Updatable aPAKE: Security Against Bulk Precomputation Attacks

Dennis Dayanikli
Hasso-Plattner-Institute, University of Potsdam
Potsdam, Germany
dennis.dayanikli@hpi.de

Anja Lehmann
Hasso-Plattner-Institute, University of Potsdam
Potsdam, Germany
anja.lehmann@hpi.de

Abstract

Asymmetric Password-Authenticated Key Exchange (aPAKE) enables secure key establishment between a client and a server using a pre-shared password, while providing security against offline attacks. However, aPAKE does not guarantee any precomputation resistance, and considers passwords to become immediately available upon server compromise. A recent work by Dayanikli and Lehmann (EuroS&P'24) observed that many existing aPAKE protocols provide stronger precomputation attack resistance than what is guaranteed through the aPAKE model: they often rely on salted password hashes, where a unique salt makes precomputation attacks more difficult. While these salts are sent in clear to the client during authentication, and thus trivial to obtain for an attacker, this makes a difference in multi-user settings with millions of user accounts per server. In order to run bulk precomputation attacks on all users' passwords, the attacker needs to start an authentication session on behalf of every user to obtain their salts. However, this protection is still limited as salts are static, and the attacker can gradually extract all salt values for precomputation attacks.

In this work, we build upon the observation that many aPAKE protocols include salts for their password protection, and propose a new aPAKE variant that makes such bulk precomputation attacks practically infeasible. We propose *updatable* aPAKE which employs updatable salts. In updatable aPAKE, the salt is implicitly refreshed with each successful user authentication, forcing an attacker to rebuild their precomputation table after every honest user's login – offering a level of precomputation resistance similar to that of strong aPAKE protocols. We formalize the security of updatable aPAKE in the Universal Composability framework and show how OKAPE-HMQV, the currently most efficient aPAKE protocol, can be lifted to the updatable aPAKE setting in a provably secure way. The core idea is that this salt update can be integrated through relying on the password-based server-side authentication, that is already guaranteed through aPAKE. We also observe that OKAPE-HMQV is very similar to SRP-6a, the currently most widely deployed aPAKE protocol, and explain how the same idea can be used to upgrade this legacy protocol to achieve strong bulk precomputation attack resistance with minimal overhead.

1 Introduction

Asymmetric Password-Authenticated Key Exchange (aPAKE) enables a user and a server to establish a strong shared key from a low-entropy password. The "asymmetric" aspect lies in the fact that the user must know the plaintext password to establish this shared key, while the server relies on a *password file* – which is derived from the password during user registration – instead of using the password in plaintext. aPAKEs have been extensively studied [7, 20, 23, 26, 27, 29, 30, 33, 35–38], with some protocols being widely deployed in commercial applications such as Apple [3], Telegram [40] or Whatsapp [17]. The core security property of aPAKE protocols is their resilience to offline attacks: Even if an adversary eavesdrops on protocol messages or tricks the user into running the aPAKE protocol with a malicious server, it cannot efficiently recover the user's password via brute-force guessing. This guarantee holds as long as the server that holds the password file is not compromised.

Once the server is compromised, and the attacker obtains a user's password file, offline dictionary attacks become feasible. Specifically, the attacker can iterate through a dictionary of potential passwords to recover the password that is consistent with the stored password file. This is inherent in the password-only setting, as an attacker can internally simulate aPAKE sessions to verify password guesses.

Precomputation Attacks in aPAKE. To contain the impact of server compromise, the majority of the computational effort required for offline password attacks should have to occur *after* the compromise. However, the standard aPAKE security model provides no such guarantee, as it is does not include any security against precomputation attacks. Thus, aPAKE security for the stored password files is equivalent to that of unsalted password hashes – $H(pw)$ – where an attacker can precompute a large database of such values (e.g., a rainbow table) for candidate passwords. Then, upon compromising the server, the attacker can instantly check if a precomputed password value matches the password file.

Thus, the security guarantees of aPAKE for stored password files are significantly weaker than what is guaranteed by conventional password-based authentication, where the server typically stores a salted hash $H(s, pw)$ to prevent such precomputation attacks.

Strong aPAKE: Security against Precomputation Attacks. This lack of precomputation attack resistance has only been solved in 2018, where Jarecki et al. [33] introduced the concept of strong aPAKE. This stronger version extends the security of standard aPAKE and additionally prevents any precomputation attacks on a user's password. Since then, several strong aPAKE protocols have been proposed [12, 16, 27, 33, 37], and OPAQUE [33] is currently undergoing standardization by the IETF [11]. However, most strong aPAKEs either lack practical efficiency [12, 16] or rely on an oblivious pseudorandom function (OPRF) as building block [27, 33]. Although it is trivial to build OPRFs from discrete logarithms, realizing them *efficiently* from quantum-safe assumptions remains an open research challenge (see e.g., [1, 2, 4, 8, 9, 21, 28]. Recently, McQuoid and Xu [37] provided efficient strong aPAKE protocols, which do not rely on OPRFs and can be instantiated based on the isogeny-based CSIDH [15], which is assumed to be quantum-safe. However, their proof relies on the algebraic group (action) model whose validity is unclear [44].

Salted aPAKE: Limited Security against Precomputation Attacks. Recently, Dayanikli and Lehmann [19] observed that many existing

aPAKE protocols already provide some resistance against precomputation attacks, i.e., go beyond what the aPAKE model guarantees. Their observation was that many protocols essentially use salted hashes of the form $(H(s, pw), s)$ as password files, where s is a user-specific high-entropy salt. In these protocols, which we refer to as *salted* aPAKE, the attacker can only perform precomputation attacks against a user's password if it knows the used salt.

The salt s is used in a quite peculiar manner in all such protocols though: it is sent in clear to the client as the first message in the authentication session, prior to user authentication. Thus, the salt is easily accessible to the attacker, who can obtain it with a single query to the server. In the single-user setting in which most aPAKE protocols have been designed and analyzed, this salt does not provide any additional security then. However, when considering the practical *multi-user* setting with potentially millions of users in which these aPAKEs are deployed, this salt *does* make a difference: to launch *bulk* precomputation attacks against all users, the attacker would need to initiate a session on behalf of *every* user.

Grasping the security of such salted aPAKEs is not very clear though, and lies somewhere in between standard and strong aPAKE. If one considers mainly smash-and-grab style attacks, or accidental leakage through severs, i.e., one does not assume that the adversary actively engages with the server, then salted aPAKE and strong aPAKE are essentially equivalent. Against more sophisticated attacks, where an attacker might gradually query the salts from all users, salted aPAKE security falls back to standard aPAKE.

However, the observation that many aPAKE protocols deploy a salt, and open reveal this during the authentication session, raises the question whether we can turn this weird behaviour into an advantage, and lift security from existing aPAKE protocols through continuously updating the salt.

1.1 Our Contributions

In this work, we answer that question in a positive manner and propose a new aPAKE variant that provides security that is practically equivalent to strong aPAKE, and can be built from classic aPAKEs through the use of standard building blocks only.

This new variant are *updatable* aPAKEs, which we build upon the salted aPAKE model [19]. The core idea is that we can simply let the user send a fresh password file $H((s', pw), s')$ for a new salt s' upon every successful authentication session. The server then replaces the previous file, and uses the new one for its next session with the user — sending s' as the first message. Thus, we still openly reveal the salt with every query, but now it is useful only until the honest user logs in again, and replaces the salt and file with a new version. This significantly reduces the time in which the adversary can build precomputation tables *and* must compromise the server to learn also $H((s, pw)$. If such an updatable aPAKE is used in a scenario where users log-in on a daily or weekly basis, this practically voids any precomputation advantage.

Stronger Security Model for aPAKE. We first provide a formal security model for updatable aPAKE, capturing the added precomputation resistance. Our definition is given in form of an ideal functionality in the Universal Composability (UC) framework, the defacto gold-standard for aPAKE security. Our updatable aPAKE functionality $\mathcal{F}_{\mathsf{uaPAKE}}$ extends the well-established (multi-user) aPAKE functionality [18–20, 26] to incorporate the improved security: precomputation attacks are now limited to the time window between a user's last successful login and the moment of server compromise.

Boosting Security of aPAKE Protocols. Our work then shows how standard aPAKE protocols can be easily lifted to achieve this stronger security. As already sketched, the main idea is to continuously update the password file with a new salt and salted "hash" at the end of every user authentication. As knowing the password file will always enable offline attacks on the password, it is crucial that the user cannot be tricked into sending such a file to a malicious server. Interestingly, we can ensure this through implicit properties that are provided by any standard aPAKE protocol.

In a nutshell, the client and server first execute the aPAKE protocol to agree on a shared key k. The client then generates a new password file using a fresh salt, encrypts it under k, and sends it to the server. The server decrypts the password file, and updates its stored password file. What we now rely on is the aspect of *server-authentication* as part of the aPAKE. That is, only a server that uses a correct password file $(H(s, pw), s)$ arrives at the same key k as the user that starts the authentication session with password pw. Thus, sending $(H(s', pw), s')$ to such a server reveals no more information than what the server already knew. If the user accidentally runs the session with a malicious server that does not have such an original password file, the server will not learn anything about the key k that the user obtained on her side. And thus, sending $(H(s, pw), s)$ encrypted under k to such a malicious server leaks no information about the password file.

Any secure aPAKE protocol already provides such (implicit) server authentication, but this aspect is rarely used explicitly. aPAKE is mostly seen as a method to authenticate clients towards the server, while servers are authenticated through the usual TLS certificates anyway. Our work now highlights a case where server authentication in aPAKE is crucial and enables this simple update mechanism.

As the standard aPAKE ideal functionality does not expose the existence of password files in the protocol, we cannot build our transform in a generic manner from any aPAKE though. Instead, we show how two concrete variants, OKAPE-HMQV [20] and SRP-6a [42] can be modified following this blueprint, and prove their security with minor modification to the original security proofs.

UPAKE: Boosting OKAPE-HMQV. Our first updatable aPAKE protocol, UPAKE, builds on OKAPE-HMQV – currently the most efficient UC-secure aPAKE protocol [20]. UPAKE follows our blueprint and lets the client generate updated password files using fresh salts and encrypt it using an encryption key derived from the OKAPE-HMQV key via a pseudorandom function (PRF). The encryption uses an authenticated encryption (AE) scheme. Since the client sends the final message, the encrypted password file can be incorporated within the existing round, requiring no additional communication. The result is a round-optimal 2-round updatable aPAKE protocol with only 3 exponentiations per party.

The security analysis reveals an interesting gap: UPAKE allows a malicious client to change its password within an authentication session. While we consider this a benign and unavoidable behavior – real-world applications typically allow authenticated users to change their password anyway – this is not permitted by $\mathcal{F}_{\mathsf{uaPAKE}}$. Including this behaviour would unnecessarily inflate the

ideal functionality, and still require additional protocol changes for extractability reasons. Instead, we apply a proof technique that detects and excludes attackers which change a user's password, and state security for "behaving" adversaries. Finally, we also show how password consistency can be enforced even against malicious clients by adding a non-interactive zero-knowledge proof (NIZK) to the updated password files.

UPAKE-SRP: Boosting the most widely deployed aPAKE. Finally, we apply our blueprint to upgrade the security of the Secure Remote Password protocol SRP-6a [42], an aPAKE protocol designed in the early 2000s. Despite receiving limited academic scrutiny, SRP-6a remains the most widely deployed aPAKE protocol. It is used in numerous applications, including Telegram [40], Apple HomeKit [3], ProtonMail [13], ProtonPass [43], and 1Password [22], and has been standardized by IETF (RFC2945 [41], RFC5054 [39]), IEEE (P1363.2 [31]), and ISO (11770-4 [32]).

We show that SRP-6a shares a structure very similar to that of OKAPE-HMQV – it uses a similar Diffie-Hellman-based key exchange, but differs in how the server's public key is encrypted. Thus, we can re-use the same transformation and most of the analyis of UPAKE to boost the security of SRP, too. The resulting protocol, UPAKE-SRP, achieves updatable aPAKE security with minimal overhead: it adds just one protocol message and requires only standard additional building blocks while preserving the existing password file format of SRP-6a. In essence, we show how a widely deployed legacy protocol can be upgraded to achieve security close to that of strong aPAKE with minimal overhead.

2 Preliminaries

Here, we introduce the building blocks needed in our work.

Encryption. A *symmetric encryption scheme* SE consists of key generation $\mathsf{KGen}(1^n) \to k$, encryption $\mathsf{Enc}(k, m) \to c$, and decryption algorithm $\mathsf{Dec}(k, c) \to m$. We require SE to be an authenticated encryption scheme. i.e. have CCA-secure and ciphertext integrity. The underlying PAKE we use, assumes SE to be an *ideal cipher* IC. This is an idealized version of an encryption scheme, where encryption and decryption can only be performed by interacting with an IC oracle $\mathsf{IC}^*.\mathsf{Enc}$ and $\mathsf{IC}^*.\mathsf{Dec}$. A *public key encryption scheme* PKE consists of key generation $\mathsf{KGen}(1^n) \to (sk, pk)$, an encryption $\mathsf{Enc}(pk, m) \to c$, and decryption algorithm $\mathsf{Dec}(sk, c) \to m$. In our paper, we use standard IND-CPA and IND-CCA security of PKE. We recall all these standard properties in App. A.

Hash/Pseudorandom Function. We use *hash functions* $H : \{0,1\}^* \to \{0,1\}^\lambda$, modelled as random oracle (RO), and *pseudorandom functions* (PRF) $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$. The latter is secure, if its outputs are indistinguishable from a truly random function f.

NIZK. We denote a non-interactive zero-knowledge proof of knowledge (NIZK) by $\pi := NIZK\{(w) : \text{statement}(w)\}(\text{ctx})$, where π proves knowledge of a witness w such that $\text{statement}(w) = 1$, and the proof is bound to the context ctx. We assume that the NIZK scheme is zero-knowledge and sound.

3 The Updatable aPAKE Functionality

In this section, we formally define updatable aPAKE and the security it guarantees against bulk precomputation attacks. We define updatable aPAKE through an ideal functionality $\mathcal{F}_{\mathsf{uaPAKE}}$ in the Universal Composability (UC) framework. We first recap the standard UC functionality $\mathcal{F}_{\mathsf{aPAKE}}$, as well as the stronger model(s) from [19], and describe the different strengths they provide w.r.t. precomputation attacks. We then explain how these existing models must be changed to capture the idea of updating the password files with every honest user's login – thereby drastically limiting the time an adversary has for precomputation attacks.

Universal Composability. We analyze security in the Universal Composability (UC) framework [14], which has evolved as the gold standard for aPAKE protocols. In the UC framework, the desired properties of a protocol are specified through an ideal functionality $\mathcal{F}$. A real-world protocol π is considered secure if it can be indistinguishably simulated by an efficient algorithm – the simulator – which interacts solely with the ideal functionality $\mathcal{F}$. In this case, π is said to UC-realize functionality $\mathcal{F}$. UC-security is particularly advantageous for password-based protocols because it ensures security under arbitrary protocol compositions and does not require any idealized assumptions on the distribution or handling of passwords.

3.1 Basic aPAKE Functionality

The ideal functionality $\mathcal{F}_{\mathsf{aPAKE}}$ [23] models the asymmetric password-authenticated key exchange between a user *uid* (communicating from a client machine C) and a server S. We adopt the multi-user functionality used in [19]. Therein, each instance of the ideal functionality is tied to a single server S, identified by a unique session identifier $\mathsf{sid} := (\mathsf{S}, \mathsf{sid}')$ and allows multiple users to access the functionality. In each protocol execution, a client C acts on behalf of a single user *uid*, with each user uniquely identified by *uid*.

To register a user, the server accesses the StorePwdFile interface with user identifier *uid* and password *pw*. This triggers $\mathcal{F}_{\mathsf{aPAKE}}$ to internally store password file $\langle \mathsf{file}, \mathsf{S}, uid, pw \rangle$, enabling subsequent key exchanges between a client C (on behalf of *uid*) and S.

To initiate a key exchange, the functionality provides different interfaces for the two parties: a client utilizes the CltSession interface, providing the *uid* and password *pw′* she wants to use for the session. Meanwhile, the server uses the SvrSession interface, supplying only the intended user identifier *uid*, which acts as a pointer to the internal password file $\langle \mathsf{file}, \mathsf{S}, uid, pw \rangle$. Specifically, an honest server will always use the registered password *pw* stored in this file for the key exchange session. If there is no active attack, and if the client's password *pw′* matches the password *pw* stored in the file, both parties will output the same session key (provided via the NewKey interface).

The aPAKE functionality $\mathcal{F}_{\mathsf{aPAKE}}$ also models several inevitable attacks. First, an attacker can actively engage in the protocol and, if it guesses the counterparty's password correctly, establish a key with the other party. This online attack is captured through the TestPwd interface and is limited to one password guess per session. Additionally, the attacker can compromise a server to learn the stored password file of a user *uid*, modeled through the StealPwdFile interface. Compromising the password file of a user *uid* enables impersonation and offline attacks for this user, which

Password Registration

- On (StorePwdFile, sid, uid, pw) from S, create record $\langle$file, S, uid, $pw\rangle$ marked fresh.

Stealing Password Data

- On (StealPwdFile, sid, S, uid) from $\mathcal{A}$, if there is no record $\langle$file, S, uid, $pw\rangle$, return "no password file". Otherwise mark this record stolen, and if there is a record $\langle$offline, S, uid, $pw\rangle$, send pw to $\mathcal{A}$.
- On (OfflineTestPwd, sid, S, uid, pw^*) from $\mathcal{A}$, do:
  - If $\exists$ record $\langle$file, S, uid, $pw\rangle$ marked stolen, do the following: If $pw^* = pw$ return "correct guess" to $\mathcal{A}$, else return "wrong guess".
  - Else, if $\exists$ record $\langle$file, S, uid, $pw\rangle$ marked unlocked, record $\langle$offline, S, uid, $pw^*\rangle$.

Password Authentication

- On (CltSession, sid, ssid, S, uid, pw') from $\mathcal{P} \in \{\mathsf{C}, \mathcal{A}\}$, if there is no record $\langle$ssid, C, ...$\rangle$ then record $\langle$ssid, C, S, uid, pw', cl$\rangle$ marked fresh and send (CltSession, sid, ssid, C, S, uid) to $\mathcal{A}$. If $\mathcal{P} = \mathcal{A}$, $\exists$ record $\langle$ssid, S, C, uid, pw, sr$\rangle$ and if $\exists$ record $\langle$file, S, uid, $pw\rangle$ marked fresh, mark it unlocked.
- On (SvrSession, sid, ssid, C, uid) from S, if there is no record $\langle$ssid, S, ...$\rangle$ then retrieve record $\langle$file, S, uid, $pw\rangle$, and if it exists then create record $\langle$ssid, S, C, uid, pw, sr$\rangle$, mark it fresh and send (SvrSession, sid, ssid, S, C, uid) to $\mathcal{A}$. If $\exists$ record $\langle$ssid, C, S, uid, pw, cl$\rangle$ with $\mathsf{C} = \mathcal{A}$ and if record $\langle$file, S, uid, $pw\rangle$ is marked fresh, mark it unlocked.

Active Session Attacks

- On (TestPwd, sid, ssid, P, pw^*) from $\mathcal{A}$, if there is a record $\langle$ssid, P, P$'$, uid, pw, role$\rangle$ marked fresh, then do: If $pw^* = pw$ then mark it compromised and return "correct guess" to $\mathcal{A}$; else mark it interrupted and return "wrong guess".
- On (Impersonate, sid, ssid, C, S, uid) from $\mathcal{A}$, if there is a record $\langle$ssid, C, S, uid, pw, cl$\rangle$ marked fresh, then do: If there is a record $\langle$file, S, uid, $pw\rangle$ marked stolen then mark $\langle$ssid, C, S, uid, pw, cl$\rangle$ compromised and return "correct guess" to $\mathcal{A}$; else mark it interrupted and return "wrong guess".

Key Generation and Authentication

- On (NewKey, sid, ssid, P, K^*) from $\mathcal{A}$, if there is a record rec = $\langle$ssid, P, P$'$, uid, pw, role$\rangle$ not marked completed, then do:
  - If rec is compromised set $K \leftarrow K^*$;
  - Else if role = sr, rec is fresh, there is a record $\langle$ssid, P$'$, P, uid, pw, cl$\rangle$ s.t. $\mathcal{F}_{\mathsf{aPAKE}}$ sent (sid, ssid, K') to P$'$ while that record was marked fresh, set $K \leftarrow K'$; delete all records $\langle$offline, S, uid, $pw^*\rangle$. If $\langle$file, S, uid, $pw\rangle$ is marked unlocked, mark it fresh.
  - Else if role = cl, rec is fresh, there is a record $\langle$ssid, P$'$, P, uid, pw, sr$\rangle$ which is marked fresh, pick $K \xleftarrow{r} \{0,1\}^\lambda$;
  - Else set $K \leftarrow \perp$.

  Finally, mark rec as completed. If $K = \perp$, provide public delayed output (sid, ssid, $\perp$) to P, otherwise provide private delayed output (sid, ssid, K) to P.

Figure 1: The multi-user ideal functionality $\mathcal{F}_{\mathsf{uaPAKE}}$ The functionality is tied to a server S, encoded in sid = (S, sid$'$) for a unique sid$'$ and models explicit mutual authentication. Our additions compared to the ideal functionality for salted aPAKE [19] are highlighted.

are captured via two distinct interfaces. The impersonation attack is modeled by the Impersonate interface, and allows the attacker to impersonate the server towards uid using the stolen password file. The offline attack, modeled by the OfflineTestPwd interface, enables the attacker to recover uid's password in plaintext by performing offline tests against the stolen password file.

3.2 Levels of Precomputation Attack Resistance

Existing aPAKE models differ in their handling of precomputation attacks. In such an attack, an adversary precomputes a table of password-derived values, e.g., hashes based on a password dictionary. If the attacker later compromises the server, it can instantly recover the user's password through a look-up from the precomputed values. We first explain how this is handled in standard aPAKE, strong aPAKE [33] and salted aPAKE [19], with a focus on [19], as our new updatable aPAKE model builds upon this model.

aPAKE: No precomputation resistance. Standard aPAKE does not provide *any* security against precomputation attacks. The attacker can perform precomputation attacks on a user uid's password as soon as uid is registered with the server. This attack is captured by the OfflineTestPwd interface. If the attacker queries OfflineTestPwd before stealing the password file, the functionality stores a record $\langle$offline, S, uid, $pw^*\rangle$ representing values of a precomputed table. This enables the attacker to instantly recover the user's password in case of server compromise: StealPwdFile returns the correct password pw if it was previously tested through OfflineTestPwd.

strong aPAKE: Full precomputation resistance. Jarecki et al. [33] introduced the concept of strong aPAKE (saPAKE) which is resistant to precomputation attacks. This is modeled in the saPAKE functionality, as OfflineTestPwd does not log password guesses before the file is compromised. This ensures that OfflineTestPwd can only be used for offline attacks after server compromise.

salted aPAKE: Limited precomputation resistance. Dayanikli and Lehmann [19] adjusted the aPAKE model to reflect the security of existing protocols whose resistance against precomputation attacks lies between aPAKE and saPAKE. In these existing protocols, such as SRP[42], OKAPE[20], KHAPE[26], precomputation attacks require knowledge of a user-specific, high-entropy salt – think of salted hashes $v = H(s, pw)$. The server sends the salt s to the client during authentication – but before the client is authenticated. This forces the adversary to initiate at least one authentication session per user in order to learn the salt that is necessary for performing precomputation attacks on the users' passwords. While this attack remains straightforward – the attacker does not need to know the user's password to initiate a session – the security impact becomes clear in the multi-user setting with potentially millions of users: If an attacker aims to perform a bulk precomputation attack across millions of users, it would need to initiate a session with the honest server for each user to retrieve the corresponding salt. In contrast, without this added layer of resistance, an attacker could precompute values for all users without ever interacting with the server.

To model the precomputation attack resistance provided by these existing protocols, the ideal salted aPAKE functionality introduces an additional password file state – unlocked – next to the existing states fresh and stolen. Our updatable aPAKE functionality adopts the same states which determine the behavior of OfflineTestPwd as follows:

fresh: Offline or precomputation attacks are not possible, i.e. querying OfflineTestPwd returns nothing.

unlocked: Precomputation attacks are possible. Querying interface OfflineTestPwd logs the password guess as $\langle$offline, S, uid, $pw^*\rangle$.

stolen: Offline attacks are possible. OfflineTestPwd can be used to immediately test passwords.

Initially, password files are created in the fresh state, and become unlocked if the attacker initiates a CltSession such that there is a corresponding SvrSession started by the honest server. If the attacker steals the password file via query StealPwdFile, its state moves from either fresh or unlocked to stolen.

This state-based approach also lets us highlight the differences between existing aPAKE models: in aPAKE, password files start in the unlocked state by default and become stolen upon a StealPwdFile query. In contrast, strong aPAKE restricts password files to either fresh or stolen (after a StealPwdFile query), as they can never become unlocked.

3.3 Updatable aPAKE Functionality $\mathcal{F}_{\mathsf{uaPAKE}}$

Our updatable aPAKE functionality $\mathcal{F}_{\mathsf{uaPAKE}}$, given in Fig. 1, extends the level of protection against precomputation attacks provided by salted aPAKE by further restricting the attacker's ability to perform precomputations.

First, recall the type of precomputation resistance we aim to model: in salted aPAKE protocols, password files include a high-entropy, user-specific salt s, e.g., in the form of a salted hash $v = H(s, pw)$ and an attacker must first obtain s before being able to compute values $v_i = H(s, pw_i)$ for password guesses pw_i. If the file (s, v) stored by the sever gets continuously replaced with a fresh version (s', v') with $v' = H(s', pw)$ for a new salt s' upon every successful user authentication, this significantly limits the time where the adversary can mount precomputation attacks. In particular, the attacker's precomputed values become entirely useless whenever the honest user successfully authenticates to the server.

Thus, the ideal functionality must capture two key effects:

Resetting Precomputed Values In the (salted) aPAKE model, precomputation attacks are logged through interface OfflineTestPwd by creating record $\langle \mathsf{offline}, \mathsf{S}, uid, pw^* \rangle$ if the password file is in state unlocked. In updatable aPAKE we model that the attacker's precomputed values become useless upon a successful client authentication by deleting all such records $\langle \mathsf{offline}, \mathsf{S}, uid, pw^* \rangle$ whenever user uid successfully authenticates. Note that this reset occurs only for *uncompromised* user authentication sessions – in a compromised session the adversary has already guessed the password, so the protocol cannot offer protection against precomputation attacks anymore.

Lock Offline Attacks Upon successful authentication, the salt is updated, and the attacker must first learn the new salt before launching further offline attacks. This is reflected in the functionality by transitioning the password file state from unlocked back to fresh, thereby making OfflineTestPwd queries unavailable until the attacker actively engages with the server again.

Password File Handling. As in the standard aPAKE functionality, our ideal functionality $\mathcal{F}_{\mathsf{uaPAKE}}$ is agnostic of any concrete handling of password files. It only keeps the idealized record $\langle \mathsf{file}, \mathsf{S}, uid, pw \rangle$ which may be accessed through the functionality's interfaces. When considering aPAKE protocols with updatable password files realizing this functionality, we want to highlight two interesting aspects of this modeling which makes a difference in the security analysis compared to regular aPAKE:

Table 1: Conditions under which an attacker can mount precomputation attacks on a user's password in various security models.

aPAKE	Salt Model	Precomputation attack *uid*'s *pw*
regular	no salt	Always feasible
salted	static salt, revealed to active attacker	If the attacker starts a session with S on behalf of *uid*
updatable	updatable salt, revealed to active attacker	If the attacker starts a session with S on behalf of *uid* between the last successful user login and server compromise
strong	private salt	not feasible (resistant to precomputation attacks)

Password Consistency The ideal functionality enforces that for each user, there is only a single password. Thus, a protocol realizing the ideal functionality must ensure that when updating the password file of a user, the updated password file corresponds to the same password. Notably, enforcing this property will be surprisingly difficult with our blueprint. However, we will discuss in Sec. 4 why password changes within the protocol are acceptable and do not undermine our security goals.

Stealing Multiple Files In updatable aPAKE instantiations where password files are updated, an attacker might compromise the server at different times and obtain multiple versions of a user's password file. However, possessing multiple versions of the password file should not help the attacker in recovering the user's password, i.e., we still require the attacker to perform a brute-force attack on the password. It is therefore important that our model allows multiple queries to StealPwdFile. While multiple queries are also allowed in the standard aPAKE model, we mention this explicitly because it was previously of little significance – in standard aPAKE, password files are typically static, meaning StealPwdFile would always return the same password file.

Comparison to Existing Models. Our updatable aPAKE model offers stronger protection against precomputation attacks than salted aPAKE by significantly reducing the time window in which an attacker can capture the salt and perform precomputation attacks. For example, if a user logs in daily, updatable aPAKE limits an attacker to a maximum of one day to conduct precomputation attacks, while in salted aPAKE this window can span multiple years.

While updatable aPAKE does not provide the same level of resistance against precomputation attacks as strong aPAKE, it can still match its security under certain conditions. For instance, if an attacker does not initiate a session on behalf of the user between the user's last successful login and server compromise, updatable aPAKE prevents precomputation attacks and, in this case, provides the same level of security against precomputation attacks as strong aPAKE.

Table 1 gives an overview of the precomputation attack resistance guaranteed by existing aPAKE security models. Note that all models can also be interpreted through the lens of how they manage the salt: standard aPAKE does not use a salt; both salted aPAKE and updatable aPAKE expose it to an active attacker, though updatable aPAKE periodically refreshes it; and in strong aPAKE, the salt is never leaked before server compromise.

4 Warmup: Salted aPAKE Transform

In this section, we show how aPAKE protocols can be lifted to achieve our stronger updatable aPAKE security. Our starting point are *salted* aPAKE protocols that already use a salt to protect the password in the stored file – but still reveal the salt with every authentication. The core observation is that we can simply let the user create a new password file for a fresh salt with each authentication session. The user then transmits the encrypted new password file to the server, who replaces the existing file with the new one. To ensure that this is not harming security, the new password file gets encrypted under the fresh key established through aPAKE. Here the server-side authentication plays a crucial role: when the user obtains a key through aPAKE, she is ensured that only a server already knowing her correct password file can arrive at the same key too. Thus, only a server that already had a valid password file, can obtain the new one. As the password file is an offline-attackable value, this protection is essential here to prevent leaking passwords when accidentally authenticating to malicious servers.

4.1 Explicit Password File Handling

We begin with a secure salted aPAKE protocol according to Dayanikli and Lehmann [19], and our goal is to turn this into a secure updatable aPAKE protocol by modifying its password file handling. However, the salted aPAKE functionality is agnostic of any concrete password file, which prevents us from directly reasoning about the structure or even existence of such files in the protocol design. Making these files available in a formal manner, would require changing the UC definition and reproving all (salted) aPAKE protocols for this new extended UC definition.

Thus, we only informally sketch the additional structure that we assume from the underlying salted aPAKE protocol, to explain the high-level idea of our transform. In the following sections we then construct updatable aPAKE protocols following this blueprint from concrete salted aPAKE protocols and prove their security.

Registration: We assume that upon registration of a user uid with password pw, the server S generates a password file via a deterministic function f. More precisely, the server samples a random salt $s \xleftarrow{r} \{0,1\}^\lambda$, and computes the password verifier $v := f(\mathsf{S}, uid, pw, s)$. The server then stores the password file for uid consisting of (s, v). Note that all existing aPAKE protocols follow this structure, or can be easily transformed into such a salted version, as shown in [19].

Authentication: The client and server engage in a key exchange protocol. We now explicitly assume that the server uses the currently stored password file $\mathsf{file}[uid] = (s, v)$. In salted aPAKE protocols, this file is static, whereas our updatable aPAKE transform will continuously refresh the file with every authentication session. Further, we assume that the salt s is simply revealed in the first message from the server to the client.

Making this registration procedure and password file explicit, allows us to to express that the registration is re-run (locally) with every authenticated key exchange session and how the password file gets transmitted from the user to the server.

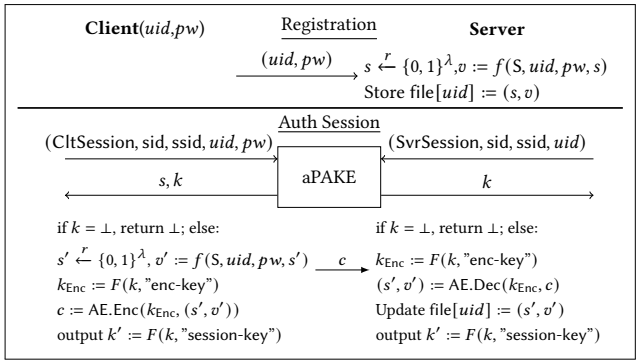


Figure 2: Blueprint for our UPAKE solution building on a salted aPAKE protocol with password registration function f. We assume that SvrSession **always utilizes the user's current password file** file$[uid]$**.**

4.2 Salted to Updatable aPAKE Transform

We now sketch our blueprint which extends any salted aPAKE protocol with the aforementioned password-file handling, into a secure updatable aPAKE protocol. The only additional building blocks are an authenticated encryption scheme $\mathsf{AE} := (\mathsf{AE.KGen}, \mathsf{AE.Enc}, \mathsf{AE.Dec})$ and a pseudorandom function $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$.

High-Level Idea. Let Π be a salted aPAKE where the password file is determined through a function f. The server initially registers the user with uid and pw through $v := f(\mathsf{S}, uid, pw, s)$ for a random salt $s \xleftarrow{r} \{0,1\}^\lambda$ and stores the user's currently stored password file $\mathsf{file}[uid] := (s, v)$. Then, in the authentication phase, the parties first run the aPAKE authentication protocol using $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{ssid}, uid, pw)$ and $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{ssid}, uid)$ where the client uses her password pw and we assume that SvrSession uses the user's password file $\mathsf{file}[uid] := (s, v)$ stored on the server. This results in both parties holding a key k, if the *mutual* authentication was successful. The client then picks a random salt $s' \xleftarrow{r} \{0,1\}^\lambda$, creates a new password verifier $v' := f(\mathsf{S}, uid, pw, s')$ and sends (s', v') to the server, encrypted under an encryption key k_{Enc} derived from k via PRF F, i.e. $c := \mathsf{Enc}(k_{\mathsf{Enc}}, (s', v'))$. The server decrypts c to obtain (s', v'). It securely deletes the old password file and stores (s', v') as the new $\mathsf{file}[uid]$. Finally, both parties derive the session key k' from k using PRF F to ensure proper key separation. The construction is given in Fig. 2.

4.3 Security Argumentation

For security to hold, we argue two things: first, updating the salt invalidates the attacker's precomputation up to that point, and second, by sending the updated password file encrypted under a key derived from the salted aPAKE key, no other party than the authenticated server will learn the updated password file.

Updating Salt Negates Precomputation Efforts. The precomputation security provided by salted aPAKE assumes that an adversary must know (S, uid, s) to generate a table of precomputed values

using the deterministic function f. Crucially, this requires the adversary to know the user-specific salt s, which is uniformly random over $\{0,1\}^\lambda$. By refreshing this salt with every successful user authentication, and by securely deleting the old password file, our transform ensures that any prior precomputed values become obsolete. To mount a new precomputation attack, the adversary must first obtain the updated salt s' which is required to compute $f(\mathrm{S}, uid, s', pw^*)$ for password guesses pw^*. Since the salt is not exposed to passive eavesdroppers, an attacker must actively initiate a session on behalf of the user in order to recover it.

Encryption Protects Updated Password File. The client sends a password file encrypted under a key derived from the aPAKE key k to the server. Since knowledge of the password file enables an attacker to perform offline attacks, it is crucial that the protocol does not reveal any information about the password file to the attacker. In particular, a malicious server must not be able to learn a valid password file, as such a phishing attack resistance is exactly what aPAKE aims to prevent.

The security of the updated password file is guaranteed through the authenticated encryption which uses an encryption key derived from the aPAKE-established key k. Specifically, security relies on the server authentication of aPAKE which guarantees that only a server holding a password file for the user's password can establish the session key. This server authentication can come in two forms. In explicit authentication, the client only derives a key $k \neq \perp$ if the server holds a valid password file. In implicit authentication, the client always derives a key k, but only a server with the correct password file can compute the same key. Our blueprint is compatible with both variants and in fact, our protocol UPAKE leverages the implicit server authentication provided by OKAPE-HMQV, while UPAKE-SRP builds on SRP-6a, which achieves explicit server authentication.

Another important security aspect is that the password file must arrive at the server in an unaltered form: as the new password file will be used in the subsequent login session of the user, an attacker must not be able to modify the file and thereby lock the user out of her account. This is achieved via the use of *authenticated* encryption, and only completing the session – including the password file update – when decryption of the file succeeded.

Overall, our transform replaces the user's password file with every successful login, without leaking any information about the new password file to an attacker.

4.4 Password Consistency

In our transform, a successfully authenticated client updates the password file with a new (s', v'). Since there is no check whether (s', v') belongs to the same password as before, this essentially allows changing the user password within the aPAKE authentication session. Importantly, such a password change can only be performed by an attacker that already knows the password. This means that it does not open any new attack vectors: most applications allow an authenticated attacker (i.e. one who knows the password) to change the password within the application interface anyway. Such a change is however not allowed in the $\mathcal{F}_{\mathsf{uaPAKE}}$ functionality, which implicitly ensures that the stored user password must be consistent throughout all sessions. Thus, as the sketched transform does allow such changes, it does not fully satisfy the $\mathcal{F}_{\mathsf{uaPAKE}}$ security.

To address this mismatch with the ideal functionality, we considered three potential approaches. The first option would be to extend $\mathcal{F}_{\mathsf{uaPAKE}}$ to support password changes explicitly. However, this would significantly increase the complexity of the model. Note that our blueprint allows the client to upload an arbitrary password file. As a result, an attacker who correctly guessed the user's password could provide a file unrelated to any actual password yet still enable the attacker to authenticate. For instance, if the aPAKE password file is (s, g^x) where $x := H(s, pw)$ is used for authentication, the attacker could simply choose a random x and later authenticate with the server using x directly. Modeling this would require the functionality to track maliciously created password files, and provide an interface for future sessions where authentication succeeds at the attacker's discretion – effectively decoupling authentication from any password. To avoid this complexity, we chose to the stick to the simpler functionality that enforces password consistency.

Now, the question is how to design or prove that a protocol satisfies this property. On the protocol level, we can enforce strict password consistency across sessions. In fact, our protocol in Section 5.4 adds such consistency enforcement through a NIZK where the client shows that the updated password file is derived from the same password as before. While technically interesting, we view this as more of a conceptual add-on, as it adds complexity to the protocol merely to prevent a benign behavior (i.e. changing the password) that is unavoidable on the application level anyway.

Another option – and the one we chose – is to simply exclude this adversarial behavior in the security analysis. Specifically, we assume a restricted attacker model in which the adversary does not change the user's password. This assumption simplifies both the protocol and the security analysis, and excluding this behaviour can be done efficiently in the proof.

5 UPAKE Construction

In this section, we present our provably secure updatable aPAKE protocol UPAKE. The construction instantiates our blueprint from the previous section, for a concrete (salted) aPAKE protocol: OKAPE-HMQV [20]. We selected OKAPE-HMQV as the underlying aPAKE due to its low computational and communicational cost, and its established UC security.

We begin by reviewing the original OKAPE-HMQV protocol, and then present our changes to obtain UPAKE. The core change is that we use the shared key established through OKAPE-HMQV to securely transmit a new password file from the client to the server. Since the client already sends the final message in OKAPE-HMQV, transmitting the encrypted password file requires no additional communication round. Our proof comes with one caveat though – we assume "behaving adversaries", i.e. the attacker is not allowed to change an honest user's password through the protocol. While we view this attack as benign – since password changes by authenticated users are typically permitted in password-based applications – we also propose an extension UPAKE+ which ensures that user passwords cannot be changed when updating the salt by adding a NIZK to the protocol.

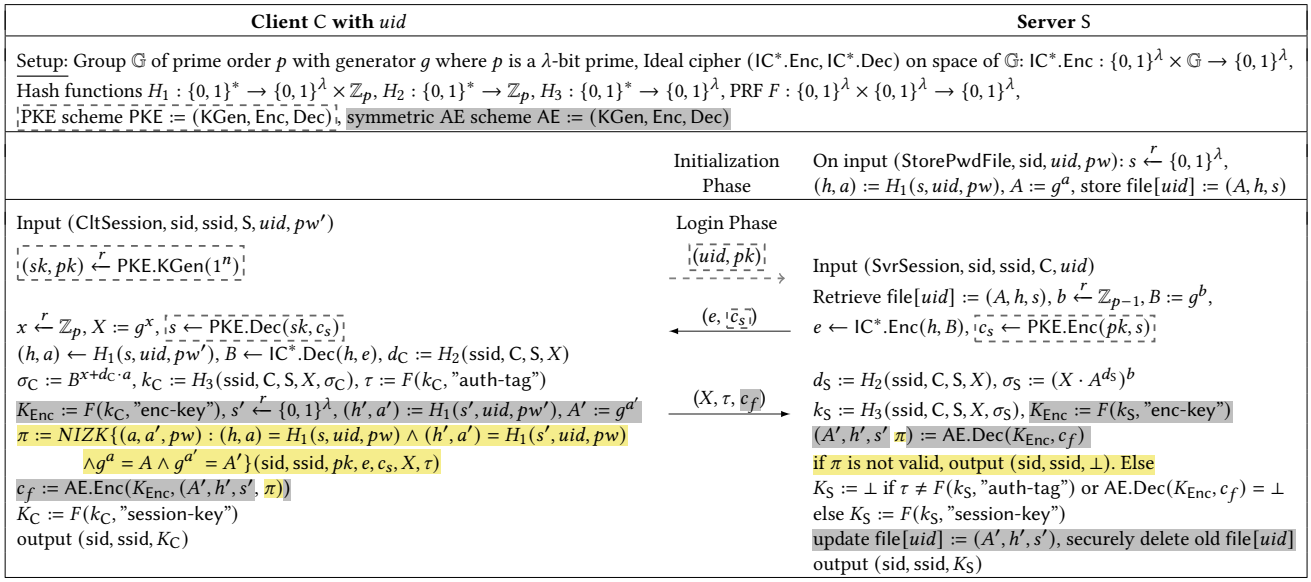


Figure 3: The UPAKE and UPAKE+ protocol. The steps added with respect to OKAPE-HMQV are highlighted: steps to encrypt the salt are highlighted with dashed boxes and are optional if the protocol runs over an encrypted channel. UPAKE additionally includes the grey text, and UPAKE+ includes all highlighted text.

5.1 Starting Point: OKAPE-HMQV

We build our protocol upon the aPAKE protocol OKAPE-HMQV presented in [20]. OKAPE [20] is a compiler which transforms a key-hiding one-time-key authenticated key exchange (otkAKE) into an UC-secure aPAKE utilizing the random oracle (RO) and an ideal cipher (IC). The otkAKE building block allows two parties to securely establish a shared key such that one party is authenticated based on a long-term key-pair while the other party uses one-time keys each time the parties want to exchange keys.

The core idea of the OKAPE compiler is to run the otkAKE protocol in each login session, where the client has a long-term AKE key pair (a, A), while the server employs a one-time key pair (b, B). To achieve password-based user authentication, the client's key pair is derived from her password pw and a salt s, i.e. $a = H(s, uid, pw)$. The server stores the corresponding public key and salt together with a value h which is also derived from the password hash as the user's password file. Precisely, during registration, the server computes the user's password file using a deterministic function $f(S, uid, s, pw)$ which returns (g^a, h) for $(h, a) := H_1(s, uid, pw)$.

To authenticate the server – that is, to ensure that it uses the correct password file established during registration – the server sends its ephemeral public key B ideal-cipher encrypted under the password-derived hash h which is also part of the user's password file. The use of an ideal cipher ensures that the client can correctly decrypt and recover B only if the server used the correct h. Once the client has obtained B, the parties execute the otkAKE protocol using (a, B) on the client side and (A, b) on the server side to derive a shared session key.

The key-hiding property of otkAKE further guarantees that no information on the AKE public keys (A, B) is revealed unless an attacker actively engages in the protocol and correctly guesses the counterparty's public key. Otherwise, leaking A or B would enable offline attacks for an attacker with knowledge of s and e since A is derived from $H(s, pw)$ and it holds that $e = \text{IC}^*.\text{Enc}(H(s, pw), B)$.

OKAPE-HMQV Description. Dos Santos et al. also present a highly efficient instantiation [20] which uses the one-pass HMQV protocol [34] as otkAKE whose security is based on the Gap CDH assumption. The protocol, depicted in Fig. 3, operates over a group $\mathbb{G}$ of prime order p. The client's HMQV long-term key-pair is (a, g^a), derived from the salted hash $(h, a) := H_1(s, uid, pw)$. The server uses a one-time HMQV key-pair (b, g^b) where $b \xleftarrow{r} \mathbb{Z}_p$ is chosen at random. Additionally, the client generates an ephemeral key-pair (x, g^x). The HMQV shared key is $\sigma := g^{bx+bd_C a}$, where $d_C := H_2(\text{ssid}, C, S, X)$ serves as a challenge value to ensure that the client possesses knowledge of a rather than merely g^a, similar to the role of challenges in zero-knowledge proofs. Importantly, both parties are able to compute σ using their own HMQV private key and the counterparty's public key: the client computes $\sigma := (g^b)^{x+d_C a}$, while the server derives $\sigma := (g^x \cdot (g^a)^{d_C})^b$.

5.2 UPAKE: Updating Salts in OKAPE-HMQV

In order to achieve updatable aPAKE security, we follow our blueprint and update the salt with each successful user authentication. Specifically, the client generates a new salt and password file and sends it to the server, encrypted under an encryption key derived from the OKAPE-HMQV key. With this change, the attacker is forced to recompute its precomputation table every time the user successfully authenticates. To prevent salts being leaked immediately to passive eavesdroppers, we further apply a transform described in Dayanikli and Lehmann [19] which encrypts the salt (originally sent in clear in OKAPE-HMQV), thus forcing the attacker to actively participate in the protocol in order to get it.

Encrypting the Salt. In OKAPE-HMQV, the server sends the salt s to the client so she can derive her AKE private key a and the hash h from $H_1(s, uid, pw)$. However, since s is sent in plaintext, it is exposed to passive eavesdroppers, enabling an attacker to launch precomputation attacks after observing just a single login session between server and client. Note that the UC framework assumes an all-powerful adversary with full network control, which turns this benign observation into a massive-scale attack. To formally achieve UC-security, this salt must be encrypted, as argued in [19]. Hence, the client sends a public encryption key pk and the server responds with $c_s := \mathsf{Enc}(pk, s)$.

In practice, however, if we assume a TLS-encrypted channel or a more realistic attacker who does not observe all network communications, the salt does not need to be additionally encrypted within the protocol, and we view this encryption more as a formality for UC security.

Updating the Salt. The core change is now in continuously updating the salt with every authentication session, but without introducing any offline attackable information. Following the blueprint in Sec. 4, the client, with each successful authentication, generates a new password file using a fresh salt, and sends it to the server encrypted under a key derived from the aPAKE key. The server then replaces the stored password file with the updated version.

In more detail, during authentication, the client and server first establish a key k using OKAPE-HMQV. The client then selects a fresh salt s' and derives a new password file $(A' := g^{a'}, h', s')$ from $H_1(s', uid, pw)$. The client derives an encryption key k_{Enc} from k using PRF F and encrypts the new password file under k_{Enc}, i.e. computes $c := \mathsf{Enc}(k_{\mathsf{Enc}}, (A', h', s'))$, and sends c_f to the server. The server can derive k_{Enc} from its own key k to decrypt c_f and update the password file. Finally, the overall session key is derived from k using F. Note that due to domain separation, we can securely re-use the same PRF F which is used to derive the authentication tag in OKAPE-HMQV for this purpose. Moreover, adding the encrypted password file c_f to the protocol incurs no extra communication round as it can be sent together with the client's OKAPE-HMQV message (X, τ).

5.3 Security Analysis of UPAKE

As mentioned in Sec. 4.4, UPAKE, as an instantiation of our transform, allows password changes within the protocol. Precisely, an attacker who starts a session with the honest server and guesses the user's password pw correctly, will be able to compute a key k_{C} which matches the server's key k_{S}. The attacker can then choose a different password pw^*, compute the password file (A^*, h^*, s^*) derived from pw^* and send it to the server encrypted under the encryption key k_{Enc} derived from k_{C}. As a result, the server updates the user uid's password file to (A^*, h^*, s^*). In a subsequent authentication attempt by the legitimate user with pw, authentication will fail as the password file now maps to pw^*, effectively locking the user out of her account.

Behaving Adversaries. Since password changes in the protocol are not allowed by $\mathcal{F}_{\mathsf{uaPAKE}}$, we restrict our analysis to adversaries which do not change a user's password. That is, if the attacker successfully authenticates as a user uid, then the password file is updated correctly and remains consistent with the original password. We call an adversary who adheres to this restriction as "behaving". Note that the adversary can still behave arbitrarily with other protocols, so UC composition security remains unaffected. To ensure the soundness of our security analysis we also need to verify that an adversary adheres to this assumption. Specifically, we must be able to *efficiently check* whether the adversary causes the server to update the password file correctly. In our security analysis, we can use the additional abilities of the simulator (i.e. its control of the random oracle) to verify this.

We now prove the security of UPAKE, showing that the proposed blueprint can lift the security of OKAPE-HMQV:

Security statement (informal): If OKAPE-HMQV is a secure aPAKE protocol, and AE *is an authenticated encryption scheme, F a secure PRF, and* PKE *is CPA-secure, then UPAKE is a secure updatable aPAKE in the presence of behaving adversaries.*

Rather than proving this in a generic way (which would require explicit properties on the OKAPE-HMQV password files), we prove security of the entire protocol, including the OKAPE-HMQV part.

Theorem 5.1. *If* IC *is an ideal cipher, F a secure PRF, H_1, H_2, H_3 random oracles,* AE *an authenticated encryption scheme, and* PKE *is a CPA-secure public key encryption scheme,if the Gap CDH assumption hold in $\mathbb{G}$, then UPAKE securely UC-realizes the functionality $\mathcal{F}_{\mathsf{uaPAKE}}$ in the presence of behaving adversaries.*

Proof Sketch. The full proof is given in App. B and builds upon the proof of the OKAPE compiler from Dos Santos et al. [20], but uses the concrete otkAKE protocol one-pass HMQV. Here, we provide a proof sketch focusing on the novel parts introduced by UPAKE's salt update mechanism. Specifically, we argue how c_f protects the client's updated password file from being leaked to an adversary, and how we can detect non-behaving adversaries.

Security provided by c_f. In UPAKE, the client creates a new password file (A', h', s'), encrypts it as $c_f := \mathsf{AE.Enc}(k_{\mathsf{Enc}}(A', h', s'))$ using encryption key k_{Enc} derived from key k_{C}, and sends the resulting ciphertext to the server. We consider two cases:

Authenticated Server Here, the server can derive k_{Enc} and decrypt c_f to obtain the updated password file. The simulator's challenge is to create a password file based on the real user password. It can extract the server's password from (e, c_s), as it controls the ideal cipher (e must be a result of an $\mathsf{IC}^*.\mathsf{Enc}(h, B)$ query). With the correct pw, it can create an updated file and encryption key which allows perfect simulation.

Unauthenticated server If the server is not authenticated, then $B := \mathsf{IC}^*.\mathsf{Dec}(h, e)$ for $(h, a) := H_1(s, uid, pw)$ with the client's password pw is a random group element. In this case, the adversary cannot compute the encryption key k_{Enc} as this would require computing $B^{x+d_C a}$ and thus breaking the Gap CDH assumption. Since the adversary has no information about the encryption key, the IND-EAV security of AE protects the updated password file. In this case, the simulator can just encrypt random data which will be indistinguishable.

Detecting Behaving Adversaries. We require the simulator to detect whether the adversary is behaving, and abort simulation otherwise. The simulator can do this with its power of controlling the ideal cipher and the random oracle. Specifically, it can embed a trapdoor b into the IC^*.Dec oracle by answering fresh queries (h, e) to IC^*.Dec with g^b. Then, it can extract the adversary's password from message (X, τ, c_f), as there must be $g^b := \mathsf{IC}^*.\mathrm{Dec}(h, e)$ and $(h, a) := H(pw, s)$ such that $\tau = F(k_C, \text{"auth-tag"})$ for $k_\mathsf{C} := H_3(\mathrm{ssid}, \mathsf{C}, \mathsf{S}, X, \sigma_\mathsf{C})$ with $\sigma_\mathsf{C} = (X \cdot (g^a)^{d_\mathsf{S}})^b$. This is the same strategy as in the OKAPE-HMQV proof to extract the adversarial client's password – one of the main challenges in UC aPAKE proofs. Then, using the password it can check whether the new password file (A', h', s') corresponds to the same password by computing $H_1(s', uid, pw')$ and comparing the result to (A', h').

5.4 Ensuring Password Consistency with NIZK

For UPAKE, we had to restrict the analysis to attackers which do not change a user's password. We now extend the protocol to achieve updatable aPAKE security without this restriction. To achieve this, we add a NIZK to the protocol which prevents password changes. Note that we only prevent the client – or an attacker impersonating the client – from changing the password *within the authentication protocol*. However, password changes at the application level, outside of the aPAKE protocol, might still be possible.

The Challenge of Checking Password Consistency. In UPAKE, during each login, the client authenticates to the server, which holds a password file (A, h, s). Upon successful authentication, the client generates a new password file (A', h', s') using the same password but a fresh salt, and transmits it to the server, who overwrites the stored password file. To ensure that the password itself remains unchanged during this update, the server must verify that both the old and new password files were derived from the same underlying password. To address this, we employ a NIZK: the client provides a NIZK proof that both the original and updated password files were computed from the same password. This enables the server to verify password consistency efficiently and securely, without learning the password.

UPAKE+: Adding NIZK to UPAKE. The password files in UPAKE take the form (g^a, h, s), where s is a randomly chosen salt, and $(a, h) := H_1(s, uid, pw)$. To update the password file, the client selects a new salt s' at random and constructs an updated password file $\mathsf{file}' := (g^{a'}, h', s')$, where $(a', h') := H_1(s', uid, pw)$. The client then generates a NIZK prove that both of these password files correspond to the same password, and binds it to the session by including sid, ssid and the protocol transcript (pk, e, c_s, X, τ) as context. Note that the NIZK also hides the client's HMQV secret keys (a, a') from the server, as knowing these would allow the server to impersonate the client (without knowing her password) – something that is not allowed for aPAKE security. The client sends this NIZK to the server, also encrypted under the encryption key k_{Enc}. After decryption, the server holds the updated password file together with π, and updates the password file to $\mathsf{file}' = (g^{a'}, h', s')$, only if π verifies. The protocol UPAKE+ is given in Fig. 3.

Security Analysis. We prove that UPAKE+ is a secure updatable aPAKE protocol. Since it only extends UPAKE by incorporating a NIZK proof, the protocol largely inherits its security properties from UPAKE. However, the addition of the NIZK provides additional security by preventing adversaries from inserting a new password file without knowing the correct password. Thus, UPAKE+ is even secure in the presence of non-behaving adversaries which try to change a user's password in a protocol execution. We provide an extended proof sketch in App. B.1. The main insight of the proof is that the NIZK provides two properties: (1) the zero-knowledge property ensures that the server when receiving π does not learn anything about the witness, i.e. the (a, a', pw) which would allow it to impersonate the client, and (2) the simulation soundness ensures that when the server receives a valid proof, the statement must be true and the updated password file remains consistent with the original password file.

Theorem 5.2. *Let the building blocks be as in Thm 5.1. If additionally, π is a simulation-sound NIZK, then UPAKE+ securely UC-realizes the functionality $\mathcal{F}_{\mathsf{uaPAKE}}$ in the presence of static corruption.*

Efficiency of NIZK Proof. To evaluate the computational overhead introduced by UPAKE+ compared to UPAKE, we analyze the cost of generating the NIZK proof required by the protocol. To instantiate the NIZK, we use Groth16 [25], a state-of-the-art scheme for pairing-based zkSNARKS known for its succinct proofs and efficient verification. First, we employ Circom [6] to translate the NIZK into circuits of a Rank-1 Constraint System (R1CS). To instantiate the hash function H_1, we use Poseidon [24], a hash function optimized for zk-SNARKS which is indifferentiable from a random oracle up to a certain capacity [24]. The resulting R1CS circuit consists of 1212 constraints. To estimate proving cost, we use the established estimates for Groth16 proofs of 3 exponentiations in G_1 and 1 exponentiation in G_2 per constraint [25]. We estimate the total proving time using the zkalc benchmarking tool (https://www.zka.lc). On the BLS12-381 curve, proof generation takes under 60ms. As Groth16 is succinct, the resulting proof is small (128B) and allows fast verification (approx. 3 pairings or 4ms).

6 UPAKE-SRP: Boosting a Legacy aPAKE

In this section, we apply the techniques of our blueprint to another aPAKE protocol that relies on salted password hashes: the Secure Remote Password protocol (SRP-6a). SRP-6a is the most widely deployed aPAKE protocol, with notable deployments in applications such as 1Password [22], Apple [3], and Telegram[40]. The modifications we propose to achieve updatable aPAKE security can be integrated into existing systems with minimal overhead, allowing them to retain the same architecture. We thus show that the security of this widely used protocol can be brought close to that of strong aPAKE, without significant implementation cost.

We begin by describing the SRP-6a protocol. Our analysis highlights that SRP-6a shares several structural similarities with the more recent OKAPE-HMQV protocol – both rely on a Diffie–Hellman-based key exchange and differ primarily in how the server's public key is encrypted. We then present our updatable aPAKE variant, denoted UPAKE-SRP, and argue its secure by building on the existing proof for SRP-6a [18] and our proof for UPAKE.

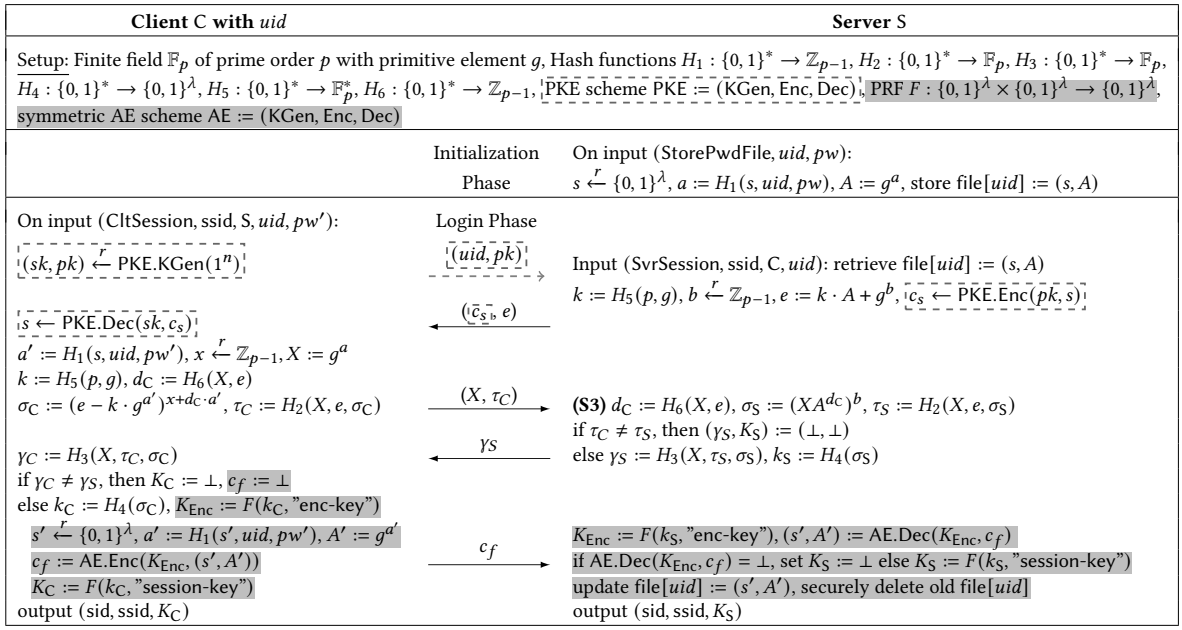


Figure 4: The UPAKE-SRP protocol based on the SRP-6a protocol description of [18]. The added steps to encrypt the salt are highlighted with dashed boxes, while the added steps to update the password file are highlighted in gray. We changed the notation to match the notation of OKAPE-HMQV. An overview of our notational changes can be found in Table 3 in App. C.

6.1 Building Block: SRP-6a

The SRP-6a protocol is an aPAKE protocol which operates in a finite field $\mathbb{F}_p$ of prime order p with primitive element g. Structurally, SRP-6a bears a close resemblance to OKAPE-HMQV, as it utilizes a slightly modified variant of the otkAKE one-pass HMQV in a similar fashion as OKAPE: the client's long term key-pair is derived from a salted password hash and the server uses a HMQV one-time key-pair in each authentication session. The main difference lies in how the server's one-time public key is encrypted to the client: In OKAPE, this key is encrypted using an ideal cipher, while in SRP-6a the server masks this key by adding it (in the finite field) to the client's long-term HMQV public key. Since this masking technique does not provide server authentication, SRP-6a includes an additional key confirmation message from the server to the client after the shared HMQV key has been established.

Protocol Description. To register a client, the server stores the client's HMQV long-term public key $A := g^a$, where $a := H_1(s, uid, pw)$ is a hash of the salt s, user identifier uid, and password pw.

During each authentication session, the server sends the salt s to the client, allowing it to recompute its HMQV long-term secret key a. Additionally, the server sends its HMQV one-time public key $B := g^b$ in a blinded form as $e := k \cdot A + B$ where k is a hash of public parameters (p, g). Upon receiving e, the client generates an ephemeral key pair (x, X), recomputes its HMQV secret key $a := H_1(s, uid, pw)$ and uses it to unblind e and recover $B = e - k \cdot A$. Using (a, x, B), the client computes the HMQV shared key $\sigma_C := B^{x + d_C a}$, where $d_C := H_6(X, e)$ is a hash of the exchanged messages similar to d_C in HMQV.

The client sends X to the server and demonstrates knowledge of σ_C by transmitting $\tau_C := H_2(X, e, \sigma_C)$. Knowing its own HMQV secret key b, the client's ephemeral public key X, and the stored HMQV long-term key A, the server can compute the HQMV key $\sigma_S := (X \cdot A^u)^b$, which equals σ_C if the password is correct. The server verifies τ_C and responds with $\gamma_S := H_3(X, \tau_C, \sigma_S)$ to prove its own knowledge of σ_C. Finally, both parties derive a session key from the shared value $\sigma_C = \sigma_S$.

SRP-6a is OKAPE-HMQV in Disguise? We observe that SRP-6a can be seen as a (weird) variant of OKAPE-HMQV, where the ideal cipher encryption $e := \mathsf{IC}^*.\mathsf{Enc}(h, B)$ is replaced by an additive mask $e := B + k \cdot A$ where h and A are each part of the user's password file. Clearly, this additive masking is far from being an ideal cipher, which is what makes the security proof of SRP-6a more challenging [18]: (1) limits on online password guessing follow naturally from an ideal cipher model in OKAPE-HMQV, whereas the proof of SRP-6a requires a new assumption to argue the same property. [18] introduces a novel finite field assumption to ensure σ_S can only be computed for a single $A := g^{H_1(s, uid, pw^*)}$, i.e., one $B := e - k \cdot A$. (2) Extractability of passwords comes also for free in the ideal cipher model used by OKAPE-HMQV, whereas the SRP-6a proof utilizes a DDH oracle to for this extraction in the security proof. We further elaborate these differences and similarities in App. C. The core insight of our observation is, that we can use the security argumentation and ideas from our UPAKE protocol and adapt them to SRP-6a as well.

6.2 UPAKE-SRP Protocol

To elevate the security of SRP-6a to meet the updatable aPAKE security notion, we apply the same modifications as we did in UPAKE. First, to prevent passive leakage of the salt, the server encrypts the user's salt using an ephemeral encryption public key provided by the user. Second, we incorporate the salt update mechanism where

Table 2: A comparison of our schemes UPAKE, UPAKE+ and UPAKE-SRP with their underlying aPAKE protocols OKAPE-HMQV and SRP-6a, as well as with existing strong aPAKE protocols. For the computational costs, we focused on the number of exponentiation per party, where FF denotes that the exponentiation happens in the finite field.

	aPAKE Security	Client cost	Server cost	Assumption	Model	No. of rounds
OKAPE-HMQV [20]	regular	2 exp. +1 IC dec.	3 exp. +1 IC enc.	GCDH	RO, IC	2
SRP-6a [42]	regular	3 exp. (FF)	3 exp. (FF)	aSDH, GCDH	RO, angel-based UC	3
MX-EKE [37]	strong	3 exp.	4 exp.	CDH/GACDH	RO, GG(A)M, AG(A)M	2
OPAQUE-HMQV[33]	strong	5 exp.	4 exp.	OMGDH	RO	3
AuCPace [27]	strong	6 exp.	5 exp.	sSDH, OMGDH	RO	3
CKEM-saPAKE [12]	strong	13 exp.	8 exp.	2-SDH, DDH	RO + GGM	2
UPAKE [Our work]	updatable	3 exp. +1 IC dec.	3 exp. +1 IC enc.	GCDH	RO, IC	2
UPAKE+ [Our work]	updatable	3 exp. +1 IC dec. +1 NIZK	3 exp. +1 IC enc.	GCDH	RO, IC	2
UPAKE-SRP [Our work]	updatable	4 exp. (FF)	3 exp. (FF)	aSDH, GCDH	RO, angel-based UC	4

the client creates a password file (s', A') with $A' := g^{H_1(s', uid, pw)}$ using a new salt s' and sends it to the server, encrypted under a key k_{Enc} derived from the SRP key k_C. The server updates the password file and securely deletes the old password file of the user. The protocol is given in Fig. 4.

More Efficient UPAKE-SRP? Since SRP-6a concludes with a server-to-client key confirmation message, our salt update introduces an additional round from the client to the server on top of the SRP-6a protocol. Given the similarities with OKAPE-HMQV, one may wonder whether the salt update could instead be sent earlier – specifically, along with the client's message (X, τ_C). At that point, the server is already implicitly authenticated, i.e. it should only be able to derive the shared key if it has guessed the correct password (file), and we believe this should also yield a secure protocol. However, formally proving its security is more involved and requires additional assumptions on the AE building block. In particular, since the addition $e = B + k \cdot A$ does not bind the server to a password guess, the simulator cannot extract an adversary's password from its first message (c_s, e). The simulator must therefore simulate c_f without knowing the corresponding plaintext – yet in a way that allows it to later match c_f to the correct plaintext, once the adversary's password becomes extractable and can be verified against the client's. This would require equivocable encryption, a primitive also used in the security analysis of OPAQUE [33]. To avoid relying on such additional assumptions, we chose a less efficient variant with simpler building blocks.

Security Analysis. Since UPAKE-SRP also allows password changes, we again show security in restrict the presence of behaving adversaries. We state the security statement to lift security from SRP-6a, and show security for the concrete UPAKE-SRP protocol. Due to the protocols' similarities, the proof follows closely the proof of UPAKE, only that the server is now explicitly authenticated when the client sends the updated password file c_f. The formal theorem and proof sketch are deferred to App. D.

Security statement (informal): If SRP-6a is a secure aPAKE protocol, AE *an authenticated encryption scheme, F a secure PRF, and* PKE *is CPA-secure, then UPAKE-SRP is a secure updatable aPAKE in the presence of behaving adversaries.*

7 Comparison & Discussion

We now compare our updatable aPAKE protocols to strong aPAKE, focusing on efficiency and post-quantum friendliness. We argue that updatable aPAKE is superior in both aspects while essentially guaranteeing the same security as strong aPAKE. Finally, we discuss the privacy implications of salt exposure as metadata.

Efficiency. Table 2 compares the efficiency of our updatable aPAKE protocols with their underlying aPAKEs and with existing strong aPAKE protocols. For this comparison, we ignore the cost of encrypting the salt in our protocols, as this is mainly a UC artifact, where all communication is adversarially controlled, and the salt thus immediately considered leaked if it was sent in plaintext. In practice, where the attacker does not control all communication channels, or if we assume a TLS-encrypted channel, encrypting the salt would not be necessary.

First, as UPAKE-SRP is based on SRP-6a which operates in finite fields, it is significantly slower than the other protocols which can be instantiated over elliptic curves. Nevertheless, SRP-6a is the most widely deployed aPAKE, and our strengthened version UPAKE-SRP induces minimal overhead compared to SRP-6a: 1 client exponentiation and one extra communication round.

For UPAKE and UPAKE+, we observe that they achieve optimal round efficiency with 2 rounds and requires only 3 exponentiations and one ideal cipher operation per party, with UPAKE+ additionally adding the cost of proving a NIZK on the client side. Note that the ideal cipher in this setting operates over group elements, as in [5, 7, 20, 26]. Realizing an ideal cipher over group elements is non-trivial, and current implementation techniques are discussed in [20, 26]. The most efficient known instantiation introduces an overhead of one exponentiation for both encryption and decryption. Notably, UPAKE and UPAKE+ are more efficient than OPAQUE [33], the strong aPAKE protocol currently being standardized by the IETF [11], and are slightly more efficient than the strong aPAKE protocol MX-EKE of McQuoid and Xu [37]. Further, note that the security of MX-EKE relies on the algebraic group (action) model, an idealized model whose usefulness is not entirely clear [44], whereas the security of our protocols are based on more natural assumptions.

Post-Quantum Friendliness. An important feature of our updatable aPAKE blueprint is its compatibility with post-quantum (PQ) secure schemes. Specifically, our transformation can be applied to any PQ-secure aPAKE, such as the construction by Lyu et al. [36], and only requires the addition of a PQ-secure authenticated encryption scheme and PRF to achieve a PQ-secure updatable aPAKE. As symmetric building blocks can simply be lifted to be quantum-safe by doubling the key length, we can essentially achieve updatable aPAKE security at the cost of standard (PQ) aPAKE.

In contrast, the OPAQUE compiler additionally relies on an Oblivious Pseudorandom Function (OPRF), a primitive that remains challenging to instantiate securely in the post-quantum setting. Moreover, most existing strong aPAKE protocols are not quantum-safe [12, 16, 27, 33]. The sole exception is the saPAKE by McQuoid and Xu [37], which achieves PQ security assuming cryptographic group actions – such as the isogeny-based CSIDH [15]. However, this restricts the protocol to this specific assumption.

Privacy Considerations: Leaking Login Metadata. All our updatable aPAKE constructions reveal additional metadata about the users: each successful login updates the salt, and an (active) attacker can learn this information and infer whether a user has logged in recently. This can have both advantages and disadvantages. On the one hand, usage patterns can aid attackers, e.g., rarely used accounts give more time for precomputation, while active ones may be high-value targets. Additionally, this information could have unintended privacy implications (e.g., a spouse discovering continued use of a dating app). On the other hand, clients can also use this metadata for detecting unauthorized access to their accounts: by remembering the salt from their last login, a user can check if it changes unexpectedly, possibly indicating unauthorized access.

References

[1] Martin R Albrecht, Alex Davidson, Amit Deo, and Daniel Gardham. 2024. Crypto dark matter on the torus: Oblivious prfs from shallow prfs and tfhe. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 447–476.

[2] Martin R Albrecht, Alex Davidson, Amit Deo, and Nigel P Smart. 2021. Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In *IACR International Conference on Public-Key Cryptography*. Springer, 261–289.

[3] Apple. 2022. HomeKit Communication Security. https://support.apple.com/guide/security/communication-security-sec3a881ccb1/web.

[4] Andrea Basso. 2023. A post-quantum round-optimal oblivious PRF from isogenies. In *International Conference on Selected Areas in Cryptography*. Springer, 147–168.

[5] Mihir Bellare, David Pointcheval, and Phillip Rogaway. 2000. Authenticated key exchange secure against dictionary attacks. In *International conference on the theory and applications of cryptographic techniques*. Springer, 139–155.

[6] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2022), 4733–4751.

[7] Steven Michael Bellovin and Michael Merritt. 1992. Encrypted key exchange: Password-based protocols secure against dictionary attacks. (1992).

[8] Ward Beullens, Lucas Dodgson, Sebastian Faller, and Julia Hesse. 2025. The 2Hash OPRF framework and efficient post-quantum instantiations. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 332–362.

[9] Dan Boneh, Dmitry Kogan, and Katharine Woo. 2020. Oblivious pseudorandom functions from isogenies. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 520–550.

[10] Dan Boneh and Victor Shoup. 2023. A Graduate Course in Applied Cryptography. (2023).

[11] Daniel Bourdrez, Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. 2024. *The OPAQUE Augmented PAKE Protocol*. Internet-Draft draft-irtf-cfrg-opaque-18. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-irtf-cfrg-opaque/18/ Work in Progress.

[12] Tatiana Bradley, Stanislaw Jarecki, and Jiayu Xu. 2019. Strong asymmetric PAKE based on trapdoor CKEM. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 798–825.

[13] Bart Butler. 2021. Improved Authentication for Email Encryption and Security. https://protonmail.com/blog/encrypted_email_authentication/.

[14] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.

[15] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. 2018. CSIDH: an efficient post-quantum commutative group action. In *ASIACRYPT 2018, Brisbane, QLD, Australia, Proceedings, Part III 24*. Springer, 395–427.

[16] Cas Cremers, Moni Naor, Shahar Paz, and Eyal Ronen. 2022. CHIP and CRISP: protecting all parties against compromise through identity-binding PAKEs. In *Annual International Cryptology Conference*. Springer, 668–698.

[17] Gareth T Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. 2023. Security analysis of the whatsapp end-to-end encrypted backup protocol. In *Annual International Cryptology Conference*. Springer, 330–361.

[18] Dennis Dayanikli and Anja Lehmann. 2024. Provable security analysis of the secure remote password protocol. In *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. IEEE, 620–635.

[19] Dennis Dayanikli and Anja Lehmann. 2024. (Strong) aPAKE Revisited: Capturing Multi-User Security and Salting. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 415–439.

[20] Bruno Freitas Dos Santos, Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. 2022. Asymmetric PAKE with low computation and communication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 127–156.

[21] Sebastian Faller, Astrid Ottenhues, and Johannes Ottenhues. 2023. Composable oblivious pseudo-random functions via garbled circuits. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 249–270.

[22] Rick Fillion. 2018. Developers: How we use SRP, and you can too: 1Password. https://blog.1password.com/developers-how-we-use-srp-and-you-can-too/.

[23] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. 2006. A method for making password-based key exchange resilient to server compromise. In *Annual International Cryptology Conference*. Springer, 142–159.

[24] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A new hash function for {Zero-Knowledge} proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 519–535.

[25] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 305–326.

[26] Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. 2021. KHAPE: asymmetric PAKE from key-hiding key exchange. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part IV 41*. Springer, 701–730.

[27] Björn Haase and Benoît Labrique. 2018. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *Cryptology ePrint Archive* (2018).

[28] Lena Heimberger, Tobias Hennerbichler, Fredrik Meisingseth, Sebastian Ramacher, and Christian Rechberger. 2024. OPRFs from isogenies: designs and analysis. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 575–588.

[29] Julia Hesse. 2020. Separating symmetric and asymmetric password-authenticated key exchange. In *International Conference on Security and Cryptography for Networks*. Springer, 579–599.

[30] Jung Yeon Hwang, Stanislaw Jarecki, Taekyoung Kwon, Joohee Lee, Ji Sun Shin, and Jiayu Xu. 2018. Round-reduced modular construction of asymmetric password-authenticated key exchange. In *SCN 2018, Amalfi, Italy, September 5–7, 2018, Proceedings 11*. Springer, 485–504.

[31] IEEE. 2000. IEEE Standard Specifications for Public-Key Cryptography. *IEEE Std 1363-2000* (2000), 1–228. https://doi.org/10.1109/IEEESTD.2000.92292

[32] ISO. 2006. *Information Technology - Security Techniques - Key Management - Part 4: Mechanisms based on Weak Secrets*. ISO 11770-4:2006.

[33] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. 2018. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part III 37*. Springer, 456–486.

[34] Hugo Krawczyk. 2005. HMQV: A high-performance secure Diffie-Hellman protocol. In *Annual international cryptology conference*. Springer, 546–566.

[35] Xiangyu Liu, Shengli Liu, Shuai Han, and Dawu Gu. 2023. EKE meets tight security in the universally composable framework. In *IACR International Conference on Public-Key Cryptography*. Springer, 685–713.

[36] You Lyu, Shengli Liu, and Shuai Han. 2025. Efficient asymmetric PAKE compiler from KEM and AE. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 34–65.

[37] Ian McQuoid and Jiayu Xu. 2023. An Efficient Strong Asymmetric PAKE Compiler Instantiable from Group Actions. *Cryptology ePrint Archive* (2023).

[38] Victor Shoup. 2020. Security Analysis of SPAKE2+. In *Theory of Cryptography Conference*. Springer, 31–60.

[39] David Taylor, Trevor Perrin, Thomas Wu, and Nikos Mavrogiannopoulos. 2007. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. RFC 5054.

[40] Telegram. 2023. Two-factor Authentication. https://core.telegram.org/api/srp.

[41] Thomas Wu. 2000. The SRP Authentication and Key Exchange System. RFC 2945.

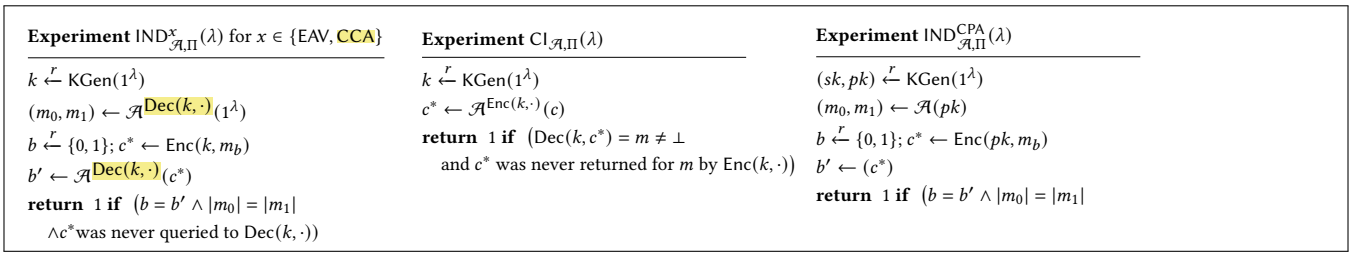


Figure 5: Security experiments for IND-EAV security, CCA and ciphertext integrity for an AE scheme, as well as IND-CPA security for a PKE scheme.

[42] T. Wu. 2002. SRP-6: Improvements and Refinements to the Secure Remote Password Protocol. *http://srp. stanford. edu/srp6.ps* (2002).

[43] Andy Yen. 2023. Proton Pass is now in beta. https://proton.me/blog/proton-pass-beta.

[44] Cong Zhang, Hong-Sheng Zhou, and Jonathan Katz. 2022. An analysis of the algebraic group model. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 310–322.

A Extended Preliminaries

Notation. Throughout this paper, λ denotes the security parameter. For an algorithm A, $y \leftarrow A(x)$ signifies running A on input x to produce output y. With $r \xleftarrow{r} S$ we represent that a value r is chosen uniformly at random from set S. We say that a function f is negligible if for any positive polynomial p, we have $f(\lambda) < 1/p(\lambda)$ for sufficiently large λ. For simplicity, we use negl to denote an unspecified negligible function.

A.1 Encryption

We recall IND-EAV, IND-CCA and ciphertext integrity for a symmetric encryption scheme, as well as CPA security for a PKE scheme. These definitions can also be found in [10].

Definition A.1 (IND-EAV). An AE scheme $\pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ is IND-EAV secure if for every PPT adversary $\mathcal{A}$ it holds that $\Pr[\mathsf{IND}^{\mathsf{EAV}}_{\mathcal{A},\pi}(\lambda) = 1] \leq 1/2 + \mathsf{negl}$ where the experiment $\mathsf{IND}^{\mathsf{EAV}}_{\mathcal{A},\pi}(\lambda)$ is defined in Fig. 5.

Definition A.2 (IND-CCA). An AE scheme $\pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ is IND-EAV secure if for every PPT adversary $\mathcal{A}$ it holds that $\Pr[\mathsf{IND}^{\mathsf{CCA}}_{\mathcal{A},\pi}(\lambda) = 1] \leq 1/2 + \mathsf{negl}$ where the experiment $\mathsf{IND}^{\mathsf{CCA}}_{\mathcal{A},\pi}(\lambda)$ is defined in Fig. 5.

Definition A.3 (CI). An AE scheme $\pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ has ciphertext integrity if for every PPT adversary $\mathcal{A}$ it holds that $\Pr[\mathsf{CI}_{\mathcal{A},\pi}(\lambda) = 1] \leq \mathsf{negl}$ where the experiment $\mathsf{CI}_{\mathcal{A},\pi}(\lambda)$ is defined in Fig. 5.

Definition A.4 (AE). An encryption scheme $\pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ is is an authenticated encryption scheme if it has ciphertext integrity and CCA security.

A PKE scheme is IND-CPA secure, if an attacker, when given a public key pk and a challenge ciphertext c of a message, should not be able to derive any information about the underlying plaintext m. In this scenario, the attacker can obtain multiple encryptions of arbitrary messages as he knows the public key pk.

Definition A.5 (IND-CPA Security). A PKE scheme $\pi = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ is IND-CPA secure if for every PPT adversary $\mathcal{A}$ it holds that $\Pr[\mathsf{IND}^{\mathsf{CPA}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq 1/2 + \mathsf{negl}$ where the experiment $\mathsf{IND}^{\mathsf{CPA}}_{\mathcal{A},\pi}(\lambda)$ is defined in Fig. 5.

A.2 Gap CDH Assumption

Let G be a cyclic group of prime order q with generator g. The Computational Diffie-Hellman (CDH) assumption on G states that when given two group elements $(X, Y) = (g^x, g^y)$ for $(x, y) \xleftarrow{r} \mathbb{Z}_q$ it is hard to compute $\mathsf{cdh}_g(X, Y) := g^{xy}$. The Gap CDH assumption states that CDH is hard even when given a Decisional Diffie-Hellman oracle $\mathsf{ddh}_g(\cdot,\cdot,\cdot)$ which on input (A, B, C) returns 1 if $C = \mathsf{cdh}_g(A, B)$ and 0 otherwise.

B Proof of Theorem 5.1

To show that UPAKE UC-realizes $\mathcal{F}_{\mathsf{uaPAKE}}$, we need to show that there is a simulator SIM which, while interacting only with the ideal functionality $\mathcal{F}_{\mathsf{uaPAKE}}$, can create a view of the protocol which the environment cannot distinguish from an execution of the real protocol. This requires describing how SIM simulates the behaviour of honest parties without knowing their private inputs (i.e. their passwords). Specifically, SIM needs to simulate the leakage of the password file stored by an honest server (i.e., server compromise), as well as all interactions performed by honest parties in a key exchange session. To handle sessions involving an adversary, the simulator's strategy is to extract the adversary's private input (i.e. the password), and utilize the TestPwd interface of $\mathcal{F}_{\mathsf{uaPAKE}}$ to compare it against the honest party's password. The complete description of the simulator is given in Fig. 6. We first describe the proof strategy for the different scenarios before providing a series of game hops which establish indistinguishability of the real and the ideal world.

Password File Handling. Throughout the protocol, the simulator must simulate how an honest server manages password files. This involves simulating protocol executions that rely on the password file and handling its leakage in the event of a server compromise, i.e. through an adversary's StealPwdFile query. When simulating stolen password files, the simulator must ensure that the password file given to the attacker remains consistent with the user's real password, especially since the attacker might know the user's password. Precisely, if the adversary compromises the password file of a *uid* by issuing a $(\mathsf{StealPwdFile}, \mathsf{sid}, uid)$ query, the simulator has

Initialization

- *Convention:* First call to SvrSession or StealPwdFile for (S, uid): pick $s \xleftarrow{r} \{0,1\}^\lambda$, record file$[uid] := (\perp, \perp, s)$. If we explicitly refer to the salt contained in the current password file, it is denoted with s^*

Stealing Password File

- On $\mathcal{Z}$'s permission to do so, send $(\mathsf{StealPwdFile}, \mathsf{S}, uid)$ to $\mathcal{F}$. If $\mathcal{F}$ sends "no password file", SIM passes it to $\mathcal{A}$, otherwise:

(1) If $\mathcal{F}$ returns pw, retrieve record $(H_1, [s, uid, pw], (h, a))$; else, pick $a \xleftarrow{r} \mathbb{Z}_p$
(2) Set $A := g^a$, $h := h$
(3) Update file$[uid] := (A, h, s)$, send file$[uid]$ to $\mathcal{A}$ and add $(\mathsf{sid}, uid, A, h, s, a)$ to T_{comp}.

Simulation of Random Oracles

On $H_1(s, uid, pw)$ from $\mathcal{A}$:

if this is the first such query, then
  if $s = s^* \wedge \nexists$ record $(\mathsf{ssid}, s^*, e, c_s)$ abort.
  if $\exists (A_j, h_j, a_j)$ s.t. $(\mathsf{sid}, uid, A_j, h_j, s, a_j) \in T_{comp}$, send $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{S}, uid, pw)$ to $\mathcal{F}$
    upon answer "correct guess", set $h := h_j$, $a := a_j$, $A := g^a$, $h := h$ and update file$[uid] := (A, h, s)$
    upon answer "wrong guess", set $h \xleftarrow{r} \{0,1\}^\lambda$, $a \xleftarrow{r} \mathbb{Z}_p$
    else set $h \xleftarrow{r} \{0,1\}^\lambda$, $a \xleftarrow{r} \mathbb{Z}_p$
  if $s = s^*$, send $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{S}, uid, pw)$ to $\mathcal{F}$
    upon answer "correct guess", set $h \xleftarrow{r} \{0,1\}^\lambda$, $a \xleftarrow{r} \mathbb{Z}_p$, $A := g^a$, and update file$[uid] := (A, h, s)$
    upon answer "wrong guess", set $h \xleftarrow{r} \{0,1\}^\lambda$, $a \xleftarrow{r} \mathbb{Z}_p$
    else set $h \xleftarrow{r} \{0,1\}^\lambda$, $a \xleftarrow{r} \mathbb{Z}_p$
  else set $h \xleftarrow{r} \{0,1\}^\lambda$, $a \xleftarrow{r} \mathbb{Z}_p$
  if $\exists$ record $(H_1, [*, *, *], (h, a))$ abort
  store $(H_1, [s, uid, pw], (h, a))$ and reply with (h, a)
else retrieve record $(H_1, [s, uid, pw], (h, a))$ and reply with (h, a).

On $H_i(m)$ from $\mathcal{A}$ for $i = 2, 3$:

if this is the first such query, then
  pick $h \xleftarrow{r} \mathcal{R}(H_i)$
  if $\exists$ record $(H_i, [*], h)$ abort
  store $(H_i, [m], h)$ and reply with h
else retrieve record $(H_i, [m], h)$ and reply with h

Simulation of Ideal Cipher

On $\mathsf{IC}^*.\mathsf{Enc}(h, B)$ from $\mathcal{A}$:

if this is the first such query, then
  pick $e \xleftarrow{r} \{0,1\}^\lambda$
  if $\exists$ record $(IC, [h, *, *], e)$ abort
  store $(IC, [h, B, \perp], e)$ and reply with e
else retrieve record $(IC, [h, B, *], e)$ and reply with e

On $\mathsf{IC}^*.\mathsf{Dec}(h, y)$ from $\mathcal{A}$:

if this is the first such query, then
  pick $b \xleftarrow{r} \mathbb{Z}_p$, set $B := g^b$
  store $(IC, [h, B, b], y)$ and reply with B
else retrieve record $(IC, [h, B, *], y)$ and reply with B.

Figure 6: Simulation of Password File, Random Oracles and IC.

to send file$[uid]$ to $\mathcal{A}$ in a way that is indistinguishable from the real execution of the protocol.

In the real protocol, the password file is defined as file$[uid] = (A := g^a, h, s)$ for $(h, a) := H_1(s, uid, pw)$. To simulate the file, the simulator forwards the StealPwdFile query to $\mathcal{F}$, and checks whether the attacker has performed a successful precomputation attack on the password. If this is the case, $\mathcal{F}$ responds with the user's password pw, and the simulator can set $(h, a) := H_1(s, uid, pw)$ for the real user password pw. If there is no successful precomputation attack, i.e. if $\mathcal{F}$ does not respond with a password, the simulator simulates the password file without knowing the password. Therefore, it picks a random (h, a), but stores it to ensure consistency for future offline attacks on the password. Crucially, SIM marks this password file as compromised.

To ensure consistency between the simulated password file and the real user password, whenever the adversary queries OfflineTestPwd with a password pw and a salt s that matches a compromised password file, the simulator tests the password using OfflineTestPwd. If the password is correct, the simulator ensures that the result of H_1 matches the stored (h, a).

Additionally, the simulator must detect valid precomputation attacks by the adversary and forward tested passwords to the functionality using the OfflineTestPwd interface. Therefore, the simulator also forwards passwords used in queries to H_1 when they involve the salt s from the current password file of a user uid, even if

Login Phase

On (CltSession, ssid, C, S, uid) from $\mathcal{F}$:

Pick $(sk, pk) \xleftarrow{r} \mathsf{PKE.KGen}(n)$
Send pk to $\mathcal{A}$ intended for S, keep record (ssid, sk, pk)

On (SvrSession, ssid, S, C, uid) from $\mathcal{F}$ and receiving pk from $\mathcal{A}$:

Retrieve file$[uid] = (*, *, s)$. If $\nexists$ file$[uid]$ do nothing and halt
Pick $e \xleftarrow{r} \{0,1\}^\lambda$
Compute $c_s \leftarrow \mathsf{PKE.Enc}(pk, s)$
Send (e, c_s) to $\mathcal{A}$ intended for C, keep record (ssid, s, e, c_s)

Simulation of client upon receiving (e, c_s) from $\mathcal{A}$:

Retrieve record (ssid, sk, pk)
Compute $s \leftarrow \mathsf{PKE.Dec}(sk, c_s)$
Pick $x \xleftarrow{r} \mathbb{Z}_p$, set $X := g^x$
If $\exists$ record (ssid, s, e, c_s), pick $\tau \xleftarrow{r} \{0,1\}^\lambda, K_\mathsf{C} \xleftarrow{r} \{0,1\}^\lambda, K_\mathsf{Enc} \xleftarrow{r} \{0,1\}^\lambda, c_f \leftarrow \mathsf{AE.Enc}(K_\mathsf{Enc}, (g, 0^\lambda, 0^\lambda))$,
keep record (ssid, s, e, c_s, X, τ, c_f)
If $\exists A_j, h_j, a_j, B$: (sid, uid, A_j, h_j, s, a_j) $\in T_{comp}$ and $\exists$ record $(IC, [h_j, B, *], e)$:
Send (Impersonate, sid, ssid, C, S, uid) to $\mathcal{F}$:
upon answer "correct guess", compute $d_\mathsf{C} := H_2(\text{ssid}, \mathsf{C}, \mathsf{S}, X)$, $\sigma_\mathsf{C} := B^{x + d_\mathsf{C} \cdot a_j}$,
$k_\mathsf{C} := H_3(\text{ssid}, \mathsf{C}, \mathsf{S}, X, \sigma_\mathsf{C})$, $\tau := F(k_\mathsf{C}, \text{"auth-tag"})$, $K_\mathsf{C} := F(k_\mathsf{C}, \text{"session-key"})$, $K_\mathsf{Enc} := F(k_\mathsf{C}, \text{"enc-key"})$,
$s' \xleftarrow{r} \{0,1\}^\lambda$, $a' \xleftarrow{r} \mathbb{Z}_p$, $A' := g^{a'}$, $h' \xleftarrow{r} \{0,1\}^\lambda$, $c_f := \mathsf{AE.Enc}(K_\mathsf{Enc}, (A', h', s'))$,
add (sid, uid, A', h', s', a') to T_{comp}.
upon answer "wrong guess", pick $\tau \xleftarrow{r} \{0,1\}^\lambda, K_\mathsf{C} \xleftarrow{r} \{0,1\}^\lambda, K_\mathsf{Enc} \xleftarrow{r} \{0,1\}^\lambda, c_f \leftarrow \mathsf{AE.Enc}(K_\mathsf{Enc}, (g, 0^\lambda, 0^\lambda))$
If $\exists pw, h, a, B$ s.t. $\exists$ record $(H_1, [s, uid, pw], (h, a))$ and $(IC, (h, B, *), e)$ (abort if not unique)
Send (TestPwd, sid, ssid, C, pw) to $\mathcal{F}$:
upon answer "correct guess", compute $d_\mathsf{C} := H_2(\text{ssid}, \mathsf{C}, \mathsf{S}, X)$, $\sigma_\mathsf{C} := B^{x + d_\mathsf{C} \cdot a}$,
$k_\mathsf{C} := H_3(\text{ssid}, \mathsf{C}, \mathsf{S}, X, \sigma_\mathsf{C})$, $\tau := F(k_\mathsf{C}, \text{"auth-tag"})$, $K_\mathsf{C} := F(k_\mathsf{C}, \text{"session-key"})$, $K_\mathsf{Enc} := F(k_\mathsf{C}, \text{"enc-key"})$,
$s' \xleftarrow{r} \{0,1\}^\lambda$, $(h', a') := H_1(s', uid, pw)$, $A' := g^{a'}$, $c_f := \mathsf{AE.Enc}(K_\mathsf{Enc}, (A', h', s'))$,
add (sid, uid, A', h', s', a') to T_{comp}.
upon answer "wrong guess", pick $\tau \xleftarrow{r} \{0,1\}^\lambda, K_\mathsf{C} \xleftarrow{r} \{0,1\}^\lambda, K_\mathsf{Enc} \xleftarrow{r} \{0,1\}^\lambda, c \leftarrow \mathsf{AE.Enc}(K_\mathsf{Enc}, (g, 0^\lambda, 0^\lambda))$
In all other cases, pick $\tau \xleftarrow{r} \{0,1\}^\lambda, K_\mathsf{C} \xleftarrow{r} \{0,1\}^\lambda, K_\mathsf{Enc} \xleftarrow{r} \{0,1\}^\lambda, c_f \leftarrow \mathsf{AE.Enc}(K_\mathsf{Enc}, (g, 0^\lambda, 0^\lambda))$
Send (NewKey, sid, ssid, C, K_C) to $\mathcal{F}$ and (X, τ, c_f) to $\mathcal{A}$ intended for S.

Simulation of server upon receiving (X, τ, c_f) from $\mathcal{A}$:

Retrieve record (ssid, s, e, c_s) and compute $d_\mathsf{S} := H_2(\text{ssid}, \mathsf{C}, \mathsf{S}, X)$
If $\exists$ record (ssid, s, e, c_s, X, τ, c_f), send (NewKey, sid, ssid, S, $\top$) to $\mathcal{F}$
Upon receiving public delayed output (ssid, $\perp$) finish simulation,
else, pick $s' \xleftarrow{r} \{0,1\}^\lambda$, and update file$[uid] := (\perp, \perp, s')$
If $\exists a, h, B, b$ s.t. $\exists$ records $(H_1, [s, uid, pw], (a, h))$, $(IC, [h, B, b], e)$
and $(H_2, [\text{ssid}, \mathsf{C}, \mathsf{S}, X, \sigma_\mathsf{C}], k_\mathsf{C})$ with $\sigma_\mathsf{C} = (X \cdot (g^a)^{d_\mathsf{S}})^b$ s.t. $\tau = F(k_\mathsf{C}, \text{"auth-tag"})$
Send (TestPwd, sid, ssid, S, pw) to $\mathcal{F}$:
upon "correct guess", compute $K_\mathsf{S} := F(k_\mathsf{C}, \text{"session-key"})$, $K_\mathsf{Enc} := F(k_\mathsf{S}, \text{"enc-key"})$, $(A', h', s') \leftarrow \mathsf{AE.Dec}(K_\mathsf{Enc}, c_f)$
compute $(h', a') := H(s', uid, pw)$; if $h' \neq h'$ or $A' \neq g^{a'}$ abort
update file$[uid] := (A', h', s')$ and add (sid, uid, A', h', s', a') to T_{comp}
upon "wrong guess", set $K_\mathsf{S} \xleftarrow{r} \{0,1\}^\lambda$
Send (NewKey, sid, ssid, C, K_S) to $\mathcal{F}$.

Figure 7: Simulator SIM for UPAKE.

the file has not been compromised. However, if the attacker queries H_1 on the current salt without first initiating a client session with the server for uid, the simulator aborts, as this would violate the intended precomputation attack security of the protocol. Nevertheless, since the salt has high entropy and is always sent encrypted under the IND-CPA secure encryption scheme PKE, where a passive attacker does not have access to the secret key, the probability of the attacker guessing the correct salt, and forcing the simulator to abort is therefore negligible.

Simulate Honest Client Interacting with Adversarial Server. The simulator must emulate the behaviour of an honest client interacting with an adversarial server without knowing the client's password. The challenge for the simulator is to extract the password used by the server from the protocol messages. Once the simulator has extracted this password, it can utilize the TestPwd or Impersonate interface of $\mathcal{F}$ to determine whether both parties are using the same password and proceed accordingly with the simulation.

The simulator extracts the password following the approach in the proof of [20]. This is facilitated by the use of the ideal cipher

which the simulator controls, making the extraction straightforward. When the client receives (e, c_s), the simulator can extract the inputs (h, B) which must have been queried to $\mathsf{IC}^*.\mathsf{Enc}$ to obtain e. If the adversary used pw to create h, then there must be a query $H_1(s, uid, pw)$ with output (a, h). In this case, the simulator extracts the adversary's password and tests it against the client's password using TestPwd and continues simulation. On the other hand, h can be part of a stolen password file (A, h, s) in which case the simulator tests whether the client runs on the same password using interface Impersonate and continues simulation.

In UPAKE the client additionally creates an encrypted password file update $c_f := \mathsf{AE.Enc}(k_{\mathsf{Enc}}(A', h', s'))$ using k_{Enc} derived from key k_{C}. If both parties share the same password, the simulator correctly derives k_{Enc} and constructs a valid password file (A', h', s') using the correct password, ensuring indistinguishability from the real protocol execution. On the other hand, if the passwords do not match, the simulator must simulate c_f without knowing the password used by the client. In this case, it encrypts random "garbage" under a randomly chosen key K_{Enc}. Due to the IND-EAV security of AE (note that K_{Enc} is never reused), the environment cannot distinguish this simulation from a real execution.

Simulate Honest Server Communicating with an Adversarial Client. The simulator must emulate behaviour of an honest server interacting with an adversarial client. Again, the challenge for the simulator is to extract the password used by the client from the protocol messages, and verify it against the server's password using the TestPwd interface.

Extraction again follows the proof of OKAPE [20] and is facilitated by the use of the ideal cipher and the random oracle. First, SIM incorporates a "trapdoor" into $\mathsf{IC}^*.\mathsf{Dec}$ queries: For every fresh query (h, y) to $\mathsf{IC}^*.\mathsf{Dec}$, SIM samples a random $b \xleftarrow{r} \mathbb{Z}_q$ and answers the decryption query with $B := g^b$ while keeping track of this query by recording (h, B, b, y).

During the login phase, instead of computing $e \leftarrow \mathsf{IC}^*.\mathsf{Enc}(h, B)$, the simulator selects a random e from the IC-ciphertext space. Upon receiving the client's response (X, τ, c_f), the simulator checks the client's $\mathsf{IC}^*.\mathsf{Dec}$ queries and uses its trapdoor b to extract the client's pw. Specifically, if the client used pw to create (X, τ, c_f), then there must be $g^b := \mathsf{IC}^*.\mathsf{Dec}(h, e)$ and $(h, a) := H(pw, s)$ such that $\tau = F(k_C, \text{"auth-tag"})$ for $k_{\mathsf{C}} := H_3(\mathsf{ssid}, \mathsf{C}, \mathsf{S}, X, \sigma_{\mathsf{C}})$ with $\sigma_{\mathsf{C}} = (X \cdot (g^a)^{d_{\mathsf{S}}})^b$.

After extracting the client's password, the simulator can test it against the server's password using the TestPwd interface. If the password is correct, the simulator continues simulation with $\sigma_{\mathsf{S}} = \sigma_{\mathsf{C}}$, and also updates the password file as in the real execution.

In this scenario, we also need to ensure that the attacker is a behaving adversary. To this end, the simulator decrypts c_f to obtain (A', h', s') and computes $(h, a) = H(s', uid, pw)$. If $(A', h', s') \neq (g^a, h, s')$, then the attacker is not behaving and the simulator aborts.

Simulating Honest Client and Honest Server. Finally, the simulator also needs to emulate the protocol execution betwen two honest parties, showing the protocol's security against passive attackers. The challenge in this case is to create a transcript of the protocol between the two honest parties without knowing their passwords (while the environment might know the password but should still not be able to distinguish the simulation from a real execution). The only information the simulator receives is whether the key exchange succeeds.

Therefore, the simulator selects random protocol messages: e and τ as random values and c_s and c_f as encryptions of random values. We briefly argue why this is indistinguishable: (i) Replacing e with a random value is indistinguishable, as the attacker can decrypt e to different possible values $B_1, B_2, \ldots, B_n$, but the passive attacker does not learn which B was used by the server in the protocol. This was formalized in [20] as key-hiding property of the otkAKE, and is achieved by one-pass HMQV. (ii) PKE is CPA-secure, and hence changing c_s is indistinguishable. (iii) Changing τ is indistinguishable due to the PRF-property of F. Furthermore, the gapCDH assumption makes it so that the adversary cannot compute $\sigma := B^{x+d_{\mathsf{C}} \cdot a}$ even if it knows X, A, B, thus $\mathcal{A}$ has no information about the PRF key. (iv) Changing c_f is indistinguishable since AE is an authenticated encryption, and the attacker has no information about the underlying key K_{Enc} with the same argument as in (iii).

Game Hops. We now construct a sequence of games that incrementally transforms a real-world execution of the protocol UPAKE into a simulated execution using the $\mathcal{F}_{\mathsf{uaPAKE}}$ functionality such that no efficient environment $\mathcal{Z}$ can distinguish the two worlds. The simulator for the ideal world is given in Fig. 6 and Fig. 7. We write $\Pr[G_i]$ for the probability that the environment outputs 1 in game G_i. Throughout our game hops, we sometimes differentiate the simulation based on different corruption settings. We assume that the adversary relays the messages between the parties, and the honest client/honest server settings means that all messages arrive at the other party unaltered. To note this, the simulator keeps track of all exchanged messages.

Game G_0: The real protocol execution The environment $\mathcal{Z}$ runs the real protocol with adversary $\mathcal{A}$.

Game G_1: Restructure entities In this game, we shift all actions performed by the protocol parties to the simulator, which internally executes all parties. Additionally, we introduce an ideal functionality that merely forwards every received input to the simulator without any modification. Notably, this means that SIM still obtains the parties' passwords as input, which we will eliminate in subsequent games. To ensure that these changes remain indistinguishable to the environment, we incorporate dummy parties that relay inputs from $\mathcal{Z}$ to the functionality. Furthermore, SIM now simulates the random oracle and the ideal cipher, doing so via lazy sampling. Finally, the functionality is extended with dummy interfaces that allow the simulator to let any party produce any output chosen by the simulator. As these are only syntactical changes, we have

$$\Pr[G_1] = \Pr[G_0]$$

Game G_2: Abort upon RO collision In this game, the simulator aborts whenever it samples an answer h for random oracle H_i which was chosen before. Let q_{H_i} be the number of queries to H_i. This game hop is only noticeable if a collision occurs, and the probability for this is given by the birthday bound:

$$|\Pr[G_2] - \Pr[G_1]| \leq \frac{q_{H_1}^2 + q_{H_2}^2 + q_{H_3}^2}{2^\lambda}$$

Game G_3: Abort upon IC collision In this game, the simulator aborts whenever it samples an answer y for ideal cipher encryption oracle $\mathsf{IC}^*.\mathsf{Enc}$ which was chosen before. Let q_{ICE} be the number of queries to $\mathsf{IC}^*.\mathsf{Enc}$. This game hop is only noticeable if a collision occurs, and the probability for this is given by the birthday bound:

$$|\Pr[G_3] - \Pr[G_2]| \leq \frac{q_{ICE}^2}{2^\lambda}$$

Game G_4: Bookkeeping $\mathsf{IC}^*.\mathsf{Dec}$ trapdoors In this game, when answering fresh $\mathsf{IC}^*.\mathsf{Dec}$ queries (h, y), i.e. if there is no record $(IC, [h, *, *], y)$, SIM samples $b \xleftarrow{r} \mathbb{Z}_p$, outputs g^b as output to this query and keeps a record $(IC, [h, g^b, b], y)$. This is only a bookkeeping change, and thus not noticeable to the environment.

$$\Pr[G_4] = \Pr[G_3]$$

Game G_5: Honest client/ honest server: Change k_C to random When simulating the interaction between an honest client and an honest server, instead of using $k_\mathsf{C} := H_3(\mathsf{ssid}, \mathsf{C}, \mathsf{S}, X, \sigma_\mathsf{C})$, the simulator uses $k_\mathsf{C} \xleftarrow{r} \{0,1\}^\lambda$ to derive the client's protocol messages. To simulate the honest server, the simulator uses the same k_C to derive session key K_S if $pw = pw'$, and outputs $\perp$ otherwise.

Indistinguishability argument: The behaviour of the honest parties is identical, the only change is how k_C is constructed. The adversary may see protocol messages τ, c_f or the session keys derived from this k_C. But due to the random oracle property, the adversary can distinguish the two games only if he can query H_3 on exactly $(\mathsf{ssid}, \mathsf{C}, \mathsf{S}, X, \sigma_\mathsf{C})$. But this involves the adversary having to compute $\sigma_\mathsf{C} = B^{x+d_\mathsf{C}\cdot a}$. We show that the adversary can compute this value only with negligible probability if the CDH assumption holds in G. To show this, we construct the following reduction. Let (g, g^u, g^v) be a CDH challenge given to an adversary $\mathcal{B}$ who embeds the challenge as follows: it picks $h \xleftarrow{r} \{0,1\}^\lambda$, $a \xleftarrow{r} \mathbb{Z}_p$ and sets $(h, a) := H_1(s, uid, pw')$ where pw' is the client's password. Recall that the simulator in this game still knows the passwords of both parties. Then, SIM simulates the protocol with $X := g^u$, and $e = \mathsf{IC}^*.\mathsf{Enc}(h, g^v)$. $\mathcal{B}$ then picks one of the q_{H_3} queries to H_3 of the form $(\mathsf{ssid}, \mathsf{C}, \mathsf{S}, X, *)$ and outputs $\sigma_\mathsf{C} \cdot ((g^v)^{d_\mathsf{C}\cdot a})^{-1}$ in the CDH experiment. If the adversary $\mathcal{A}$ managed to query H_3 using $\sigma_\mathsf{C} = B^{x+d_\mathsf{C}\cdot a}$, then it holds that $\sigma_\mathsf{C} \cdot ((g^v)^{d_\mathsf{C}\cdot a})^{-1} = g^{uv}$, thus $\mathcal{B}$ wins the CDH experiment with probability $1/q_{H_3}$. Therefore, $\mathcal{A}$ cannot query H_3 on this value if the CDH assumption holds, and the games cannot be distinguished.

$$|\Pr[G_5] - \Pr[G_4]| \leq q_{H_3} \cdot \mathbf{Adv}^{\mathrm{CDH}}_{\mathcal{B}}(\lambda)$$

Game G_6: Honest client/honest server: Change $\tau, k_\mathsf{Enc}, K_\mathsf{C}$ to random In this game, the simulator picks random $\tau, k_\mathsf{Enc}, K_\mathsf{C}$ instead of $F(k_\mathsf{C}, \text{"auth-tag"})$, $F(k_\mathsf{C}, \text{"enc-key"})$, and $F(k_\mathsf{C}, \text{"session-key"})$. More specifically, the simulator picks a random function f and computes them as $f(\text{"auth-tag"})$, $f(\text{"enc-key"})$, $f(\text{"session-key"})$. If both parties run on the same password, i.e. if $pw = pw'$ (which the simulator can still check as he knows the passwords of both parties), SIM simulates server behavior with $k_\mathsf{Enc} = k_\mathsf{Enc}$ and $K_\mathsf{S} = K_\mathsf{C}$. If $pw \neq pw'$, the server outputs $\perp$.

Indistinguishability argument: The behaviour of the parties is identical. This game hop is distinguishable only if the adversary can distinguish the PRF outputs $F(k_\mathsf{C}, x)$ from $f(x)$ for a random key k_C. Note that $\mathcal{A}$ has no information on k_C due to the previous game hop. Thus, the indistinguishability follows from the pseudorandomness of the PRF. To see this, note that we can construct a PRF distinguisher from a distinguisher for these games. The PRF distinguisher $\mathcal{B}$ queries its oracle O with "auth-tag", "session-key", "enc-key" and simulates the protocol with the resulting values. Clearly, if the oracle returns $F(k, x)$, the simulation corresponds to G_5, and if the oracle returns $f(x)$ for a truly random function f, the simulation corresponds to G_6. Thus,

$$|\Pr[G_6] - \Pr[G_5]| \leq \mathbf{Adv}^{\mathrm{PRF}}_{\mathcal{B}}(\lambda)$$

Game G_7: Honest client/honest server: Change c_f to be random In this game, the simulator computes $c_f := \mathsf{AE}.\mathsf{Enc}(k_\mathsf{Enc}, (g, 0^\lambda, 0^\lambda))$ instead of $c_f := \mathsf{AE}.\mathsf{Enc}(k_\mathsf{Enc}, (A', h', s'))$. In case both parties run on the same password, the server, instead of decrypting c_f, updates the password file directly with (A', h', s').

Indistinguishability argument: The behaviour of the parties is identical. If an adversary can distinguish the two games, he would break CCA security of AE. To see this, consider a CCA adversary $\mathcal{B}$ who queries the challenge oracle with messages $m_0 := (A', h', s')$ and $m_1 := (g, 0^\lambda, 0^\lambda)$ and simulates the protocol by using the resulting challenge ciphertext c^* as c_f. Clearly, if c^* is an encryption of m_0, the simulation corresponds to game G_6, and if it is an encryption of m_1 it corresponds to game G_7. Note that K_Enc is a truly random value unknown to the adversary due to the last game hop. Thus, it holds that

$$|\Pr[G_7] - \Pr[G_6]| \leq \mathbf{Adv}^{\mathrm{CCA}}_{\mathcal{B}}(\lambda)$$

Game G_8: Bookkeeping compromised password file Whenever the adversary compromises a password file, the simulator records an entry $(\mathsf{sid}, uid, A, h, s, a)$ in T_{comp}. This is only a bookkeeping change, and thus

$$\Pr[G_8] = \Pr[G_7]$$

Game G_9: Honest client/adversarial server In this game, the simulator simulates the honest client as follows: When receiving (e, c_s), decrypt c_s. If e matches an $\mathsf{IC}^*.\mathsf{Enc}$ output on (h, B), check if h belongs to a compromised password file (list T_{comp}) or if there is a query to H_1 with $(h, a) \leftarrow H_1(s, uid, pw)$ for some pw. If this matches the client's password, i.e. if $pw = pw'$, continue simulation as in the protocol; in all other cases simulate the client's messages as in the honest client/honest server setting in G_7.

Indistinguishability argument: In case the password is correct, the behaviour is the same. If not, the indistinguishability of this game hop follows the same argument as the game hops for G_5 to G_7. Note that for the game hop of G_5 it is important that $\mathsf{IC}^*.\mathsf{Dec}(h, e)$ is undefined (i.e. not queried by the adversary before) such that the simulator can embed a CDH challenge into the protocol.

$$|\Pr[G_7] - \Pr[G_6]| \leq \mathbf{Adv}^{\mathrm{CCA}}_{\mathcal{B}}(\lambda) + \mathbf{Adv}^{\mathrm{PRF}}_{\mathcal{B}}(\lambda) + q_{H_3} \cdot \mathbf{Adv}^{\mathrm{CDH}}_{\mathcal{B}}(\lambda)$$

Game G_{10}: Adversarial client/honest server The server chooses a random IC ciphertext e instead of $e := \mathsf{IC}^*.\mathsf{Enc}(h, B)$. When

receiving X, τ, c_f, the simulator checks if there are queries to H_1, H_2, H_3 and $\mathsf{IC}^*.\mathsf{Dec}$ such that $\tau = F(k_\mathsf{C}, 1), k_\mathsf{C} = H_3(\mathsf{ssid}, \mathsf{C}, \mathsf{S}, X, \sigma_\mathsf{C}), B = \mathsf{IC}^*.\mathsf{Dec}(h, e), (h, a) = H_1(s, uidpw), d_\mathsf{C} = H_2(\mathsf{ssid}, \mathsf{C}, \mathsf{S}, X)$ and $\sigma_\mathsf{C} = B^{x+d_\mathsf{C}\cdot a}$. If this is the case, the simulator extracts the password pw from the H_1 query. If this password matches the server's password, i.e. if $pw = pw'$, the simulator follows the protocol correctly using the extracted values, otherwise output $\perp$.

Indistinguishability argument: Due to the simulator's ability to extract the adversary's input, this game hop is indistinguishable.

$$\Pr[G_{10}] = \Pr[G_9]$$

Game G_{11}: Simulation of Password File The complete password file (A, h, s) is only created upon the first query to StealPwdFile. Before, only the salt s is created, i.e. the password file consists of $(\perp, \perp, s)$. On the first query to StealPwdFile, if the correct password was queried to H_1 before on the current salt s, the simulator returns password file (g^a, h, s) where (a, h) is set to the output of the corresponding H_1 query. Otherwise, a random (a, h) is chosen and the simulator adds (A, h, s, a) to the list of compromised password files T_{comp}. If there is a query to $H_1(s, uid, pw)$ using s and the correct pw, the answer of this H_1 query is then set to (h, a) to ensure consistency. Further, while the password is not fully instantiated, (i.e. in the form $(\perp, \perp, s)$), in the simulation between an honest client and honest server, the client does not compute a new password file (A', h', s'). Instead, the server updates the password file to $(\perp, \perp, s')$ for a random s' in case of $pw = pw'$.

Indistinguishability argument: This change is indistinguishable, as the behaviour is the same. Furthermore, note that while the password file is not stolen, the protocol execution is independent of A, h.

$$\Pr[G_{11}] = \Pr[G_{10}]$$

Game G_{12}: Simulate honest server: abort if c_f changes If the server receives (X, τ, c_f) s.t. (X, τ) was created by a simulated honest client but c_f was changed, simulate that the server outputs abort.

Indistinguishability argument: This game hop is only distinguishable if an adversary is able to pass off a valid ciphertext c_f for a key unbeknownst to him. We can reduce the probability of this happening to the ciphertext integrity. Let $\mathcal{B}$ be an adversary in the INT-CTXT experiment. $\mathcal{B}$ queries its encryption oracle to receive c_f. Then, when $\mathcal{B}$ acts as the server and receives a modified c'_f, he outputs c'_f as a forgery in the ciphertext integrity experiment. $\mathcal{B}$ is clearly in PPT, and wins the INT-CTXT experiment. Thus,

$$|\Pr[G_{12}] - \Pr[G_{11}]| \leq \mathbf{Adv}^{\text{INT-CTXT}}_{\mathcal{B}}(\lambda)$$

Game G_{13}: Introduce $\mathcal{F}$, password check is handled by $\mathcal{F}$, simulator does not receive private inputs. In this game, we add an ITI $\mathcal{F}$ external to the simulator. $\mathcal{F}$ has all interfaces of $\mathcal{F}_{\mathsf{uaPAKE}}$ and manages all password checks in the protocol. Furthermore, the ideal functionality provides the session keys to the parties. We thus remove the simulator's access to the parties' private inputs. The changes are only internal, and thus

$$\Pr[G_{13}] = \Pr[G_{12}]$$

Finally, G_{13} is identical to the ideal world execution of the protocol which concludes the proof.

B.1 Proof Sketch of Thm 5.2

We provide a proof sketch for Thm. 5.2. The proof follows the proof of Theorem 5.1. The main difference is the incorporation of the zero-knowledge proof, and we highlight how the proof needs to be adjusted for that. Therefore, consider the three scenarios where the simulator needs to handle the NIZK.

Honest Client and Honest Server: In this scenario, the NIZK is encrypted as part of c_f, so the adversary does not get information about it.

Honest Client and Adversarial Server: In this case, the simulator must send a proof π to the server to verify that the password used in the old and updated password files is the same. Note that in the proof of UPAKE, the simulator first extracts the adversarial server's password and then constructs the updated password file accordingly. If the passwords match, this file is identical to the one in the real world. The simulator then generates a simulated NIZK proof π. Although the simulator knows the client's password and could produce a real proof using the password as a witness, generating a simulated proof ensures that the adversary learns nothing about the password. This approach prevents the server or any adversary impersonating the server with a stolen password file from learning the plaintext password. The security of this change follows from the zero-knowledge property of the NIZK.

Adversarial Client and Honest Server: In this case, the simulator must additionally verify the NIZK proof sent by the client and update the password file only if the proof is valid. The soundness property of the NIZK guarantees that an adversarial client can produce a valid proof π only if they know the correct password embedded in both the old and new password file. Importantly, the simulator does not need to rewind the prover to extract the password – an action that would violate UC security – as it can already extract the password from the protocol values (X, τ). Instead, the simulator relies solely on the soundness property of the NIZK to ensure that a valid proof implies the client's knowledge of the correct password.

C Comparing Proofs of SRP-6a and OKAPE-HMQV

We provide a more detailed comparison of the proofs of SRP-6a [18] and OKAPE-HMQV [20]. Due to the structural similarities of the two protocols, their proofs are also similar. However, since the server's public key B is masked using addition in the finite field instead of an ideal cipher encryption, the proof of SRP-6a introduces two significant changes compared to the proof of OKAPE-HQMV:

Limit Server to One Password Guess. Note that instead of IC-encrypting B as in OKAPE, the server sends $e := B + k \cdot A$ to the client where k is derived from public values. For security to hold, e must bind the server to one password guess. In OKAPE, this is inherently

Our Notation	Notation in [18]
Client's HMQV long-term key-pair (a, A)	(x, v)
Client's ephemeral key-pair (x, X)	(a, A)
Server's masked HMQV public key $e := k \cdot A + g^b$	$B := k \cdot v + g^b$
Scrambling parameter d_C	u
Shared HMQV key σ_C/σ_S	T_C/T_S
Client's key confirmation message τ_C/τ_S	M_1^C/M_1^S
Server's key confirmation message γ_C/γ_S	M_2^C/M_2^S

Table 3: Mapping of protocol values from our notation of the SRP-6a protocol to the protocol values in [18].

provided by the ideal cipher, since the IC-ciphertext corresponds with high probability to only one query (h, B) to IC*.Enc which commits an attacker to one value h derived from $H_1(s, uid, pw)$. In SRP-6a, however, security requires that the server can only compute σ_S for a unique password guess $A = g^{H_1(s,uid,pw^*)}$, i.e., for only one valid $g^b = e - k \cdot A$ the server should be able to compute $\sigma_S := (X \cdot A^{d_C})^b$. Dayanikli and Lehmann [18] show that this holds under a novel finite field assumption – the additive Strong Diffie–Hellman (aSDH) assumption, which assumes that it is computationally hard to find two CDH tuples that are additively related by a random offset Δ.

Angel-Based UC: Extract Client's Password. In OKAPE-HMQV, when simulating the server, the simulator can embed a trapdoor into IC*.Dec queries. Specifically, for each fresh query (h, e) to IC*.Dec, the simulator responds with g^b and records the corresponding exponent b. This setup allows the simulator – given the client's ephemeral key X and candidate password-derived values $(h, a) := H_1(s, uid, pw)$ – to test whether the client's key τ_C was computed as $(X \cdot (g^a)^{d_C})^b$ for some d_C, thereby enabling password extraction.

In contrast, in SRP-6a, g^b is constructed from $e - k \cdot A$ where $A := g^{H_1(s,uid,pw)}$. In this case, the simulator cannot embed a trapdoor in the same way as in OKAPE-HMQV. Instead, the proof in [18] employs a DDH oracle to check whether the client's key confirmation value τ_C corresponds to a value σ_C computed as $\sigma_C = \mathsf{cdh}_g(g^a \cdot X^{d_C}, e - k \cdot g^a)$ for some $a = H_1(s, uid, pw)$. The simulator then iterates over all password candidates and uses the DDH oracle to identify the correct password. Although this DDH oracle is not efficiently instantiable, one can resort to it as a super-polynomial helper, and its use is permitted within the angel-based UC framework if it is made accessible to both the simulator and the adversary.

D Proof Sketch for UPAKE-SRP

We now give the proof sketch for the security of UPAKE-SRP, which shows the following statement.

Theorem D.1. *[18] Let $\mathbb{F}_p$ be a finite field where the Gap CDH and Gap aSDH assumptions are hard, and let F be a secure PRF,* PKE *a CPA-secure encryption scheme, and* AE *an authenticated encryption scheme. Then, UPAKE-SRP is a secure updatable aPAKE protocol in the angel-based UC framework where all parties have access to a DDH oracle, in the presence of static corruption in the random oracle model.*

Proof Sketch. The security intuition is the same as for our blueprint: updating the salt invalidates prior precomputed values by an attacker, the mutual authentication of SRP-6a ensures that the password file is only sent to the correct server (thus avoiding phishing attacks), and the authenticated encryption of AE ensures that c_f leaks no information about the password file, and that the server receives the password file unaltered. We sketch how the simulator simulates c_f. All other protocol values are simulated as in the proof of SRP-6a [18], and the simulation of c_s is as described in the transform of [19]

Simulating c_f. Note that in UPAKE-SRP, the ciphertext c_f is only sent if the server is explicitly authenticated, in contrast to UPAKE where server authentication was implicit when sending c_f. In the proof there are two cases where we need to simulate c_f for an authenticated server:

Honest Client with an Adversarial Server An adversary may authenticate as a server by guessing the user's password correctly, or impersonating the server using a stolen password file. In this case, the adversary expects to receive an updated password file which is built from the same password as the existing password file. The simulator can create this updated password file by extracting the server's password using techniques from the SRP-6a proof and thus compute a valid password file (s', v') and session key k_C using this password.

Honest Client and Honest Server In this case, both parties must share the same password. The simulator in this case generates c_f as an encryption of a random value of appropriate length. Due to the security of SRP-6a against passive adversaries, the adversary obtains no information about the session key $k_C = k_S$, and therefore learns nothing about the derived encryption key k_{Enc}. The IND-EAV security of the authenticated encryption scheme ensures that the simulated ciphertext c_f is computationally indistinguishable from a real ciphertext. If the ciphertext arrives with the honest sever unaltered, the simulator updates the salt in the password file with a random salt s', otherwise the simulator outputs $\perp$. Due to the ciphertext integrity of AE, this is indistinguishable.