 Upon sending or receiving the final wPoPWK message do the following:

(a) If $\mathsf{P} = \mathsf{S}$ and $b = 0$ sample $h', s' \xleftarrow{\$} \{0,1\}^\lambda$ record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n, s'), h', x, n+1, j, [\mathsf{pw}] \rangle$. If pwfile is marked LEAKED or `invalid` record $\langle \mathsf{H}, h_n \| s', h' \rangle$. Send TestPwd with $\emptyset$ if no TestPwd was sent to S, in any case then send NewKey, $*, \perp$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and ignore any subsequent message for ssid to S.

(b) Else if $\mathsf{P} = \mathsf{S}$ and $b = 1$ wait for the update message from the client

(c) If $\mathsf{P} = \mathsf{C}$ sample $h', s' \xleftarrow{\$} \{0,1\}^\lambda$. Moreover if C was successfully attacked with TestPwd and pw^* record $\langle \mathsf{H}, \mathsf{pw}^* \| s', h' \rangle$. If instead C was attacked via Impersonate append (s', h') to L

(d) If $\mathsf{P} = \mathsf{C}$, P was not attacked and $\langle \mathsf{pwfile}, \ldots, [\mathsf{pw}'] \rangle$ is marked LEAKED and $\mathsf{pw}' \neq \perp$ record LEAKED record $\langle \mathsf{H}, \mathsf{pw}' \| s', h' \rangle$

Fig. 8. PAKE-to-saPAKE simulator 2

(e) If $\mathsf{P} = \mathsf{C}$ and P was not attacked encrypt a random plaintext to c. If P was successfully attacked, let c be the encryption of (s', h').
(f) Finally, if $\mathsf{P} = \mathsf{C}$ send c to S and send $\mathsf{NewKey}, *, \mathsf{H}(K)$ to $\mathcal{F}_{\mathsf{saPAKE}}$.

On input $(\mathbf{Prove}, \mathsf{sid}^*, \mathsf{ssid}^*, \mathsf{pw}^*)$ **or** $(\mathbf{Verify}, \mathsf{sid}^*, \mathsf{ssid}^*, \mathsf{pw}^*)$ **from** $\mathcal{S}_{\mathbf{wPoPWK}}$ **on behalf of an adversarial prover or verifier:**

PrVe.1 Execute $\mathcal{F}_{\text{wPoPWK}}$ code. If the honest VERIFY (resp. PROVE) input for this proof execution has not been submitted, use $b = \perp$. Otherwise, compute b as follows, where P denotes the honest party whose password is compared:
(a) Let pw^* be the input to the adversarial VERIFY query if $\mathsf{P} = \mathsf{C}$, or to the adversarial PROVE query if $\mathsf{P} = \mathsf{S}$.
(b) If $\mathsf{pw}^* = \perp$ and $\mathsf{P} = \mathsf{C}$ set $b = 1$ if SIM sent $\mathsf{Impersonate}$ to $\mathcal{F}_{\mathsf{saPAKE}}$. If $\mathsf{TestPwd}, *, \mathsf{pw}''$ was sent and a record $\langle \mathsf{pwfile}, * \rangle$ is marked LEAKED send $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}'')$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and set $b = 1$ on $\mathsf{CorrectGuess}$ and $b = 0$ otherwise.
(c) Else if $\mathsf{pw}^* \neq \perp$ and P was attacked with $\mathsf{TestPwd}, *, \mathsf{pw}''$ query, set $b = 1$ if $\mathsf{pw}^* = \mathsf{pw}''$ and $b = 0$ otherwise.
(d) Else if $\mathsf{pw}^* \neq \perp$ and $\mathsf{P} = \mathsf{S}$ was attacked with the leaked password file send $\mathsf{TestPwd}, *, \mathsf{P}, \mathsf{pw}^*$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and set $b = 1$ on $\mathsf{CorrectGuess}$ and $b = 0$ otherwise.
(e) Else if $\mathsf{pw}^* \neq \perp$ and $\mathsf{P} = \mathsf{C}$ was attacked with $\mathsf{Impersonate}$, send $\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}^*$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and set $b = 1$ on $\mathsf{CorrectGuess}$ and $b = 0$ otherwise.

On input $(\mathbf{Finalize}, \mathsf{sid}^*, \mathsf{ssid}^*)$ **from** $\mathcal{S}_{\mathbf{wPoPWK}}$**:**

PrVe.1 Execute $\mathcal{F}_{\text{wPoPWK}}$ code, using b already computed. If $b = \perp$ execute the code for $b = 0$

On final message c **from** $\mathcal{A}$ **to** S**:**

u.1 Retrieve $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), h_n, x, n, j, [\mathsf{pw}] \rangle$
u.2 If c was sent by the Simulator on behalf of the C, let (s', h') be the update value sampled and record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, s', h', x, 1, j+1, \mathsf{pw} \rangle$
u.3 Else if S was successfully attacked and decryption fails, or S was not attacked and c is adversarially computed sample $h', s' \xleftarrow{\$} \{0,1\}^\lambda$ and record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n, s'), h', x, n+1, j, \mathsf{pw} \rangle$. Moreover, if $\langle \mathsf{pwfile} \rangle$ is marked LEAKED record $\langle \mathsf{H}, h_n \| s', h' \rangle$
u.4 Else if S was successfully attacked with pw^* and c decrypts to (s', h') record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s'), h', x, 1, j+1, \mathsf{pw}^* \rangle$. Moreover, if $(s', h') \notin L$ and there is no j' such that $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s'), h', x, 1, j', \ldots \rangle$ exists and $fL[\mathsf{sid}, \mathsf{C}, j'] = 1$ retrieve $\langle \mathsf{H}, [\mathsf{pw}'] \| s', h' \rangle$: if $\mathsf{pw}' \neq \mathsf{pw}^*$ mark the file $\texttt{invalid}, \mathsf{pw}'$, if no pw' is found mark it $\texttt{invalid}, \perp$.
u.5 If c was honestly generated or successfully decrypted send $\mathsf{NewKey}, \mathsf{H}(K)$ to $\mathcal{F}_{\mathsf{saPAKE}}$. Else if $\mathsf{TestPwd}$ was not sent to S send $\mathsf{TestPwd}$ with $\emptyset$, in any case then send $\mathsf{NewKey}, \perp$

On input $(\mathsf{OfflineTestPwd}, *, \mathsf{pw}^*)$ **from** $\mathcal{S}_{\mathbf{wPoPWK}}$ **on behalf of** $\mathcal{A}$**:**

OTP.1 Execute $\mathcal{F}_{\text{wPoPWK}}$ code. If $\mathcal{F}_{\text{wPoPWK}}$ needs to compare passwords, proceed as follows.
OTP.2 If $\mathcal{F}_{\text{wPoPWK}}$ would test against verifier transcript (resp. prover) and VERIFY (resp. PROVE) was queried by SIM on behalf of the honest party and a successful $\mathsf{TestPwd}$ with pw' was sent output $\mathsf{CorrectGuess}$ if $\mathsf{pw}' = \mathsf{pw}^*$.
OTP.3 Else, if there exists a $\langle \mathsf{pwfile} \rangle$ marked LEAKED, send the $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}^*)$ query of step RO.3; on $\mathsf{CorrectGuess}$, execute steps RO.4–RO.5 without returning the value. On $\mathsf{CorrectGuess}$, send it to $\mathcal{S}_{\text{wPoPWK}}$.

On input m **from** $\mathcal{A}$ **to** $\mathcal{F}_{\mathsf{RO}}$**:**

RO.1 If there exists a record $\langle \mathsf{H}, m, [h] \rangle$ return h.
RO.2 Check if the following holds:
(a) There exists $([s], [h]) \in L$ such that $m = [\mathsf{pw}^*] \| s$ and there exists a $\langle \mathsf{pwfile} \rangle$ marked LEAKED
(b) There exists $n = fL[\mathsf{sid}, \mathsf{C}, j]$ such that $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), h_n, x, n, j, * \rangle$ is marked LEAKED and $m = [\mathsf{pw}^*] \| s_1$
(c) There exists $n = fL[\mathsf{sid}, \mathsf{C}, j]$ such that $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), h_n, x, n, j, * \rangle$ is marked LEAKED and $m = [h_{t-1}] \| s_t$ with $1 < t \leq n$ and there exists $\langle \mathsf{H}, h_{i-1} \| s_i, h_i \rangle$ for $i = \{1, \ldots, t-1\}$. If this is the case let $\mathsf{pw}^* \leftarrow h_0$.
RO.3 If any of the conditions holds send $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and on $\mathsf{WrongGuess}$, sample $r \xleftarrow{\$} \{0,1\}^\lambda$, record $\langle \mathsf{H}, m, r \rangle$ and return r.
RO.4 On $\mathsf{CorrectGuess}$, execute $\text{SIM}_{\text{wPoPWK}}$ code for successful $\mathsf{OfflineTestPwd}$ query to $\mathcal{F}_{\text{wPoPWK}}$ on input pw^*. Moreover, for every $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), [h_n], x, *, *, \perp \rangle$, overwrite the last entry with pw^*.
RO.5 On $\mathsf{CorrectGuess}$, for each $(s, h) \in L$, if $\langle \mathsf{H}, \mathsf{pw}^* \| s, [h'] \rangle$ exists and $h' \neq h$, abort; otherwise record $\langle \mathsf{H}, \mathsf{pw}^* \| s, h \rangle$. Moreover, do the following for all j such that $fL[\mathsf{sid}, \mathsf{C}, j]$ is defined:
(a) Let n be the biggest such that $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_n), [h_n], x, n, j, * \rangle$ exists and set $h_0 \leftarrow \mathsf{pw}^*$. Execute the following steps for $i = 1, \ldots, n$.
(b) If $\langle \mathsf{H}, h_{i-1} \| s_i, [h'] \rangle$ exists, set $h_i \leftarrow h'$ if $i < n$; if $i = n$ and $h' \neq h_n$, abort.
(c) Otherwise, if $i \neq fL[\mathsf{sid}, \mathsf{C}, j]$, sample $h_i \xleftarrow{\$}$ and record $\langle \mathsf{H}, h_{i-1} \| s_i, h_i \rangle$, with $h_0 = \mathsf{pw}^*$
(d) If $i = fL[\mathsf{sid}, \mathsf{C}, j]$ retrieve record $\langle \mathsf{pwfile}, \mathsf{sid}, \mathsf{C}, (s_1, \ldots, s_i), [h_i], x, i, j, * \rangle$. If $\langle \mathsf{H}, h_{i-1} \| s_i, [h'] \rangle$ exists and $h' \neq h_i$, abort; otherwise record $\langle \mathsf{H}, h_{i-1} \| s_i, h_i \rangle$.
Finally, retrieve the recorded $\langle \mathsf{H}, m, h \rangle$ and return h
RO.6 Otherwise, sample $h \xleftarrow{\$} \{0,1\}^\lambda$, record $\langle \mathsf{H}, m, h \rangle$ and return h.

Fig. 9. PAKE-to-saPAKE simulator 3

If a retrieved record does not exist, the simulator ignores the query.

On $(\textbf{GenStatement}, \mathsf{sid})$ **from** $\mathcal{F}_{\textbf{wPoPWK}}$**:** //Generate file

POK-gs.1 Sample $h_0 \xleftarrow{s} \{0,1\}^\lambda$, $s \xleftarrow{s} \{0,1\}^\lambda$

POK-gs.2 $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KEM.KGen}(h_0)$

POK-gs.3 Record $(s, \mathsf{pk}, \mathsf{sk}, h_0, \perp)$

POK-gs.4 Return 1 to $\mathcal{F}_{\text{wPoPWK}}$

On $(\textbf{Prove}, \mathsf{sid}, \mathsf{ssid}, b)$ **from** $\mathcal{F}_{\textbf{wPoPWK}}$**:**

POK-prv.1 Create record $(\mathsf{ssid}, \mathcal{P}, b, \perp)$ //W.l.o.g. Verifier got input first, so $b \neq \perp$

POK-prv.2 Return activation to $\mathcal{A}$ //Waiting for protocol message to arrive

On $(\textbf{Verify}, \mathsf{sid}, \mathsf{ssid}, b)$ **from** $\mathcal{F}_{\textbf{wPoPWK}}$**:** //Honest verifier

POK-vry.1 Retrieve record $(s, \mathsf{pk}, \mathsf{sk}, h_0, *)$

POK-vry.2 $(c, K) \leftarrow \mathsf{KEM.Encap}(pk)$

POK-vry.3 Store record $(\mathsf{ssid}, \mathcal{V}, c, K)$

POK-vry.4 Send (ssid, s, c) to $\mathcal{A}$ as message from $\mathcal{P}$

On (ssid, s, c) **from** $\mathcal{A}$ **as message to** $\mathcal{P}$**:** //Honest prover

POK-sc.1 Retrieve record $(\mathsf{ssid}, \mathcal{P}, b, \mathsf{pw})$

POK-sc.2 If $b = 1$ and s, c is simulated, retrieve $(\mathsf{ssid}, \mathcal{V}, c, K)$ and set $h \leftarrow H_1(\mathsf{ssid}, K, c)$ //Honest matching session

POK-sc.3 Else, if $\mathsf{pw} = \perp$, set $h \xleftarrow{s} \{0,1\}^\lambda$ //$\mathcal{Z}$ can't compute Prover's sk yet; honest mismatching session, or attack on prover with mismatching password

POK-sc.4 Else, do: //$\mathcal{Z}$ can compute prover's sk; honest mismatching session, or attack on prover

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow H_0(\mathsf{pw}, s)$
- $K \leftarrow \mathsf{KEM.Decap}(\mathsf{sk}, c)$
- $h \leftarrow H_1(\mathsf{ssid}, K, c)$

POK-sc.5 Send h to $\mathcal{A}$ as message to $\mathcal{V}$

On (ssid, h) **from** $\mathcal{A}$ **as message to** $\mathcal{V}$**:** //Honest Verifier

POK-h.1 Retrieve record $(s, \mathsf{pk}, \mathsf{sk}, h_0, \mathsf{pw})$

POK-h.2 If there are records $(H_1, ([\mathsf{ssid}], [K], [c]), h)$, and $(\mathsf{ssid}, \mathcal{V}, c, K)$, do: //If $\mathcal{Z}$ can compute the honest KEM key, there must be a password hash

- Abort the simulation if $\mathsf{pw} = \perp$ //IND-CPA security
- Send $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pw})$ and $(\textsc{Finalize}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{F}_{\text{wPoPWK}}$ //Successful attack on verifier

POK-h.3 Else: //h not hashed correctly, from wrong password, or at all: verifier outputs 0

- Send $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \perp)$ and $(\textsc{Finalize}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{F}_{\text{wPoPWK}}$

On $H_0(\mathsf{pw}, s)$ **from** $\mathcal{A}$**:**

POK-ro0.1 If there is a record $(H_0, (\mathsf{pw}, s), h)$, reply with h

POK-ro0.2 If there is a record $(s, *, *, h_0, \perp)$, send $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \perp, \mathcal{V}, \mathsf{pw})$ to $\mathcal{F}_{\text{wPoPWK}}$

POK-ro0.3 Upon receiving back $\mathsf{CorrectGuess}$, do: //Statement password found!

- Overwrite $\perp$ in the record with pw
- Set $h \leftarrow h_0$ //$\mathcal{Z}$ can compute simulation keypair now!

POK-ro0.4 For each record $(\mathsf{ssid}, \mathcal{P}, \perp)$, do:

- Send $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}_{\text{wPoPWK}}$
- Upon receiving back $\mathsf{CorrectGuess}$, do:
  - Overwrite $\perp$ in the record with pw //Prover's password in session ssid found!
  - If prover message $(ssid, h)$ was already simulated, do: //Program already simulated h to the password
    - $(\mathsf{pk}, \mathsf{sk}) \leftarrow H_0(\mathsf{pw}, s)$
    - Retrieve message $(\mathsf{ssid}, s, [c])$ sent to $\mathcal{P}$
    - $K \leftarrow \mathsf{KEM.Decap}(\mathsf{sk}, c)$
    - Record $(H_1, (\mathsf{ssid}, K, c), h)$
- If h is undefined, sample $h \xleftarrow{s} \{0,1\}^\lambda$
- Record $(H_0, (\mathsf{pw}, s), h)$ and reply with h

On $H_1(\mathsf{ssid}, K, c)$ **from** $\mathcal{A}$**:** //

POK-ro1.1 If there is a record $(H_1, (\mathsf{ssid}, K, c), h)$, reply with h

POK-ro1.2 Else, sample $h \xleftarrow{s} \{0,1\}^\lambda$, record $(H_1, (\mathsf{ssid}, K, c), h)$ and reply with h

On $(\textbf{LeakStatement}, \mathsf{sid})$ **from** $\mathcal{A}$**:** //

POK-ls.1 Forward the query to $\mathcal{F}_{\text{wPoPWK}}$

POK-ls.2 Retrieve record $(s, \mathsf{pk}, \mathsf{sk}, h_0, \perp)$

POK-ls.3 Upon receiving back $(\textsc{LeakStatement}, \mathsf{pw})$, do:

- Overwrite $\perp$ in the record with pw
- Record $(H_0, (\mathsf{pw}, s), h_0)$
- Send (s, pk) to $\mathcal{A}$

Fig. 10. Simulator for the KEM-based wPoPWK protocol in Figure 6.

Method	Communication Costs		Client Computation		Server Computation		Storage Costs after i KEx.
	Reg.=KEx.CS	ith KEx.SC	Reg.=KEx.Off	ith KEx.On	Reg.	ith KEx.	
1a	$B(h+s)$	s	$B(H+S)$	H	-	-	$(B-i)(s+h)$
1b	$B \cdot h + s$	s	$S + (2B-1)H$	H	$(B-1)H$	-	$(B-i)(s+h)$
1c	$h+s$	$2s$	$H+S$	$2H$	$B(S+H)$	-	$(B-i)(s+h)+s$
1d	$h+2s$	$s+\ell_B$	$S+(B+1)H$	$(2i-1)H$	-	$(B-i+1)H$	$h+2s+\ell_B$
2a	$h+s$	$i \cdot s$	$H+S$	$i \cdot H$	-	$S+H$	$(i+1)s+h$

Table 2. Comparison of salt update approaches. Notations: H: one hashing, S: sampling a random salt value, h: size of a hash output, s: size of a salt value, $\ell_B = \lceil \log_2(B+1) \rceil$: size of the salt index in bits. For 1b, the server expands the seed at initialization and each successful reset, stores all salt/hash pairs, and deletes each pair after use. For 1d, initialization and reset also transmit the password hash; the server stores only the seed, current salt, password hash and index, recomputing each next salt from the seed. CS denotes client to server communication and SC is the other direction (both meant for the key-exchange phase). Client computation (of salted hashing) during key exchange can be split up to an offline part (KEx.Off) that only depends on the client's input and an online part (KEx.On), that cannot be precomputed.