(Strong) aPAKE Revisited: Capturing Multi-User Security and Salting

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

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

***Abstract*—Asymmetric Password-Authenticated Key Exchange (aPAKE) protocols, particularly Strong aPAKE (saPAKE) have enjoyed significant attention, both from academia and industry, with the well-known OPAQUE protocol currently undergoing standardization. In (s)aPAKE, a client and a server collaboratively establish a high-entropy key, relying on a previously exchanged password for authentication. A main feature is its resilience against offline and precomputation (for saPAKE) attacks. OPAQUE, as well as most other aPAKE protocols, have been designed and analyzed in a single-user setting, i.e., modelling that only a single user interacts with the server. By the composition framework of UC, security for the actual multi-user setting is then conjectured. As any real-world (s)aPAKE instantiation will need to cater multiple users, this introduces a dangerous gap in which developers are tasked to extend the single-user protocol securely and in a UC-compliant manner.**

In this work, we extend the (s)aPAKE definition to directly model the multi-user setting, and explicitly capture the impact that a server compromise has across user accounts. We show that the currently standardized multi-user version of OPAQUE might not provide the expected security, as it is insecure against offline attacks as soon as the file for one user in the system is compromised. This is due to using shared state among different users, which violates the UC composition framework. However, we show that another change introduced in the standardization draft which also involves a shared state does not compromise security. When extending the aPAKE security in the multi-client setting, we notice that the widely used security definition captures significantly weaker security guarantees than what is offered by many protocols. Essentially, the aPAKE definition assumes that the server stores *unsalted* password-hashes, whereas several protocols explicitly use a salt to protect against precomputation attacks. We therefore propose a definitional framework that captures different salting approaches – thus showing that the security gap between aPAKE and saPAKE can be smaller than expected.

1. Introduction

Asymmetric Password-Authenticated Key Exchange (aPAKE) allows a server and user to establish a cryptographic session key based on the user's knowledge of a low-entropy password. The asymmetry refers to the fact that the server does not need to store the password in plain, but creates a *password file* upon the user's registration, which is then used for re-authentication in the actual key exchange phase. The core feature of aPAKE protocols is their resilience to offline attacks, i.e., none of the values transmitted in the protocol allows to recover the password via brute-force guessing attempts. This feature must hold as long as the server is not compromised. Several provably-secure aPAKE protocols exist [5], [11], [13], [15], [17], [18], [23], [28], [31], with some being widely deployed to secure applications such as Telegram [33], Apple Homekit [2], ProtonMail [6], ProtonPass [38] or 1Password [14].

In 2018, Jarecki et al. [23] introduced the concept of strong aPAKE (saPAKE), which improves upon aPAKE by requiring that no pre-computation attacks on the users's password can be performed before the server gets compromised. OPAQUE [23] was the first protocol to satisfy this stronger notion, and has inspired a line of work providing protocols based on alternative building blocks and more efficient constructions [5], [28]. The OPAQUE protocol has already enjoyed significant real-world attention: it was chosen as the (s)aPAKE winner in the standardization initiative led by the IRTF [30] and is used to protect encrypted backups in the WhatsApp messenger [10].

Single-User Security Models. In terms of security, the Universal Composability (UC) [7] framework has emerged as the de-facto gold standard for modelling and analyzing (s)aPAKE protocols. The UC framework is particularly well-suited for password-based protocols because it does not make any assumptions about password distributions, models real-world behaviour such as password re-use and guarantees secure composition with arbitrary other protocols – or itself.

In fact, the self-composition aspect of UC has been used to study a simplified setting of (s)aPAKE: the *single*-user variant, where only a *single* user can register with the server and establish keys. This allows to focus the model, protocol design and analysis to this simpler setting, and rely on the composition theorem [9] when the protocol is used for multiple users. So far, UC-secure (s)aPAKE protocols have predominantly been designed and analyzed for such a single-user setting [5], [13], [17], [18], [22], [23], [24], [28]. Exceptions are the recent works of [11], [13], [17].

Multi-User Reality. In practice, (s)aPAKE is typically deployed in a setting where a single server serves up to millions of users, all running the same protocol for password-based authentication and key-exchange. Technically, the self-composition requires the server to run several and fully independent instances of the single-user protocol [20] . In particular, the server must not re-use any shared state, such as secret keys across the instances if it

wants to benefit from the composability guarantees and provide the same security as the provably secure single-user version.

This requires application developers to understand the limitations of the single-user (s)aPAKE protocols and be able to extend the protocol in a way that is compliant with the UC framework. Thus, the focus on the single-user setting leaves a dangerous gap between the provably-secure protocol variant and the version that would actually be needed for real-world deployment.

That this ambiguity can lead to debatable design choices can be seen in the practical adaptation of OPAQUE: the multi-user variant of OPAQUE that is currently in the process of standardization, does *not* adhere to such a strict state separation as it recommends "The $\text{oprf}_{\text{seed}}$ value *[which is a crucial server-side value in the protocol]* SHOULD be used for all clients" [4]. In fact, there has been a discussion in the irtf-cfrg mailing list [26] whether it is okay to use the same seed for multiple clients, with the draft authors still stating they believe in the security of this design choice.

1.1. Multi-User Security for strong aPAKE

We strongly believe that the formal security model should be as close to the practical setting as possible. To this end, we propose the first security model for strong aPAKE that explicitly captures the multi-user setting, based on similar efforts in recent aPAKE works [11], [13], [17]. In the case of strong aPAKE, this requires dedicated care to correctly model the impact that a server-compromise should have across several user accounts. In reality, when servers get compromised and leak password files, the breach often contains data of a subset of the registered users only [1], [12], [16], [34], [36], and users not included in the breach are not impacted. Clearly, strong aPAKE should provide the same guarantee and our model captures a fine-grained corruption handling, requiring that a compromise of a server's file on a user uid must have no impact on any other $\mathsf{uid}' \neq \mathsf{uid}$.

We then study the standardization draft of the OPAQUE protocol in the multi-user setting, and show that the recommended sharing of the OPRF seed among multiple users is insecure when the seed is accessible at every login (as currently demanded by the standard) [4]. We describe an attack on the draft version that exploits these cross-client dependencies. Roughly, when the password file of a user's uid is compromised, an attacker learns the global seed and can derive the OPRF key of all users. The key alone cannot be used to breach security of any other user, but only a single benign interaction with the server is needed: when the adversary makes a login-attempt (on a random password) for another user uid', it can use the server's response and OPRF key to offline attack the password for uid' too. This is a substantial gap compared to the expected saPAKE security, which limits offline attacks to the user whose password file has been compromised.

Interestingly, we show that another and similar modification related to the multi-user setup in the standardization draft does not undermine security: the draft permits the server to utilize the same long-term key pair for the authenticated key exchange building block for all users. We show that re-using this key pair indeed results in a secure protocol.

1.2. Stronger Security for aPAKE

When turning our attention to standard aPAKE, i.e., the weaker variant that does not provide security against precomputation attacks, we notice that several real-world protocols actually do offer better resistance against these attacks than what is required from the security model. The better resistance is only apparent when directly studying and formalizing the multi-user setting, again demonstrating the need to model the real-world as closely as possible.

According to the security model [15], the adversary can precompute a list of possible passwords – which usually translates to precomputing a list of hashes for guessed passwords – at any time. As soon as the user's password file is compromised, the adversary can instantly determine if a precomputed password matches the file. Thus, in case of an eventual server compromise, the aPAKE guarantees are interestingly significantly weaker than that of conventional password-based authentication, where the server typically stores a salted hash only (along with the salt) to thwart such precomputation attacks [29].

In fact, a similar salting approach has already been used in some UC-secure aPAKE protocols such as KHAPE [17], AuCPace [18] or SRP-6a [37], as well, where the password file uses a user-specific salt value in the hash computation. In contrast to strong aPAKE, this salt does not remain hidden but is sent in clear to any user trying to authenticate as uid. When studying the single-user setting, this extra salt might not appear to add much extra security – the adversary can learn the salt with a single login query.

When turning the model to reflect the multi-user setting, where a single server might cater up to million users, this makes a difference though. In order to start a bulk precomputation attack targeting all users, the adversary must now intercept or even start login sessions for all millions of users. This is still significantly easier than having to compromise the user's file on the server (as demanded by strong aPAKE), but might thwart some attacker or at least make their attacks more costly and easier to detect.

Thus, for a more realistic study and comparison of aPAKE protocols within the aPAKE family – as well as comparing it to their stronger saPAKE sibling – we formalize these salting approaches as part of the existing UC definitions for (s)aPAKE. More precisely, we formalize a framework of salting levels spanning from aPAKE to saPAKE, and propose newer definitions that are in between both.

Level 1: No Salt (aPAKE): These aPAKEs have no user-specific salt, and correspond to the established security definition of aPAKE.

Level 2: Passively-Revealed Salt: These protocols incorporate a salt, which is transmitted to the user during the login session. The user and any passive eavesdropper can learn the user-specific salt and start precomputation attacks on that user's password after that.

Level 3: Actively-Revealed Salt: These aPAKEs also employ a salt but ensure it is protected from eaves-

droppers during transmission to the user. Again, learning the salt will allow precomputation attacks – but only on that user, and this time requiring an active login session from the adversary for each user account it wants to attack via precomputation.

Level 4: Private Salt (strong aPAKE): These are classified as strong aPAKEs and represent the highest level of security in the aPAKE framework.

While Level 1 and 4 correspond to aPAKE and strong aPAKE respectively, we propose modifications to the standard aPAKE definition to yield Level 2 and 3 security. We then categorize existing aPAKE protocols into these security levels, showing that several protocols achieve stronger security than what was advertised so far. In fact, we also show that there are easy transformations to lift any aPAKE that is secure on Level 1 to Level 2 security, and again lift Level 2 security to Level 3. Overall, while strong aPAKE is still the strongest of all variants, our work reveals that the difference between aPAKE and strong aPAKE – in the multi-user setting – can be smaller than originally thought.

2. Multi-User Security for saPAKE

In this section, we describe the existing single-user model for saPAKE which has been used in various works [5], [23], [28], and show how it can be extended to explicitly model the real-world setting where multiple users interact with the same server instance. Our new model clearly articulates the security users can expect in such a setting. As a core goal of saPAKE is to provide security against precomputation attacks before the server gets compromised, we put dedicated care in modeling this feature in the strongest (and most realistic) setting possible.

Universal Composability. We express and study security in the Universal Composability (UC) model [7], which has evolved as the gold standard for (s)aPAKE protocols and has been used in all recent works. In the UC model, the desired properties of a protocol are represented as an ideal functionality $\mathcal{F}$ and security is achieved if the real-world protocol π can be indistinguishably mimicked by a simulator which only interacts with the ideal functionality $\mathcal{F}$. In this case, π is said to UC-realize functionality $\mathcal{F}$. UC security is particularly suited for password-based protocols, as it does not require any idealized assumptions on the distribution or handling of passwords. One of the main benefits of UC security is that UC-secure protocols remain secure under arbitrary composition of protocols and under concurrent executions – but only when the composition is done in a manner that is compliant with the framework.

2.1. Single-User saPAKE Model

The ideal functionality $\mathcal{F}_{\mathsf{saPAKE}}$ proposed by Jarecki et al. [23] models the strong asymmetric password-authenticated key exchange between two parties – a client C and a server S. It allows the server to register a client with a password pw using the StorePwdFile interface. The original functionality does not include usernames, but implicitly assumes that the global session identifier sid is a combination of the user name uid and the identity of the server. The functionality internally stores a password file $\langle \mathsf{file}, \mathsf{S}, pw \rangle$ and from then on allows the client to engage in a key generation session with the server.

In order to perform a key exchange, the client uses the CltSession interface of $\mathcal{F}_{\mathsf{saPAKE}}$ where she inputs the password pw' she wants to use for the session. The client also has to use the same user-specific sid as input to the CltSession interface as in the registration, and do so consistently for every key exchange. The server uses the SvrSession interface where the functionality will use the internal file for the password-based authentication. 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 (s)aPAKE functionality $\mathcal{F}_{\mathsf{saPAKE}}$ also models several inevitable attacks. The online guessing attack is modeled through the TestPwd interface which may be accessed once per session. The attacker further has the ability to compromise a server which is modeled through the StealPwdFile interface and which allows the attacker to learn a user's password file. Stealing the password file then allows for impersonation and offline attacks. In the impersonation attack (modeled by query Impersonate), the attacker impersonates the server to the client using the stolen password file, and in the offline attack (modeled by OfflineTestPwd), the attacker can test passwords offline against the stolen password file. The advantage of saPAKE over regular aPAKE, is that queries to OfflineTestPwd are only possible after server compromise, meaning that no precomputation attacks on the user is possible before the adversary compromises the file held by the server.

2.2. Multi-User Security From UC Composition

The original functionality for $\mathcal{F}_{\mathsf{saPAKE}}$ is a single-user functionality, meaning it only models the interaction between a single user and a single server. However, in practical real-world applications, aPAKE schemes are typically deployed in multi-user environments, where up to millions of users register with the same server.

Composition of Single-User Functionalities. Conveniently, the modularity of the UC framework [7], allows to extend the UC security guarantee to the multi-user setting. Therefore, one can construct a multi-user saPAKE protocol π_{MU} which runs the single-user $\mathcal{F}_{\mathsf{saPAKE}}$ between each user-server pair as subprotocols. Using the universal composition theorem of the UC framework [7], all the ideal $\mathcal{F}_{\mathsf{saPAKE}}$ protocols can be subsequently replaced with their realizations (i.e. saPAKE protocol π_{SU} which UC-realizes the single-user functionality $\mathcal{F}_{\mathsf{saPAKE}}$), from which the UC-security of the multi-user protocol follows.

As a caveat though, in the classical UC composition theorem [7], constructing a multi-user UC-secure protocol in this way requires all subprotocols to be independent of each other. That is, the composed protocol instances cannot share an internal state and all internal random choices in the different subprotocols have to be independent. In the self-composition needed for real-world (s)aPAKE, the same server will run multiple instances with every user and thus must ensure that no long-term keys or shared

Password Registration

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

Stealing Password Data

– On $(\mathsf{StealPwdFile}, \mathsf{sid}, \mathsf{S}, \mathsf{uid})$ from $\mathcal{A}$, if there is no record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$, return "no password file". Otherwise mark this record stolen, and if there is a record $\langle \mathsf{offline}, \mathsf{S}, \mathsf{uid}, pw \rangle$ then send pw to $\mathcal{A}$.

– On $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{S}, \mathsf{uid}, pw^*)$ from $\mathcal{A}$, do:

  – If $\exists$ record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked stolen, do the following: If $pw^* = pw$ return "correct guess" to $\mathcal{A}$, else return "wrong guess".

  – Else, record $\langle \mathsf{offline}, \mathsf{S}, \mathsf{uid}, pw^* \rangle$.

Password Authentication

– On $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \mathsf{uid}, pw')$ from C, if there is no record $\langle \mathsf{ssid}, \mathsf{C}, \dots \rangle$ then record $\langle \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \mathsf{uid}, pw', \mathtt{cl} \rangle$ marked fresh and send $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \mathsf{uid})$ to $\mathcal{A}$.

– On $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{uid})$ from S, if there is no record $\langle \mathsf{ssid}, \mathsf{S}, \dots \rangle$ then retrieve record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$, and if it exists then create record $\langle \mathsf{ssid}, \mathsf{S}, \mathsf{C}, \mathsf{uid}, pw, \mathtt{sr} \rangle$ marked fresh and send $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \mathsf{C}, \mathsf{uid})$ to $\mathcal{A}$.

Active Session Attacks

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

Key Generation and Authentication

– On $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, K^*)$ from $\mathcal{A}$, if there is a record $\mathsf{rec} = \langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \mathsf{uid}, pw, \mathsf{role} \rangle$ not marked completed, then do:

  – If rec is compromised set $K \leftarrow K^*$;

  – Else if $\mathsf{role} = \mathtt{cl}$, rec is fresh, there is a record $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \mathsf{uid}, pw, \mathtt{sr} \rangle$ s.t. $\mathcal{F}_{\mathsf{aPAKE}}$ sent $(\mathsf{sid}, \mathsf{ssid}, K')$ to P' while that record was marked fresh, set $K \leftarrow K'$;

  – Else if $\mathsf{role} = \mathtt{sr}$, rec is fresh, there is a record $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \mathsf{uid}, pw, \mathtt{cl} \rangle$ 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 $(\mathsf{sid}, \mathsf{ssid}, \perp)$ to P, otherwise provide private delayed output $(\mathsf{sid}, \mathsf{ssid}, K)$ to P.

Figure 1: The multi-user ideal functionalities $\mathcal{F}_{\mathsf{saPAKE}}$ and $\mathcal{F}_{\mathsf{aPAKE}}$ (including highlighted text) with explicit authentication where the server receives the key first. The functionality is tied to a server S, and we assume that S is encoded in sid, e.g. as $\mathsf{sid} = (\mathsf{S}, \mathsf{sid}')$ for a unique sid'.

state is used in different subprotocols. Not adhering to this requirement will void the UC security guarantees provided by the classical UC composition theorem.

UC with Joint State. To enhance the practical usability of the composability property, Canetti and Rabin [9] introduced the notion of Universal Composability with Joint State (JUC), which allows parties to have a joint state across different protocols. In the JUC framework, the composition of protocols which share a subprotocol can be shown to remain UC-secure. In fact, several aPAKE protocols have been proven in the JUC framework [18], [20] for a shared random oracle. However, the JUC framework is mostly suited when the joint subprotocols are publicly available functions without secret keys such as a random oracle, where a secure instantiation mostly has to handle domain separation only. In keyed subprotocols which also have to handle key compromise, the JUC framework is not the most suitable choice.

Server Compromise in the Composed Protocol. In a multi-user saPAKE protocol which is composed from single-user saPAKE protocols using one of the methods mentioned above, the server compromise is modeled via $\mathsf{StealPwdFile}$ in the single-user functionality $\mathcal{F}_{\mathsf{saPAKE}}$. The UC security guarantees that this compromise exclusively impacts the user who resides within this functionality. The security of all other protocol instances is independent of the affected protocol instance, ensuring that compromising the file of a user uid (registered as sid) does not impact the security of users $\mathsf{uid}' \neq \mathsf{uid}$ in this model.

2.3. Our Multi-User saPAKE Model

Our goal is to model strong aPAKE directly in the multi-user setting it will be used in, and guarantee the same security properties that the self-composition of the single-user protocols would yield. The advantage of the multi-user variant is that any protocol is explicitly designed for the real-world setting. In contrast to the self-composition approach, the multi-user variant can re-use key material and shared state on the server side, as long as it does not harm the strong security guarantees required from the protocol. In particular, compromising the password file of a user uid must have no impact on any other $\mathsf{uid}' \neq \mathsf{uid}$,

Multi-User Functionality. The changes to transform the single-user functionality to the multi-user setting are rather subtle. In fact, for regular aPAKE, newer works have turned to the multi-user setting already [11], [13], [17], and we adopt their handling to the strong aPAKE model. The detailed functionality for the multi-user setting is given in Fig. 1 and we explain the main changes and impact on the server compromise handling below.

In the functionality, the global session identifier sid now identifies a multi-user functionality, i.e., is no longer assumed to encode the user name. Instead, a dedicated

user-name uid has to be provided to the StorePwdFile interface along the password pw. This registration interface can now be called multiple times, and the functionality maintains individual password files $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ for all registered users. The uid then needs to be provided by the parties as an additional input when initiating a new session through CltSession or SvrSession queries. Furthermore, the Impersonate query requires the attacker to indicate which uid should be targeted. The TestPwd and NewKey interfaces do not require the uid, as they are only called on existing sessions which are identified by the combination of (sub)-session identifiers $(\mathsf{sid}, \mathsf{ssid})$ and already linked to the username uid.

Modeling Server Compromise. Compromising a server in the single-user setting essentially means compromising the single user's password file. When transitioning from the single-user to the multi-user setting, there are different options to model server compromise. It could either be modeled as a full server compromise, which compromises the files of all users, or individual server compromise which only targets the files of individual users.

We chose to model the compromise of individual password files as it gives us a more flexible and fine-grained security model, which caters for possible cross-user dependencies. After compromising the file of uid, we still have security for all users $\mathsf{uid}' \neq \mathsf{uid}$, and the security guarantee also extends to new users who want to register with the server after the password file of some user uid is compromised, essentially modeling *post-compromise security*.

Individual or partial server compromise is a realistic scenario, as servers usually do not handle all their data simultaneously, and partial leaks can occur. Consider, for instance, the login process, where the server retrieves the password file of a single user. If there is an issue with the server, such as a software bug or malware, the password file of that user could be inadvertently exposed, while the password files of other users may remain secret.

In the functionality, the individual file compromise is represented by the StealPwdFile interface which takes as input a username uid and only compromises the file of uid. The ideal functionality marks the internal password file $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ as stolen. Compromising the file of user uid should only have contained impact. It should only enable the inevitable offline password tests on the compromised user uid and it should not downgrade the security guarantees for any other user $\mathsf{uid}' \neq \mathsf{uid}$. The functionality therefore allows OfflineTestPwd queries only for users whose password file has been marked stolen. The ideal functionality for the multi-user setting is given in Fig. 1.

3. Multi-User (In)security of Draft-OPAQUE

OPAQUE [23], introduced in 2018, is a strong aPAKE protocol, that was chosen as the winner of the IETF selection process for aPAKE in 2020 and is presently undergoing standardization by the IRTF [4]. In this section, we first give an overview of the original OPAQUE protocol by Jarecki et al. [23]. Following this, we highlight some changes introduced in the current draft version 12 from the IRTF. We then show that one of these changes – using a single-seed OPRF for multiple users – makes the draft version insecure. We isolate this change for our analysis in a version we call *single-seed OPAQUE* (ssOPAQUE). We detail an attack for this simplified version, and show how the attack can be mitigated.

3.1. OPAQUE

Jarecki et al. [23] describe two ways of building a strong aPAKE from an Oblivious Pseudorandom Function (ORPF). The OPAQUE compiler[1] combines the OPRF with an authenticated key exchange (AKE). We first describe the building blocks used in OPAQUE. In contrast to the original description [23], we capture the OPRF building block through an algorithmic representation instead of an ideal functionality. This makes it easier to explain the change made in the standardization draft, and the attack thereon.

OPRF. An Oblivious Pseudorandom Function (OPRF) is a protocol run between a client and a server to jointly compute the pseudorandom function $F(k, x)$ where the client provides input x and the server provides key k. A secure OPRF protocol ensures that at the end of the protocol, the client learns $F(k, x)$ while the server learns nothing. More formally, we define the OPRF as a tuple of algorithms $(\mathsf{KGen}, \mathsf{Blind}, \mathsf{BlindEval}, \mathsf{Eval}, \mathsf{Finalize})$ where

- $\mathsf{KGen}(1^\lambda) \to k$ creates the PRF key k.
- $\mathsf{Blind}(x) \to (\overline{x}, r)$ is run by the client to blind input x. It returns the blinded input $\overline{x}$ and randomness r.
- $\mathsf{BlindEval}(k, \overline{x}) \to \overline{y}$ is run by the server to blindly evaluate the OPRF on $\overline{x}$ using key k. This returns a blinded evaluation $\overline{y}$.
- $\mathsf{Finalize}(x, r, \overline{y}) \to y$ is run by the client to unblind the blinded evaluation $\overline{y}$ to y using randomness r.
- $\mathsf{Eval}(k, x) \to y$ directly generates output $y = F(k, x)$ from input x and key k.

In the modified OPAQUE version ssOPAQUE, the KGen algorithm runs deterministic when given randomness rnd. In this case, we write $k := \mathsf{KGen}(1^\lambda; rnd)$.

AKE. The second building block used in OPAQUE is an authenticated key exchange (AKE) protocol which allows two parties $\mathcal{P}$ and $\mathcal{P}'$ to exchange a session key over an insecure communication channel while also ensuring the authenticity of each other's identities based on their long-term public keys. OPAQUE additionally requires the AKE to achieve key compromise impersonation resistance (KCI) which ensures that even if an attacker obtains or compromises the long-term secret key of party $\mathcal{P}$, he should still be unable to impersonate party $\mathcal{P}'$ to $\mathcal{P}$. For simplicity, we model the AKE as a protocol between C and S in which the parties first create ephemeral key-pairs and exchange the ephemeral public keys. These ephemeral keys are combined with the long-term keys (where each party uses its own secret key part and the counterparty's public key part) to derive a key. The key is derived using a key exchange formula similar as in [23]. The AKE protocol further encompasses key confirmation messages to verify the authenticity of the counterparty.

1. OPAQUE is actually described as a concrete instantiation of the compiler using 2HashDH and HMQV. We refer to OPAQUE as the generic compiler akin to the current draft specification.

The construction of the AKE is given in Fig. 5 in the Appendix and uses the following algorithms:

- $\mathsf{KGen}_{\mathsf{lt}}(1^\lambda) \to (sk, pk)$: generates a party's long-term key-pair.
- $\mathsf{KGen}_{\mathsf{eph}}(1^\lambda) \to (x, X)$: generates a party's ephemeral key-pair.
- $\mathsf{KE}(sk_P, pk_{P'}, x_P, X_{P'}) \to k$: key exchange formula that computes a key k for party $\mathcal{P}$ based on its long-term and ephemeral secret key, and the counterparty $\mathcal{P}'$'s long-term and ephemeral public key.

The AKE further uses a pseudorandom function $F : \{0,1\}^\lambda \times \mathcal{X} \to \mathcal{Y}$ to perform explicit key confirmation. The AKE protocols HMQV [25] and 3DH [35] with additional key confirmation are an instantiation of this 3-flow protocol.

Authenticated Encryption and PRF. OPAQUE further uses an authenticated encryption scheme and a pseudorandom function. The authenticated encryption scheme consists of algorithms $(\mathsf{KGen}, \mathsf{AuthEnc}, \mathsf{AuthDec})$ with

- $\mathsf{KGen}(1^\lambda) \to k$: creates key k.
- $\mathsf{AuthEnc}(k, m) \to c$: encrypts message m using key k to obtain ciphertext c.
- $\mathsf{AuthDec}(k, c) \to m/\perp$: decrypts ciphertext c using key k. Returns either message m or, failure message $\perp$ in case decryption fails.

OPAQUE requires that the authenticated encryption scheme fulfills the standard notions of authenticated encryption – CCA-security and ciphertext integrity. It is further required to be *random-key robust*, i.e. a ciphertext should not decrypt successfully under two different randomly generated keys. This condition ensures that a ciphertext will only decrypt successfully under the key it was created with. Looking ahead, the random-key robustness will be important to our attack, as it provides the attacker with a way to test passwords correctly.

The OPAQUE Protocol. The core idea of the OPAQUE compiler is to employ an AKE protocol to establish a shared key, where the client's long-term keys are stored on the server in encrypted form (the so-called envelope). In the login phase, the server sends the envelope to the client, and the client can only decrypt the envelope if she uses the correct password. The encryption key is the output of the OPRF on the server's secret key and the users' password, providing the desired offline attack resistance. If the client's decryption is successful, she retrieves her AKE keys, and both parties can run the AKE protocol.

In more detail, the OPAQUE protocol consists of two parts – a registration and a login phase. During registration, the server samples AKE key-pairs for server $(sk_\mathsf{S}, pk_\mathsf{S})$ and client $(sk_\mathsf{C}, pk_\mathsf{C})$ as well as an OPRF key k_O. It encrypts $(sk_\mathsf{C}, pk_\mathsf{C}, pk_\mathsf{S})$ under the OPRF evaluation $rw = F(k, pw)$ to create the envelope c where pw is the client's password. The file then consists of $\mathsf{file}[\mathsf{uid}] = (k_O, sk_\mathsf{S}, pk_\mathsf{S}, pk_\mathsf{C}, c)$.

In the login phase, both parties first engage in the OPRF protocol where the client inputs password guess pw' and the server the OPRF key k_O. At the end of the OPRF protocol the client learns an envelope decryption key rw'. The server also sends the envelope c to the client. If the password guess pw' was correct, the client is able to decrypt c with rw', and the client obtains her AKE long-term keys and the server's AKE public key $(sk_\mathsf{C}, pk_\mathsf{C}, pk_\mathsf{S})$. Both parties further generate AKE ephemeral key-pairs and exchange the ephemeral public keys. If the client's password was correct, both parties can now derive the shared AKE-key via the KE function. Both parties further send key confirmation messages t_1, t_2 in order to achieve explicit authentication. If the check is successful, both parties will output the shared session key K_{sess}. A description of the OPAQUE protocol is given in Fig. 2.

3.2. Insecurity of Shared OPRF Key

The OPAQUE protocol is presently undergoing standardization efforts led by the IRTF, with the 12th iteration of the draft version – henceforth called Draft-OPAQUE – currently under review [4]. The draft version deviates from the version proposed by Jarecki et al. [23] in several aspects. Among the most significant changes, it models the registration as an interactive protocol, allows the server to re-use its AKE key-pair and to use a single OPRF seed to derive per-client OPRF keys. Furthermore, it uses transportable keys, meaning that the client's encrypted key-pair is not directly included in the envelope. Instead, the envelope contains information which can be used by the client to deterministically derive her AKE key-pair. An overview of the changes can be found in [4], and parts of the protocol have been analyzed in [10], [21].

A formal security analysis of the currently standardized draft version has not been conducted yet, and is a significant endeavor on its own. In our work, we focus on two specific modifications aimed at adapting the single-user protocol into the multi-user setting: re-using the same OPRF seed for all users, and re-using the server's AKE keys. To gain a deeper understanding of the implications of these changes, we analyze them separately.

We initiate our analysis with the changes to the OPRF keys. We capture this change by analyzing a version that is equivalent to the provably-secure OPAQUE protocol, except for using the single seed approach of Draft-OPAQUE and being expressed for multiple clients directly. We then show that this single-seed OPAQUE (ssOPAQUE) version does not achieve the expected security guarantees.

ssOPAQUE. The single-seed OPAQUE protocol is identical to OPAQUE with two main changes: First, it allows to register multiple-clients with the same server, whereas OPAQUE was designed for a single client only. To run a UC-secure multi-user version of OPAQUE, the server would have to run fully independent instances of the protocol for each client, as discussed in Section 2.2. In particular, the server cannot re-use any key material for different clients.

However, instead of having independent OPRF keys for different clients, the OPRF keys in ssOPAQUE are derived from a single seed. To this end, the server first initializes the seed $\mathsf{seed}_{\mathsf{oprf}}$ in a setup phase. During the registration and login phase, whenever the server requires the OPRF key, it deterministically derives the OPRF key from $\mathsf{seed}_{\mathsf{oprf}}$ and user identifier uid as $k_O := \mathsf{OPRF.KGen}(1^\lambda; H(\mathsf{seed}_{\mathsf{oprf}}, \mathsf{uid}))$ where $H(\mathsf{seed}_{\mathsf{oprf}}, \mathsf{uid})$ is the explicit randomness used for the KGen algorithm. This approach ensures that each client gets a different

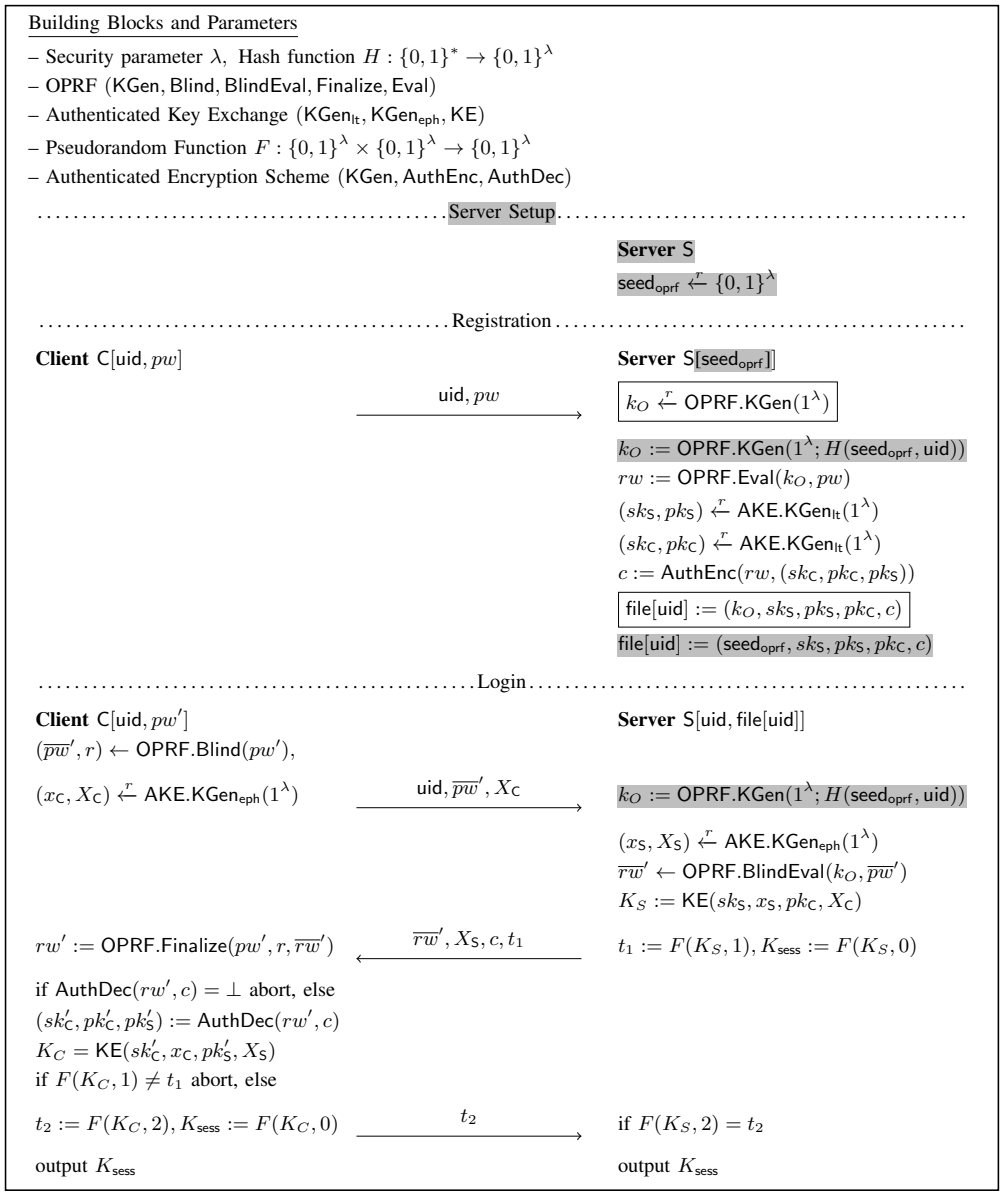

Figure 2: The OPAQUE [23] protocol (with boxed text and without grey text) and the ssOPAQUE protocol (with grey text and without boxed text) reflecting the single seed approach of Draft-OPAQUE. We assume all messages and inputs to the hash functions are prefixed by the session identifier sid, and all messages specific to the login phase are further prefixed by ssid. We omit these values as a writing convention.

OPRF key which is a good practice. However, as we will see, the dependency of the keys can become problematic, particularly if the seed gets compromised.

Modeling the Password File in Draft-OPAQUE. The draft is somewhat ambiguous on how a user's password file should be interpreted with regards to the OPRF seed. In theory, the password file for a user uid is interpreted as the output of the registration process, which the server stores in his database. During a login session, the server retrieves this password file and authenticates the user based solely on the password file.

Draft-OPAQUE differs from this approach in that it includes a setup phase where long-term keys such as the $\mathsf{seed_{oprf}}$ are established. The output of the registration process is a record which does not contain the OPRF seed or the OPRF key (as would be the case for OPAQUE), instead the draft states that "*the values* $\mathsf{seed_{oprf}}$ *and* [...] *from the server's setup phase must also be persisted*" [4, Sec. 5].

Furthermore, in the authentication phase, the $\mathsf{seed_{oprf}}$ is an additional input from the server to generate its response in the login phase [4, APIs 6.2.2 and 6.3.2.2.], and is thus used to authenticate the user. Since $\mathsf{seed_{oprf}}$ is a long-term value stored by the server, and used to authenticate the client in every key exchange, it fulfills our definition of (being part of) a password file and we include $\mathsf{seed_{oprf}}$ in every user's password file in ssOPAQUE. As a consequence of this, as soon as a single file gets compromised, the OPRF seed is leaked to the attacker. However, since $\mathsf{seed_{oprf}}$ is handled by the server in every key exchange protocol (i.e. hot storage), we believe that this is a realistic scenario and the possible effects of

leaking $\mathsf{seed}_{\mathsf{oprf}}$ should be analyzed.

For comparison, the secure multi-user OPAQUE and ssOPAQUE are depicted in Fig. 2.

Insecurity of ssOPAQUE. We show that ssOPAQUE does not achieve the desired security properties of a strong aPAKE. The main issue is that in ssOPAQUE, compromising the password file of one client allows an attacker to perform offline attacks on all other clients who have registered with the same server.

In order to show the attack, we consider a system with two honest users, uid and uid'. Both users have registered with the server, meaning the server has executed $\mathsf{StorePwdFile}(\mathsf{uid}, pw)$ and $\mathsf{StorePwdFile}(\mathsf{uid}', pw')$ queries where the attacker does not know the passwords pw and pw'. The attacker then compromises the server's file for uid, which immediately destroys all security for uid. Importantly, this should not impact the security of user uid', as the attacker should not be able to test passwords offline for other users.

This does not hold for ssOPAQUE, as demonstrated by the following attack:

Compromise password file of uid: The adversary $\mathcal{A}$ issues a $\mathsf{StealPwdFile}(\mathsf{S}, \mathsf{uid})$ query to obtain the password file of uid. He receives $\mathsf{file}[\mathsf{uid}] = (\mathsf{seed}_{\mathsf{oprf}}, sk_{\mathsf{S}}, pk_{\mathsf{S}}, pk_{\mathsf{C}}, c)$, and thus learns $\mathsf{seed}_{\mathsf{oprf}}$. With this information, the adversary can now conduct offline password tests for uid, but is not expected to do anything beyond that.

Compute OPRF key of uid': $\mathcal{A}$ uses the stolen seed to compute the OPRF key of uid' as $k'_O := \mathsf{OPRF.Eval}(1^\lambda; H_0(\mathsf{seed}_{\mathsf{oprf}}, \mathsf{uid}'))$. This information alone does not allow to break security for uid' yet, but only a single benign interaction with the server is needed to do so.

Retrieve envelope c' of uid': $\mathcal{A}$ initiates a client session with the server to obtain the envelope c' stored for user uid'. To achieve this, he picks a random pw, computes $(\overline{pw}, r) \leftarrow \mathsf{OPRF.Blind}(pw)$ and $(x_{\mathsf{C}}, X_{\mathsf{C}}) \xleftarrow{r} \mathsf{AKE.KGen}(1^\lambda)$ as outlined in the protocol. He sends $\overline{pw}, X_{\mathsf{C}}$ to the server and receives the server's answer $(\overline{rw}, X_{\mathsf{S}}, c', t_1)$. The value c' is part of the user's password file and should not allow for offline attacks on the password.

Offline Password Guess on uid': Knowing c' and the OPRF key k'_O, the attacker can now conduct offline password guesses. Let pw^* be a password that the attacker wishes to test. In order to do so, he computes the potential decryption key $rw^* := \mathsf{OPRF.Eval}(k_O, pw^*)$ and attempts to decrypt c' by executing $m/\perp \leftarrow \mathsf{AuthDec}(rw^*, c')$. Through the authenticated encryption and random key-robustness property, the decryption will only succeed in case the attacker has guessed the password correctly, giving the attacker a way to check his password guess. Notably, the attacker can verify arbitrary many password guesses without requiring interaction with the server, making this an offline attack.

This attack presents a clear contradiction to saPAKE security. Specifically, saPAKE states that offline password guesses for client uid' should be infeasible unless the password file of uid' is compromised. Our attack, however, demonstrates that in ssOPAQUE, as soon as the password file of a single user within the system is compromised, offline password guesses become possible for all users.

Mitigating the Attack. The problem that the draft version of OPAQUE exhibits is that the OPRF keys are not independent of each other, and knowing a user's OPRF key immediately opens up offline attacks. In order to mitigate the attack, the OPRF keys need to be instantiated individually and independently for each client. Security of the protocol then follows from the single-client security of OPAQUE [23] and the UC composition guarantees [7].

As a middle ground, it might seem tempting to use a single-seed to derive the different OPRF keys during registration, but store the resulting OPRF key as part of the password file instead of the OPRF seed. The OPRF seed could be stored in some secure enclave where we have higher protection of it, as suggested in [21]. This version is then secure as long as the OPRF seed is not compromised. In order to formally analyze this version though, one would have to adjust the ideal functionality to capture these two different types of file and key compromise of the server. After defining the new functionality, the adapted protocol must be proven secure in this model. Thus, we believe that independently creating the OPRF keys is a better approach.

Implications for Draft-OPAQUE. Even without analyzing the other changes of Draft-OPAQUE it is clear that the currently proposed standard is not secure, when the recommendation of using a shared seed for multiple client is implemented. For the sake of completeness, we also detail our attack to the Draft-OPAQUE protocol in Appendix B. In fact, this change has already led to discussions on the mailing list [26]. In the response, it was still stated that storing and re-using $\mathsf{seed}_{\mathsf{oprf}}$ for each client is believed to be secure, but it is OK to use multiple seeds if desired. As shown by our analysis this is not correct, and using multiple seeds should be the default.

3.3. Security of Shared AKE Key

The Draft-OPAQUE specification also allows the server to use the same AKE key-pair across multiple clients [4, Sec. 3.1]. Similar to the treatment of $\mathsf{seed}_{\mathsf{oprf}}$, we can view the AKE key-pair as part of each user's password file. Consequently, if an attacker compromises a user uid's password file, he will learn the server's AKE key-pair $(sk_{\mathsf{S}}, pk_{\mathsf{S}})$ used for all clients registered with the server. This protocol variant is depicted in Fig. 8 in Appendix C.

Contrary to a shared OPRF seed, sharing AKE keys for multiple clients (while assuming independent OPRF keys) may still result in a secure instantiation of OPAQUE. In Appendix C, we show the security of an instantiation of OPAQUE with the AKE protocol 3DH proposed in the IETF Draft [4], which utilizes a single server key-pair $(sk_{\mathsf{S}}, pk_{\mathsf{S}})$ for all clients. Our proof uses a simulator which is very similar to the one described in [23] with three essential tweaks: (1) the honest server is simulated with the same AKE key-pair across all clients, (2) the simulator is adjusted to the multi-user setting by replacing the sid with $(\mathsf{S}, \mathsf{uid})$, and (3) our simulator simulates the AKE subprotocol 3DH directly, instead of interacting with a simulator $\mathsf{SIM}_{\mathsf{AKE}}$. The analysis of the simulator

further needs to encompass that when the password file $\mathsf{file}[\mathsf{uid}] = (k_O, sk_\mathsf{S}, pk_\mathsf{S}, pk_\mathsf{C}, c)$ of user uid is compromised, it will leak the server's AKE key-pair $(sk_\mathsf{S}, pk_\mathsf{S})$ which is part of the password file of all users $\mathsf{uid}' \neq \mathsf{uid}$ registered with the server. We briefly sketch here why this does not compromise the security guarantees for other users $\mathsf{uid}' \neq \mathsf{uid}$:

No Additional Offline Attacks: If the adversary compromises the password file of uid, offline attacks on the password of uid become inevitable. Nevertheless, even though the attacker learns $(sk_\mathsf{S}, pk_\mathsf{S})$ which is also part of $\mathsf{file}[\mathsf{uid}']$, this does not open the possibility of offline attacks on the password of user uid'. This is due to two key factors: (1) $(sk_\mathsf{S}, pk_\mathsf{S})$ is independent of the password, and (2) to execute an offline attack on uid', the attacker needs to know the OPRF key k'_O, which is independent of the OPRF key k_O associated with uid and which is not leaked if uid' is not compromised.

No Impersonation of S towards uid': Compromising the password file of uid inevitably allows the adversary to impersonate the server towards uid. However, he should not be able to impersonate S towards other users uid'. This is not possible since the adversary does not know the whole password file of uid' which includes the values $(k'_O, pk'_\mathsf{C}, c')$ needed for authentication. While the adversary might attempt to initiate the authentication phase with different values $(k'_O, pk'_\mathsf{C}, c')$, the adversary will only succeed and be able to determine the key of uid' if he guesses the password of uid' correctly. Crucially, this counts as an online attack which is already possible if the attacker does not have the keys $(sk_\mathsf{S}, pk_\mathsf{S})$ as additional knowledge. Furthermore, the simulator can detect this online attack and extract the password of the attacker.

No Impersonation of uid' towards S: Stealing the password file of uid does not allow the adversary to impersonate uid towards S: Even though the attacker knows $(pk_\mathsf{C}, sk_\mathsf{S}, pk_\mathsf{S})$, he cannot authenticate as uid due to the KCI resistance of the AKE protocol. This also extends to uid': the adversary is unable to authenticate as uid' towards S even if the adversary knows $pk_\mathsf{S}, sk_\mathsf{S}$ because the adversary does not know the key-pair $(sk'_\mathsf{C}, pk'_\mathsf{C})$ of uid' which is created independently from the $(sk_\mathsf{C}, pk_\mathsf{C})$ of any other client. Thus, learning the AKE keys $(sk_\mathsf{C}, pk_\mathsf{C}, sk_\mathsf{S}, pk_\mathsf{S})$ for any other client uid will not help in the AKE session of uid'. The only way for the adversary to authenticate as uid' is by guessing the password of uid' correctly.

A full description of the simulator is given in Appendix C.

4. aPAKE Security with Salting

We now take a look at standard aPAKE protocols – the weaker version of saPAKE which does not offer resistance against precomputation attacks. As in our analysis of saPAKE, we consider a multi-user setting that realistically addresses multiple clients in the system. It becomes apparent in this multi-user setting that existing aPAKE protocols, where password files incorporate a user-specific salt, such as [11], [13], [18], exhibit better resistance against precomputation attacks compared to what is required by the existing aPAKE security model. Consequently, we re-evaluate the handling of precomputation attacks within the security model. To this end, we provide a refined security model capable of capturing protocols that offer more resistance against precomputation attacks than what is required from standard aPAKE.

4.1. Existing aPAKE Security Model

aPAKEs have been studied extensively in the past [19] and the security guarantees for an aPAKE protocol are identical to that of a saPAKE protocol, with one important change: aPAKE is not resistant against precomputation attacks on a user's password file.

Precomputation Attacks. In a precomputation attack, the attacker's objective is to reduce the time it takes to learn a user's password after compromising the user's password file. To this end, before compromising the server, the attacker can precompute a table of values based on a dictionary of passwords. This usually translates to precomputing a list of hashes for the most common passwords [3]. When the attacker compromises the server at a later point and obtains the user's password file, he will instantly learn the user's password if it matches one of the passwords included in the precomputation table. In the hashed password context, this translates to looking up whether the password file matches one of the hashes in the precomputation table.

Ideal Functionality $\mathcal{F}_{\mathsf{aPAKE}}$. Much like in the saPAKE setting, most of the analysis on aPAKE was conducted in a single-user UC framework [15], with multi-user security achieved through the composability theorem [7]. However, a recent development introduced an ideal functionality which explicitly handles the multi-user setting [11], [13], [17]. In our analysis, we adopt this multi-user ideal functionality for aPAKE.

The ideal functionality $\mathcal{F}_{\mathsf{aPAKE}}$ provides the same interfaces as $\mathcal{F}_{\mathsf{saPAKE}}$, only the handling of the OfflineTestPwd interface is changed. The attacker can access this interface at any time with password guess pw^* for client uid. In case the password file is not stolen, the password guess is now logged as $\langle \mathsf{offline}, \mathsf{S}, \mathsf{uid}, pw^* \rangle$. When the attacker then issues a StealPwdFile query to compromise the server at a later point, he learns the password used to create a password file if he has queried the OfflineTestPwd query on the correct password before. On a more technical level, upon receiving the $\mathsf{StealPwdFile}(\mathsf{sid}, \mathsf{S}, \mathsf{uid})$ query, the functionality retrieves the internal password file $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ and checks if there is a logged offline guess $\langle \mathsf{offline}, \mathsf{S}, \mathsf{uid}, pw^* \rangle$ with $pw^* = pw$. If this is the case, pw is sent to the attacker. The aPAKE functionality is given in Fig. 1.

Gap To Real World. Upon a closer examination of the ideal functionality, we observe that the handling of precomputation attacks is too permissive. The functionality permits an attacker to log an offline password guess – thus effectively creating a precomputation table – *at any time* after the user is registered. This is significantly weaker than the security guarantees of conventional password-based authentication [29], where, in an effort to mitigate

precomputation attacks, the server creates a high-entropy random value (the salt) for each user, and stores the salted hash along with the salt.

Given the fact that there are UC-secure aPAKE protocols which use similar salting techniques to thwart precomputation attacks [17], [11], [13], [18], we want to understand their exact resistance to precomputation attacks. To this end, we observe that the password file in every aPAKE protocol contains values (s, v), where we call s the salt value and v the password verifier, such that v is computed with a deterministic function f as

$$v := f(\mathsf{S}, \mathsf{uid}, pw, s)$$

This notation allows us to express a precomputation attack. Since f is deterministic, an attacker who knows $(\mathsf{S}, \mathsf{uid}, s)$ can precompute a possible table of values $(v_1, v_2, \ldots, v_n)$ where $v_i = f(\mathsf{S}, \mathsf{uid}, pw_i, s)$ for password guesses pw_i. Upon compromise, the attacker learns v and checks whether $v = v_i$ for one of the precomputed values. Clearly, for this precomputation attack to work, the attacker needs to know S, uid and s. Since we assume S and uid to be public, precomputation attacks only become possible if the attacker learns the salt s. A protocol's resistance against precomputation attacks is therefore mainly tied to the question: In which moment does the attacker learn the salt s which allows him to run a precomputation attack?

Therefore, we propose a framework based on the UC (s)aPAKE definition to capture different approaches for aPAKE protocols based on their leakage of the salt. In our framework, we outline four different salt levels, which we present in the following. Salt level 1 offers the weakest security guarantee against precomputation attacks, and salt level 4 the strongest.

4.2. Level 1: No Salt (aPAKE)

The most basic form of aPAKE is an aPAKE with no salt, i.e. $s = \perp$. In this case, we cannot assume any resistance against precomputation attacks as the attacker who knows $(\mathsf{S}, \mathsf{uid})$ can perform a precomputation attack, using $f(\mathsf{S}, \mathsf{uid}, pw_i, \perp)$ for password guesses pw_i. The security for level 1 is expressed with the classic aPAKE functionality which we outlined in Sec. 4.1, and which allows precomputation attacks against a user at any time.

It is important to note here, that offline password guesses in aPAKE are required to be targeted to a specific user-server pair. This is evident from the functionality which requires $(\mathsf{S}, \mathsf{uid})$ to be an input to the $\mathsf{OfflineTestPwd}$ query. This essentially means that password files need to be tied to a specific user (e.g. by including the username in the hash). Having this constraint gives a better security guarantee than pure dictionary attacks where the attacker can build a global user-independent precomputation table and use it to recover the passwords of multiple users upon compromise.

Weakening aPAKE? Modeling offline password guesses as targeted to a single user is actually inherent in the single-user setting for which the aPAKE functionality was first defined, since there is only a single user per functionality. In our multi-user setting, we do not encounter this limitation and could formalize a weaker aPAKE model which allows for scenarios in which the attacker computes a global user-independent precomputation table to recover the passwords of multiple users. However, we chose not to model this because we believe that this is not a desirable security property. Modeling this property would also not be as straightforward, as it would additionally require handling additional leakage in the functionality (i.e. if two users share the same password this would be leaked to an adversary, because their password verifier would be equal).

4.3. Level 2: Passively-Revealed Salt

The level 2 security for aPAKE is inspired by our observation that existing UC-secure aPAKE protocols such as KHAPE [17], AuCPace [18], or SRP-6a [11], use salting, i.e. they create a user-specific salt value in the hash computation to thwart precomputation attacks. The user's password file thus constains a high-entropy salt s, but in contrast to strong aPAKE, this salt s does not remain hidden but is sent in clear to any user trying to authenticate as uid. Any eavesdropping attacker can therefore learn s and compute the precomputation table using $f(\mathsf{S}, \mathsf{uid}, pw_i, s)$. Sending the salt in these protocols, however, is needed, because in order to execute the protocol the client needs to be able to reconstruct the salted hash. In the single-user setting, this extra salt may not seem to provide a significant security boost, as an attacker can learn the user's salt with a single login query and subsequently use it to build a precomputation table for that user.

However, when transitioning to the multi-user setting, where a single server may serve millions of users, the dynamics change. In order for an attacker to have the same precomputation attack capabilities as in level 1, he needs to intercept or start login sessions for all million users.

For level 2 security, we therefore require that the attacker can build a precomputation table for user uid only after eavesdropping on a session between uid and the server or by starting a session with the honest server where the attacker pretends to be uid.

In our security model, we do not assume authenticated channels between parties as this would defeat the purpose of a password-based authentication protocol. In the UC framework, this is modeled by an attacker who controls all messages sent between the parties. The attacker can therefore be seen as a global network adversary who eavesdrops all messages. In the presence of this attacker, we can only model the following security guarantee for level 2: Precomputation attacks on uid are not possible before the honest server has started a session with uid.

This essentially means that a precomputation table can be built for any user who has started a session at least once with the server. While this security guarantee seems rather weak, it is the strongest security level one can achieve for a protocol which sends the salt in clear over an unauthenticated channel such as [11], [13], [17], [18] (see Sec. 5.2 for a further analysis). We see level 2 therefore more as a stepping stone towards defining level 3, which offers substantially more resistance against precomputation attacks.

We model this in the functionality by introducing a flag for the internal password file $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ which

is marked fresh upon instantiation. The first SvrSession for uid will change the status of this flag to unlocked. The functionality then logs password guesses for queries $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{S}, \mathsf{uid}, pw^*)$ only if the status of $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ is unlocked. If the server's password file gets stolen, the process remains unchanged compared to the aPAKE functionality: If there has been a correct offline password guess logged, the attacker learns the password immediately. The changes to the ideal functionality are given in Fig. 3.

4.4. Level 3: Actively-Revealed Salt

Level 2 offers only weak protection against a global network adversary. Therefore, we want to strengthen our model and protect the salt against eavesdropping adversaries. In level 3 protocols, only the user who wishes to authenticate uid will learn the salt s for user uid, essentially meaning that the salt is not transmitted in clear and protected from eavesdroppers. However, an active attacker who authenticates as uid still learns s and can run precomputation attacks using $f(\mathsf{S}, \mathsf{uid}, pw_i, s)$.

For level 3 security, we therefore require that in order to run a precomputation attack, the attacker needs to *actively engage* in a session by starting a session on behalf of uid.

Starting a session on behalf of uid is still an easy task for an adversary, especially since, due to the password-only setting, no authentication happens prior to starting a session. Hence, anyone in the system can start a session on behalf of uid. However, in the multi-user setting with millions of users, the attacker would need to start a session for all millions of users in order to perform precomputation attacks (on all users). While this is still considerably easier than compromising a user's file on the server, as required by strong aPAKE, it can deter some attackers or, at the very least, make their attacks more costly and easier to detect.

To model this security level in the UC functionality, we again assume that the internal password file $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ has by default a status fresh. We now change the condition when the file becomes unlocked. Therefore, we first have to change the CltSession interface to be accessible by either the client or the adversary. This allows us to express whether an honest client runs a session, or an adversary pretending to be the client. In case the adversary starts a CltSession, and there is a corresponding SvrSession initiated by the honest server, the file is marked unlocked. The OfflineTestPwd password guesses are only logged if the password file is marked unlocked. The changes to the ideal functionality are given in Fig. 4

4.5. Level 4: Private Salt (saPAKE)

Interestingly, the strong aPAKE security guarantee can also be expressed in the salting framework. In an saPAKE protocol, the password file contains a salt s, where s is never transmitted in the protocol. It is only available upon compromising the password file. Therefore no precomputation attack on uid should be possible before the password file of uid is compromised. This level corresponds to the strong aPAKE security level which we already discussed in Sec. 2 with the functionality given in Fig. 1.

5. Categorizing Existing aPAKE Protocols

In this section, we categorize the existing aPAKE solutions into the different security levels. We only consider aPAKE protocols that have been proven secure in the UC framework. Table 1 gives an overview of the security levels achieved. We first argue how protocols which are instantiated in the single-user setting can be translated to our multi-user setting, before we inspect why the protocols are categorized as they are. For the analyzed level 4 protocols, their categorization follows directly from security in the saPAKE framework, and we do not include them in our discussion.

Multi-User Security. Most of the protocols we analyzed have been proven in the single-user UC framework. To adapt these protocols to our multi-user scenario, we outline the process of combining multiple single-user aPAKE protocols to achieve a secure multi-user aPAKE.

To create a secure multi-user protocol π, we combine *independent* copies of the single-user aPAKE protocols where the session identifier of the individual copies are $(\mathsf{S}, \mathsf{uid})$. Since the combination of $(\mathsf{S}, \mathsf{uid})$ yields a unique identifier for the single-user session, the security of this approach follows from the universal composability theorem [7], [8], [9]. Importantly, for the composability theorem to hold, this requires all protocol instances to be independent of each other. In the aPAKE context, this especially refers to the password files stored by the server. Looking at the password files used in aPAKE protocols, they include things such as salts, hashes, (O)PRF keys, AKE keys or encryption keys. We have to assume that all of these values which are created by the server in different copies of the protocol are independent of each other. Furthermore, it is important to note that the session identifiers of the individual copies are $(\mathsf{S}, \mathsf{uid})$, which means that all queries to hash functions stemming from this copy need to be prefixed by $(\mathsf{S}, \mathsf{uid})$, thus including the user identifier in the hash functions.

5.1. Level 1 Protocols

Since level 1 corresponds to the classic aPAKE security definition, it is no surprise that most of the existing UC-secure aPAKEs fall into this category [15], [22], [24], [27], [31].

Construction. The common theme to all of the level 1 protocols is that the password file includes a hash of the password together with the client-server specific values $(\mathsf{S}, \mathsf{uid})$. As an example, in KC-SPAKE2+ [31] the password file is constructed as

$$\mathsf{file}[\mathsf{uid}] := (H_0(\mathsf{S}, \mathsf{uid}, pw), g^{H_1(\mathsf{S}, \mathsf{uid}, pw)})$$

Precomputation Attacks. In all of the protocols, precomputation attacks on user uid are possible as soon as uid is registered with the server since we assume that the attacker knows the values S and uid. The precomputation attack itself is straightforward, the attacker just computes

Salt Level 2

Stealing Password Data

– On $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{S}, \mathsf{uid}, pw^*)$ from $\mathcal{A}$, do:
  – If $\exists$ record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked stolen, do the following: If $pw^* = pw$ return "correct guess" to $\mathcal{A}$, else return "wrong guess".
  – Else, if $\exists$ record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked unlocked, record $\langle \mathsf{offline}, \mathsf{S}, \mathsf{uid}, pw^* \rangle$.

Password Authentication

– On $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{uid})$ from S, if there is no record $\langle \mathsf{ssid}, \mathsf{S}, \dots \rangle$ then retrieve record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$, and if it exists then create record $\langle \mathsf{ssid}, \mathsf{S}, \mathsf{C}, \mathsf{uid}, pw, \mathtt{sr} \rangle$ marked fresh and send $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \mathsf{C}, \mathsf{uid})$ to $\mathcal{A}$. Additionally, if rec. $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ is marked fresh, mark it unlocked.

Figure 3: The changes in the ideal functionality $\mathcal{F}_{\mathsf{aPAKE}}$ for level 2 (changes relative to level 1).

Salt Level 3

Stealing Password Data

– On $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{S}, \mathsf{uid}, pw^*)$ from $\mathcal{A}$, do:
  – Else, if $\exists$ record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked stolen, do the following: If $pw^* = pw$ return "correct guess" to $\mathcal{A}$, else return "wrong guess".
  – Else, if $\exists$ record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked unlocked, record $\langle \mathsf{offline}, \mathsf{S}, \mathsf{uid}, pw^* \rangle$.

Password Authentication

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

Figure 4: The changes in the ideal functionality $\mathcal{F}_{\mathsf{aPAKE}}$ for level 3 (changes relative to level 2).

$H_0(\mathsf{S}, \mathsf{uid}, pw^*)$ for password guesses pw^*. Upon compromising the server, the attacker can compare the password file with his list of hashed passwords and recover the password immediately. These protocols offer therefore no resistance against precomputation attacks.

5.2. Level 2 Protocols

Four of the UC-secure protocols we analyzed fall into the level 2 category which allows precomputation attacks for user uid only after there was a session between an honest server and someone claiming to be uid. Three of these protocols, SRP-6a [11], OKAPE [13] and AuCPace[18], use the salted hash approach in the creation of their password files. That is, upon registration, a random high-entropy salt s is chosen, and stored in the password file together with values derived from the salted hash $H(s, pw)$ of the password. In these protocols, the salt is sent to the client during a login session, such that the client can recompute the salted hash.

OKAPE-HMQV. We sketch the OKAPE-HMQV protocol as a representative of these protocols. The precomputation attack in the other protocols is similar. OKAPE uses AKE as building block where the client derives her AKE long-term key deterministically from the password, and receives the server's (client-specific) long term key encrypted under a salted password hash.

During registration, the server picks a random salt $s \xleftarrow{r} \{0,1\}^\lambda$, and creates an AKE long-term key-pair (a, g^a) and an encryption key h from $H(pw, s)$, i.e. $(h, a) := H(pw, s)$. The password file then consists of

$$\mathsf{file}[\mathsf{uid}] = (g^a, h, s)$$

In the login session, the server creates a client-specific AKE long-term key-pair (b, B) and sends the public key B encrypted under h to C, along with the salt s, i.e. the client receives $(\mathsf{Enc}(h, B), s)$. It is important that the salt s is sent to the client, because the client uses s to recompute $(h, a) := H(s, pw)$. She then computes an ephemeral key pair (x, X), and derives the AKE key using her own private keys a, x and the server's public key B. She sends X to the server along with a key confirmation message. The server also derives the AKE key using its private key b and the client's public keys A, X. If the key confirmation message verifies, both parties will output the session key.

Precomputation Attacks. In OKAPE, an attacker is able to build a precomputation table upon seeing the salt s, which is part of the first message sent from the server to the client. This salt is available to an attacker who eavesdrops in an honest session or engages with S on behalf of uid. After learning the salt s, the attacker can compute $(h^*, a^*) \leftarrow H(s, pw^*)$ for password guess pw^* and store h^*. Upon compromise, the attacker can see immediately whether the h value from the password file $\mathsf{file}[\mathsf{uid}] = (g^a, h, s)$ matches one of the stored values h^*, in which case he learns the password.

KHAPE. The fourth UC-secure level 3 protocol is KHAPE [17] which uses the encrypted envelope structure for the password file (similar to OPAQUE). In KHAPE, the server registers a client by creating two AKE key-pairs, one for the server (b, B) and one for the client (a, A). The server then encrypts (a, B) under the user's password, i.e. $e := \mathsf{Enc}(pw, (a, B))$, and stores this encryption together with his private key b and the client's public key A in the password file:

$$\mathsf{file}[\mathsf{uid}] = (e, b, A)$$

Table 1: Overview of the security guarantees achieved by existing UC-secure aPAKEs. We list the parts of the password file which are precomputable and which values are a prerequisite for the attacker in order to perform a precomputation attack. For a better comparison, in protocols with * we have translated the sid from the single-user setting to its corresponding equivalent $(\mathsf{S}, \mathsf{uid})$ in the multi-user setting.

Scheme	file[uid] =	Precomputable	Prerequisite	Salt Level
Sigma-Method* [15]	$\left(H(\mathsf{S},\mathsf{uid},pw), pk, c := \mathsf{Enc}(pw, sk)\right)$	$H(\mathsf{S},\mathsf{uid},pw)$	S, uid (public)	1
Hwang et al.* [22]	$\left(H_1(\mathsf{S},\mathsf{uid},pw), g^{H_0(\mathsf{S},\mathsf{uid},pw)}\right)$	$H_1(\mathsf{S},\mathsf{uid},pw)$	S, uid (public)	1
KC-SPAKE2+ [31]	$\left(H_0(\mathsf{S},\mathsf{uid},pw), g^{H_1(\mathsf{S},\mathsf{uid},pw)}\right)$	$H_0(pw,\mathsf{S},\mathsf{uid})$	S, uid (public)	1
2DH-aEKE [27]	$\left(h, g^{v_1}, g^{v_2}\right)$ with $(h, v_1, v_2) \leftarrow H(\mathsf{S},\mathsf{uid},pw)$	$H(\mathsf{S},\mathsf{uid},pw)$	S, uid (public)	1
Jutla-Roy [24]	$\left(g^{H(\mathsf{S},\mathsf{uid},pw)}\right)$	$g^{H(\mathsf{S},\mathsf{uid},pw)}$	S, uid (public)	1
SRP-6a [11]	$(s, g^{H(s,\mathsf{uid},pw)})$	$H(s,\mathsf{uid},pw)$	uid (public) and s (passively-revealed)	2
OKAPE-HMQV* [13]	$\left(g^a, h, s\right)$ with $s \xleftarrow{r} \{0,1\}^\lambda$ and $(h, a) \leftarrow H(pw, s)$	$H(pw, s)$	s (passively-revealed)	2
KHAPE-HMQV* [17]	$\left(e, (b, A)\right)$ with AKE key-pairs (a, A), (b, B) and $e := \mathsf{Enc}(pw, (a, B))$	$\mathsf{Dec}(pw, e)$	e (passively-revealed)	2
AuCPace [18]	$\left(s, g^{H(pw,\mathsf{uid},s)}\right)$	$g^{H(pw,\mathsf{uid},s)}$	uid (public) and s (passively-revealed)	2
CKEM-saPAKE* [5]	$\left(s, f_s(H(\mathsf{S},\mathsf{uid},pw)\right)$ where f_s is a salted tight one-way function	–	–	4
OPAQUE-HMQV [23]	$\left(k_O, sk_\mathsf{S}, pk_\mathsf{S}, pk_\mathsf{C}, c\right)$ with $c := \mathsf{AuthEnc}(F(k_O, pw), (sk_\mathsf{C}, pk_\mathsf{C}, pk_\mathsf{S}))$	–	–	4
MX-Compilers* [28]	$\left(g^s, (g^s)^{H(\mathsf{S},\mathsf{uid},pw)}\right)$	–	–	4

In a login session, the first message from the server to the client consists of the ciphertext e. The client decrypts e using her password, and both parties run the AKE with long-term keys (a, A) and (b, B).

Precomputation Attacks. In KHAPE, the client can run precomputation attacks for uid after seeing the ciphertext e of uid. To run a dictionary attack, the attacker decrypts the envelope e with password guess pw^* and obtains some message $(x, Y) := \mathsf{Dec}(pw, e)$. Note that unlike OPAQUE, KHAPE does not use authenticated encryption, and the AKE possesses a key-hiding property which ensures that the validity of guesses can only be checked upon compromising the server for uid. Compromising the server, the attacker will learn (e, b, A). Since (b, A) are the public-private counterparts of the values encrypted in e, the attacker can verify whether one of the decrypted ciphertexts are the counterparts of values (b, A).

Adjusting the security proofs for level 2 protocols. The security of these protocols has only been shown using the regular $\mathcal{F}_{\mathsf{aPAKE}}$ functionality which achieves level 1. We therefore need to argue why these protocols actually achieve level 2. Thus, we sketch how the proofs need to be adjusted to show security as a level 2 protocol.

The only change in the functionalities between level 1 and level 2 protocols is that password guesses to OfflineTestPwd are not logged by the functionality unless the password file of the user is marked unlocked, and a file becomes unlocked with the first query to SvrSession from the honest server. In order to show that these protocols achieve level 2, we therefore need to show that the attacker cannot make a precomputation attack before unlocking the file. We thus need to show that in the proof, the simulator does not query OfflineTestPwd before a session with the honest server has been started.

We first take a look at the protocols which use the salted password file approach (SRP-6a,OKAPE and AuCPace). The proof for all of these protocols is in the random oracle model, and the handling of offline password guesses is identical . In order to simulate the handling of password files, the simulator creates a random salt $s \xleftarrow{r} \{0,1\}^\lambda$ for user uid. One of the challenges of the simulator is that he has to detect an offline password guess from the attacker, and extract the tested password. In the random oracle model, this is an easy task for the simulator as he can control the in- and outputs of all queries to the random oracle H. Since password guesses for uid must contain the user-specific salt s, the simulator can detect that any query to the random oracle consisting of (s, uid, pw) corresponds to an offline password guess by the attacker. After extracting the password, the simulator can pass it to the functionality using the OfflineTestPwd interface.

Crucially, in order to show that the proofs achieve level 2 security, we need to show that the probability that an attacker queries $(s, \mathsf{uid}, \cdot)$ before the first session for uid with an honest server is established, is negligible. The proofs consist of a series of game hops and we can show level 2 security by including the following game hop at any point throughout the security proofs:

Additional Game Hop for Level 2: If there is a query to the random oracle H of the form $(s, \mathsf{uid}, \cdot)$ where s is identical to the salt value chosen for uid, before the first call to $(\mathsf{SvrSession}, \mathsf{ssid}, \mathsf{C}, \mathsf{uid})$ by S, abort.

For the indistinguishability of the game hop, we need to show that the probability of an abort is negligible. Since the salt is a random value from $\{0,1\}^\lambda$, independent from all other salts in the system, and the simulator only keeps this value internally if there is no query to SvrSession or StealPwdFile, the probability that the adversary queries the random oracle on $(s, \mathsf{uid}, \perp)$ is bounded by $q/2^\lambda$, where q is the number of queries to H and λ is the security parameter of the system. Since $q/2^\lambda$ is negligible, the probability of an abort is negligible. In case

there is no abort, we have not changed anything in the simulation, but we are sure that the attacker never issues an OfflineTestPwd query before the first honest server session. The rest of the proof remains the same, yielding level 2 security.

In the case of KHAPE, a similar argument holds, only that the attacker's password guesses are now extracted from queries to the ideal cipher decryption oracle. The attacker learns the user's envelope e only after a session is started. Before there is a session with an honest server, e is only kept internally, random and independent from all other values due to the ideal cipher model. The attacker can therefore query the ideal cipher on e before a session starts only with negligible probability. Formally, we can introduce a game hop where the simulation aborts if e is queried to the ideal cipher oracle before the first SvrSession with the honest server has been started. Otherwise, the proof again proceeds as before, and level 2 security of KHAPE follows.

Compiler Level 1 → Level 2. Level 2 security can also be achieved by lifting a level 1 protocol to level 2. To do so, one can use the following approach suggested by [13]:

Let π be a level 1 aPAKE protocol. In the modified protocol π', the server registers the user by picking a random salt $s \xleftarrow{r} \{0,1\}^\lambda$ and running the registration of π with modified password $pw' = pw|s$. In the login phase, the server then sends s to C as the first message, and both parties can run the level 1 aPAKE protocol π on modified password $pw' = pw|s$. Since s is not leaked before the first session between uid and S, this yields level 2 security.

5.3. Level 3 Protocols

None of the aPAKE protocols we analyzed are level 3 protocols because they either use no salt or send the salt in clear. However, we argue that any level 2 protocol can be transformed to a level 3 protocol in a generic and simple way by encrypting the salt. One way to encrypt the salt is by assuming confidential channels, e.g. through a TLS connection. Then the salt would be protected from eavesdroppers. However, we do not want to rely on a confidential channel outside of the aPAKE protocol as this leaves a gap for security issues in the coordination of aPAKE and the implementation of the confidential channel. In fact, there are protocols which combine TLS with aPAKE protocols such as TLS-SRP [32] and TLS-OPAQUE [21]. While OPAQUE is already a strong aPAKE and does not need confidential channels to protect its salt, one would hope that TLS-SRP raises the security of SRP to level 3 through the TLS channel. However, this is not the case, as the aPAKE messages are transmitted before the TLS handshake, eventually leaking the salt to eavesdroppers.

Compiler Level 2 → Level 3. In light of this, we want to design a protocol which includes the encryption as part of the aPAKE protocol. Let π be a level 2 aPAKE protocol. We assume that the client sends the first message m_1 in the protocol and the server's response m_2 includes the salt while no other protocol value leaks information about the salt. In order to protect the salt in π from eavesdropping adversaries, the client first generates an encryption keypair (sk, pk) and sends pk to the server along with m_1. The server then sends $c := \mathsf{Enc}(pk, m_2)$ to the client. The rest of the protocol remains the same. This way, only the client can decrypt the server's message m_2 containing the salt, and the salt is protected from eavesdroppers. This yields level 3 security.

The attacker can still get the salt of user uid by pretending to be uid towards the server. In this case, the attacker will create a key-pair herself and send the public key to the server, where the salt value will be encrypted under the attacker's public key.

Lower Risk of Shared Seeds in Level 2 and 3? The multi-user variant of Draft-OPAQUE lost crucial security guarantees, because it relies on a single-seed to derive the OPRF keys (which can be seen as salt) for multiple clients. When discussing the multi-user aPAKE variants in this section, we assumed that they are securely composed from their single-user protocols. Interestingly though, the level 2 and 3 protocols seem less vulnerable if a seed re-use for the salt is done analogously to Draft-OPAQUE. Also here, application developers could be tempted to derive the users salt deterministically as $s := PRF(s_{\mathsf{seed}}, \mathsf{uid})$ relying on a long-term key s_{seed} that needs to be accessible at every login. Even if s_{seed} is leaked as part of a user file for uid, the only consequence is that it will allow precomputation attacks against all users, essentially downgrading to level 1 security. It particularly does not allow offline attacks on uid' as in (ss)OPAQUE. The reason is that the leaked seed would only allow to recompute $s := PRF(s_{\mathsf{seed}}, \mathsf{uid}')$ for uid', which is what the adversary can learn from a single benign login-query with the honest server anyway. Thus, the impact of such a single-seed is much less severe than in a level 4 protocol that crucially relies on the secrecy of the user-specific salts.

6. Conclusion

In our work, we revisited the security definitions for both standard aPAKE and strong aPAKE within a multi-user setting. Concerning strong aPAKE, our investigation revealed that the currently standardized draft version of OPAQUE may result in insecure implementations in the multi-user context due to a shared state.

Regarding standard aPAKE, we observed that the concept of salting was not accurately represented in the original aPAKE security definition. Our examination showed that real-world protocols employing salting methods exhibit better resistance against precomputation attacks compared to the requirements outlined in the security definition. To address this, we refined the security definitions of aPAKE to properly capture the security assurances provided by different salting approaches. Additionally, we classified existing schemes within our framework. Ultimately, we presented a straightforward method to transform a regular aPAKE protocol into an aPAKE which achieves better security guarantees.

References

[1] 2012 Linkedin Breach had 117 Million Emails and Passwords Stolen, Not 6.5M. https://www.trendmicro.com/vinfo/us/security/news/cyber-attacks/2012-linkedin-breach-117-million-emails-and-passwords-stolen-not-6-5m, May 2016.

[2] Apple. HomeKit communication security. https://support.apple.com/guide/security/communication-security-sec3a881ccb1/web, May 2022.

[3] Jeremiah Blocki, Benjamin Harsha, and Samson Zhou. On the economics of offline password cracking. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 853–871. IEEE, 2018.

[4] Daniel Bourdrez, Dr. Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-irtf-cfrg-opaque-12, Internet Engineering Task Force, October 2023. Work in Progress.

[5] Tatiana Bradley, Stanislaw Jarecki, and Jiayu Xu. 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*, pages 798–825. Springer, 2019.

[6] Bart Butler. Improved authentication for email encryption and security. https://protonmail.com/blog/encrypted_email_authentication/, Aug 2021.

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

[8] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *Advances in Cryptology—EUROCRYPT 2002: International Conference on the Theory and Applications of Cryptographic Techniques Amsterdam, The Netherlands, April 28–May 2, 2002 Proceedings 21*, pages 337–351. Springer, 2002.

[9] Ran Canetti and Tal Rabin. Universal composition with joint state. In *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*, pages 265–281. Springer, 2003.

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

[11] Dennis Dayanikli and Anja Lehmann. Provable security analysis of the secure remote password protocol. *Cryptology ePrint Archive*, 2023.

[12] Bojana Dobran. 1.6 million PayPal customer details stolen in Major Data Breach. https://phoenixnap.com/blog/paypal-customer-details-stolen, Jan 2022.

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

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

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

[16] Jonathan Greig. Nvidia says employee credentials, proprietary information stolen during cyberattack, Mar 2022.

[17] Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. 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*, pages 701–730. Springer, 2021.

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

[19] Feng Hao and Paul C van Oorschot. Sok: Password-authenticated key exchange–theory, practice, standardization and real-world lessons. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 697–711, 2022.

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

[21] Julia Hesse, Stanislaw Jarecki, Hugo Krawczyk, and Christopher Wood. Password-authenticated TLS via OPAQUE and post-handshake authentication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 98–127. Springer, 2023.

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

[23] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. 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*, pages 456–486. Springer, 2018.

[24] Charanjit S Jutla and Arnab Roy. Smooth NIZK arguments. In *Theory of Cryptography: 16th International Conference, TCC 2018, Panaji, India, November 11–14, 2018, Proceedings, Part I 16*, pages 235–262. Springer, 2018.

[25] Hugo Krawczyk. HMQV: A high-performance secure diffie-hellman protocol. In *Annual international cryptology conference*, pages 546–566. Springer, 2005.

[26] Kevin Lewi. Re: [cfrg] [crypto-panel] request for review: OPAQUE, September 19 2023. Message to the CFRG mailing list.

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

[28] Ian McQuoid and Jiayu Xu. An efficient strong asymmetric PAKE compiler instantiable from group actions. *Cryptology ePrint Archive*, 2023.

[29] Abhijit Menon-Sen, Alexey Melnikov, Nicolás Williams, and Chris Newman. Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms. RFC 5802, July 2010.

[30] Joern-Marc Schmidt. Requirements for Password-Authenticated Key Agreement (PAKE) Schemes. RFC 8125, April 2017.

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

[32] David Taylor, Trevor Perrin, Thomas Wu, and Nikos Mavrogiannopoulos. Using the secure remote password (SRP) protocol for TLS authentication. RFC 5054, November 2007.

[33] Telegram. Two-factor authentication. https://core.telegram.org/api/srp, April 2023.

[34] Karim Toubba. Notice of Recent Security Incident. https://blog.lastpass.com/2022/12/notice-of-recent-security-incident/, Dec 2022.

[35] Nihal Vatandas, Rosario Gennaro, Bertrand Ithurburn, and Hugo Krawczyk. On the cryptographic deniability of the Signal protocol. In *Applied Cryptography and Network Security: 18th International Conference, ACNS 2020, Rome, Italy, October 19–22, 2020, Proceedings, Part II 18*, pages 188–209. Springer, 2020.

[36] Martyn Williams. Inside the Russian hack of Yahoo: How they did it. https://www.csoonline.com/article/3180762/inside-the-russian-hack-of-yahoo-how-they-did-it.html, Oct 2017.

[37] T. Wu. SRP-6: Improvements and refinements to the secure remote password protocol. *http://srp. stanford. edu/srp6.ps*, 2002.

[38] Andy Yen. Proton pass is now in beta. https://proton.me/blog/proton-pass-beta, April 2023.

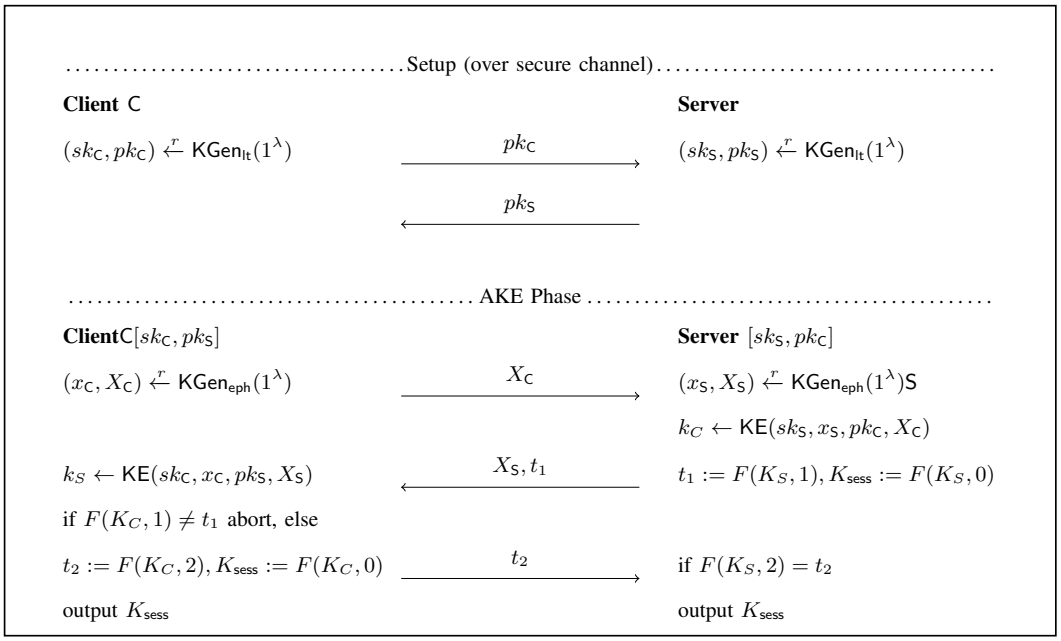

Figure 5: The 3-flow AKE protocol used in our description of ssOPAQUE.

A. Draft-OPAQUE

In this section, we present the attack on Draft-OPAQUE [4]. The Draft-OPAQUE protocol is given in Fig. 6 and 7. We highlight the most notable changes between Draft-OPAQUE and OPAQUE [23]

- While all the keys established during registration in OPAQUE are client-specific, Draft-OPAQUE allows the server to re-use keys: He can use the same AKE key-pair (sk_S, pk_S) for multiple clients and he can derive the per-client OPRF keys from an OPRF seed shared for multiple clients. In fact, the draft states that "the OPRF seed value SHOULD be used for all clients". In our description, we model this with a setup phase, where the server initializes the AKE key-pair and the OPRF seed. The values $(sk_S, pk_S, \mathsf{seed}_{\mathsf{oprf}})$ then become part of the state of the server which will be an additional input when registering a client.
- The registration phase in Draft-OPAQUE is interactive. OPAQUE considers a non-interactive registration phase where the server learns the client's password and performs all registration steps by himself. This is a standard approach in most UC aPAKE papers [15], [23]. In a real-world application however, an interactive registration phase where the server never learns the password, is preferred. Hesse et al. [21] recently considered the integration of OPAQUE into TLS where they model the registration in an interactive way, albeit not for the $\mathcal{F}_{\mathsf{saPAKE}}$ functionality but to the related $\mathcal{F}_{\mathsf{PHA}}$ functionality.
- The protocol outputs an export key for the client in addition to the session key. The export key can be used for application-specific purposes. This key is independent of all other protocol values and has no influence on the security analysis as it can be simulated with a random output.
- The Draft-OPAQUE protocol uses transportable keys as described in [21]. Transportable keys means that the envelope does not contain the encrypted client key-pair, but contains information which can be used to derive the AKE key by the client. More formally, the envelope contains an envelope nonce n_E, from which together with the OPRF-output rw the client-key pair can be derived deterministically.
- In Draft-OPAQUE, envelopes are masked with a per-client masking key k_{mask} derived from rw. The main motivation behind this is to prevent against client-enumeration attacks, where an attacker tries to learn whether a certain client is registered with the server. Draft-OPAQUE recommends storing a fake credential on the server which is accessible in similar time as a real credential. Draft-OPAQUE derives a One-Time-Pad key from the masking key and a masking nonce and encrypts the envelope with this key. The server sends the masking nonce to the client who can reconstruct the OTP key if he inputs the correct password and learns rw. Otherwise, the response of the server is indistinguishable and the attacker does not learn whether the client is registered.
- In Draft-OPAQUE, the parties exchange nonces n_C, n_S and bind all protocol values to these nonces, mainly to prevent replay attacks. In OPAQUE, security against replay attacks is covered by binding protocol values to the unique session identifiers ssid which are necessary in the UC framework [7].

A more exhaustive list of changes can be found in the draft version itself [4].

B. Attacking Draft-OPAQUE

We now give a concrete attack on Draft-OPAQUE following our attack approach to ssOPAQUE. We again consider two honest users uid and uid′ who have registered with the server, i.e. the server has executed $\mathsf{StorePwdFile}(\mathsf{uid}, pw)$ and $\mathsf{StorePwdFile}(\mathsf{uid}', pw')$ and

the attacker does not know passwords pw and pw'. The attack then proceeds as follows:

Compromise password file of uid**:** The attacker $\mathcal{A}$ issues $\mathsf{StealPwdFile}(\mathsf{S}, \mathsf{uid})$ to learn the password file of uid:

$$\mathsf{file}[\mathsf{uid}] = (pk_{\mathsf{C}}, k_{\mathsf{mask}}, t_{\mathsf{auth}}, n_E, sk_{\mathsf{S}}, pk_{\mathsf{S}}, \mathsf{seed}_{\mathsf{oprf}})$$

Compute OPRF key of uid$'$**:** Use the stolen $\mathsf{seed}_{\mathsf{oprf}}$ to compute the OPRF key of uid$'$:

$$k_O := \mathsf{OPRF.KGen}(1^\lambda; H_0(\mathsf{seed}_{\mathsf{oprf}} \| \mathsf{uid}'))$$

Retrieve envelope of uid$'$**:** The attacker now engages in a session with the server trying to impersonate uid$'$. He therefore chooses a random pw, and creates the first protocol message, i.e. he blinds the password to $\overline{pw}$, chooses a nonce n_{C} and an ephemeral key-pair $(x_{\mathsf{C}}, X_{\mathsf{C}})$ as outlined in the protocol. The attacker sends $\mathsf{uid}, \overline{pw}, n_{\mathsf{C}}, X_{\mathsf{C}}$ to the server and receives the server's response $((\overline{rw}, n_M, \mathsf{res}_{\mathsf{mask}}), (n_{\mathsf{S}}, X_{\mathsf{S}}, t_S))$.

Offline Password Guess on uid$'$**:** This is all the information the attacker needs in order to perform offline guesses on the password: For a password guess pw^*, the attacker computes $rw^* := \mathsf{OPRF.Eval}(k_O, pw)$, and computes the subsequent steps of the client in the protocol using rw^*. Crucially, if the check $\mathsf{Vf}(k_2, H(\mathsf{pa}), t_{\mathsf{S}}) = 1$ verifies, the attacker learns whether the password is correct, otherwise it is incorrect. Since the attacker never compromised the password file of uid$'$, this is a contradiction to the strong aPAKE security guarantees.

C. OPAQUE with Shared AKE Keys

We have seen that sharing the $\mathsf{seed}_{\mathsf{oprf}}$ in OPAQUE is insecure. In this section, we show that for the 3DH instantiation of OPAQUE, sharing the server's AKE key for multiple users remains secure. Therefore, we consider the OPAQUE protocol instantiated with 3DH, where the server uses the same AKE key-pair $(sk_{\mathsf{S}}, pk_{\mathsf{S}})$ for all users. The protocol is given in Fig. 8 and a UC-compliant description is given in Fig. 9.

Weakened Functionality. The OPAQUE protocol has actually only been shown to realize a weaker functionality $\mathcal{F}^-_{\mathsf{saPAKE}}$ [23], whose multi-user adoption we describe in Fig. 10. Our proof also uses this functionality, and we briefly outline the two main relaxations:

Allowing Compromise of all Open Server Sessions: $\mathcal{F}^-_{\mathsf{saPAKE}}$ does not guarantee the security of sessions in which the attacker actively interferes and which are not completed when the attacker learns the password. This is modeled by introducing a flag $\mathsf{flag}[\mathsf{sid}, \mathsf{uid}]$ which is marked uncompromised upon initialization, and which is set to compromised, if the password of uid is guessed correctly, either in an online (modeled by TestPwd interface) or offline way (modeled by StealPwdFile and OfflineTestPwd interface). If this is the case, then the attacker can determine the session key of all open sessions of the server with uid in which the attacker interfered, i.e. all sessions marked compromised or interrupted.

Delayed Extraction for Server Session: The second relaxation allows for late password tests, meaning the adversary's password guess can occur even after the session completes, i.e. after a NewKey query. In this scenario, the attacker learns whether the password guess is correct, but cannot influence the session key. This is modeled in $\mathcal{F}^-_{\mathsf{saPAKE}}$ by adding a new query Interrupt, which models an active attack against an S session in which the adversary does not immediately provide password guess pw^*. A session $(\mathsf{S}, \mathsf{ssid})$ subject to this attack is flagged with $\mathsf{dPT}(\mathsf{ssid}) := 1$, which allows the adversary to make a delayed password test on this session.

On Not Proving a Generic Compiler from Multi-User AKE. The OPAQUE protocol is a generic compiler from any AKE with the KCI property to a strong aPAKE. OPAQUE therefore uses an AKE subprotocol in a black box way and is secure if the underlying AKE realizes some functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$. Since the original OPAQUE protocol was in the single-user setting, also the functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$ is a single-user AKE functionality which is only accessed by one client C and one server S. When translating to the multi-user setting where the AKE building block has a shared state among multiple instantiations between different users, we would therefore need to develop a multi-user functionality for the AKE building block, show that the existing AKE protocols achieve this notion, and adjust the OPAQUE proof to work with the multi-user AKE functionality.

Instead, we opted for a more direct approach, and show security of the shared AKE-key OPAQUE when instantiated with the concrete 3DH protocol as AKE. The 3DH protocol is also one of the AKE protocol recommended in the IRTF draft. Our proof also has the benefit of giving a less abstract security proof for the OPAQUE protocol.

Building Blocks: 3DH and $\mathcal{F}_{\mathsf{OPRF}}$. In our protocol, we instantiate the AKE protocol with 3DH which works in a group $\mathbb{G}$ of prime oder q with generator g and uses the following algorithms:

$\mathsf{KGen}_{\mathsf{lt}}(1^\lambda)$: Pick $sk \xleftarrow{r} \mathbb{Z}_q$, set $pk := g^{sk}$. Output (sk, pk)

$\mathsf{KGen}_{\mathsf{eph}}(1^\lambda)$: Pick $x \xleftarrow{r} \mathbb{Z}_q$, set $X := g^x$. Output (x, X).

$\mathsf{KE}(sk_P, pk_{P'}, x_P, X_{P'})$: If $P = \mathsf{C}$ compute $k := H(X_P, X_{P'}, (pk_{P'})^{x_P}, (X_{P'})^{sk_P}, (X_{P'})^{x_P})$, else $k := H(X_{P'}, X_P, (X_{P'})^{sk_P}, (pk_{P'})^{x_P}, (X_{P'})^{x_P})$.

Our proof is further conducted in the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid world, which means that parties have access to an ideal functionality $\mathcal{F}_{\mathsf{OPRF}}$ which instantiates the OPRF protocol. We take the $\mathcal{F}_{\mathsf{OPRF}}$ functionality from [23] which uses a ticketing system to keep track of executions of the OPRF and only allows as many online evaluations for a client as there are sessions started by the server. The functionality further uses prefixes which allows to determine which sessions are actively interfered (if their prefixes do not match), and which are executed honestly.

Authenticated Encryption Properties. We recall the two security properties of the authenticated encryption scheme – *random-key robustness* and *encryption equivocability*– which were introduced in [23] to prove security of OPAQUE.

Random-key Robustness [23]: An authenticated encryption scheme $\Pi = (\mathsf{KGen}, \mathsf{AuthEnc}, \mathsf{AuthDec})$

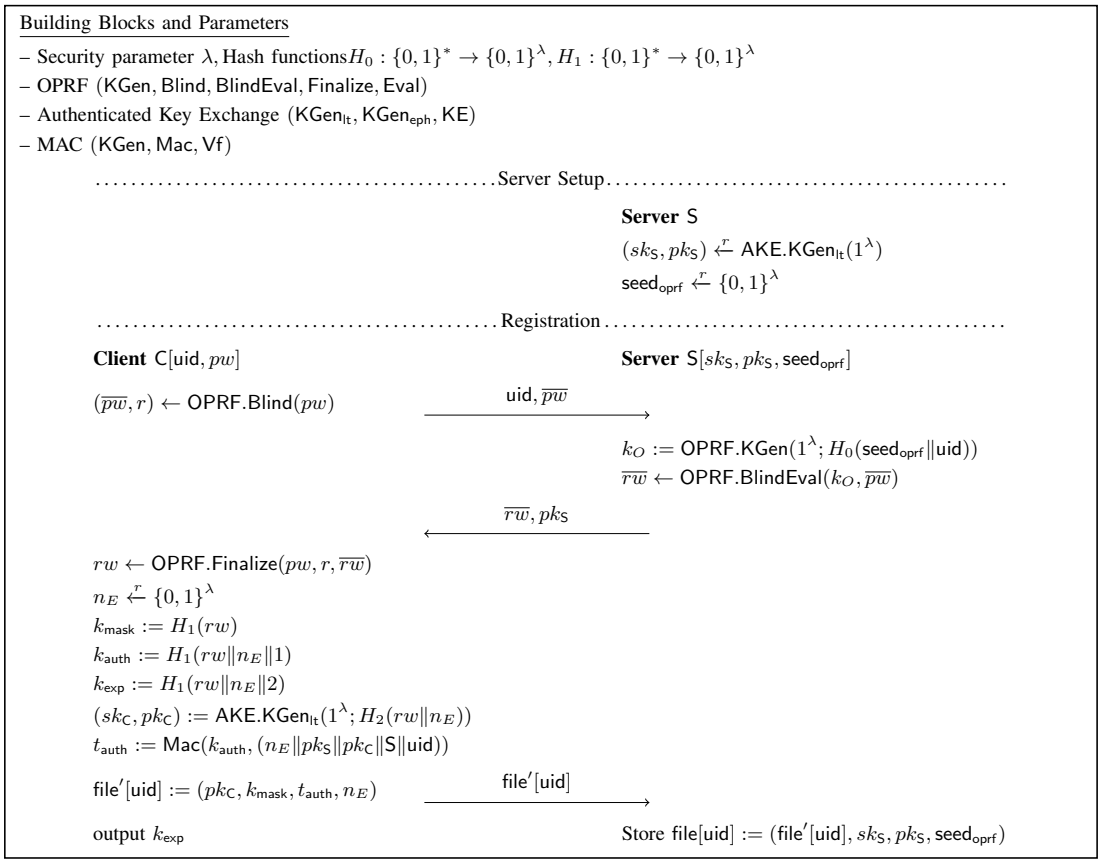

Figure 6: Setup and Registration Phase for Draft-OPAQUE

is called *random-key robust*, if for all efficient adversaries $\mathcal{A}$, the probability

$$\Pr[c \leftarrow \mathcal{A}(k_1, k_2) \text{ s.t. } \mathsf{AuthDec}(k_1, c) \neq \perp \wedge \mathsf{AuthDec}(k_2, c) \neq \perp]$$

where $k_1 \xleftarrow{r} \mathsf{KGen}(1^\lambda), k_2 \xleftarrow{r} \mathsf{KGen}(1^\lambda)$, is negligible in the security parameter λ.

Encryption Equivocability [23]: An authenticated encryption scheme $\Pi = (\mathsf{KGen}, \mathsf{AuthEnc}, \mathsf{AuthDec})$ is called equivocable if for any efficient adversary $\mathcal{A}$, there is an efficient stateful simulator $\mathsf{SIM}_{\mathsf{EQV}}$ such that the distinguishing advantage of $\mathcal{A}$'s view in the following two worlds is a negligible function of λ:

- The real world: $\mathcal{A}$ sends out a message m, and computes its final output given (c, k) produced as $k \xleftarrow{r} \mathsf{KGen}(1^\lambda)$ and $c := \mathsf{AuthEnc}(k, m)$.
- The ideal world: $\mathcal{A}$ sends out a message m, and computes its final output given (c, k) produced as $c \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(|m|)$ and $k \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(m)$.

Given these properties, we are now able to formalize the security of the shared AKE key OPAQUE variant, and provide the proof sketch.

Theorem 1. The Shared AKE Key OPAQUE protocol with 3DH securely realizes functionality $\mathcal{F}^{-}_{\mathsf{saPAKE}}$ in the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid model, if the Authenticated Encryption scheme is random-key robust and equivocable, if the Gap DH assumption holds in $\mathbb{G}$, and if F is a pseudorandom function.

Proof Sketch. Since we combine the OPAQUE protocol with the 3DH instantiation, our proof closely follows the proof of the OPAQUE protocol with individual keys in the single-user setting from [23], and the proof that 3DH is a secure AKE [17]. We give a high-level description of the simulation strategy. The detailed description of the simulator can be found in Fig. 12 and 13.

Recall that in a UC proof, the goal is to show that any attack that is possible in the real-world can be simulated indistinguishably by an adversary only interacting with the ideal functionality. The goal is thus to describe a simulator SIM only interacting with the ideal functionality $\mathcal{F}^{-}_{\mathsf{saPAKE}}$ (henceforth also referred to as just $\mathcal{F}$) which mimicks all aspects of the protocol's execution in a way that it is indistinguishable from the real-world execution of the protocol. Crucially, in the real-world, parties run the protocol on private inputs (i.e. the password/password file) while in the ideal world the simulator has to mimick the execution without access to these inputs. The simulation itself can be divided into three parts: (1) simulating the handling of password files and their compromise, (2) simulating the behaviour of an honest client and (3) simulating the behaviour of an honest server.

Password File Storage. First, we describe how the password file is simulated. Therein, the simulator SIM picks the server's global 3DH key-pair $sk_S \xleftarrow{r} \mathbb{Z}_q$, $pk_S := g^{sk_S}$, and for each user uid generates a virtual OPRF instance with $\mathsf{sid}_O = (\mathsf{sid} \| \mathsf{uid})$, instead of computing $rw = F_S(pw)$. Recall here that we use the multi-user setting with independent OPRF keys for each user which

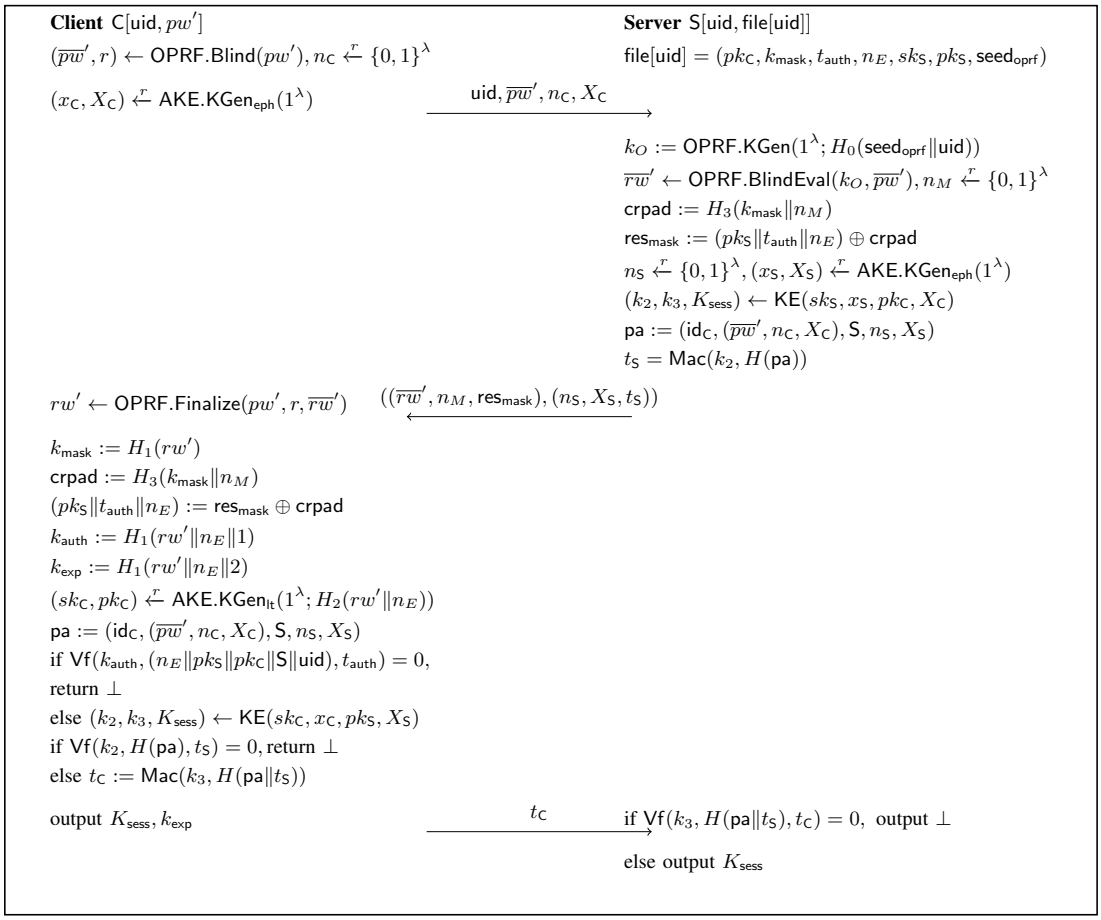

Figure 7: Login-Phase for Draft-OPAQUE

is modeled by instantiating the OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ with a different sid_O for each user uid. The simulator further generates the ciphertext c using the simulator $\mathsf{SIM}^{\mathsf{uid}}_{\mathsf{EQV}}$ given by the equivocability of the authenticated encryption scheme. If the adversary later compromises the password file of a user uid stored at the server by issuing a $(\mathsf{StealPwdFile}, \mathsf{sid}, \mathsf{uid})$ query, the simulator has to send $\mathsf{file}[\mathsf{sid}, \mathsf{uid}]$ to $\mathcal{A}$ such that the adversary cannot distinguish this from the password file created in the real execution of the protocol where the password file is defined as $\mathsf{file}[\mathsf{sid}, \mathsf{uid}] = (sk_{\mathsf{S}}, pk_{\mathsf{S}}, pk_{\mathsf{C}}, c)$ with $c = \mathsf{AuthEnc}(rw, (sk_{\mathsf{C}}, pk_{\mathsf{C}}, pk_{\mathsf{S}}))$ for $rw = F_{\mathsf{S}}(pw)$. Therefore, the simulator creates a 3DH key-pair for the client as $sk_{\mathsf{C}} \xleftarrow{r} \mathbb{Z}_q$, $pk_{\mathsf{C}} := g^{sk_{\mathsf{C}}}$ and sends the password file $\mathsf{file}[\mathsf{sid}, \mathsf{uid}] = (sk_{\mathsf{C}}, pk_{\mathsf{C}}, pk_{\mathsf{S}}, c)$ to the adversary while also granting the adversary access to offline evaluation queries to $\mathcal{F}_{\mathsf{OPRF}}$ for sid_O. This allows the adversary to perform offline dictionary attacks on the password file, by sending $(\mathsf{OfflineEval}, \mathsf{sid}_O, x)$ queries to $\mathcal{F}_{\mathsf{OPRF}}$. The simulator can detect these queries and send a corresponding $(\mathsf{OfflineTestPwd}, \mathsf{sid}, x)$ query to $\mathcal{F}^{-}_{\mathsf{saPAKE}}$. If the functionality responds with "wrong guess", SIM answers the OfflineEval query with a random $F_S(x) \xleftarrow{r} \{0,1\}^{\lambda}$. Otherwise, if $\mathcal{F}_{\mathsf{saPAKE}}$ replies with "correct guess", then SIM learns that the adversary's password guess x is equal to the password pw for which this uid was initialized, and in this case SIM "backpatches" c s.t. it matches to the password, i.e. it uses $\mathsf{SIM}^{\mathsf{uid}}_{\mathsf{EQV}}$ to compute a randomized password rw s.t. $c = \mathsf{AuthEnc}(rw, (sk_{\mathsf{C}}, pk_{\mathsf{C}}, pk_{\mathsf{S}}))$ and "programs" F s.t. $F_S(x) = pw$. By the OPRF security and the equivocability of the encryption scheme, the adversary's view of this interaction is identical to the real protocol. The simulator's handling of initialization and offline attacks is given in Fig. 12.

Login Phase. Regarding the login phase, we follow the notation of [23] and let i^* denote the function pointer used by the adversary $\mathcal{A}$ in $(\mathsf{RcvComplete}, \mathsf{sid}_O, \mathsf{ssid}, \mathsf{C}, i^*)$ for an honest C's OPRF session, and $(X^*_{\mathsf{S}}, c^*, t^*_1)$ to denote the message which $\mathcal{A}$ passes to C after OPRF evaluation. The details of the simulation procedure regarding the online phase are divided between Fig. 12, where we show how SIM simulates the $\mathcal{F}_{\mathsf{OPRF}}$ functionality towards $\mathcal{A}$, and the first message from C and S, and Fig. 13, where we show how SIM simulates the second messages from C and S and the random oracle H. However, the main ideas of the simulation can be explained by considering the two cases of the simulator: (1) simulate the behaviour of an honest client against an adversarial server (or a man-in-the-middle attacker who replaces protocol messages) and (2) simulate the behaviour of an honest server against an adversarial client (or a man-in-the-middle attacker who replaces protocol messages). In the following, we describe how SIM handles these two cases.

Simulating Honest Client. When simulating the honest client, the key observation is that C outputs $(\mathsf{ssid}, \perp)$ with overwhelming probability, except for either of the following three cases (corresponding to cases 1a), 1b) and

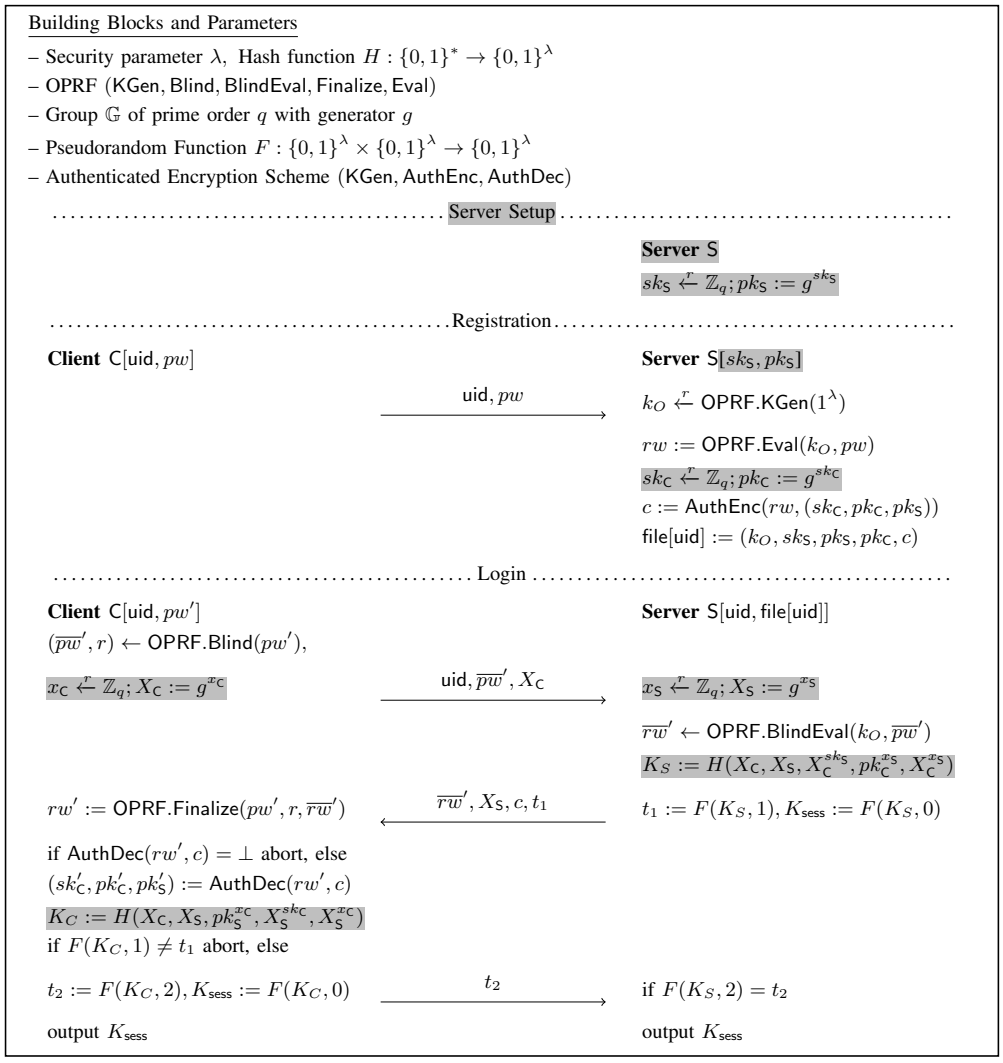

Figure 8: The multi-user OPAQUE [23] protocol with a shared server AKE key and with AKE protocol 3DH (changes highlighted in grey). We assume all messages and inputs to the hash functions are prefixed by the session identifier sid, and all messages specific to the login phase are further prefixed by ssid. We omit these values as a writing convention.

1c)) of the simulation of the second messages from C and S:

a) If $\mathcal{A}$ remains passive, meaning the simulator emulates both S and C, and the adversary simply relays the message $(\mathsf{ssid}, X_S, c, t_1)$ from honest S to C without interfering in the OPRF execution (thus ensuring both parties possess the same prfx), then C will output a key if the passwords of both parties match ($pw' = pw$). The simulator can verify this by issuing a NewKey query to $\mathcal{F}$, which, in case of mismatching passwords, yields a public delayed output of $(\mathsf{ssid}, \perp)$. If the passwords match, the simulator can substitute the 3DH-related messages with random ones. The security holds due to the GapDH assumption, as the attacker remains passive. This assumption is also used in the proof of 3DH to argue security against eavesdropping attacks.

In case $\mathcal{A}$ actively interferes, there are two possibilities which lead to C not outputting $(\mathsf{ssid}, \perp)$ at the end.

b) In the first scenario, the adversary may have compromised the user's password file on the server and uses it to impersonate the server against C. The simulator can detect this situation, as he can detect whether $c^* = c$ for the stolen password file, and in this case he knows that $\mathsf{file}[\mathsf{sid}, \mathsf{uid}] = (sk_S, pk_S, pk_C, c)$ and he additionally knows the secret key sk_C corresponding to pk_C since he created the password file. This enables the simulator to check whether t_1 is constructed correctly for this password file. Therefore he verifies whether $t_1 = F(K_C, 1)$ holds true, where $K_C = H(\mathsf{sid}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, pk_S^{x_C}, X_S^{sk_C}, X_S^{x_C})$. Notably, the simulator, having also created x_C and X_C for the client, can compute all these values. If this is the case, the simulator can issue an Impersonate query to $\mathcal{F}$ and in case of "correct guess" simulate the rest of the protocol using the correct K_C. It is important to note here that the compromise of the password file of a user uid' does not allow impersonating the server to user uid even though the server uses the same AKE key-pair for all users. This is because the attacker who only knows (sk_S, pk_S) and may have obtained the ciphertext c for uid by eavesdropping, does not know the AKE key-pair (sk_C, pk_C) to which the ciphertext c decrypts. Therefore the attacker can-

Public Components

– Random-key robust and equivocable authenticated encryption scheme $(\mathsf{AuthEnc}, \mathsf{AuthDec})$ with (2λ)-bit keys;
– Functionality $\mathcal{F}_{\mathsf{OPRF}}$ with output length parameter $\ell = 2\lambda$.
– Hash function $H : \{0,1\}^* \to \{0,1\}^\lambda$
– Group $\mathbb{G}$ of prime order q with generator g
– Pseudorandom Function $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$

Server Setup

– On the first command $(\mathsf{StorePwdFile}, \mathsf{sid}, \cdot, \cdot)$, S generates $sk_{\mathsf{S}} \xleftarrow{r} \mathbb{Z}_q$, sets $pk_{\mathsf{S}} := g^{sk_{\mathsf{S}}}$, and stores $(\mathsf{ServerState}, \mathsf{sid}, \mathsf{S}, sk_{\mathsf{S}}, pk_{\mathsf{S}})$.

Password Registration

1) On input $(\mathsf{StorePwdFile}, \mathsf{sid}, \mathsf{uid}, pw)$, S retrieves $(\mathsf{ServerState}, \mathsf{sid}, \mathsf{S}, sk_{\mathsf{S}}, pk_{\mathsf{S}})$, generates $sk_{\mathsf{C}} \xleftarrow{r} \mathbb{Z}_q$, sets $pk_{\mathsf{C}} := g^{sk_{\mathsf{C}}}$, $\mathsf{sid}_O := (\mathsf{sid} \| \mathsf{uid})$ and sends $(\mathsf{Init}, \mathsf{sid}_O)$ and $(\mathsf{OfflineEval}, \mathsf{sid}_O, \mathsf{S}, pw)$ to $\mathcal{F}_{\mathsf{OPRF}}$.
2) On $\mathcal{F}_{\mathsf{OPRF}}$'s response $(\mathsf{OfflineEval}, \mathsf{sid}_O, rw)$, compute $c := \mathsf{AuthEnc}(rw, (sk_{\mathsf{C}}, pk_{\mathsf{C}}, pk_{\mathsf{S}}))$ and record $\mathsf{file}[\mathsf{sid}, \mathsf{uid}] := (sk_{\mathsf{S}}, pk_{\mathsf{S}}, pk_{\mathsf{C}}, c)$.

Server Compromise

– On $(\mathsf{StealPwdFile}, \mathsf{sid}, \mathsf{S}, \mathsf{uid})$ from $\mathcal{A}$, S retrieves $\mathsf{file}[\mathsf{sid}, \mathsf{uid}]$ and sends it to $\mathcal{A}$.

Password Authentication and Key Generation

1) On $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \mathsf{uid}, pw')$, C sets $\mathsf{sid}_O := (\mathsf{sid} \| \mathsf{uid})$, sends $(\mathsf{Eval}, \mathsf{sid}_O, \mathsf{ssid}, \mathsf{S}, pw')$ to $\mathcal{F}_{\mathsf{OPRF}}$ and records $\mathcal{F}_{\mathsf{OPRF}}$'s response $(\mathsf{Prefix}, \mathsf{ssid}, \mathsf{prfx})$. C also generates $x_{\mathsf{C}} \xleftarrow{r} \mathbb{Z}_q$, sets $X_{\mathsf{C}} := g^{x_{\mathsf{C}}}$ and sends $(\mathsf{ssid}, X_{\mathsf{C}})$ to S.
2) On $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{uid})$ and upon receiving $(\mathsf{ssid}, X_{\mathsf{C}})$ from C, S parses $\mathsf{file}[\mathsf{sid}, \mathsf{uid}] = (sk_{\mathsf{S}}, pk_{\mathsf{S}}, pk_{\mathsf{C}}, c)$, sets $\mathsf{sid}_O := (\mathsf{sid} \| \mathsf{uid})$ and sends $(\mathsf{SndrComplete}, \mathsf{sid}_O, \mathsf{ssid})$ to $\mathcal{F}_{\mathsf{OPRF}}$. On $\mathcal{F}_{\mathsf{OPRF}}$'s response $(\mathsf{Prefix}, \mathsf{ssid}, \mathsf{prfx}')$, S generates $x_{\mathsf{S}} \xleftarrow{r} \mathbb{Z}_q$, sets $X_{\mathsf{S}} := g^{x_{\mathsf{S}}}$, computes $K_{\mathsf{S}} := H(\mathsf{sid}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}', X_{\mathsf{C}}^{sk_{\mathsf{S}}}, pk_{\mathsf{C}}^{x_{\mathsf{S}}}, X_{\mathsf{C}}^{x_{\mathsf{S}}})$, records $(\mathsf{ssid}, K_{\mathsf{S}})$, computes $t_1 := F(K_{\mathsf{S}}, 1)$ and sends $(\mathsf{ssid}, X_{\mathsf{S}}, c, t_1)$ to C.
3) On $(\mathsf{Eval}, \mathsf{sid}_O, \mathsf{ssid}, rw')$ from $\mathcal{F}_{\mathsf{OPRF}}$ and $(\mathsf{ssid}, X_{\mathsf{S}}, c, t_1)$ from S, C decrypts $m := \mathsf{AuthDec}(rw', c)$. If m can be parsed as $(sk'_{\mathsf{C}}, pk'_{\mathsf{C}}, pk'_{\mathsf{S}})$, then C retrieves $(\mathsf{Prefix}, \mathsf{ssid}, \mathsf{prfx})$, computes $K_{\mathsf{C}} := H(\mathsf{sid}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, pk_{\mathsf{S}}^{x_{\mathsf{C}}}, X_{\mathsf{S}}^{sk_{\mathsf{C}}}, X_{\mathsf{S}}^{x_{\mathsf{C}}})$, and if $t_1 = F(K_{\mathsf{C}}, 1)$, sets $t_2 := F(K_{\mathsf{C}}, 2)$, sends (ssid, t_2) to S, computes $K_{\mathsf{sess}} := F(K_{\mathsf{C}}, 0)$ and outputs $(\mathsf{ssid}, K_{\mathsf{sess}})$. Else outputs $(\mathsf{ssid}, \perp)$.
4) On (ssid, t_2) from C, S retrieves $(\mathsf{ssid}, K_{\mathsf{S}})$. If $t_2 = F(K_{\mathsf{S}}, 2)$, S computes $K_{\mathsf{sess}} := F(K_{\mathsf{S}}, 0)$ and outputs $(\mathsf{ssid}, K_{\mathsf{sess}})$, else outputs $(\mathsf{ssid}, \perp)$.

Figure 9: The shared AKE key OPAQUE variant in the UC-compliant notation where the AKE building block is instantiated with the AKE protocol 3DH.

not compute the correct t_1 which is necessary to impersonate the server to uid.

c) If $\mathcal{A}$ actively interferes without using a stolen password file (in which case it holds that $c^* \neq c$), then C will only output a session key, if c^* decrypts to some $(sk'_{\mathsf{C}}, pk'_{\mathsf{C}}, pk'_{\mathsf{S}})$ under key $rw' = F_{i^*}(pw')$ such that t_1 is computed correctly w.r.t. these values. In other words, $t_1 = F(K_{\mathsf{C}}, 1)$ should hold, where $K_{\mathsf{C}} = H(\mathsf{sid}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, pk_{\mathsf{S}}^{\prime x_{\mathsf{C}}}, X_{\mathsf{S}}^{sk'_{\mathsf{C}}}, X_{\mathsf{S}}^{x_{\mathsf{C}}})$.
But in this case, the adversary must have computed $rw' = F_{i^*}(pw')$ through three possible methods: (i) in an online OPRF instance between $\mathcal{A}$ and S if $i^* = \mathsf{S}$, (ii) via offline computation $(\mathsf{OfflineEval}, \mathsf{sid}_O, i^*, pw')$ if $i^* = \mathsf{S}$ and S is compromised or corrupted, or (iii) via offline computation $(\mathsf{OfflineEval}, \mathsf{sid}_O, i^*, pw')$ if $i^* \neq \mathsf{S}$. In all of these cases the adversary can choose $(sk^*_{\mathsf{C}}, pk^*_{\mathsf{C}}, pk^*_{\mathsf{S}})$ and set the ciphertext $c^* := \mathsf{AuthEnc}(rw', (sk^*_{\mathsf{C}}, pk^*_{\mathsf{C}}, pk^*_{\mathsf{S}}))$. However, SIM, who manages the OPRF functionality sees the adversary's OPRF queries, and will learn the same information as well. Thus, it can check if there was a query x such that c^* decrypts to a valid client-key pair which results in a correct t_1. Thus, the simulator can extract the password and test it against the client's. If the guess is correct, the simulator proceeds to simulate the protocol accordingly using the correct K_{C}. The random-key robustness of authenticated encryption is crucial in this context, ensuring that c^* decrypts to a valid message $m \neq \perp$ for at most one key (with overwhelming probability), ensuring that the server is limited to one password guess per interaction.

Simulating Honest Server. When simulating the server towards a client, the server always uses the same ciphertext c initialized for uid, but instead of deriving t_1 from the real K_{S}, a random $K_{\mathsf{S}} \xleftarrow{r} \{0,1\}^\lambda$ is chosen. Nevertheless, this K_{S} can later be backpatched to match the client's computation of K_{C} if both parties use the same password. This can be done using the simulator's power of programming the random oracle. Thus, the simulator checks when answering a query to H whether the client instance has been compromised (in which case the adversary knows the client's keys $(sk_{\mathsf{C}}, pk_{\mathsf{C}}, pk_{\mathsf{S}})$) and whether these keys are used in computing the 3DH pre-secret $(X_{\mathsf{C}}^{sk_{\mathsf{S}}}, pk_{\mathsf{C}}^{x_{\mathsf{S}}}, X_{\mathsf{C}}^{x_{\mathsf{S}}})$ which is also an input to the random oracle. The simulator can check this with his knowledge of $(sk_{\mathsf{S}}, pk_{\mathsf{S}}, pk_{\mathsf{C}}, x_{\mathsf{S}}, X_{\mathsf{C}})$. If this is the case, the random oracle is backpatched to match K_{S} chosen by the server, otherwise a random K_{S} is chosen as output.

A client instance uid can become compromised in two ways: (1) Through online evaluations to $\mathcal{F}_{\mathsf{OPRF}}$ with the correct password or (2) through offline evaluations to $\mathcal{F}_{\mathsf{OPRF}}$ with the correct password, if the password file of uid is stolen. The simulator can detect these queries by managing the $\mathcal{F}_{\mathsf{OPRF}}$ functionality, and test the password against the server's either via a TestPwd query (in case 1) or via an OfflineTestPwd query (in case 2). Both of these queries will, in case of success, trigger $\mathcal{F}$ to mark the file as compromised, which allows the simulator to determine the session key for all compro-

Password Registration

– On $(\mathsf{StorePwdFile}, \mathsf{sid}, \mathsf{uid}, pw)$ from S, create record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked fresh, set $\mathsf{flag}[\mathsf{sid}, \mathsf{uid}] := \mathsf{uncompromised}$.

Stealing Password Data

– On $(\mathsf{StealPwdFile}, \mathsf{sid}, \mathsf{S}, \mathsf{uid})$ from $\mathcal{A}$, if there is no record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$, return "no password file". Otherwise mark this record stolen.

– On $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{S}, \mathsf{uid}, pw^*)$ from $\mathcal{A}$, do:

  – If $\exists$ record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked stolen, do the following: If $pw^* = pw$ return "correct guess" to $\mathcal{A}$ and set $\mathsf{flag}[\mathsf{sid}, \mathsf{uid}] := \mathsf{compromised}$, else return "wrong guess".

Password Authentication

– On $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \mathsf{uid}, pw')$ from C, if there is no record $\langle \mathsf{ssid}, \mathsf{C}, \dots \rangle$ then record $\langle \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \mathsf{uid}, pw', \mathtt{cl} \rangle$ marked fresh and send $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \mathsf{uid})$ to $\mathcal{A}$.

– On $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{uid})$ from S, if there is no record $\langle \mathsf{ssid}, \mathsf{S}, \dots \rangle$ then retrieve record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$, and if it exists then create record $\langle \mathsf{ssid}, \mathsf{S}, \mathsf{C}, \mathsf{uid}, pw, \mathtt{sr} \rangle$ marked fresh and send $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \mathsf{C}, \mathsf{uid})$ to $\mathcal{A}$.

Active Session Attacks

– On $(\mathsf{Interrupt}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S})$, if there is a record $(\mathsf{ssid}, \mathsf{S}, \mathsf{C}, \mathsf{uid}, pw, \mathtt{sr})$ marked fresh, mark it interrupted and set $\mathsf{dPT}(\mathsf{ssid}) := 1$.

– On $(\mathsf{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, pw^*)$ from $\mathcal{A}$, if there is a record $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \mathsf{uid}, pw, \mathsf{role} \rangle$ then do:

  – If the record is 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".

  – If $\mathsf{P} = \mathsf{S}$ and $\mathsf{dPT}(\mathsf{ssid}) = 1$, then set $\mathsf{dPT}(\mathsf{ssid}) := 0$ and if $pw^* = pw$ then return "correct guess" to $\mathcal{A}$, else return "wrong guess".

  In either case, if $\mathsf{P} = \mathsf{S}$ and $pw^* = pw$, set $\mathsf{flag}[\mathsf{sid}, \mathsf{uid}] := \mathsf{compromised}$.

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

Key Generation and Authentication

– On $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, K^*)$ from $\mathcal{A}$, if there is a record $\mathsf{rec} = \langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \mathsf{uid}, pw, \mathsf{role} \rangle$ not marked completed, then do:

  – If rec is compromised, or ($\mathsf{P} = \mathsf{S}$, rec is interrupted and $\mathsf{flag}[\mathsf{sid}, \mathsf{uid}] = \mathsf{compromised}$) set $K \leftarrow K^*$;

  – Else if $\mathsf{role} = \mathtt{cl}$, rec is fresh, there is a record $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \mathsf{uid}, pw, \mathtt{sr} \rangle$ s.t. $\mathcal{F}_{\mathsf{aPAKE}}$ sent $(\mathsf{sid}, \mathsf{ssid}, K')$ to P' while that record was marked fresh, set $K \leftarrow K'$;

  – Else if $\mathsf{role} = \mathtt{sr}$, rec is fresh, there is a record $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \mathsf{uid}, pw, \mathtt{cl} \rangle$ 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 $(\mathsf{sid}, \mathsf{ssid}, \perp)$ to P, otherwise provide private delayed output $(\mathsf{sid}, \mathsf{ssid}, K)$ to P.

Figure 10: The relaxed multi-user ideal functionality $\mathcal{F}^{-}_{\mathsf{saPAKE}}$ (additions to $\mathcal{F}_{\mathsf{saPAKE}}$ highlighted) used for the proof of security of OPAQUE [23].

mised or interrupted sessions for uid. Therefore, when receiving message (ssid, t_2^*), the simulator checks whether $t_2^* = F(K_\mathsf{S}, 2)$ and sends a NewKey query to $\mathcal{F}$ either with $K_{\mathsf{sess}} := F(K_\mathsf{S}, 0)$ or with $\perp$. In case the server's password matches the client's in any subsession started by the uid, and if the client has computed t_2^* correctly, this will output the correct K_{sess} to the server ensured by the programming of the random oracle.

Public Parameters: PRF output length ℓ, polynomial in security parameter λ.
Conventions: For every i, x value $F_{\mathsf{sid},i}(x)$ is initially undefined, and if undefined value $F_{\mathsf{sid},i}(x)$ is referenced then $\mathcal{F}_{\mathsf{OPRF}}$ assigns $F_{\mathsf{sid},i}(x) \xleftarrow{r} \{0,1\}^{\ell}$.
Initialization:
On message $(\mathsf{Init}, \mathsf{sid})$ from party S, if this is the first Init message for sid , set $\mathsf{tx} = 0$ and send $(\mathsf{Init}, \mathsf{sid}, S)$ to $\mathcal{A}$. From now on use tag "S" to denote the unique entity which sent the Init message for the session identifier sid. (Ignore all subsequent Init messages for sid.)
Server Compromise:
On $(\mathsf{Compromise}, \mathsf{sid}, \mathsf{S})$ from $\mathcal{A}$, declare server S as compromised. (If S is corrupted then it is declared compromised from the beginning.)
Note: Message $(\mathsf{Compromise}, \mathsf{sid}, \mathsf{S})$ *requires permission from the environment.*
Offline Evaluation:
On $(\mathsf{OfflineEval}, \mathsf{sid}, i, x)$ from $\mathsf{P} \in \{\mathsf{S}, \mathcal{A}\}$, send $(\mathsf{OfflineEval}, \mathsf{sid}, F_{\mathsf{sid},i}(x))$ to P if any of the following hold: (i) S is corrupted, (ii) $\mathsf{P} = \mathsf{S}$ and $i = \mathsf{S}$, (iii) $\mathsf{P} = \mathcal{A}$ and $i \neq \mathsf{S}$, (iv) $\mathsf{P} = \mathcal{A}$ and S is compromised.
Online Evaluation:

- On $(\mathsf{Eval}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}', x)$ from $\mathsf{P} \in \{\mathsf{C}, \mathcal{A}\}$, send $(\mathsf{Eval}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \mathsf{S}')$ to $\mathcal{A}$. On prfx from $\mathcal{A}$, ignore this message if prfx was used before. Else record $\langle \mathsf{ssid}, \mathsf{P}, x, \mathsf{prfx} \rangle$ and send $(\mathsf{Prefix}, \mathsf{sid}, \mathsf{ssid}, \mathsf{prfx})$ to P.
- On $(\mathsf{SndrComplete}, \mathsf{sid}, \mathsf{ssid}')$ from S, send $(\mathsf{SndrComplete}, \mathsf{sid}, \mathsf{ssid}', \mathsf{S})$ to $\mathcal{A}$. On prfx' from $\mathcal{A}$, send $(\mathsf{Prefix}, \mathsf{sid}, \mathsf{ssid}', \mathsf{prfx}')$ to S. If there is a record $\langle \mathsf{ssid}, \mathsf{P}, x, \mathsf{prfx} \rangle$ for $\mathsf{P} \neq \mathcal{A}$ and $\mathsf{prfx} = \mathsf{prfx}'$, change it to $\langle \mathsf{ssid}, \mathsf{P}, x, OK \rangle$, else set $\mathsf{tx}++$.
- On $(\mathsf{RcvComplete}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, i)$ from $\mathcal{A}$, ignore this message if there is no record $\langle \mathsf{ssid}, \mathsf{P}, x, \mathsf{prfx} \rangle$ or if $(i = \mathsf{S}, \mathsf{tx} = 0$, and $\mathsf{prfx} \neq OK)$. Else send $(\mathsf{Eval}, \mathsf{sid}, \mathsf{ssid}, F_{\mathsf{sid},i}(x))$ to P, and if $(i = \mathsf{S}$ and $\mathsf{prfx} \neq OK)$ then set $\mathsf{tx}--$.

Figure 11: Functionality $\mathcal{F}_{\mathsf{OPRF}}$ with adaptive compromise from [23].

Initialization

Set $\mathsf{tx} := 0$. Initialize empty table T_H to manage hash queries. Initialize function family $\{F_i\}$ s.t. for all (i, x), including $i = \mathsf{S}$, $F_i(x)$ is undefined. Whenever SIM references undefined value $F_i(x)$ below, set $F_i(x) \xleftarrow{r} \{0,1\}^\lambda$. Pick $sk_\mathsf{S} \xleftarrow{r} \mathbb{Z}_q$, set $pk_\mathsf{S} := g^{sk_\mathsf{S}}$. For the first SvrSession or StealPwdFile query for uid, set $c \leftarrow \mathsf{SIM}^{\mathsf{uid}}_{\mathsf{EQV}}(k_{pr} + 2k_{pb})$, where $k_{pr} = |\mathbb{Z}_q|$, $k_{pb} = |\mathbb{G}|$ are the lengths of private/public keys in 3DH, and record $\mathsf{file}[\mathsf{sid}, \mathsf{uid}] := (sk_\mathsf{S}, pk_\mathsf{S}, \perp, c)$.

Stealing Password Data and Offline Queries

1) On $(\mathsf{Compromise}, \mathsf{sid}_O)$ aimed at $\mathcal{F}_{\mathsf{OPRF}}$ with $\mathsf{sid}_O = (\mathsf{sid}\|\mathsf{uid})$, and $(\mathsf{StealPwdFile}, \mathsf{sid}, \mathsf{S}, \mathsf{uid})$ aimed at S from $\mathcal{A}$ (we assume $\mathcal{A}$ sends these commands together), send $(\mathsf{StealPwdFile}, \mathsf{sid}, \mathsf{S}, \mathsf{uid})$ to $\mathcal{F}$. If $\mathcal{F}$ returns "no password file", pass this message to $\mathcal{A}$ on behalf of S. Otherwise declare $\mathsf{file}[\mathsf{sid}, \mathsf{uid}]$ as stolen, pick $sk_\mathsf{C} \xleftarrow{r} \mathbb{Z}_q$, set $pk_\mathsf{C} := g^{sk_\mathsf{C}}$, reset $\mathsf{file}[\mathsf{sid}, \mathsf{uid}] := (sk_\mathsf{S}, pk_\mathsf{S}, pk_\mathsf{C}, c)$ and send it to $\mathcal{A}$ on behalf of S. Keep record $(\mathsf{stolen}, \mathsf{sid}, \mathsf{uid}, sk_\mathsf{C}, pk_\mathsf{C})$.
2) On $(\mathsf{OfflineEval}, \mathsf{sid}_O, i^*, x)$ from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{OPRF}}$ with $\mathsf{sid}_O = (\mathsf{sid}\|\mathsf{uid})$, do the following:
   - If $i^* = \mathsf{S}$ and $\mathsf{file}[\mathsf{sid}, \mathsf{uid}]$ is not marked stolen, ignore this message.
   - If $i^* = \mathsf{S}$ and $\mathsf{file}[\mathsf{sid}, \mathsf{uid}]$ is marked stolen, send $(\mathsf{OfflineTestPwd}, \mathsf{sid}, \mathsf{S}, \mathsf{uid}, x)$ to $\mathcal{F}$. If $\mathcal{F}$ returns "correct guess", retrieve $(\mathsf{stolen}, \mathsf{sid}, \mathsf{uid}, sk_\mathsf{C}, pk_\mathsf{C})$ and set $rw \leftarrow \mathsf{SIM}^{\mathsf{uid}}_{\mathsf{EQV}}(sk_\mathsf{C}, pk_\mathsf{C}, pk_\mathsf{S})$ and $F_\mathsf{S}(x) := rw$. Keep record $(\mathsf{compromised}, \mathsf{sid}, \mathsf{uid}, sk_\mathsf{C}, pk_\mathsf{C}, pw)$.

   Finally, send $(\mathsf{OfflineEval}, \mathsf{sid}_O, F_{i^*}(x))$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{OPRF}}$.

OPRF Evaluation + First Message from C and S

1) On $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \mathsf{uid})$ from $\mathcal{F}$, set $\mathsf{sid}_O := (\mathsf{sid}\|\mathsf{uid})$, send $(\mathsf{Eval}, \mathsf{sid}_O, \mathsf{ssid}, \mathsf{C}, \mathsf{S})$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{OPRF}}$. On prfx from $\mathcal{A}$, record $(\mathsf{uid}, \mathsf{ssid}, \mathsf{C}, \mathsf{prfx})$ if prfx is new, else reject. Pick $x_\mathsf{C} \xleftarrow{r} \mathbb{Z}_q$, set $X_\mathsf{C} := g^{x_\mathsf{C}}$, send $(\mathsf{ssid}, X_\mathsf{C})$ to S on behalf of C and keep record $(\mathsf{hbsC}, \mathsf{C}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, x_\mathsf{C}, X_\mathsf{C})$ (needed for AKE simulation).
2) On $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \mathsf{C}, \mathsf{uid})$ from $\mathcal{F}$, and upon S receiving $(\mathsf{ssid}, X_\mathsf{C})$ from $\mathcal{A}$, retrieve $\mathsf{file}[\mathsf{sid}, \mathsf{uid}] = (sk_\mathsf{S}, pk_\mathsf{S}, \cdot, c)$, set $\mathsf{sid}_O := (\mathsf{sid}\|\mathsf{uid})$, send $(\mathsf{SndrComplete}, \mathsf{sid}_O, \mathsf{ssid}, \mathsf{S})$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{OPRF}}$, and given $\mathcal{A}$'s response prfx' do the following in order:
   a) If there is record $(\mathsf{uid}, \mathsf{ssid}, \mathsf{C}, \mathsf{prfx}')$, then replace it with $(\mathsf{uid}, \mathsf{ssid}, \mathsf{C}, OK)$; Else record $(\mathsf{uid}, \mathsf{ssid}, \mathsf{act})$, set $\mathsf{tx}{+}{+}$, send $(\mathsf{Interrupt}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S})$ to $\mathcal{F}$.
   b) Pick $x_\mathsf{S} \xleftarrow{r} \mathbb{Z}_q$, set $X_\mathsf{S} := g^{x_\mathsf{S}}$, pick $K_\mathsf{S} \xleftarrow{r} \{0,1\}^\lambda$, set $t_1 := F(K_\mathsf{S}, 1)$, send $(\mathsf{ssid}, X_\mathsf{S}, c, t_1)$ to $\mathcal{A}$ on behalf of S and record $(\mathsf{hbsS}, \mathsf{S}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}', X_\mathsf{C}, x_\mathsf{S}, K_\mathsf{S}, X_\mathsf{S}, c, t_1)$.
3) On $(\mathsf{RcvComplete}, \mathsf{sid}_O, \mathsf{ssid}, \mathsf{C}, i^*)$ from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{OPRF}}$, retrieve $(\mathsf{uid}, \mathsf{ssid}, \mathsf{C}, \mathsf{prfx})$ (ignore the message if such record not found) and do in order:
   a) If $i^* = \mathsf{S}$, $\mathsf{file}[\mathsf{sid}, \mathsf{uid}]$ is not stolen, and there is no record $(\mathsf{uid}, \mathsf{ssid}, \mathsf{C}, OK)$, then do: Ignore this message if $\mathsf{tx} = 0$, else set $\mathsf{tx}{-}{-}$.
   b) Augment record $(\mathsf{uid}, \mathsf{ssid}, \mathsf{C}, \mathsf{prfx})$ to $(\mathsf{uid}, \mathsf{ssid}, \mathsf{C}, \mathsf{prfx}, i^*)$.
4) On $(\mathsf{Eval}, \mathsf{sid}_O, \mathsf{ssid}, \mathsf{S}, x)$ followed by $(\mathsf{RcvComplete}, \mathsf{sid}_O, \mathsf{ssid}, \mathcal{A}, i^*)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{OPRF}}$ (string prfx chosen by $\mathcal{A}$ for this Eval can be ignored), send $(\mathsf{Eval}, \mathsf{sid}_O, \mathsf{ssid}, \mathcal{A}, \mathsf{S})$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{OPRF}}$ and do in order:
   a) If $i^* \neq \mathsf{S}$, then send $(\mathsf{Eval}, \mathsf{sid}_O, \mathsf{ssid}, F_{i^*}(x))$ to $\mathcal{A}$.
   b) If $i^* = \mathsf{S}$ and $\mathsf{tx} > 0$, but there is no record $(\mathsf{uid}, \mathsf{ssid}', \mathsf{act})$ then output halt.
   c) If $i^* = \mathsf{S}$ and there are some records $(\mathsf{uid}, \mathsf{ssid}', \mathsf{act})$ then do in order:
      i) If there is record $(\mathsf{uid}, \mathsf{ssid}', \mathsf{act})$ which is not marked completed then choose ssid' of any such record, but if all records $(\mathsf{uid}, \mathsf{ssid}', \mathsf{act})$ are marked completed then choose ssid' of any of those.
      ii) Ignore this message if $\mathsf{tx} = 0$, else set $\mathsf{tx}{-}{-}$ and send $(\mathsf{TestPwd}, \mathsf{sid}, \mathsf{ssid}', \mathsf{S}, x)$ to $\mathcal{F}$.
      iii) If $\mathcal{F}$ returns "correct guess": retrieve $(\mathsf{stolen}, \mathsf{sid}, \mathsf{uid}, sk_\mathsf{C}, pk_\mathsf{C})$ if it exists, otherwise generate $sk_\mathsf{C} \xleftarrow{r} \mathbb{Z}_q$ and set $pk_\mathsf{C} := g^{sk_\mathsf{C}}$. Set $rw \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(sk_\mathsf{C}, pk_\mathsf{C}, pk_\mathsf{S})$ and $F_\mathsf{S}(x) := rw$. Keep record $(\mathsf{compromised}, \mathsf{sid}, \mathsf{uid}, sk_\mathsf{C}, pk_\mathsf{C}, pw)$.
      iv) Send $(\mathsf{Eval}, \mathsf{sid}_O, \mathsf{ssid}, F_\mathsf{S}(x))$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{OPRF}}$, and modify the chosen record $(\mathsf{uid}, \mathsf{ssid}', \mathsf{act})$ into $(\mathsf{uid}, \mathsf{ssid}', \mathsf{used})$.

Figure 12: Simulator SIM showing that multi-user shared 3DH key UC-realizes $\mathcal{F} = \mathcal{F}^{-}_{\mathsf{saPAKE}}$, part 1: Initialization, Offline Attacks, OPRF Evaluation and First Message from C and S.

Second Message from C and S

1) For any ssid, as soon as $(\mathsf{uid}, \mathsf{ssid}, \mathsf{C}, \mathsf{prfx})$ is augmented to $(\mathsf{uid}, \mathsf{ssid}, \mathsf{C}, \mathsf{prfx}, i^*)$ and $\mathcal{A}$ sends $(\mathsf{ssid}, X^*_{\mathsf{S}}, c^*, t^*_1)$ to C, retrieve $(\mathsf{hbsC}, \mathsf{C}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, x_{\mathsf{C}}, X_{\mathsf{C}})$ and do one of the following:
   a) If there is a record $(\mathsf{hbsS}, \mathsf{S}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, X_{\mathsf{C}}, x_{\mathsf{S}}, K_{\mathsf{S}}, X_{\mathsf{S}}, c, t_1)$ with $(X^*_{\mathsf{S}}, c^*, t^*_1, i^*) = (Xs, c, t_1, \mathsf{S})$, then send $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \top)$ to $\mathcal{F}$. If $\mathcal{F}$ sends public delayed output $(\mathsf{ssid}, \perp)$, deliver it to C and set $t_2 := \perp$. Else, set $t_2 := F(K_{\mathsf{S}}, 2)$ and send (ssid, t_2) to $\mathcal{A}$ on behalf of C.
   b) Otherwise if $\mathsf{file}[\mathsf{sid}, \mathsf{uid}] = (sk_{\mathsf{S}}, pk_{\mathsf{S}}, \cdot, c)$ is marked stolen, $c^* = c$, there is record $(\mathsf{stolen}, \mathsf{sid}, \mathsf{uid}, sk_{\mathsf{C}}, pk_{\mathsf{C}})$ and $t_1 = F(K_{\mathsf{C}}, 1)$ for $K_{\mathsf{C}} = H(\mathsf{sid}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, pk_{\mathsf{S}}^{x_{\mathsf{C}}}, X_{\mathsf{S}}^{sk_{\mathsf{C}}}, X_{\mathsf{S}}^{x_{\mathsf{C}}})$ then send $(\mathsf{Impersonate}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \mathsf{S}, \mathsf{uid})$ to $\mathcal{F}$. Upon answer "correct guess", set $(K_{\mathsf{sess}}, t_2) := (F(K_{\mathsf{C}}, 0), F(K_{\mathsf{C}}, 2))$, else set $(K_{\mathsf{sess}}, t_2) := (\perp, \perp)$. In both cases, send $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, K_{\mathsf{sess}})$ to $\mathcal{F}$ and (ssid, t_2) to $\mathcal{A}$ on behalf of C.
   c) Otherwise for every x s.t. $y = F_{i^*}(x)$ is defined, check if $\mathsf{AuthDec}(y, c^*)$ output parses as $(sk'_{\mathsf{C}}, pk'_{\mathsf{C}}, pk'_{\mathsf{S}})$ such that $t_1 = F(K_{\mathsf{C}}, 1)$ for $K_{\mathsf{C}} = H(\mathsf{sid}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, (pk'_{\mathsf{S}})^{x_{\mathsf{C}}}, (X^*_{\mathsf{S}})^{sk'_{\mathsf{C}}}, (X^*_{\mathsf{S}})^{x_{\mathsf{C}}})$, and do one of the following:
      i) If there is no such x, send $(\mathsf{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \perp)$ followed by $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \perp)$ to $\mathcal{F}$, set $t_2 := \perp$ and send (ssid, t_2) to $\mathcal{A}$ on behalf of C.
      ii) If there are more than one such x's, output halt and abort.
      iii) If there is a unique such x, send $(\mathsf{TestPwd}, \mathsf{sid}, \mathsf{ssid}, C, x)$ to $\mathcal{F}$.
         - If $\mathcal{F}$ replies "wrong guess", send $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, \perp)$ to $\mathcal{F}$, set $t_2 := \perp$ and send (ssid, t_2) to $\mathcal{A}$ on behalf of C.
         - If $\mathcal{F}$ replies "correct guess", set $(K_{\mathsf{sess}}, t_2) := (F(K_{\mathsf{C}}, 0), F(K_{\mathsf{C}}, 2))$. Send $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{C}, K_{\mathsf{sess}})$ to $\mathcal{F}$ and (ssid, t_2) to $\mathcal{A}$ on behalf of C.
2) When $\mathcal{A}$ sends (ssid, t^*_2) to S, retrieve $(\mathsf{hbsS}, \mathsf{S}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, X_{\mathsf{C}}, x_{\mathsf{S}}, K_{\mathsf{S}}, X_{\mathsf{S}}, c, t_1)$ and set $K_{\mathsf{sess}} := F(K_{\mathsf{S}}, 0)$. If $t^*_2 = F(K_{\mathsf{S}}, 2)$, send $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, K_{\mathsf{sess}})$ to $\mathcal{F}$, else send $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \perp)$.

Simulation of random oracle H

On input $(\mathsf{sid}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, A, B, C)$ to H: If there exists record $(T_H, \mathsf{sid}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, A, B, C, K)$, respond with K. Else do:

- if there exist records $(\mathsf{hbsS}, \mathsf{S}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, X_{\mathsf{C}}, x_{\mathsf{S}}, K_{\mathsf{S}}, X_{\mathsf{S}}, c, t_1)$ and $(\mathsf{compromised}, \mathsf{sid}, \mathsf{uid}, sk_{\mathsf{C}}, pk_{\mathsf{C}}, pw)$, check if $A = (X_{\mathsf{C}})^{sk_{\mathsf{S}}}$, $B = (pk_{\mathsf{C}})^{x_{\mathsf{S}}}$ and $C = (X_{\mathsf{C}})^{x_{\mathsf{S}}}$. If this is the case, set $h := K_{\mathsf{S}}$. Otherwise, pick $h \xleftarrow{r} \{0,1\}^{\lambda} \setminus \{K_{\mathsf{S}}\}$.

If there is a record $(T_H, \mathsf{sid}, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, h)$, abort; else respond with h and record $(T_H, \mathsf{sid}, \mathsf{uid}, \mathsf{ssid}, \mathsf{prfx}, A, B, C, h)$.

Figure 13: Simulator SIM showing that multi-user shared 3DH key UC-realizes $\mathcal{F} = \mathcal{F}^{-}_{\mathsf{saPAKE}}$, part 2: AKE Simulation.