OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks *

Stanislaw Jarecki[1], Hugo Krawczyk[2], and Jiayu Xu[1]

[1] University of California, Irvine. Email: {stasio@ics.,jiayux@}uci.edu.
[2] IBM Research. Email: hugo@ee.technion.ac.il.

Abstract. Password-Authenticated Key Exchange (PAKE) protocols allow two parties that only share a password to establish a shared key in a way that is immune to offline attacks. *Asymmetric* PAKE (aPAKE) strengthens this notion for the more common client-server setting where the server stores a mapping of the password and security is required even upon server compromise; that is, the only allowed attack in this case is an (inevitable) offline exhaustive dictionary attack against individual user passwords. Unfortunately, current aPAKE protocols (that do not rely on PKI) allow for *pre-computation attacks* that lead to the *instantaneous compromise* of user passwords upon server compromise, thus forgoing much of the intended aPAKE security. Indeed, these protocols use – in essential ways – deterministic password mappings or use random "salt" transmitted *in the clear* from servers to users, and thus are vulnerable to pre-computation attacks.

We initiate the study of *Strong aPAKE* protocols that are secure as aPAKE's but *are also secure against pre-computation attacks.* We formalize this notion in the Universally Composable (UC) settings and present two modular constructions using an Oblivious PRF as a main tool. The first builds a Strong aPAKE from *any* aPAKE (which in turn can be constructed from any PAKE [26]) while the second builds a Strong aPAKE from *any* authenticated key-exchange protocol secure against KCI attacks. Using the latter transformation, we show *a practical instantiation of a UC-secure Strong aPAKE* in the Random Oracle model. The protocol ("OPAQUE") consists of 3 messages, requires 3 and 4 exponentiations for server and client, respectively (including a multi-exponentiation and 1 or 2 fixed-base per party), provides forward secrecy and explicit mutual authentication, is PKI-free, supports user-side password hardening, has a built-in facility for password-based storage-and-retrieval of secrets and credentials, and accommodates a user-transparent server-side threshold implementation.

1 Introduction

Passwords constitute the most ubiquitous form of authentication in the Internet, from the mundane to the most sensitive applications. The almost

* This is a revised ePrint version of the paper which appeared in Eurocrypt 2018 [33]. See revision notes in Sec. 1.2.

universal password authentication method in practice relies on TLS/SSL and consists of the user sending its password to the server under the protection of a client-to-server confidential TLS channel. At the server, the password is decrypted and verified against a one-way image typically computed via hash iterations applied to the password and a random "salt" value. Both the password image and salt are stored for each user in a so-called "password file." In this way, an attacker who succeeds in stealing the password file is forced to run an exhaustive *offline dictionary attack* to find users' passwords given a set ("dictionary") of candidate passwords. The two obvious disadvantages of this approach are: (i) the password appears in cleartext at the server during login[1]; and (ii) security breaks if the TLS channel is established with a compromised server's public key (a widespread concern given today's too-common PKI failures[2]).

Password protocols have been extensively studied in the cryptographic literature – including in the above client-server setting where the user is assumed to possess an authentic copy of the server's public key [27, 29], but the main focus has been on *password-only* protocols where the user does not need to rely on any outside keying material (such as public keys). The basic setting considers two parties that share the same low-entropy password with the goal of establishing shared session keys secure against *offline dictionary attacks*, namely, against an active attacker that possesses a small dictionary from which the password has been chosen. The only viable option for the attacker should be the inevitable *online* impersonation attack with guessed passwords. Such model, known as *password-authenticated key exchange (PAKE)*, was first studied by Bellovin and Merritt [7] and later formalized by Bellare et al. [6] in the game-based indistinguishability approach. Canetti et al. [15] formalized PAKE in the Universally Composable (UC) framework [14], which better captures PAKE security issues such as the use of arbitrary password distributions, the inputting of wrong passwords by the user, and the common practice of using related passwords for different services.

Whereas the cryptographic literature on PAKE's focuses on the above basic setting, in practice the much more common application of password protocols is in the client-server setting. However, sharing the same password between user and server would mean that a break to the server leaks plaintext passwords for all its users. Thus, what's needed is that upon a server compromise, and the stealing of the password file, an attacker is forced to perform an exhaustive offline dictionary attack as in the above TLS scenario. No other attack, except for an inevitable online guessing attack, should be feasible. In particular, *the*

[1] See [2, 3] for examples of the detrimental effect of such exposure even for servers that are not under malicious attack.

[2] PKI failures include stealing of server private keys, software that does not verify certificates correctly, users that accept invalid or suspicious certificates, certificates issued by rogue CAs, servers that share their TLS keys with others – e.g., CDN providers or security monitoring software, information (including passwords) that traverses networks in plaintext form after TLS termination; and more.

two main shortcomings of password-over-TLS mentioned earlier - reliance on public keys and exposure of the password to the server - *need to be eliminated*. This setting, known as *aPAKE*, for *asymmetric PAKE* (also called *augmented* or *verifier-based*), was introduced by Bellovin and Merrit [8], later formalized in the simulation-based approach by Boyko et al. [13], and in the UC framework by Gentry et al. [26]. Early protocols proven in the simulation-based model include [13,42,43]. Later, Gentry et al. [26] presented a compiler that transforms any UC-PAKE protocol into a UC-aPAKE (adding an extra round of communication and a client's signature). This was followed by [34] who show the first simultaneous one-round adaptive UC-aPAKE protocol. In addition, several aPAKE protocols targeting practicality have been proposed, most with ad-hoc security arguments, and some have been (and are being) considered for standardization (see below).

A common deficiency of all these aPAKE protocols, including those being proposed for practical use, is that they are *all vulnerable to pre-computation attacks*. Namely, the attacker $\mathcal{A}$ can *pre-compute* a table of values based on a passwords dictionary D, so as soon as $\mathcal{A}$ succeeds in compromising a server it can *instantly* find a user's password. This weakens the benefits of security against server compromise that motivate the aPAKE notion in the first place. Moreover, while current definitions require that the attacker cannot exploit a server compromise without incurring a workload proportional to the dictionary size $|D|$, these definitions allow all this workload to be spent *before* the actual server compromise happens. Indeed, this weakening in the existing aPAKE security definition [26] is needed to accommodate aPAKE protocols that store a one-way *deterministic* mapping of the user's password at the server, say $H(\mathsf{pw})$. Such protocols trivially fall to a pre-computation attack as the attacker $\mathcal{A}$ can build a table of $(H(\mathsf{pw}), \mathsf{pw})$ pairs for all $\mathsf{pw} \in D$, and once it compromises the server, it finds the value $H(\mathsf{pw})$ associated with a user and immediately, in $\log(|D|)$ time, finds that user's password. Such devastating attack can be mitigated by "personalizing" the password map, e.g., hashing the password together with the user id. This forces $\mathcal{A}$ to pre-compute separate tables for individual users, yet all this effort can still be spent before the actual server compromise.

Note that the standard password-over-TLS scheme prevents pre-computation by hashing passwords with a random salt visible to the server only. In contrast, existing aPAKE protocols that do not rely on PKI, either don't use salt or if they do, the salt is transmitted from server to user during login *in the clear*[3]. Given that password stealing via server compromise is the main avenue for collecting billions of passwords by attackers, the above vulnerability of existing aPAKE protocols to pre-computation attacks is a serious flaw (particularly applicable to targeted attacks), and in this aspect password-over-TLS is more secure than all known aPAKE schemes.

[3] Note that even if the aPAKE protocol runs over TLS, the transmitted salt is open to a straightforward active attack.

1.1 Our contributions

We initiate the study of *Strong aPAKE (SaPAKE)* protocols that strengthen the aPAKE security notion by *disallowing pre-computation attacks.* We formalize this notion in the Universally Composable (UC) model by modifying the aPAKE functionality from [26] to eliminate an adversarial action which allowed such pre-computation attacks. As we explain above, allowing pre-computation attacks was indeed necessary to model the security of existing aPAKE protocols.

The next contribution is building Strong aPAKE (SaPAKE) protocols. For this we present two generic constructions. The first (Section 4) builds the SaPAKE protocol from any aPAKE protocol (namely one that satisfies the original definition from [26]) so that one can "salvage" existing aPAKE protocols. To do so we resort to Oblivious PRF (OPRF) functions [25, 31], namely, a PRF with an associated two-party protocol run between a server S that stores a PRF key k and a user U with a password pw. At the end of the interaction, U learns the PRF output $F_k(\mathsf{pw})$ and S learns nothing (in particular, nothing about pw). We show that by preceding any aPAKE protocol with an OPRF interaction in which U computes the value $\mathsf{rw} = F_k(\mathsf{pw})$ with the help of S and uses rw as the password in the aPAKE protocol, one obtains a Strong aPAKE protocol. We show that if the OPRF and the given aPAKE protocol are, respectively, UC realizations of the OPRF functionality we present (based on [31]) and the original aPAKE functionality from [26], the resultant scheme realizes our UC functionality $\mathcal{F}_{\mathsf{saPAKE}}$.

Our second transformation (Section 5) consists of the composition of an OPRF with a regular authenticated key exchange protocol AKE. We require UC security (with forward secrecy) for the AKE protocol as well as resistance to KCI attacks. The latter is a common feature of public-key AKE protocols that ensures that an attacker that learns the secret keys of one party P, but does not actively control P, cannot use this information to impersonate another party P' to P. In our OPRF-AKE composition, U first runs the OPRF with S to compute $\mathsf{rw} = F_k(\mathsf{pw})$; then it runs the AKE protocol with S using a private key stored, encrypted under rw, at S who sends it to U. Crucial to the security of the protocol is a "random-key robustness" property of the encryption function that can be ensured, for example, by adding an HMAC to a symmetric encryption scheme. Additionally, we assume the encryption to be formally "equivocable" which requires random-oracle, or ideal cipher, modeling of the encryption function (these requirements can be relaxed by a variant of OPAQUE that dispenses with the encryption of the user's private key). We show that the aPAKE scheme resultant from the above composition realizes our UC functionality $\mathcal{F}_{\mathsf{saPAKE}}$.

We use the above second transformation to instantiate a Strong aPAKE protocol with a very efficient OPRF and any efficient UC-secure AKE with the KCI property. The OPRF scheme we use, essentially a Chaum-type blinded DH computation, has been proven UC-secure by Jarecki et al. [30, 31]. We show that this OPRF scheme, which we call DH-OPRF(called 2HashDH in [30, 31]), remains secure in spite of changes to the OPRF functionality that

we introduce for supporting a stronger OPRF notion needed in our setting. We call the result of this instantiation, the *OPAQUE protocol.* OPAQUE combines the best properties of existing aPAKE protocols and of the standard password-over-TLS. On the one hand, it greatly improves on TLS by not relying on PKI and never exposing the cleartext password to the server. At the same time, it resolves the major flaw of existing aPAKE protocols relative to password-over-TLS, namely, their vulnerability to pre-computation attacks.

In addition, OPAQUE offers significant features for practical deployment. Its modularity allows for its use with different key-exchange schemes that can provide different features and performance tradeoffs. When implemented with the DH-OPRF scheme, its cost is one exponentiation for the server and two for the client[4] in addition to the KE protocol cost which can be as little as 2.17 exponentiations per party using HMQV [38] (with full forward secrecy and mutual explicit authentication). The OPRF messages are piggy-backed on those of the AKE protocol for a total of three messages. Another feature of OPAQUE is that it allows for client-side hardening (via hashing or memory-hard functions) that increases the cost of offline dictionary attacks upon server compromise as well as the cost of online guessing attacks. In Fig. 12 in Section 6 we show an instantiation of OPAQUE in the RO model with HMQV as the AKE.

Compared to the practical aPAKE protocols that have been and are being considered for standardization (cf., [1, 48]), OPAQUE fares clearly better on the security side as the only scheme that offers resistance to pre-computation attacks (other aPAKE protocols can be made "strong" by using our methodology from Section 4). Performance-wise, OPAQUE is competitive with the more efficient among these protocols (see Section 6). OPAQUE also provides a unique functionality among aPAKE protocols in that it allows to *store and retrieve user's secrets* such as a bitcoin wallet, authentication credentials, encrypted backup keys, etc., thus offering a far more secure alternative to the practice of deriving low-entropy secrets directly from a user's password. Furthermore, OPAQUE allows for a user-transparent server-side threshold implementation [32] where the only exposure of the user password - or any stored secrets - is in case a threshold of servers is compromised and even then a full dictionary attack is required.

Finally, we comment that while OPAQUE can completely replace password authentication in TLS, it can also be used in conjunction with TLS for protecting account information, for bootstrapping TLS client authentication (via an OPAQUE-retrieved client signing key), or as an hedge against PKI failures. In other words, while we are accustomed to use TLS to protect passwords, OPAQUE can be used to protect TLS. We expand on this aspect in Section 6.2.

Prior protocol variants. OPAQUE is not a completely new protocol. Variants have been studied in prior work in several settings but none of these works presents a formal analysis of the protocol as an aPAKE, let alone as a Strong

[4] A variant of the protocol discussed in Section 6.3 allows one or both of the client's exponentiations to be fixed-base and offline.

aPAKE, a notion that we introduce here for the first time. While our treatment frames OPAQUE in the context of Oblivious PRFs [30, 31], its design can be seen as an instantiation of the Ford-Kaliski paradigm for password hardening and credential retrieval using Chaum's blinded exponentiation. Most related is Boyen's work [12] which specifies and studies an instantiation of the protocol (called HPAKE) in the setting of client-side *halting KDF* [11]. Jarecki et al. [30, 31] study a threshold version (also using the OPRF abstraction) in the context of *password-protected secret sharing (PPSS)* protocols. Because of the relation between PPSS and Threshold PAKE protocols [30], this analysis implies security of OPAQUE as a PAKE protocol in the BPR model [6] but not as an aPAKE (let alone as a strong aPAKE).

1.2 Revision Notes

The current version of this paper introduces substantial revisions in the formal treatment relative to the proceedings version [33]. The protocol specification has not changed except that the necessity for forward secrecy made explicit here implies that OPAQUE cannot be implemented with a 2-message protocol as depicted in [33] (see footnote 13 in page 48). We list the main formal changes here with details provided in the corresponding sections.

1. We strengthen (Section 5.1) the AKE functionality with adaptive security for both client and server (in [33] we only did it on the server side) which, in particular, ensures forward secrecy as needed for the security of the OPAQUE protocol.
2. We relaxed the Strong aPAKE functionality in three ways (Section 2). First, in the event that the attacker guesses the password correctly, we consider sessions that were created but not completed before the guessing event as compromised, but only if the attacker actively interfered in these sessions. Second, the attacker is allowed one password guess after a session is completed (if it guesses correctly, it learns that the guess is correct but it does not learn the key output by the completed session). Third, the attacker can test if a user's session runs on a password matching the server's even if the server does not run a sub-session whose ID matches the tested user's sub-session. While these relaxations are necessary for proving security, they do not seem to earn the attacker any practical advantages.
3. The composition of OPRF and AKE functionalities that form the OPAQUE scheme require a binding mechanism between the OPRF and AKE sub-sessions which we implement by including in the AKE's session identifier a prefix of the OPRF session transcript (e.g., for DH-OPRF function, this prefix is defined as the user's initial value a). In order to formalize this mechanism we extend the OPRF functionality in Section 3 to output a prefix of the function's transcript.
4. We identify the requirement for the authenticated encryption protecting the user's private key stored at S to be equivocable (Section 5.2).

5. Pending revision: The material in Section 4 (compiler from aPAKE to strong aPAKE) will be revised shortly to adapt the proof of Theorem 1 to the modified OPRF and SaPAKE functionalities.

2 The Strong aPAKE Functionality

In this section we present the ideal Strong aPAKE UC functionality, $\mathcal{F}_{\mathsf{saPAKE}}$, shown in Fig. 2, that will serve as our definition of Strong aPAKE security; namely, we will call a protocol a secure *Strong aPAKE* if it realizes $\mathcal{F}_{\mathsf{saPAKE}}$. Functionality $\mathcal{F}_{\mathsf{saPAKE}}$ is a relaxation of the Strong aPAKE notion defined in the proceedings version of this paper [33] to which we refer as $\mathcal{F}_{\mathsf{saPAKE}^+}$.[5] Functionality $\mathcal{F}_{\mathsf{saPAKE}}$ and its predecessor $\mathcal{F}_{\mathsf{saPAKE}^+}$ build on the UC asymmetric PAKE (aPAKE) functionality $\mathcal{F}_{\mathsf{aPAKE}}$ defined by Gentry et al. [26][6] which, in turn, adapts the (symmetric) UC PAKE defined by Canetti et al. [15] to the asymmetric case.

We present $\mathcal{F}_{\mathsf{saPAKE}}$ in three stages: First, we explain how the UC functionality $\mathcal{F}_{\mathsf{aPAKE}}$ used to define asymmetric PAKE security in [26] builds on the symmetric UC PAKE notion of [15]. Then we show how functionality $\mathcal{F}_{\mathsf{saPAKE}^+}$ strengthens the definition from [26] to eliminate the vulnerability to pre-computation attacks. Lastly, we derive our UC Strong aPAKE functionality $\mathcal{F}_{\mathsf{saPAKE}}$ by modifying $\mathcal{F}_{\mathsf{saPAKE}^+}$ to account for adversarial behavior which is possible against our main protocol OPAQUE from Section 5 but, as we argue, does not compromise practical security in any significant way.

Asymmetric PAKE. The aPAKE functionality of [26], denoted here $\mathcal{F}_{\mathsf{aPAKE}}$ and shown in Fig. 1 (full text), extends the symmetric UC PAKE functionality from [15] to the asymmetric (user-server) setting. First, in an aPAKE scheme the server and the user run different programs: The user runs an aPAKE session on a password (via command UsrSession) while the server runs it on a "password file" $\mathsf{file}[sid]$ that represents server's user-specific state corresponding to the user's password, e.g., a password hash, which the server creates on input the user's password via command StorePwdFile during aPAKE initialization. Furthermore, $\mathcal{F}_{\mathsf{aPAKE}}$ models a possible compromise of a server, via command StealPwdFile, from which the attacker obtains $\mathsf{file}[sid]$. Such compromise subsequently allows the attacker to (1) impersonate the server to the user, via command Impersonate, and (2) find the password via an offline dictionary attack, modeled by command OfflineTestPwd. The way functionality $\mathcal{F}_{\mathsf{aPAKE}}$ of [26] handles the offline dictionary attack is the focus of the Strong aPAKE functionality we define, and which we discuss next.

Strong aPAKE vs. aPAKE. As discussed in the introduction, the aPAKE functionality from [26] is too weak to ensure resistance to offline dictionary attacks. Protocols proven secure in that model may allow an attacker to build

[5] Functionality $\mathcal{F}_{\mathsf{saPAKE}^+}$ was denoted $\mathcal{F}_{\mathsf{saPAKE}}$ in [33], but we argue that it is stronger than necessary, hence we denote it here with the plus sign.

[6] Functionality $\mathcal{F}_{\mathsf{aPAKE}}$ was denoted $\mathcal{F}_{\mathsf{apwKE}}$ in [26].

In the description below, we assume $\mathsf{P} \in \{\mathsf{U}, \mathsf{S}\}$.

Password Registration

- On (StorePwdFile, $sid, \mathsf{U}, \mathsf{pw}$) from S, if this is the first StorePwdFile message, record $\langle \text{file}, \mathsf{U}, \mathsf{S}, \mathsf{pw} \rangle$ and mark it uncompromised.

Stealing Password Data

- On (StealPwdFile, sid) from $\mathcal{A}^*$, if there is no record $\langle \text{file}, \mathsf{U}, \mathsf{S}, \mathsf{pw} \rangle$, return "no password file" to $\mathcal{A}^*$. Otherwise, if the record is marked uncompromised, mark it compromised; regardless,
  - If there is a record $\langle \text{offline}, \mathsf{pw} \rangle$, send pw to $\mathcal{A}^*$.
  - Else Return "password file stolen" to $\mathcal{A}^*$.
- On (OfflineTestPwd, sid, pw^*) from $\mathcal{A}^*$, do:
  - If there is a record $\langle \text{file}, \mathsf{U}, \mathsf{S}, \mathsf{pw} \rangle$ marked compromised, do: if $\mathsf{pw}^* = \mathsf{pw}$, return "correct guess" to $\mathcal{A}^*$; else return "wrong guess."
  - Else record $\langle \text{offline}, \mathsf{pw} \rangle$.

Password Authentication

- On (UsrSession, $sid, ssid, \mathsf{S}, \mathsf{pw}'$) from U, send (UsrSession, $sid, ssid, \mathsf{U}, \mathsf{S}$) to $\mathcal{A}^*$. Also, if this is the first UsrSession message for $ssid$, record $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}' \rangle$ and mark it fresh.
- On (SvrSession, $sid, ssid$) from S, retrieve $\langle \text{file}, \mathsf{U}, \mathsf{S}, \mathsf{pw} \rangle$, and send (SvrSession, $sid, ssid, \mathsf{U}, \mathsf{S}$) to $\mathcal{A}^*$. Also, if this is the first SvrSession message for $ssid$, record $\langle ssid, \mathsf{S}, \mathsf{U}, \mathsf{pw} \rangle$ and mark it fresh.

Active Session Attacks

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

Key Generation and Authentication

- On (NewKey, $sid, ssid, \mathsf{P}, SK^*$) from $\mathcal{A}^*$ where $|SK^*| = \ell$, if there is a record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ not marked completed, do:
  - If the record is compromised, or P or P' is corrupted, set $SK := SK^*$.
  - Else if the record is fresh, a $(sid, ssid, SK')$ tuple was sent to P', and at that time there was a record $(ssid, \mathsf{P}', \mathsf{P})$ marked fresh, set $SK := SK'$.
  - Else pick $SK \leftarrow_{\text{R}} \{0,1\}^\ell$.

  Finally, mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ completed and send $(sid, ssid, SK)$ to P.
- On (TestAbort, $sid, ssid, \mathsf{P}$) from $\mathcal{A}^*$, if there is a record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ not marked completed, do:
  - If it is fresh and there is a record $\langle ssid, \mathsf{P}', \mathsf{P}, \mathsf{pw}' \rangle$, send Succ to $\mathcal{A}^*$.
  - Else send Fail to $\mathcal{A}^*$ and (abort, $sid, ssid$) to P, and mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ completed.

Fig. 1: Functionalities $\mathcal{F}_{\mathsf{aPAKE}}$ (full text) and $\mathcal{F}_{\mathsf{saPAKE}^+}$ (shadowed text omitted)

Password Registration

- On (StorePwdFile, sid, U, pw) from S, if this is the first such message, record $\langle$file, U, S, $\mathsf{pw}\rangle$, mark it uncompromised, set flag := uncompromised.

Stealing Password Data

- On (StealPwdFile, sid) from $\mathcal{A}^*$, if there is no record $\langle$file, U, S, $\mathsf{pw}\rangle$, return "no password file" to $\mathcal{A}^*$. Otherwise, if the record is marked uncompromised, mark it compromised; regardless, return "password file stolen" to $\mathcal{A}^*$.
- On (OfflineTestPwd, sid, pw^*) from $\mathcal{A}^*$, if there is a record $\langle$file, U, S, $\mathsf{pw}\rangle$ marked compromised, do: if $\mathsf{pw}^* = \mathsf{pw}$, return "correct guess" to $\mathcal{A}^*$ and set flag := compromised; otherwise return "wrong guess."

Password Authentication

- On (UsrSession, sid, $ssid$, S, pw') from U, send (UsrSession, sid, $ssid$, U, S) to $\mathcal{A}^*$. Also, if this is the first UsrSession message for $ssid$, record $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}'\rangle$ and mark it fresh.
- On (SvrSession, sid, $ssid$) from S, retrieve $\langle$file, U, S, $\mathsf{pw}\rangle$, and send (SvrSession, sid, $ssid$, U, S) to $\mathcal{A}^*$. Also, if this is the first SvrSession message for $ssid$, record $\langle ssid, \mathsf{S}, \mathsf{U}, \mathsf{pw}\rangle$ and mark it fresh.

Active Session Attacks

- On (Interrupt, sid, $ssid$, S) from $\mathcal{A}^*$, if there is a record $\langle ssid, \mathsf{S}, \mathsf{U}, \mathsf{pw}\rangle$ marked fresh, mark it interrupted and set $\mathsf{dPT}(ssid) := 1$.
- On (TestPwd, sid, $ssid$, P, pw^*) from $\mathcal{A}^*$, retrieve record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}'\rangle$ and:
  - If the record is fresh then do the following: If $\mathsf{pw}^* = \mathsf{pw}'$ return "correct guess" to $\mathcal{A}^*$ and mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}'\rangle$ compromised, otherwise return "wrong guess" and mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}'\rangle$ interrupted.
  - If $\mathsf{P} = \mathsf{S}$ and $\mathsf{dPT}(ssid) = 1$ then set $\mathsf{dPT}(ssid) := 0$ and if $\mathsf{pw}^* = \mathsf{pw}'$ then return "correct guess" to $\mathcal{A}^*$ else return "wrong guess."

  In either case, if $\mathsf{P} = \mathsf{S}$ and $\mathsf{pw}^* = \mathsf{pw}'$ then set flag := compromised.
- On (Impersonate, sid, $ssid$) from $\mathcal{A}^*$, if there is a record $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}'\rangle$ marked fresh, do: If there is a record $\langle$file, U, S, $\mathsf{pw}\rangle$ marked compromised and $\mathsf{pw}' = \mathsf{pw}$, mark $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}'\rangle$ compromised and return "correct guess" to $\mathcal{A}^*$; otherwise mark it interrupted and return "wrong guess."

Key Generation and Authentication

- On (NewKey, sid, $ssid$, P, SK^*) from $\mathcal{A}^*$ where $|SK^*| = \ell$, if there is a record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}'\rangle$ not marked completed, do:
  - If the record is compromised, or ($\mathsf{P} = \mathsf{S}$, the record is interrupted and flag = compromised), or either P or P' is corrupted, set $SK := SK^*$.
  - Else, if the record is fresh and $(sid, ssid, SK')$ was sent to P' at the time there was a record $\langle ssid, \mathsf{P}', \mathsf{P}, \mathsf{pw}'\rangle$ marked fresh, set $SK := SK'$.
  - Else pick $SK \leftarrow_{\mathrm{R}} \{0,1\}^{\ell}$.

  Finally, mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}'\rangle$ completed and send $(sid, ssid, SK)$ to P.
- On (TestAbort, sid, $ssid$, P) from $\mathcal{A}^*$, if there is a record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}'\rangle$ not marked completed, do:
  - If it is fresh and there is a record $\langle ssid, \mathsf{P}', \mathsf{P}, \mathsf{pw}'\rangle$ send Succ to $\mathcal{A}^*$.
  - If it is fresh, $\mathsf{P}' = \mathsf{S}$, and $\mathsf{pw}' = \mathsf{pw}$, send Succ to $\mathcal{A}^*$.
  - Otherwise send Fail to $\mathcal{A}^*$ and (abort, sid, $ssid$) to P, and mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}'\rangle$ completed.

Fig. 2: Functionality $\mathcal{F}_{\mathsf{saPAKE}}$ with marked additions relative to $\mathcal{F}_{\mathsf{saPAKE}^+}$

a dictionary prior to compromising a server so that when the compromise occurs it immediately finds the user's password (moreover, all previous protocols proven under this functionality were indeed subject to such pre-computation attack). In contrast, our UC Strong aPAKE functionality $\mathcal{F}_{\mathsf{saPAKE}}$ disallows such attacks. To explain how, we first consider functionality $\mathcal{F}_{\mathsf{saPAKE}^+}$ that serves as a nexus between $\mathcal{F}_{\mathsf{aPAKE}}$ and $\mathcal{F}_{\mathsf{saPAKE}}$. Functionality $\mathcal{F}_{\mathsf{saPAKE}^+}$, presented in Fig. 1 (gray text omitted), was used to define Strong aPAKE in the proceedings version of this paper [33] but it turns out to be too restrictive as we explain below. As depicted in the figure, the only formal difference between functionalities $\mathcal{F}_{\mathsf{aPAKE}}$ and $\mathcal{F}_{\mathsf{saPAKE}^+}$ are in the actions upon the stealing of the password file. Specifically, $\mathcal{F}_{\mathsf{saPAKE}^+}$ omits recording the $\langle \textsc{offline}, \mathsf{pw} \rangle$ pairs and does not allow for OfflineTestPwd queries made before the StealPwdFile query. Let us explain. In $\mathcal{F}_{\mathsf{saPAKE}^+}$, the actions upon server compromise, i.e., StealPwdFile, are simple. First, a flag is defined to mark that the password file has been compromised. Second, once this event happens, the adversary is allowed to submit password guesses and be informed if a guess was correct. Note that each guess "costs" the attacker one OfflineTestPwd query. This together with the restriction that these queries can only be made after the password file is compromised ensure that shortcuts in finding the password after such compromise are not possible, namely that the attacker needs to pay with one OfflineTestPwd query for each password it wants to test. Thus, pre-computation attacks are made infeasible.

Now, consider the $\mathcal{F}_{\mathsf{aPAKE}}$ functionality [26] which includes the text in gray too. This functionality allows the attacker, via $\langle \textsc{offline}, \mathsf{pw} \rangle$ records, to make guess queries against the password even before the password file is compromised. The restriction is that the responses to whether a guess was correct or not are provided to the attacker only after a StealPwdFile event. But note that if one of these guesses was correct, the attacker learns it *immediately* upon server compromise. This provision was necessary in [26] to prove their aPAKE protocol that included in the $\mathsf{file}[sid]$ a deterministic publicly-computable hash of the password, thus allowing for a pre-computation attack which lets the adversary instantaneously identify the password with a single table lookup upon server compromise. Indeed, one can think of the pairs $\langle \textsc{offline}, \mathsf{pw} \rangle$ in the original $\mathcal{F}_{\mathsf{aPAKE}}$ functionality as a pre-computed table that the attacker builds overtime and which it can use to identify the password as soon as the server is compromised. By eliminating the ability to get guesses $\langle \textsc{offline}, \mathsf{pw} \rangle$ answered before server compromise in our $\mathcal{F}_{\mathsf{saPAKE}^+}$ functionality, we make such pre-computation attacks infeasible in the case of a Strong aPAKE.

Modeling Server Compromise and Offline Dictionary Queries. In $\mathcal{F}_{\mathsf{saPAKE}^+}$ as in $\mathcal{F}_{\mathsf{aPAKE}}$, StealPwdFile and OfflineTestPwd messages from $\mathcal{A}^*$ to $\mathcal{F}_{\mathsf{saPAKE}^+}$ are accounted for by the environment. This is consistent with the UC treatment of adaptive corruption queries and is crucial to our modeling. Note that if the environment does not observe adaptive corruption queries then the ideal model adversary, i.e., the simulator, could immediately

corrupt all parties at the beginning of the protocol, learning their private inputs and thus making the work of simulation straightforward. By making the player-corruption queries, modeled by StealPwdFile command in our context, observable by the environment, we ensure that the environment's view of both the ideal and the real execution includes the same player-corruption events. This way we keep the simulator "honest," because it can only corrupt a party if the environment accounts for it.

The same concern pertains to offline dictionary queries OfflineTestPwd, because if they were not observable by the environment, the ideal adversary could make such queries even if the real adversary does not. In particular, without environmental accounting for these queries the $\mathcal{F}_{\mathsf{aPAKE}}$ and $\mathcal{F}_{\mathsf{saPAKE}^+}$ functionalities would be equivalent because the simulator could internally gather all the offline dictionary attack queries made by the real-world adversary before server corruption, and it would send them all via the OfflineTestPwd query to $\mathcal{F}_{\mathsf{saPAKE}^+}$ after server corruption via the StealPwdFile query. Such simulator would make the ideal-world view indistinguishable from the real-world view to the environment *if* the environment does not observe the sequence of OfflineTestPwd and StealPwdFile queries.

Finally, we note that $\mathcal{F}_{\mathsf{saPAKE}^+}$ (following $\mathcal{F}_{\mathsf{aPAKE}}$) has effectively two separate notions of a server corruption. Formally, it considers a *static* adversarial model where all entities, including users and servers, are either honest or corrupt throughout the life-time of the scheme. However, it adds a crucial *adaptive capability* to the attacker against *honest* servers via the StealPwdFile action. It results in leaking to the adversary the server's private state corresponding to a particular password file, but it does not give the adversary full control over the server's entity. In particular, the accounts on the same server for which the adversary does not explicitly issue the StealPwdFile command must remain unaffected. We adopt this convention from [26] and we call a server "corrupted" if it is (statically) corrupt and adversarially controlled, and we call an aPAKE instance "compromised" if the adversary steals its password file from the server.

Functionality $\mathcal{F}_{\mathsf{saPAKE}}$: Functionality $\mathcal{F}_{\mathsf{saPAKE}^+}$ captures the core notion of security against pre-computation attacks for asymmetric PAKE protocols. It also satisfies the essential property of PAKE protocols (asymmetric and symmetric) that an attacker can only test one password guess per PAKE session. However, this functionality is stronger than necessary in three ways. First, it requires that an attacker who guesses a password cannot compromise any existing session, not even those where the adversary actively tried to impersonate a user, namely, incomplete *interrupted* sessions. This choice was made by all prior UC PAKE functionality variants, symmetric [15] and asymmetric [26], but it is an over-restrictive condition relative to definitions of secure key-exchange protocols that do not guarantee the security of incomplete sessions with active corruptions (e.g., if a party to a session is corrupted while the session is open, no security is guaranteed for that session).

Secondly, functionality $\mathcal{F}_{\mathsf{saPAKE}^+}$ (as previous UC PAKE functionalities) requires that a password tested in an active attack is efficiently extractable (from adversary's computation) before the attacked session completes. This requirement cannot be satisfied by an aPAKE protocol where the user's messages are committing to a single tested password, but this password is extractable only from the user's computation *after* the protocol completes on the server side. In this case the attacker cannot influence the computation of the session key but it can still learn if its password guess is correct.

Third, functionality $\mathcal{F}_{\mathsf{saPAKE}^+}$, like $\mathcal{F}_{\mathsf{aPAKE}}$ [26], lets a man-in-the-middle adversary observe whether some U and S sessions share the same password, by "connecting" these two sessions, i.e. routing the messages back and forth between them, and observing if either U or S aborts (which implies $\mathsf{pw}' \neq \mathsf{pw}$) or not (which *might* imply $\mathsf{pw}' = \mathsf{pw}$). This is indeed a necessary information leakage in any (sa)PAKE protocol where either party implements explicit entity authentication, and aborts if the peer fails to authenticate. In UC aPAKE functionality $\mathcal{F}_{\mathsf{aPAKE}}$ of [26] this password-matching test was modeled with interface TestAbort which on input $(sid, ssid, \mathsf{P})$ replied Succ if and only if, for the tested session $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}\rangle$ there is a record of its counterparty P' running a session $\langle ssid', \mathsf{P}', \mathsf{P}, \mathsf{pw}'\rangle$ s.t. $ssid' = ssid$ and $\mathsf{pw}' = \mathsf{pw}$. While this sub-session-specific password-equality verification makes sense in the case of U sessions, whose each sub-sessions might run on a different password, it seems an overkill in the case of S sub-sessions, which all run on the same password file, a one-way function of a fixed password pw.

In our context, either requirement prevents proving security of the protocols obtained via our general compiler from Section 5, including the OPAQUE protocol from Section 6. For this reason we relax $\mathcal{F}_{\mathsf{saPAKE}^+}$ to *obtain our definition of UC Strong aPAKE functionality* $\mathcal{F}_{\mathsf{saPAKE}}$, presented in Fig. 2. Changes with respect to $\mathcal{F}_{\mathsf{saPAKE}^+}$ are highlighted with shadowed background. $\mathcal{F}_{\mathsf{saPAKE}}$ preserves the essential guarantees of $\mathcal{F}_{\mathsf{saPAKE}^+}$, i.e. the same level of security against server compromise, the ability of the attacker to try at most one password guess per session, and the security of all sessions that complete while the user's password is uncompromised. However, $\mathcal{F}_{\mathsf{saPAKE}}$ relaxes the $\mathcal{F}_{\mathsf{saPAKE}^+}$ in three ways: First, it does not guarantee the security of sessions in which the attacker actively interferes and which are not completed at the time the attacker learns the password. Second, it allows for passwords tested by the attacker in a given session to be extractable only after the session completes. Third, it allows TestAbort to test if any U sub-session runs on the server's password without S running a sub-session with a matching ID.

The first relaxation is modeled by adding a flag, denoted flag, which indicates whether the user's password is guessed by the attacker via a TestPwd query against some session of server S. When that occurs flag is set to compromised and every actively attacked session of S, i.e. one whose record is set to either interrupted or compromised, that completes when flag = compromised, is treated as compromised and the attacker can set its session key.

The second relaxation is modeled by adding a new query INTERRUPT, which models an active attack against S session in which the adversary does not contribute the password guess pw^* straight away. Session $(\mathsf{S}, ssid)$ attacked in this way is marked with flag $\mathsf{dPT}(ssid)$ set to 1, which allows the adversary to make a *delayed* password test on this session. Namely, if $\mathsf{dPT}(ssid) = 1$ then an adversary can make *one* TESTPWD query against session $(\mathsf{S}, ssid)$ regardless of its status, i.e. either before this session completes via NEWKEY or after. In the first case, i.e. if TESTPWD is sent before the session completes *and* the password guess pw^* is correct then the adversary learns that pw^* is correct, flag is set to COMPROMISED, and all interrupted and still active S sessions, including $(\mathsf{S}, ssid)$, are treated as compromised so the attacker can set their session keys when they complete. In the second case, i.e. if TESTPWD is sent after the session completes, then the only thing the adversary gets is the information whether the password guess pw^* was correct or not, but it cannot either influence or learn the session key output by $(\mathsf{S}, ssid)$.

Either way the TESTPWD query sets flag $\mathsf{dPT}(ssid)$ to 0, and since the session is then either INTERRUPTED or COMPLETED, no new password query can be made against it. Note that the only new attack this mechanism allows for is that an active attacker can make a "postponed" TESTPWD query against this session, when the session already completes, rather than making it before. However, the password guess pw^* is committed at the time of the active attack, because only one pw^* can be tested against the session; and since making the TESTPWD query after the session completes gives less power to the attacker than making it before the session completes, and the latter is allowed by the standard (a)PAKE model, this new attack avenue seems to offer no advantages to the attacker.

The third relaxation is modeled by allowing TESTABORT on U's sub-session to reply SUCC if U's input pw' matches S's password pw, and the test proceeds whether or not S runs a sub-session with a matching $ssid$.

Non-black-box Assumptions. Note that the aPAKE functionality requires the simulator, playing the role of the ideal-model adversary, to detect offline password guesses made by the real-world adversary. As pointed out by [26], this seems to require a non-black-box hardness assumption on some cryptographic primitive, e.g., the Random Oracle Model (ROM), which would allow the simulator to extract a password guess from adversary's *local* computation, e.g., a local execution of aPAKE interaction on a password guess and a stolen password file.

Server Initialization. We note that while $\mathcal{F}_{\mathsf{aPAKE}}$ defines password registration as an internal action of server S, with the user's password as a local input, one can modify it to support an interactive procedure between user and server, e.g., to prevent S from *ever* learning the plaintext password. Such interactive initialization, would require a change in the $\mathcal{F}_{\mathsf{saPAKE}}$ functionality to allow the user to build the password file and send it to the server. For simplicity, and in order to minimize changes relative to [26], we keep the non-interactive functionality. See more about interactive initialization in Section 6.1.

Public Parameters: PRF output-length ℓ, polynomial in security parameter τ.
Conventions: For every i, x, value $F_{sid,i}(x)$ is initially undefined, and if undefined value $F_{sid,i}(x)$ is referenced then $\mathcal{F}_{\mathsf{OPRF}}$ assigns $F_{sid,i}(x) \leftarrow_{\mathrm{R}} \{0,1\}^{\ell}$.

Initialization
On message (INIT, *sid*) from party S, if this is the first INIT message for *sid*, set $\mathsf{tx} = 0$ and send (INIT, *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 (COMPROMISE, *sid*, S) from $\mathcal{A}^*$, declare server S as COMPROMISED.
(If S is corrupted then it is declared COMPROMISED from the beginning.)
Note: Message (COMPROMISE, *sid*, S) *requires permission from the environment.*

Offline Evaluation
On (OFFLINEEVAL, *sid*, i, x) from $\mathsf{P} \in \{\mathsf{S}, \mathcal{A}^*\}$, send (OFFLINEEVAL, *sid*, $F_{sid,i}(x)$) to P if any of the following hold: (i) S is corrupted, (ii) $\mathsf{P} = S$ and $i = S$, (iii) $\mathsf{P} = \mathcal{A}^*$ and $i \neq S$, (iv) $\mathsf{P} = \mathcal{A}^*$ and S is compromised.

Online Evaluation

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

Fig. 3: Functionality $\mathcal{F}_{\mathsf{OPRF}}$ with Adaptive Compromise

3 Oblivious Pseudorandom Function

An Oblivious Pseudorandom Function scheme (OPRF) is a central tool in all our constructions. An OPRF consists of a pseudorandom function family F with an associated two-party protocol executed between a server that holds a key k for F and a user with an input x. At the end of the interaction, the user learns the PRF output $F_k(x)$ and nothing else, and the server learns nothing (in particular, nothing about x). The notion of OPRF was introduced in [25]. The first UC formulation of it was given in [30], including a verifiability property that lets the user check the correct behavior of the server during the OPRF execution. Later [31] gave an alternative UC definition of OPRF which dispensed with the verifiability property, allowing for more efficient instantiations. The main idea in the OPRF formulations of [30, 31] is the use of a *ticketing mechanism* that ensures that the number of input values on which anyone can compute the OPRF

on a key held by an honest server S is no more than the number of executions of the OPRF recorded by S. This mechanism dispenses with the need to extract users' inputs as is typically needed in UC simulations and it allows much more efficient OPRF instantiations.

Here we adopt the UC OPRF formulation from [31] as the basis for our definition of *adaptively secure* OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$, presented in Fig. 3. We refer to [31,33] for detailed rationale for this functionality, but we note that it requires PRF outputs to be pseudorandom even to the owner of the PRF key k. This does not seem achievable under non-black-box assumptions, but it is achievable, indeed very efficiently, in the Random Oracle Model (ROM). (In Appendix B we show that the DH-OPRF scheme of [31] realizes $\mathcal{F}_{\mathsf{OPRF}}$ in ROM under the same One More Diffie-Hellman assumption which was used to show that the same protocol realizes the UC OPRF formalization of [31].) Note that the reliance on non-black-box assumptions like ROM is called for in the aPAKE UC context, see Section 2.

The UC OPRF definition shown in Fig. 3 is a *revision* of the definition given in [33]. In Appendix A we include detailed explanations of the differences between the UC OPRF of Fig. 3 and the UC OPRF defined in proceedings version of this paper [33], as well as the differences between both of the above definitions and the UC OPRF definition given in [31].

OPRF Transcript Prefixes. As we explain in Appendix A almost all changes between UC OPRF formulation in Fig. 3 and the corresponding formulation in the proceedings version of this paper [33] are *syntactic*. However, there is one modification which is more than syntactic, which is that here we extend the UC OPRF functionality so that both user and server sessions have additional local output, which is a *prefix of the OPRF protocol transcript*. The protocol transcript is of course application dependent, and how much of it counts as a prefix can depend on the implementation, but the functionality $\mathcal{F}_{\mathsf{OPRF}}$ does not care what these prefixes are except for the following constraint: If some subsession of server S shares a protocol prefix with some subsession of user U, then the only party which can compute function $F_k(\cdot)$ on some input x due to this interaction is that U's subsession, and not e.g. the adversary.

This property is not overly restrictive and indeed it is expected in any implementation of OPRF. It means simply that if the man-in-the-middle adversary forwards the messages exchanged between some U and S subsessions until some point (that point defined how much of the OPRF transcript counts as its prefix), then the adversary can no longer stage an active attack on S, and use it to compute $F_k(x)$ for x of its choice. The only choices the adversary has in that case is to let U and S continue their interaction, which lets U compute $F_k(\cdot)$ on U's input, or interfere with it in a way which prevents *everyone* from computing $F_k(\cdot)$ on any input in that subsession. In the OPRF scheme DH-OPRF from [31], recalled in Appendix B, which realizes $\mathcal{F}_{\mathsf{OPRF}}$ under the One More Diffie-Hellman assumption, the role of that prefix is played by the user's message a. Indeed, it is easy to see that an active attacker cannot

succeed in computing $F_k(x)$ on x of its choice if it forwards some honest U's message a to S instead of replacing it with its own message a'.

This "no successful active attack if transcript prefixes match" property of OPRF is used in the proof of security of the Strong aPAKE protocol OPAQUE, shown in the generic form in Fig. 8 in Section 5. In Section 5.2 we include a more detailed explanation of how the security argument for OPAQUE uses this property, but a general intuition is that this property forces the man-in-the-middle attacker against an OPRF scheme to decide, for every S session, whether to (I) make this session potentially useful for some U session or (II) make this S session useful only to the adversary. (This is the consequence of making S and U transcript prefixes either match or not match.) This can be useful in a higher-level protocol (e.g., OPAQUE) because it allows the simulator of this protocol decide whether a given S session can be "connected" to some honest U (case I) or it is actively attacked (case II).

4 A Compiler from aPAKE to Strong aPAKE via OPRF

Under revision. *This section is to be revised to adapt the proof to the modified SaPAKE functionality from Fig.2 and OPRF functionality from Fig. 3. We'll do so shortly.*

In Fig. 4 we specify a compiler that transforms any OPRF and any aPAKE into a Strong aPAKE protocol. In UC terms the Strong aPAKE protocol is defined in the $(\mathcal{F}_{\mathsf{OPRF}}, \mathcal{F}_{\mathsf{aPAKE}})$-hybrid world, for $\mathcal{F}_{\mathsf{OPRF}}$ with the output length parameter $\ell = 2\tau$. The compiler is simple. First, the user transforms its password pw into a randomized value rw by interacting with the server in an OPRF protocol where the user inputs pw and the server inputs the OPRF key. Nothing is learned at the server about pw (i.e., rw is indistinguishable from random as long as the input pw is not queried as input to the OPRF). Next, the user sets rw as its password in the given aPAKE protocol. Note that since the password rw is taken from a pseudorandom set, then even if the size of this set is the same as the original dictionary D from which pw was taken, the pseudorandom set is unknown to the attacker (the attacker can only learn this set via OPRF queries which require an online dictionary attack). Thus, any previous ability to run a pre-computation attack against the aPAKE protocol based on dictionary D is now lost.

We assume that $\mathcal{A}$ always simultaneously sends queries (Compromise, *sid*) and (StealPwdFile, *sid*) for the same *sid*, resp. to $\mathcal{F}_{\mathsf{OPRF}}$ to $\mathcal{F}_{\mathsf{aPAKE}}$, because in any instantiation of this scheme the server's OPRF-related state and aPAKE-related state would be part of the same $\mathsf{file}[sid]$. Consequently, for a single *sid*, S's status (compromised or not) in $\mathcal{F}_{\mathsf{OPRF}}$ and $\mathcal{F}_{\mathsf{aPAKE}}$ is always the same.

Password Registration

On input (StorePwdFile, sid, U, pw), S sends (Init, sid, pw) to $\mathcal{F}_{\mathsf{OPRF}}$. On $\mathcal{F}_{\mathsf{OPRF}}$'s response (Init, sid, rw), S sends (StorePwdFile, sid, U, rw) to $\mathcal{F}_{\mathsf{aPAKE}}$.

Server Compromise

Message (StealPwdFile, sid) from $\mathcal{A}$ to functionality $\mathcal{F}_{\mathsf{saPAKE}}$ is interpreted as if $\mathcal{A}$ sent two messages: (1) a (StealPwdFile, sid) message to functionality $\mathcal{F}_{\mathsf{aPAKE}}$ and (2) a (Compromise, S) message to functionality $\mathcal{F}_{\mathsf{OPRF}}$.

Password Authentication and Key Generation

1. On input (UsrSession, sid, $ssid$, S, pw'), U sends (Eval, sid, $ssid$, S, pw') to $\mathcal{F}_{\mathsf{OPRF}}$. On $\mathcal{F}_{\mathsf{OPRF}}$'s response (Eval, sid, $ssid$, rw'), U sends (UsrSession, sid, $ssid$, S, rw') to $\mathcal{F}_{\mathsf{aPAKE}}$.
2. On input (SvrSession, sid, $ssid$), S sends (SndrComplete, sid, $ssid$) to $\mathcal{F}_{\mathsf{OPRF}}$ and (SvrSession, sid, $ssid$) to $\mathcal{F}_{\mathsf{aPAKE}}$.
3. On (sid, $ssid$, SK) or (abort, sid, $ssid$) from $\mathcal{F}_{\mathsf{aPAKE}}$, the recipient, either U or S, outputs this message.

Fig. 4: Strong aPAKE Protocol in the ($\mathcal{F}_{\mathsf{OPRF}}$, $\mathcal{F}_{\mathsf{aPAKE}}$)-Hybrid World

4.1 Proof of Security

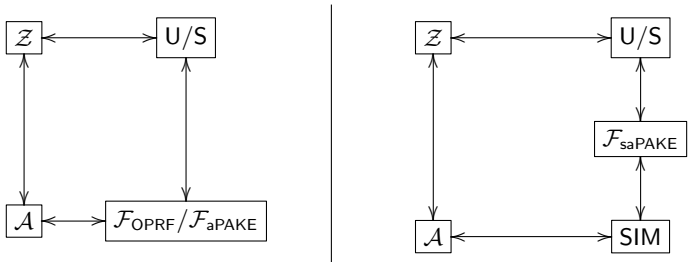
Theorem 1. *The protocol in Fig. 4 UC-realizes the $\mathcal{F}_{\mathsf{saPAKE}}$ functionality assuming access to the OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ and aPAKE functionality $\mathcal{F}_{\mathsf{aPAKE}}$.*

Proof. For any adversary $\mathcal{A}$, we construct a simulator SIM as in Fig. 5 and Fig. 6. Following [14], without loss of generality, we may assume that $\mathcal{A}$ is a "dummy" adversary that merely passes all its messages and computations to the environment $\mathcal{Z}$. We omit all interactions with corrupted U and S where SIM acts as $\mathcal{F}_{\mathsf{OPRF}}$ and $\mathcal{F}_{\mathsf{aPAKE}}$, since the simulation is trivial (SIM gains all information needed and simply follows the code of $\mathcal{F}_{\mathsf{OPRF}}/\mathcal{F}_{\mathsf{aPAKE}}$). To keep notation brief we denote functionality $\mathcal{F}_{\mathsf{saPAKE}}$ as $\mathcal{F}$.

We now show that the distinguishing advantage of $\mathcal{Z}$ between the real world and the simulated world is negligible. The argument uses a sequence of games, starting from the real world and ending at the simulated world; for any two adjacent games $\mathbf{G}_i$ and $\mathbf{G}_{i+1}$, let $\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_i,\mathbf{G}_{i+1}}$ denote the distinguishing advantage of $\mathcal{Z}$ between them, i.e.,

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_i,\mathbf{G}_{i+1}} = |\Pr[\mathcal{Z} \text{ outputs 1 in } \mathbf{G}_i] - \Pr[\mathcal{Z} \text{ outputs 1 in } \mathbf{G}_{i+1}]|.$$

($\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_i,\mathbf{G}_{i+1}}$ is a function of the security parameter τ, but we omit τ below.)



(a) real world

(b) simulated world

Initialization

(SIM assumes knowledge of server identity S and session identifier *sid* s.t. S initialized the SaPAKE instance via message StorePwdFile to $\mathcal{F}_{\mathsf{saPAKE}}$.)
Set tx := 0 and tested := $\varnothing$ and send (Init, S, *sid*) to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{OPRF}}$.

Stealing Password Data and Offline Queries

1. On (Compromise, *sid*) from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{OPRF}}$ and (StealPwdFile, *sid*) from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{aPAKE}}$, send (StealPwdFile, *sid*) to $\mathcal{F}$.
   If $\mathcal{F}$ returns "no password file," pass it to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.
   If $\mathcal{F}$ returns "password file stolen," mark S and $\langle \text{file}, \mathsf{U}, \mathsf{S}, \cdot \rangle$ compromised (record $\langle \text{file}, \mathsf{U}, \mathsf{S}, \perp \rangle$ and mark it compromised if there is no such record). Furthermore, if $\cdot$ is a string rw and rw $\in$ tested, then send rw to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$; else send "password file stolen" to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.
2. On (OfflineEval, *sid*, S, *x*) from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{OPRF}}$, if S is corrupted or marked compromised, send (OfflineEval, *sid*, $F_{\mathsf{S}}(x)$) to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{OPRF}}$ (pick $F_{\mathsf{S}}(x) \leftarrow_{\mathrm{R}} \{0,1\}^{\ell}$ if it is undefined) and (OfflineTestPwd, *sid*, *x*) to $\mathcal{F}$. If $\mathcal{F}$ returns "correct guess," retrieve $\langle \text{file}, \mathsf{U}, \mathsf{S}, \cdot \rangle$ (there must be such record, since if S is marked compromised, $\mathcal{A}$ must have sent (Compromise, *sid*) aimed at $\mathcal{F}_{\mathsf{OPRF}}$ and (StealPwdFile, *sid*) aimed at $\mathcal{F}_{\mathsf{aPAKE}}$ previously, and at that time $\langle \text{file}, \mathsf{U}, \mathsf{S}, \cdot \rangle$ was recorded); if the last item is $\perp$, replace it with $F_{\mathsf{S}}(x)$.
3. On message (OfflineTestPwd, *sid*, rw*) from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{aPAKE}}$, add rw* to tested. If there is a record $\langle \text{file}, \mathsf{U}, \mathsf{S}, \mathsf{rw} \rangle$ marked compromised, do:
   - If rw = rw*, send "correct guess" to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.
   - Else send "wrong guess" to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.

Fig. 5: The Simulator SIM in the Stealing Password Data Phase

Password Authentication

1. On (UsrSession, sid, $ssid$, U, S) from $\mathcal{F}$, send (Eval, sid, $ssid$, U, S) to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{OPRF}}$. Also, if this is the first UsrSession message for $ssid$, record $\langle ssid, \mathsf{U}, \mathsf{S}\rangle$ and mark it fresh.
2. On (SvrSession, sid, $ssid$, U, S) from $\mathcal{F}$, if there is no record $\langle \text{file}, \mathsf{U}, \mathsf{S}, \cdot\rangle$, record $\langle \text{file}, \mathsf{U}, \mathsf{S}, \perp\rangle$ and mark it uncompromised; regardless, send (SndrComplete, sid, $ssid$, S) and (SvrSession, sid, $ssid$, U, S) to $\mathcal{A}$ as messages from resp. $\mathcal{F}_{\mathsf{OPRF}}$ and $\mathcal{F}_{\mathsf{aPAKE}}$. Also, if this is the first SvrSession message for $ssid$, set $\mathsf{tx}{+}{+}$, record $\langle ssid, \mathsf{S}, \mathsf{U}\rangle$ and mark it fresh.
3. On (RcvComplete, sid, $ssid$, i^*) from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{OPRF}}$, retrieve $\langle ssid, \mathsf{U}, \mathsf{S}\rangle$; ignore this message if (i) there is no such record, or (ii) $i^* = \mathsf{S}$ and $\mathsf{tx} = 0$. Else augment the record to $\langle ssid, \mathsf{U}, \mathsf{S}, i^*\rangle$ and mark it fresh, send (UsrSession, sid, $ssid$, U, S) to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$, and if $i^* = \mathsf{S}$ then set $\mathsf{tx}{-}{-}$.

Active Session Attacks

1. On (TestPwd, sid, $ssid$, $\mathsf{P}, \mathsf{rw}^*$) from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{aPAKE}}$, if there is a record $\langle ssid, \mathsf{U}, \mathsf{S}, i^*\rangle$ (if $\mathsf{P} = \mathsf{U}$) or $\langle ssid, \mathsf{S}, \mathsf{U}\rangle$ (if $\mathsf{P} = \mathsf{S}$) marked fresh, mark it stale and check if there is an x such that $\mathsf{rw}^* = F_{i^*}(x)$ (if $\mathsf{P} = \mathsf{U}$) or $\mathsf{rw}^* = F_{\mathsf{S}}(x)$ (if $\mathsf{P} = \mathsf{S}$).
   - If there are more than one such x's, output collision and abort.
   - If there is a unique such x, send (TestPwd, sid, $ssid$, P, x) to $\mathcal{F}$ and pass it to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.
     Also, if $\mathsf{P} = \mathsf{S}$ and $\mathcal{F}$ returns "correct guess," retrieve $\langle \text{file}, \mathsf{U}, \mathsf{S}, \cdot\rangle$, and if the last item is $\perp$, replace it with rw^*.
   - If there is no such x, send "wrong guess" to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.
2. On (Impersonate, sid, $ssid$) from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{aPAKE}}$, if there is a record $\langle ssid, \mathsf{U}, \mathsf{S}, i^*\rangle$ marked fresh, mark it stale and do:
   - If $i^* = \mathsf{S}$, send (Impersonate, sid, $ssid$) to $\mathcal{F}$, and pass $\mathcal{F}$'s response ("correct guess" or "wrong guess") to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.
   - Else send "wrong guess" to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.

Key Generation and Authentication

1. On (NewKey, sid, $ssid$, P, SK^*) or (TestAbort, sid, $ssid$, P) from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{aPAKE}}$, if there is a record $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{S}^*\rangle$ (if $\mathsf{P} = \mathsf{U}$) or $\langle ssid, \mathsf{S}, \mathsf{U}\rangle$ (if $\mathsf{P} = \mathsf{S}$) not marked completed, pass the message from $\mathcal{A}$ to $\mathcal{F}$. In the case of (TestAbort, sid, $ssid$, P), also pass $\mathcal{F}$'s response (Succ or Fail) to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$. Finally, mark the record above completed.

Fig. 6: The Simulator SIM in the Login Phase

Games $\mathbf{G}_0$ and $\mathbf{G}_1$: $\mathbf{G}_0$ is the real world. In $\mathbf{G}_1$, on $(\text{UsrSession}, sid, ssid, \mathsf{S}, \mathsf{pw}')$ from $\mathcal{Z}$ to U and $(\text{RcvComplete}, sid, ssid, i^*)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{OPRF}}$, record $\langle ssid, \mathsf{U}, \mathsf{S}, i^*, \mathsf{rw}' \rangle$ (instead of $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{rw}' \rangle$). Obviously,

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0, \mathbf{G}_1} = 0.$$

Game $\mathbf{G}_2$: On $(\text{TestPwd}, sid, ssid, \mathsf{P}, \mathsf{rw}^*)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{aPAKE}}$, if there is a record $\langle ssid, \mathsf{U}, \mathsf{S}, i^*, \mathsf{rw}' \rangle$ (if $\mathsf{P} = \mathsf{U}$) or $\langle ssid, \mathsf{S}, \mathsf{U}, \mathsf{rw} \rangle$ (if $\mathsf{P} = \mathsf{S}$) marked fresh, check if there is an x such that $\mathsf{rw}^* = F_{i^*}(x)$ (if $\mathsf{P} = \mathsf{U}$) or $\mathsf{rw}^* = F_{\mathsf{S}}(x)$ (if $\mathsf{P} = \mathsf{S}$).

- If there are more than one such x's, output collision and abort.
- If there is a unique such x and $x = \mathsf{pw}'$ (if $\mathsf{P} = \mathsf{U}$) or $x = \mathsf{pw}$ (if $\mathsf{P} = \mathsf{S}$), send "correct guess" to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.
- In all other cases (i.e., $x \neq \mathsf{pw}'/\mathsf{pw}$ or there is no such x), send "wrong guess" to $\mathcal{A}$ as a message from $\mathcal{F}_{\mathsf{aPAKE}}$.

First consider event collision. collision occurs if and only if there are more than one x's such that $\mathsf{rw}^* = F_{i^*}(x)$ (if $\mathsf{P} = \mathsf{U}$) or $\mathsf{rw}^* = F_{\mathsf{S}}(x)$ (if $\mathsf{P} = \mathsf{S}$). This means that there are $x_1 \neq x_2$ such that $F_{i^*}(x_1) = F_{i^*}(x_2)$ or $F_{\mathsf{S}}(x_1) = F_{\mathsf{S}}(x_2)$. Note that $F_{\mathsf{S}}(\cdot)$ and $F_{i^*}(\cdot)$ are both random functions onto $\{0,1\}^{2\tau}$. Assuming that $\mathcal{A}$ sends q_{F} Eval and OfflineEval messages aimed at $\mathcal{F}_{\mathsf{OPRF}}$ in total and there are q_{U} U sub-sessions, there are at most $q_{\text{F}} + q_{\text{U}} + 1$ F values defined in total (q_{F} defined by $\mathcal{A}$'s actions, q_{U} defined by U's input to the protocol, and 1 by S's input to the protocol), so we have that

$$\Pr[\text{collision}] \leq \frac{(q_{\text{F}} + q_{\text{U}} + 1)^2}{2^{2\tau+1}}.$$

Next assume that collision does not occur. Consider the first message of type $(\text{TestPwd}, sid, ssid, \mathsf{P}, \mathsf{rw}^*)$ (note that $\mathcal{A}$ receives a reply for the first such message only, since $\langle ssid, \mathsf{P}, \mathsf{P}', \cdot \rangle$ becomes either compromised or interrupted after the first message). In both $\mathbf{G}_1$ and $\mathbf{G}_2$, $\mathcal{A}$ receives "correct guess" if and only if $\mathsf{rw}^* = F_{i^*}(\mathsf{pw}')$ (if $\mathsf{P} = \mathsf{U}$) or $\mathsf{rw}^* = F_{\mathsf{S}}(\mathsf{pw})$ (if $\mathsf{P} = \mathsf{S}$), so $\mathcal{Z}$'s views in $\mathbf{G}_1$ and $\mathbf{G}_2$ in this case are identical. We have that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_1, \mathbf{G}_2} \leq \Pr[\text{collision}] \leq \frac{(q_{\text{F}} + q_{\text{U}} + 1)^2}{2^{2\tau+1}},$$

which is a negligible function of τ.

Game $\mathbf{G}_3$: On $(\text{Impersonate}, sid, ssid)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{aPAKE}}$, if there is a record $\langle ssid, \mathsf{U}, \mathsf{S}, i^*, \mathsf{rw}' \rangle$ marked fresh, send "correct guess" to $\mathcal{A}$ if S is marked compromised, $i^* = \mathsf{S}$ and $\mathsf{pw}' = \mathsf{pw}$; otherwise send "wrong guess."

Similar with above, $\mathcal{A}$ receives a reply for the first Impersonate message only, so we only consider the first such message. Note that in $\mathbf{G}_2$ $\mathcal{A}$ receives "correct guess" if and only if S is compromised and $\mathsf{rw}' = \mathsf{rw}$, where $\mathsf{rw}' = F_{i^*}(\mathsf{pw}')$ and $\mathsf{rw} = F_{\mathsf{S}}(\mathsf{pw})$. $\mathcal{Z}$'s views in $\mathbf{G}_2$ and $\mathbf{G}_3$ are identical unless ($i^* \neq$

$\mathsf{S} \vee \mathsf{pw}' \neq \mathsf{pw}) \wedge F_{i^*}(\mathsf{pw}') = F_{\mathsf{S}}(\mathsf{pw})$ (in which case $\mathcal{A}$ receives "correct guess" in $\mathbf{G}_2$ and "wrong guess" in $\mathbf{G}_3$). Since $i^* \neq \mathsf{S}$ or $\mathsf{pw}' \neq \mathsf{pw}$, $F_{i^*}(\mathsf{pw}')$ and $F_{\mathsf{S}}(\mathsf{pw})$ are two independent random strings is $\{0,1\}^{2\tau}$; therefore, for a single U session, the probability that $F_{i^*}(\mathsf{pw}') = F_{\mathsf{S}}(\mathsf{pw})$ is at most $1/2^{2\tau}$. We have that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_2,\mathbf{G}_3} \leq \frac{q_{\mathrm{U}}}{2^{2\tau}},$$

which is a negligible function of τ.

Game $\mathbf{G}_4$: After $\mathcal{Z}$ sends (StorePwdFile, $sid, \mathsf{U}, \mathsf{pw}$) to S, record $\langle \text{FILE}, \mathsf{U}, \mathsf{S}, \perp \rangle$ (instead of $\langle \text{FILE}, \mathsf{U}, \mathsf{S}, \mathsf{rw} := F_{\mathsf{S}}(\mathsf{pw}) \rangle$); replace $\perp$ with $\mathsf{rw} := F_{\mathsf{S}}(\mathsf{pw})$ in the following two cases: (i) when $\mathcal{A}$ sends (OfflineEval, $sid, \mathsf{S}, \mathsf{pw}$) to $\mathcal{F}_{\mathsf{OPRF}}$, and S is corrupted or marked compromised; (ii) when $\mathcal{A}$ sends (TestPwd, $sid, ssid, \mathsf{S}, \mathsf{rw}^*$) to $\mathcal{F}_{\mathsf{aPAKE}}$, and $\mathsf{rw}^* = \mathsf{rw}$.

If neither (i) nor (ii) happens, $\mathsf{rw} = F_{\mathsf{S}}(\mathsf{pw})$ is a random string in $\{0,1\}^{2\tau}$ in $\mathcal{Z}$'s view. Therefore, replacing rw with $\perp$ in this case creates a $1/2^{2\tau}$ distinguishing advantage. We have that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_3,\mathbf{G}_4} \leq \frac{1}{2^{2\tau}},$$

which is a negligible function of τ.

Game $\mathbf{G}_5$: Postpone the recording of $\langle \text{FILE}, \mathsf{U}, \mathsf{S}, \cdot \rangle$ until (i) $\mathcal{A}$ sends (Compromise, sid) to $\mathcal{F}_{\mathsf{OPRF}}$ and (StealPwdFile, sid) to $\mathcal{F}_{\mathsf{aPAKE}}$, or (ii) S sends (SvrSession, $sid, ssid$) to $\mathcal{F}_{\mathsf{aPAKE}}$. Note that if neither (i) nor (ii) happens, $\mathbf{G}_4$ does not retrieve $\langle \text{FILE}, \mathsf{U}, \mathsf{S}, \cdot \rangle$. Therefore,

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_4,\mathbf{G}_5} = 0.$$

Note that in $\mathbf{G}_5$, $\mathsf{rw} = F_{\mathsf{S}}(\mathsf{pw})$ and $\mathsf{rw}' = F_{i^*}(\mathsf{pw}')$ are defined no matter $\mathcal{A}$ queries them (i.e., $\mathcal{A}$ sends (OfflineEval, $sid, \mathsf{S}, \mathsf{pw}$) to $\mathcal{F}_{\mathsf{OPRF}}$ when S is corrupted or marked compromised; or $\mathcal{A}$ sends (Eval, $sid, ssid, \mathsf{pw}'$) and then (RcvComplete, $sid, ssid, \mathcal{A}, i^*$) to $\mathcal{F}_{\mathsf{OPRF}}$) or not.

Game $\mathbf{G}_6$: Leave rw (resp. rw') undefined unless and until $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$ (resp. $F_{i^*}(\mathsf{pw}')$).

If $\mathcal{A}$ does not query $F_{\mathsf{S}}(\mathsf{pw})$ (resp. $F_{i^*}(\mathsf{pw}')$), rw (resp. rw') is a random string in $\{0,1\}^{2\tau}$ in $\mathcal{Z}$'s view. Since there is 1 rw and q_{U} rw''s, we have that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_5,\mathbf{G}_6} \leq \frac{q_{\mathrm{U}} + 1}{2^{2\tau}},$$

which is a negligible function of τ.

Game $\mathbf{G}_7$: $\mathbf{G}_7$ is the simulated world. We can see that the change from $\mathbf{G}_6$ to $\mathbf{G}_7$ is merely conceptual, with the game challenger split into the SaPAKE functionality $\mathcal{F}$ and the simulator SIM. We have that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_6,\mathbf{G}_7} = 0.$$

Summing up all results above, we conclude that $\mathcal{Z}$'s distinguishing advantage between the real world and the simulated world is a negligible function of τ. This completes the proof.

5 A Compiler from AKE-KCI to Strong aPAKE via OPRF

Our second transformation for building a Strong aPAKE protocol composes an OPRF with an Authenticated Key Exchange (AKE) protocol, "glued" together using authenticated encryption. We require the AKE to be secure in the UC model, namely, to realize the UC KE functionality of [18], and to also be "KCI secure." The latter notion was defined in [38] under a game-based formulation and formalized in Section 5.1 below in the UC setting.

5.1 UC Definition of AKE-KCI

The notion of KCI (for "key-compromise impersonation") security for KE protocols, concerns an attacker $\mathcal{A}$ who learns party P's long-term keys but otherwise does not actively control P. Resistance to KCI attacks, or "KCI security" for short, postulates that even though $\mathcal{A}$ can impersonate P to other parties, sessions which P itself runs with honest peers are unaffected and remain secure. A game-based definition of KCI security appears in [38], and here we formalize it in the UC model through functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$, shown in Fig. 7.[7]

Functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$ extends the standard KE functionality of [18] with two adversarial actions. The first, COMPROMISE, when issued at a party P models the leakage of P's secret keys to the attacker. In contrast to the case where P is corrupted, a COMPROMISED P is not controlled by the attacker $\mathcal{A}$ but $\mathcal{A}$ can actively impersonate P in sessions with P' by virtue of knowing P's keys. The second action, IMPERSONATE, represents this ability of the attacker: it is directed at sessions $\langle ssid, \mathsf{P}', \mathsf{P}\rangle$ where P is COMPROMISED and the result is that $\mathcal{A}$ gets to choose the session key via the NEWKEY action. Note that if P is compromised but not corrupted then sessions $\langle ssid, \mathsf{P}, \mathsf{P}'\rangle$ with an uncompromised and uncorrupted P' are not considered COMPROMISED. This captures the resistance to KCI attacks. All other elements in $\mathcal{F}_{\mathsf{AKE-KCI}}$ are the same as in the basic UC KE functionality, except of syntactic adjustments to the user-server setting.

Instantiations of AKE-KCI secure protocols. For concreteness we consider two examples of key-exchange protocols that can satisfy AKE-KCI security as defined above. While we do not include explicit proofs for these protocols here, we argue their security based on known results. The first

[7] The UC KCI-AKE functionality in Fig. 7 revises the UC KCI-AKE functionality which appeared in [33] by handling adaptive compromise of *either* party, and not only the server. The "two-sided adaptive" security of AKE is indeed needed in SaPAKE protocol shown in Section 5.2, as we explain in Section 5.3.

In the description below, we assume $\mathsf{P}, \mathsf{P}' \in \{\mathsf{U}, \mathsf{S}\}$.

- On (UsrSession, sid, $ssid$, S) from U, send (UsrSession, sid, $ssid$, U, S) to $\mathcal{A}^*$. If $ssid$ was not used before by U, record $\langle ssid, \mathsf{U}, \mathsf{S}\rangle$ and mark it FRESH.
- On (SvrSession, sid, $ssid$, U) from S, send (SvrSession, sid, $ssid$, U, S) to $\mathcal{A}^*$. If $ssid$ was not used before by S, record $\langle ssid, \mathsf{S}, \mathsf{U}\rangle$ and mark it FRESH.
- On (Compromise, sid, P) from $\mathcal{A}^*$, mark P COMPROMISED.
- On (Impersonate, sid, $ssid$, P) from $\mathcal{A}^*$, if P is marked COMPROMISED and there is a record $\langle ssid, \mathsf{P}', \mathsf{P}\rangle$ marked FRESH, mark this record COMPROMISED.
- On (NewKey, sid, $ssid$, P, SK^*) from $\mathcal{A}^*$ where $|SK^*| = \tau$, if there is a record $\langle ssid, \mathsf{P}, \mathsf{P}'\rangle$ not marked COMPLETED, do:
  - If the record is COMPROMISED, or P or P' is corrupted, set $SK := SK^*$.
  - If the record is marked FRESH, a $(sid, ssid, SK')$ tuple was sent to P', and at that time record $\langle ssid, \mathsf{P}', \mathsf{P}\rangle$ was marked FRESH, set $SK := SK'$.
  - Else pick $SK \leftarrow_{\mathrm{R}} \{0,1\}^\tau$.

  Finally, mark $\langle ssid, \mathsf{P}, \mathsf{P}'\rangle$ COMPLETED and send $(sid, ssid, SK)$ to P.

Fig. 7: Adaptively Secure Functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$

protocol we consider is the SIGMA protocol from [37] which is the basis for the key exchange protocols behind TLS 1.3 and IKEv2. SIGMA has been proven secure in [17] in the UC AKE formalization of [18]. While this formalization does not include the KCI property, KCI can easily be argued for SIGMA based on the unforgeability of the signature scheme. A second example is the HMQV protocol from [38] whose KCI property was proved in [38] in the game-based Canetti-Krawczyk model [16] extended to include KCI security. Here we require UC security, namely, a protocol that realizes functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$. Fortunately, [18] proves the equivalence of the game-based definition of [16] and their UC AKE formulation (in particular, this equivalence applies to the three-message HMQV with explicit authentication). While this UC formulation does not include KCI security, the equivalence with the game-based definition extends to the KCI case. Indeed, since the original equivalence from [18] holds even in the case of adaptive party corruptions, the Compromise and Impersonate actions introduced here – which constitute a *limited* form of adaptive corruptions – follow as a special case. Finally, we note that the equivalence between the above models also preserves forward secrecy, so this property (proved in the game-based Canetti-Krawczyk model in [38]) holds in the UC too. The security of HMQV (without including security against the leakage of ephemeral exponents) is based on the CDH assumption in the RO model [38].

Public Components:

- KCI-secure AKE protocol Π with private/public keys denoted p_s, P_s, p_u, P_u;
- Random-key robust and equivocable authenticated encryption scheme $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ with (2τ)-bit keys;
- Functionality $\mathcal{F}_{\mathsf{OPRF}}$ with output length parameter $\ell = 2\tau$.

Password Registration

1. On input $(\textsc{StorePwdFile}, sid, \mathsf{U}, \mathsf{pw})$, S generates public key pairs (p_s, P_s) and (p_u, P_u), and sends $(\textsc{Init}, sid)$ and $(\textsc{OfflineEval}, sid, \mathsf{S}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{OPRF}}$.
2. On $\mathcal{F}_{\mathsf{OPRF}}$'s response $(\textsc{OfflineEval}, sid, \mathsf{rw})$, compute $c \leftarrow \mathsf{AuthEnc}_{\mathsf{rw}}(p_u, P_u, P_s)$ and record $\mathsf{file}[sid] := (p_s, P_s, P_u, c)$.

Server Compromise

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

Password Authentication and Key Generation

1. On $(\textsc{UsrSession}, sid, ssid, \mathsf{S}, \mathsf{pw}')$, U sends $(\textsc{Eval}, sid, ssid, \mathsf{S}, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{OPRF}}$ and records $\mathcal{F}_{\mathsf{OPRF}}$'s response $(\mathsf{Prefix}, ssid, \mathsf{prfx})$.
2. On $(\textsc{SvrSession}, sid, ssid)$, S parses $\mathsf{file}[sid] = (p_s, P_s, P_u, c)$, sends $(ssid, c)$ to U and $(\textsc{SndrComplete}, sid, ssid)$ to $\mathcal{F}_{\mathsf{OPRF}}$. On $\mathcal{F}_{\mathsf{OPRF}}$'s response $(\mathsf{Prefix}, ssid, \mathsf{prfx}')$, S runs Π on (p_s, P_s, P_u) and $ssid_\Pi := [ssid||\mathsf{prfx}']$.
3. On $(\textsc{Eval}, sid, ssid, \mathsf{rw}')$ from $\mathcal{F}_{\mathsf{OPRF}}$ and c from S, U decrypts $m := \mathsf{AuthDec}_{\mathsf{rw}'}(c)$. If m can be parsed as (p'_u, P'_u, P'_s) then U retrieves $(\mathsf{Prefix}, ssid, \mathsf{prfx})$ and runs Π on input (p'_u, P'_u, P'_s) and $ssid_\Pi := [ssid||\mathsf{prfx}]$; Else U outputs $(\textsc{abort}, sid, ssid)$ and halts.
4. Given Π's local output SK, each party, either U or S, outputs $(sid, ssid, SK)$.

Fig. 8: OPRF-AKE: Strong aPAKE based on AKE-KCI and OPRF

5.2 Strong aPAKE Construction from OPRF and AKE-KCI

In Fig. 8 we present the Strong aPAKE protocol we call OPRF-AKE, because it is based on OPRF and AKE-KCI, "glued" with the equivocable robust symmetric encryption. The protocol uses the same OPRF tool as the Strong aPAKE construction of Section 4, for length parameter $\ell = 2\tau$, which defines the "randomized password" value $\mathsf{rw} = F_k(\mathsf{pw})$ for user U's password pw and OPRF key k held by server S. We assume that in the AKE-KCI protocol Π each party holds a (private,public) key pair, and that the each party runs the Login subprotocol using its key pair and the public key of the counterparty as inputs. In Password Registration phase, server S generates the user U's keys, and S's password file contains S's key pair p_s, P_s; U's public key P_u; and a ciphertext c of U's private key p_u, and the public keys P_u and P_s created using an Authenticated Encryption scheme using $\mathsf{rw} = F_k(\mathsf{pw})$ as the key. After

creating the password file, value p_u is erased at S. In Login phase, S runs OPRF with U, which lets U compute $\mathsf{rw} = F_k(\mathsf{pw})$, it sends c to U, who can decrypt it under rw and retrieves its key-pair p_u, P_u together with the server's key P_s, at which point both parties have appropriate inputs to the AKE-KCI protocol Π to compute the session key.

Role of Authenticated Encryption. Protocol OPRF-AKE utilizes an *authenticated encryption* scheme $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ to encrypt and authenticate U's AKE "credential" $m = (p_u, P_u, P_s)$. We encrypt the whole payload m for simplicity; in fact, unlike U's private key p_u, values P_u, P_s could be public and need to be only authenticated, not encrypted. However, the authentication property of AE must apply to the whole payload. Intuitively, U must authenticate S's public key P_s, but if U derived even its key pair (p_u, P_u) using just the secrecy of $\mathsf{rw} = F_k(\mathsf{pw})$, e.g., using rw as randomness in a key generation, and U then executed AKE on such (p_u, P_u) pair, the resulting protocol would already be insecure. To see an example, if an AKE leaks U's public key input P_u (note that AKE does not guarantee privacy of the public key) then an adversary $\mathcal{A}$ who engages U in a single protocol instance can find U's password pw via an offline dictionary attack by running the OPRF with U on some key k^*, and then given P_u leaked in the subsequent AKE it finds pw such that the key generation outputs P_u as a public key on randomness $\mathsf{rw} = F_{k^*}(\mathsf{pw})$.

Thus the role of the authentication property in authenticated encryption is to commit $\mathcal{A}$ to a single guess of rw and consequently, given the OPRF key k^*, to a single guess pw. (Note that our UC OPRF notion implies that F is collision-resistant.) To that end we need the authenticated encryption to satisfy the following property which we call *random-key robustness*:[8] For any efficient algorithm $\mathcal{A}$,

$$\mathbf{Adv}^{\mathrm{RBST},\mathsf{AE}}_{\mathcal{A}} = \Pr_{k_1,k_2 \leftarrow_R \{0,1\}^\tau}[c \leftarrow \mathcal{A}(k_1,k_2) \text{ s.t. } \mathsf{AuthDec}_{k_1}(c) \neq \perp, \mathsf{AuthDec}_{k_2}(c) \neq \perp]$$

is a negligible function of τ. In other words, it must be infeasible to create an authenticated ciphertext that successfully decrypts under two different randomly generated keys. This property can be achieved in the standard model using e.g., encrypt-then-MAC with a MAC that is collision resistant with respect to the message and key, a property enjoyed by HMAC with full hash output. In the RO model used by our aPAKE application one can also enforce it for any authenticated encryption scheme by attaching to its ciphertext c a hash $H(k, c)$ for a RO hash H with 2τ-bit outputs.

Encryption Equivocability. In the security argument we also need the authenticated encryption scheme to be *equivocable* in the following sense: In the scenario where the adversary gets a ciphertext followed by the key, there is

[8] This notion is a weakening of *full robustness (FROB)* from [23] where the attacker is allowed to choose k_1, k_2 (in our case these keys are random). An even weaker notion, Semi-FROB, is defined in [23] where k_1, k_2 are random but only k_1 is provided to $\mathcal{A}$.

a simulator who first creates the ciphertext with no information about the plaintext, and then creates the key given the plaintext. Formally, an authenticated encryption scheme AE is equivocable if for any efficient algorithm $\mathcal{A}$, there is an efficient *stateful* simulator $\mathsf{SIM}_{\mathsf{EQV}}$ such that the distinguishing advantage of $\mathcal{A}$'s views in the following two games, $\mathbf{Adv}_{\mathcal{R}}^{\mathrm{EQV},\mathsf{AE}}$, is a negligible function of τ:

- The real game: $\mathcal{A}$ sends out message m, and computes its final output given (c, k) produced as $k \leftarrow_{\mathrm{R}} \{0,1\}^{\tau}$ and $c \leftarrow \mathsf{AuthEnc}_k(m)$.
- The ideal game: $\mathcal{A}$ sends out message m, and computes its final output given (c, k) produced as $c \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(|m|)$ and $k \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(m)$.

(Since $\mathsf{SIM}_{\mathsf{EQV}}$ is stateful, we assume that when computing $c \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(|m|)$ it creates an internal state which it then utilizes to compute $k \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(m)$.)

Common encryption modes are equivocable under some idealized assumption. For example, an encryption scheme that xor's the message m with a pad generated by a pseudorandom generator G is equivocable if we model G as a random oracle. In this case, in response to an $\mathsf{AuthEnc}$ query, $\mathsf{SIM}_{\mathsf{EQV}}$ will choose the ciphertext c at random; then, to respond to a corresponding Reveal query with message m, $\mathsf{SIM}_{\mathsf{EQV}}$ programs G's output to $c \oplus m$. If the PRG is implemented in counter mode using a block cipher a similar strategy works but in the ideal cipher model. The above can be extended to Authenticated Encryption modes. For example, if a MAC is computed on the ciphertext, $\mathsf{SIM}_{\mathsf{EQV}}$ responds to an $\mathsf{AuthEnc}$ query by choosing a random MAC key k and outputting $(c, \mathsf{mac}_k(c))$ where c is chosen as above. To output the MAC key upon a Reveal query with message m, $\mathsf{SIM}_{\mathsf{EQV}}$ outputs k (which is independent of the message hence it works with any message).

Note on not utilizing $\mathcal{F}_{\mathsf{AKE-KCI}}$. In Fig. 8 we abstract the OPRF protocol as functionality $\mathcal{F}_{\mathsf{OPRF}}$, but we use the real-world AKE-KCI protocol Π, rather than functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$. The reason for this presentation is that in the KE functionality of [18], of which $\mathcal{F}_{\mathsf{AKE-KCI}}$ is an extension, it is not clear how to support a usage of the KE protocol on keys which are computed via some other mechanism than the intended KE key generation. The KE functionality of [18] assumes that each entity keeps its private key as a permanent state, authenticates to a counterparty given its identity, and a KE party cannot specify any bitstring as one's own private key and a counterparty's public key. This is not how we use AKE in protocol OPRF-AKE in Fig. 8 precisely because U does not keep state and has to reconstruct its keys from a password (via OPRF). However, we can still use the real-world protocol Π, which UC-realizes $\mathcal{F}_{\mathsf{AKE-KCI}}$, giving it the OPRF-computed information as input. In the proof of security we utilize the simulator $\mathsf{SIM}_{\mathsf{AKE}}$, which shows that Π UC-realizes $\mathcal{F}_{\mathsf{AKE-KCI}}$, in our simulator construction, but we rely on its correctness only if U runs Π on the correctly reconstructed (p_u, P_s, P_s), and if the adversary causes U to reconstruct a different string we interpret this as a successful attack on U's login session.

Role of OPRF Transcript Prefixes. As discussed in Section 3, the UC OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ used here extends the corresponding functionality of the

proceedings version of this paper [33] by having OPRF parties U and S output an OPRF transcript prefix, whose implications are that for every S session the adversary must follow one of the following paths: (I) If the transcript prefix output by some S session matches the transcript output by some U session, then this S session can only be used to let U compute $F_k(\mathsf{pw}')$ on U's input pw'; or (II) If the transcript prefix output by S does not match any prefix output by some U, then this S session can only be used by the adversary, and in particular it cannot be "connected" to some honest U session.

This property helps in the security proof of the SaPAKE protocol of Fig. 8. The difference between that protocol and its variant shown in the proceedings version of this paper [33], is that when U and S complete their OPRF interaction, each party outputs an OPRF transcript prefix prfx, and when each party starts its end of the AKE (sub)protocol, it runs the AKE with (sub)session ID's computed as $ssid_{\Pi} := [ssid||\mathsf{prfx}]$, where $ssid$ is the identifier of a particular instance of the SaPAKE login protocol.

The security property of AKE, see Fig. 7, implies that U and S cannot establish a shared session key unless they run on the same (sub-)session ID's. With sub-session ID's defined as above, this implies that they cannot establish a shared key unless their OPRF transcript prefixes match. This can be useful in a security proof because for each server S session the adversary has to decide if this S session can be "connected" to some honest U, because its OPRF transcript prefixes match (case I), or this S session is actively attacked, and cannot be matched with any U session, because its OPRF transcript prefix does not match that of any U session (case II). This can help in the security argument, because it lets the simulator decide which S session is (potentially) actively attacked (case II), and which one is not (case I).

Here is why the simulator needs to be able to make this decision, and why without this "no active attack if transcript prefixes match" property the security argument does not quite work. (Indeed, this problem appears in the variant of this protocol which was in the proceedings version of this paper [33], and this is why we modify this protocol as described above.) Consider an adversary $\mathcal{A}$ playing a man-in-the-middle attack between n simultaneous U sessions and n S sessions. Consider an OPRF protocol which, like the DH-OPRF scheme of Fig. 13 in Appendix B, is malleable, i.e. $\mathcal{A}$ can blind U's messages to S and de-blind S's responses, and even though U-$\mathcal{A}$ and $\mathcal{A}$-S communications cannot be externally linked, $\mathcal{A}$ actually lets U compute the correct outputs in the OPRF protocol with S. In the case of protocol DH-OPRF, the adversary $\mathcal{A}$ can do this by replacing U's message a with message $a' = (a)^t$ to S, for $t \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, and then replacing S's reply $b' = (a')^k$ with reply $b = (b')^{1/t}$ to U. Note that $b = a^k$ so U's interaction with $\mathcal{A}$ is just as good as an interaction with S, but the U-$\mathcal{A}$ interaction cannot be linked, by an external observer, to any unique S session.

This creates a problem in the simulation because if the simulator observes that $\mathcal{A}$ locally computes $F_k(\mathsf{pw}^*)$ for some pw^*, e.g. by observing $\mathcal{A}$'s hash function queries, the simulator needs to make $(\textsc{TestPwd}, ..., \mathsf{S}, \mathsf{pw}^*)$ query against *some* session of S, to program the $F_k(\mathsf{pw}^*)$ output depending on

whether $\mathsf{pw}^* = \mathsf{pw}$ or not. However, once the simulator sends TestPwd to S session, that session becomes interrupted (assume $\mathsf{pw}^* \neq \mathsf{pw}$), and cannot be then connected, i.e. output the same session key, with any U session. If the $\mathcal{A}$-S and U-$\mathcal{A}$ interactions are unlinkable then the simulator can only choose S session for TestPwd query at random. If $\mathcal{A}$ makes half of S session connect to half of U sessions, while using the other half for local computation of $F_k(\cdot)$ on $n/2$ password tests, the simulator needs to make too many guesses to be successful with non-negligible probability. Fortunately, using (the hash of) the OPRF transcript prefix (defined as the value a in this case) in the AKE protocol solves this problem because now to make any U and S session connect in the AKE instance the adversary has to make $a' = a$, which the simulator can detect. Since the modified OPRF security property says that such prefix-matched OPRF sessions cannot be used by $\mathcal{A}$ to compute $F_k(\cdot)$, if the simulator detects such computation by $\mathcal{A}$, it can be tested against any S session whose prefix *does not* match message a sent by any user U.

Role of Strong aPAKE Model Relaxations. As discussed in Section 2, our notion of Strong aPAKE functionality $\mathcal{F}_{\mathsf{saPAKE}}$, Fig. 2, is a relaxation of a variant of this notion presented in the proceedings version of this paper [33], which we denote here as $\mathcal{F}_{\mathsf{saPAKE}^+}$, Fig. 1. We introduced these relaxations because they model the adversarial behavior that is possible in protocol OPRF-AKE, and although in Section 2 we explain that these relaxations are mild and should be insignificant in practice, they are nevertheless necessary, and in particular, protocol OPRF-AKE cannot be shown to realize the stronger functionality $\mathcal{F}_{\mathsf{saPAKE}^+}$.

Recall from Section 2 that there are two relaxations which are introduced in functionality $\mathcal{F}_{\mathsf{saPAKE}}$: (1) An attacker who tests the correct password guess on some server S session can compromise all other *open*, i.e. not completed, S sessions, even if the adversary already effectively tested other passwords on these sessions; (2) An attacker must commit to a tested password for each S session, but that tested password might be efficiently extractable, by the simulator, only after the attacked session completes (although the attacker in that case learns only if its password guess was correct or not, and learns no information about the session key SK which that S session outputted, even if its password guess was correct).

The need for both relaxations with regards to protocol OPRF-AKE stem from two features of our notion of UC OPRF. The first one is a lack of a binding between the server-side and client-side view of OPRF instances. Note that instances of server S OPRF evaluation are indexed by (sub)session ID's $ssid_s$, and each of them allows the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid world adversary $\mathcal{A}$ to compute function F_k on one argument x, as modeled by a sequence of queries $(\text{Eval}, sid, ssid^*, \mathsf{S}, x)$ and $(\text{RcvComplete}, sid, ssid^*, \mathcal{A}, \mathsf{S})$ from $\mathcal{A}$. However, the OPRF functionality does not enforce any correspondence between the server-side (sub)session ID $ssid_s$ of S and the client-side "session" pointer $ssid^*$ used by $\mathcal{A}$. Therefore if the environment opens n S sessions simultaneously (note that all of them operate on the same password pw), then when $\mathcal{A}$

evaluates $F_k(\mathsf{pw}^*)$ through EVAL+RCVCOMPLETE commands above this constitutes a password test (note that if $\mathsf{pw}^* = \mathsf{pw}$ then $\mathsf{rw} = F_k(\mathsf{pw}^*)$ can decrypt user-credentials (p_u, P_u, P_s) from the authenticated ciphertext c), but each such test is in effect a test against *all* open S sessions, because if n OPRF-AKE S instances trigger n OPRF S instances then $\mathcal{A}$ can test n passwords $\mathsf{pw}_1^*, \ldots, \mathsf{pw}_n^*$, and if *any* of these tests is correct then $\mathcal{A}$ can use the decrypted credentials (p_u, P_u, P_s) as inputs to AKE subprotocols of *all* of these n S sessions, and learn/determine the sessions keys $SK_1^*, ..., SK^*n$ on all of them.

Note that this behavior is not allowed in the standard notion of UC (Sa)PAKE: There each of the n simultaneously open S sessions are treated separately, and the adversary can determine/learn only SK_i^* for $\mathsf{pw}_i^* = \mathsf{pw}$, whereas the other secret keys, on sessions corresponding to the $n-1$ incorrect password guessess $\mathsf{pw}_j^* \neq \mathsf{pw}$, would remain secure. As described in Section 2, the relaxed functionality $\mathcal{F}_{\mathsf{saPAKE}}$ allows $\mathcal{A}$ to compromise all incomplete sessions if one of them is COMPROMISED, and the OPRF-AKE simulator SIM utilizes this relaxation by sending $(\text{TESTPWD}, sid, ssid', \mathsf{S}, x)$ to $\mathcal{F}_{\mathsf{saPAKE}}$, given $\mathcal{A}$'s queries $(\text{EVAL}, ..., x)$ and RCVCOMPLETE to $\mathcal{F}_{\mathsf{OPRF}}$, for $ssid'$ corresponding to *any* S session which is not completed. (See step (4c) in SIM in Fig. 10.)

Note also that the above disconnection between the server-side and the client-side OPRF evaluation is not only a feature of our abstract OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$, but also of protocol DH-OPRF, see Appendix B, which implements this OPRF notion. (Indeed, this is implied by the fact that protocol DH-OPRF realizes $\mathcal{F}_{\mathsf{OPRF}}$.) Server-side instances of DH-OPRF are formed by an in-coming message a, a random group element, and the server response $b = a^k$, while the client evaluation of $F_k(x)$ is constitued by a hash function query $H_2(x, v)$ for $v = (H_1(x))^k$, thus there is no link between the client-side evaluation of $(H_1(x))^k$ for any particular argument x, with any particular instance of server S OPRF evaluation.

The second relaxation reflects the fact that $\mathcal{A}$ can evaluate F_k on a password guess pw^* *after* it interacts with S in the AKE's sub-protocol, at which point S session completes. If none of the on-line password tests succeeded so far this AKE instance is secure, and hence is this overall OPRF-AKE instance, because $\mathcal{A}$ doesn't know the user's authentication keys (p_u, P_u, P_s). However, if $\mathcal{A}$ evaluates $F_k(\mathsf{pw}^*)$ after this session ends it still learns if $\mathsf{pw}^* = \mathsf{pw}$, because $\mathcal{A}$ can test if $F_k(\mathsf{pw}^*)$ correctly decrypts ciphertext c. In the standard UC (Sa)PAKE functionality this would not be simulatable, and thus we relax $\mathcal{F}_{\mathsf{saPAKE}}$ so it allows such "delayed" password test. The OPRF-AKE simulator SIM relies on this relaxation by sending INTERRUPT on all actively attacked S sessions in step (2a) in Fig. 10, and by reacting to $(\text{EVAL}, \ldots, x)$ message in step (4c) by doing $(\text{TESTPWD}, \ldots, x)$ on any non-completed S session if any exist, and if they are not then doing a "postponed" password test on some completed (and previously untested) S session in step (4(c)i).

Role of Adaptive Security of AKE and of Equivocable Encryption. Let us also explain why we need the AKE protocol Π to allow adaptive compromise

for both parties, and why we need the authenticated encryption scheme AE to be equivocable. Assume for now that the two parties' passwords match, i.e., $\mathsf{pw}' = \mathsf{pw}$. Recall that an adversary $\mathcal{A}$ may compromise S at any time and perform an offline dictionary attack, and that $\mathcal{A}$ can perform an on-line dictionary attacks by running the authentication protocol protocol with different password guesses. Observe the following facts:

- If $\mathcal{A}$ does not compromise S, then $\mathcal{Z}$ learns no information about p_s. On the other hand, if $\mathcal{A}$ compromises S, then $\mathcal{Z}$ learns p_s as part of $\mathsf{file}[sid]$. This is equivalent to S being compromised in Π.
- If $\mathcal{A}$ does not run an either online or offline dictionary attack then $\mathsf{rw} = F_{\mathsf{S}}(\mathsf{pw})$ is a random string in $\mathcal{Z}$'s view, so by security of AE, $\mathcal{Z}$ learns no information about p_u. On the other hand, if $\mathcal{A}$ succeeds in either online or offline dictionary attack and learns $\mathsf{rw} = F_{\mathsf{S}}(\mathsf{pw})$, then $\mathcal{Z}$ learns p_u by parsing $m = \mathsf{AuthDec}_{\mathsf{rw}}(c)$. This is equivalent to U being compromised in Π.

Note that $\mathcal{A}$ may compromise S and/or compute $F_{\mathsf{S}}(\mathsf{pw})$ at any time. Hence, we need the AKE protocol Π to tolerate adaptive corruptions of *either* party.

This also shows why we need AE to be equivocable. Consider an adversary $\mathcal{A}$ who first sees c (in a message from S to U) and then learns rw (via an offline attack). The security of Π relies on the condition that p_u is kept random in $\mathcal{A}$'s view until U is compromised (which, as argued above, is equivalent to $\mathcal{A}$ learning rw). However, $\mathcal{A}$ first sees the ciphertext c, and then learns the decryption key rw, so two things must hold about encryption AE: (1) Ciphertext $c = \mathsf{AE}(\mathsf{rw}, m)$ must hide all information about the encrypted plaintext m until rw is revealed; (2) The adversary might at some point learn the decryption key rw s.t. $c = \mathsf{AE}(\mathsf{rw}, m)$. Standard encryption security does not guarantee the security in this *adaptive* setting, and in particular the standard simulation strategy of replacing c with an encryption of an unrelated value m' would fail requirement (2) above. The equivocability of AE is exactly what is needed here, because it implies an efficient simulator which can create ciphertext c given only the length of the encrypted plaintext (hence until rw is revealed c leaks nothing about p_u), and then given any plaintext m, e.g. $m = (p_u, P_u, P_s)$, it can produce rw s.t. $c = \mathsf{AuthEnc}(\mathsf{rw}, m)$.

5.3 Proof of Security

We state and prove the Strong aPAKE security of the generic composition protocol OPRF-AKE from Fig. 8 in the following theorem:

Theorem 2. *If protocol Π UC-realizes functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$, and AE is a random-key robust and equivocable authenticated encryption scheme, then the protocol in Fig. 8 UC-realizes functionality $\mathcal{F}_{\mathsf{saPAKE}}$ in the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid model.*

Overview of Simulation Strategy. We start with a high-level description of the simulation strategy, whereas the detailed simulation algorithm SIM is contained in Fig. 9, 10, and 11, each figure dealing with the different aspect of

protocol OPRF-AKE and hence the simulation as well. Recall that the security proof must construct a simulator SIM which interacts as an ideal-world adversary with functionality $\mathcal{F}_{\mathsf{saPAKE}}$, and creates an indistinguishable view to the environment as that of a real-world adversary $\mathcal{A}$ interacting with the honest parties in protocol OPRF-AKE.

Consider first the simulation of the password file storage, i.e., the offline security of OPRF-AKE. The actions of simulator SIM regarding this phase are described in Fig. 9, but the idea is that SIM generates a virtual $\mathcal{F}_{\mathsf{OPRF}}$ instance F_{S} (instead of computing $\mathsf{rw} := F_{\mathsf{S}}(\mathsf{pw})$) and generates c using simulator $\mathsf{SIM}_{\mathsf{EQV}}$ assumed by the equivocability of encryption AE. If server S later becomes compromised, $\mathcal{A}$ learns $\mathsf{file}[sid]$ (which SIM obtains by sending $(\textsc{Compromise}, sid, \mathsf{S})$ to $\mathsf{SIM}_{\mathsf{AKE}}$) and also gains offline evaluation access to the $\mathcal{F}_{\mathsf{OPRF}}$ instance simulated by SIM. At this point $\mathcal{A}$ can stage an offline dictionary attack on the password file, by sending $(\textsc{OfflineEval}, sid, \mathsf{S}, x)$ queries aimed at $\mathcal{F}_{\mathsf{OPRF}}$. The simulator SIM services each such query by sending $(\textsc{OfflineTestPwd}, sid, x)$ to $\mathcal{F}_{\mathsf{saPAKE}}$. If $\mathcal{F}_{\mathsf{saPAKE}}$ replies "wrong guess" then SIM replies to $\mathcal{A}$ with $F_{\mathsf{S}}(x) \leftarrow_{\mathrm{R}} \{0,1\}^{\ell}$. If, however, $\mathcal{F}_{\mathsf{saPAKE}}$ replies "correct guess" then SIM learns that $\mathcal{A}$'s password guess x is equal to the password pw for which this $\mathcal{F}_{\mathsf{saPAKE}}$ instance was initialized, and in this case SIM lets $\mathsf{SIM}_{\mathsf{EQV}}$ reveal rw s.t. $c = \mathsf{AuthEnc}(\mathsf{rw}, (p_u, P_u, P_s))$ and "programs" value $F_{\mathsf{S}}(x)$ on $x = \mathsf{pw}$ to rw.[9] By the "random even to the key holder" property of UC OPRF, the adversary's view of F_{S} outputs set as above is identical to those in the real protocol.

Regarding the login phase, i.e., the online security of OPRF-AKE, let us first fix some notation. We use i^* to denote the function pointer used by $\mathcal{A}$ in $(\textsc{RcvComplete}, sid, ssid, \mathsf{U}, i^*)$ for U's OPRF session, and c^* to denote the ciphertext in the message which $\mathcal{A}$ passes to U after OPRF evaluation. As in functionality $\mathcal{F}_{\mathsf{saPAKE}}$, Fig. 2, we use pw to denote S's password, and pw' to denote U's password. The details of the simulation procedure regarding the online phase are divided between Fig. 10, where we show how SIM reacts to $\mathcal{A}$'s messages to the OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ (recall that $\mathcal{A}$ is an adversary in the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid world), and Fig. 11, where we show how SIM reacts to $\mathcal{A}$'s messages related to AKE protocol Π. However, the main ideas of the simulation can be explained by considering the two cases of the man-in-the-middle adversary, who can (1) emulate the server to honest user U instances and (2) emulate the user to honest server S instances, and below we overview how SIM handles each case.

When $\mathcal{A}$ plays as the server to some user instance $(sid, ssid, \mathsf{U})$, the key observation is that U outputs $(\textsc{abort}, sid, ssid)$ with overwhelming probability, except for either of the following two cases:

- Case $(*)$ corresponds to line (1a) in AKE Simulation part of Simulator SIM in Fig. 11, and it involves the following two conditions: (i) $\mathcal{A}$ is passive until the execution of Π begins, i.e., $i^* = \mathsf{S}$ and $c^* = c$, and (ii) the two parties' passwords match, i.e., $\mathsf{pw}' = \mathsf{pw}$. (Note that SIM can test if $\mathsf{pw}' = \mathsf{pw}$ via a

[9] Note that after S compromise, $\mathcal{A}$ could also guess the correct password pw in an *online* attack, but as we argue below SIM we can handle that in a similar way.

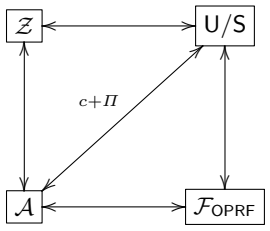
TestAbort message.) In this case U's input is the "correct" (p_u, P_u, P_s), so SIM can outsource the simulation of AKE protocol Π on behalf of this U's instance, denoted Π_u, to $\mathsf{SIM}_{\mathsf{AKE}}$.

- Case $(**)$ corresponds to line (1(b)iii) in AKE Simulation part of Simulator SIM in Fig. 11, involves the condition that $\mathcal{A}$ computes $\mathsf{rw}' = F_{i^*}(\mathsf{pw}')$, which can happen (i) in an online OPRF instance between $\mathcal{A}$ and S if $i^* = \mathsf{S}$, or (ii) via offline computation $(\textsc{OfflineEval}, sid, i^*, \mathsf{pw}')$ if $i^* = \mathsf{S}$ and S is compromised or corrupted, or (iii) via offline computation $(\textsc{OfflineEval}, sid, i^*, \mathsf{pw}')$ if $i^* \neq \mathsf{S}$. In all of these cases $\mathcal{Z}$ can choose (p_u^*, P_u^*, P_s^*) and set the ciphertext c^* as $c^* \leftarrow \mathsf{AuthEnc}_{\mathsf{rw}'}(p_u^*, P_u^*, P_s^*)$. However, SIM, who sees $\mathcal{A}$'s OPRF queries, will learn the same information as well, so it can simulate U's behavior by running Π_u on the same inputs as the real-world U would use. Here the random-key robustness of AE is needed, because it guarantees that c^* decrypts to $m \neq \perp$ for at most one key (with overwhelming probability), which allows SIM to determine (i^*, pw') such that c^* is a valid encryption under key $\mathsf{rw}' = F_{i^*}(\mathsf{pw}')$.

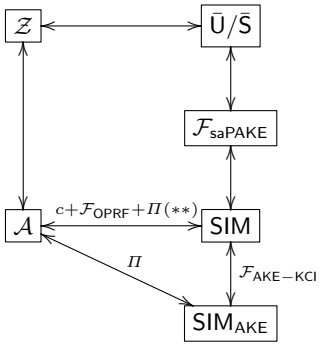
When $\mathcal{A}$ plays as the user to some server instance $(sid, ssid, \mathsf{S})$, the input of S is always the "correct" (p_u, P_u, P_s). Therefore, SIM can outsource the simulation of AKE protocol Π on behalf of this S's instance, denoted Π_s, to $\mathsf{SIM}_{\mathsf{AKE}}$. Note that user's AKE authentication token p_u is hidden to $\mathcal{A}$ unless and until $\mathcal{A}$ computes $\mathsf{rw} = F_{\mathsf{S}}(\mathsf{pw})$, via either guessing pw and then evaluating OPRF on pw, or by compromising S and then running an offline dictionary attack. In both cases SIM can extract pw by observing $\mathcal{A}$'s interaction with $\mathcal{F}_{\mathsf{OPRF}}$, send $(\textsc{TestPwd}, sid, ssid, \mathsf{S}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{saPAKE}}$ and learn that pw is S's password, and then send $(\textsc{Compromise}, sid, ssid, \mathsf{U})$ to $\mathsf{SIM}_{\mathsf{AKE}}$.

Simulation Components. Since protocol OPRF-AKE relies on the UC security of two components, OPRF and AKE, we briefly describe how the real world and the ideal world interactions involve the protocols, functionalities, or simulators of these components. We describe these two scenarios schematically in a diagram below, where for simplicity we call the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid world execution the "real world". However, since it is an execution in the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid world, it involves an interaction between $\mathcal{A}$ and U/S via functionality $\mathcal{F}_{\mathsf{OPRF}}$, while the direct interaction between $\mathcal{A}$ and U/S pertains to the other two components of OPRF-AKE, namely the AE ciphertext c, and the AKE protocol Π. Regarding the ideal world execution, recall that protocol Π realizes functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$, hence there is a simulator $\mathsf{SIM}_{\mathsf{AKE}}$ which can be used to simulate Π's execution. Indeed, our simulator SIM uses simulator $\mathsf{SIM}_{\mathsf{AKE}}$ as a sub-routine, essentially "outsourcing" to $\mathsf{SIM}_{\mathsf{AKE}}$ the interactions of $\mathcal{A}$ with (1) all Π executions run by S instances, and (2) those Π executions run by U instances which fall into case $(*)$ above, i.e. where U runs on its intended inputs (p_u, P_u, P_s). On these Π instances SIM passes all the messages between $\mathcal{A}$ and $\mathsf{SIM}_{\mathsf{AKE}}$, which we denote in part (b) of the diagram below as the direct link between $\mathcal{A}$ and $\mathsf{SIM}_{\mathsf{AKE}}$. All the other aspects of the simulation, namely (1) the $\mathsf{AuthEnc}$ ciphertext c, (2) the emulation of the OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$, and (3) Π executions of U instances which fall into case

(**), will be handled directly by SIM. Finally, note that simulator $\mathsf{SIM}_{\mathsf{AKE}}$, while interacting with $\mathcal{A}$, expects to communicate with the ideal AKE functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$. Simulator SIM will therefore internally emulate two functionalities: The OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$, used for interactions with $\mathcal{A}$, and the AKE functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$, used for interactions with $\mathsf{SIM}_{\mathsf{AKE}}$.



(a) real world



(b) simulated world

Proof by Game Changes. As is standard in UC security proofs, we assume w.l.o.g. that $\mathcal{A}$ is a "dummy" adversary who merely passes through all its messages to and from $\mathcal{Z}$. To keep notation brief we denote functionality $\mathcal{F}_{\mathsf{saPAKE}}$ as $\mathcal{F}$, and we use Π_u and Π_s for, respectively, U's and S's algorithm in the execution of protocol Π. We use S and *sid* to denote the unique server entity and session identifier such that S initialized this SaPAKE instance via command (StorePwdFile, *sid*, ...) to $\mathcal{F}_{\mathsf{saPAKE}}$, and we assume that SIM knows both identifiers.

Let $\mathbf{Adv}^{\mathrm{EQV},\mathsf{AE}}_{\mathcal{R}}$, $\mathbf{Adv}^{\mathrm{AUTH},\mathsf{AE}}_{\mathcal{R}}$, and $\mathbf{Adv}^{\mathrm{RBST},\mathsf{AE}}_{\mathcal{R}}$ denote an advantage of algorithm $\mathcal{R}$ in, respectively, the equivocability game, authenticity game, and random-key robustness game of authenticated encryption scheme AE. Fix an efficient environment $\mathcal{Z}$, and a dummy adversary $\mathcal{A}$, and let q_{F} be the number of Eval and OfflineEval messages sent by $\mathcal{A}$ or $\bar{\mathsf{U}}$ given this $\mathcal{Z}$, and let q_{U} be the number of U SaPAKE instances, i.e. the number of UsrSession queries sent to all user U entities by $\mathcal{Z}$. We will show that the advantage of $\mathcal{Z}$ in distinguishing between the real world and the simulated world executions is negligible, and we do so using a sequence of games, starting from the real world and ending at the simulated world. For any two adjacent games we use $\mathbf{Dist}^{\mathbf{G}_i,\mathbf{G}_{i+1}}_{\mathcal{Z}}$ to denote the advantage of $\mathcal{Z}$ in distinguishing $\mathbf{G}_i$ and $\mathbf{G}_{i+1}$.

Game $\mathbf{G}_0$: $\mathbf{G}_0$ is the real world.

Note that in $\mathbf{G}_0$, an instance of U and sub-session ID *ssid* results in U outputting (abort, *sid*, *ssid*) if and only if it receives (*ssid*, c^*), $\mathcal{A}$ specifies

Initialization

Set $\mathsf{tx} := 0$. Initialize $\mathsf{SIM}_{\mathsf{AKE}}$ and 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 $F_i(x) \leftarrow_{\mathrm{R}} \{0,1\}^{\ell}$. Set $c \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(k_{pr} + 2k_{pb})$, where k_{pr}, k_{pb} are the lengths of private/public AKE keys, and record $\mathsf{file}[sid] := (\perp, \perp, \perp, c)$.

Stealing Password Data and Offline Queries

1. On $(\textsc{Compromise}, sid)$ aimed at $\mathcal{F}_{\mathsf{OPRF}}$ and $(\textsc{StealPwdFile}, sid)$ aimed at S from $\mathcal{A}$ (we assume $\mathcal{A}$ sends these commands together), send $(\textsc{StealPwdFile}, sid)$ to $\mathcal{F}$. If $\mathcal{F}$ returns "no password file," pass this message to $\mathcal{A}$ on behalf of S. Otherwise declare S COMPROMISED, send $(\textsc{Compromise}, sid, \mathsf{S})$ to $\mathsf{SIM}_{\mathsf{AKE}}$, and on response (p_s, P_s, P_u) reset $\mathsf{file}[sid] := (p_s, P_s, P_u, c)$ and send it to $\mathcal{A}$ on behalf of S.
2. On $(\textsc{OfflineEval}, sid, i^*, x)$ from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{OPRF}}$, do the following:
   - If $i^* = \mathsf{S}$ and S is not COMPROMISED, ignore this message.
   - If $i^* = \mathsf{S}$ and S is COMPROMISED, send $(\textsc{OfflineTestPwd}, sid, x)$ to $\mathcal{F}$. If $\mathcal{F}$ returns "correct guess," send $(\textsc{Compromise}, sid, \mathsf{U})$ to $\mathsf{SIM}_{\mathsf{AKE}}$, and on response (p_u, P_u, P_s) set $\mathsf{rw} \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(p_u, P_u, P_s)$ and $F_{\mathsf{S}}(x) := \mathsf{rw}$, record $\langle \textsc{file}, \mathsf{U}, \mathsf{S}, x \rangle$, and declare U COMPROMISED.

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

Fig. 9: Simulator SIM for SaPAKE of Fig. 8: Initialization and Offline Attack

index i^* in the $(\textsc{RcvComplete}, sid, ssid, \mathsf{U}, i^*)$ message aimed at $\mathcal{F}_{\mathsf{OPRF}}$, and $\mathsf{AuthDec}_{\mathsf{rw}'}(c^*) = \perp$ (where $\mathsf{rw}' = F_{i^*}(\mathsf{pw}')$). In the following five games ($\mathbf{G}_1$ – $\mathbf{G}_5$) we gradually change this condition.

Game $\mathbf{G}_1$ ***(user aborts if $\mathcal{A}$ is passive before Π starts but passwords do not match)*:** In the case that $(c^* = c \wedge i^* = \mathsf{S})$ and $\mathsf{pw}' \neq \mathsf{pw}$, U outputs $(\textsc{abort}, sid, ssid)$.

$\mathcal{Z}$'s views in $\mathbf{G}_0$ and $\mathbf{G}_1$ are identical unless on some U sub-session event $c^* = c \wedge i^* = \mathsf{S} \wedge \mathsf{pw}' \neq \mathsf{pw}$ occurs but $\mathsf{AuthDec}_{\mathsf{rw}'}(c) \neq \perp$: In this case U outputs $(sid, ssid, SK)$ in $\mathbf{G}_0$ and $(\textsc{abort}, sid, ssid)$ in $\mathbf{G}_1$). Since $c \leftarrow \mathsf{AuthEnc}_{\mathsf{rw}}(p_u, P_u, P_s)$, we have that $\mathsf{AuthDec}_{\mathsf{rw}}(c) \neq \perp$. But rw' and rw are independent random strings in $\{0,1\}^{2\tau}$; therefore, we can construct a reduction $\mathcal{R}_{\mathrm{RBST1}}$ to the random-key robustness of AE where rw' and rw are the challenge AE keys: $\mathcal{R}_{\mathrm{RBST1}}$ runs the code of $\mathbf{G}_0$ except that it uses its input as rw' and rw. In every sub-session $\mathcal{R}_{\mathrm{RBST1}}$ checks if $\mathsf{AuthDec}_{\mathsf{rw}}(c) \neq \perp$ and $\mathsf{AuthDec}_{\mathsf{rw}'}(c) \neq \perp$, and if so, it outputs c (and breaks the game). We have that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0, \mathbf{G}_1} \leq \mathbf{Adv}_{\mathcal{R}_{\mathrm{RBST1}}}^{\mathrm{RBST}, \mathsf{AE}},$$

which is a negligible function of τ.

Game $\mathbf{G}_2$ ***(abort the entire game if c^* is valid under two different keys)*:** In the case that $\neg(c^* = c \wedge i^* = \mathsf{S})$, the game outputs HALT and aborts

OPRF Evaluation

1. On (UsrSession, sid, $ssid$, U, S) from $\mathcal{F}$, send (Eval, sid, $ssid$, U, S) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{OPRF}}$. On prfx from $\mathcal{A}$, record $\langle ssid, \mathsf{U}, \mathsf{prfx}\rangle$ if prfx is new, else reject.
2. On (SvrSession, sid, $ssid'$, U, S) from $\mathcal{F}$, retrieve $\mathsf{file}[sid] = (\cdot, \cdot, \cdot, c)$, send (SndrComplete, sid, $ssid'$, S) and c to $\mathcal{A}$ on behalf of, respectively $\mathcal{F}_{\mathsf{OPRF}}$ and S, and given $\mathcal{A}$'s response prfx' do the following in order:
   (a) If there is record $\langle ssid, \mathsf{U}, \mathsf{prfx}'\rangle$ then replace it with $\langle ssid, \mathsf{U}, \mathsf{OK}\rangle$;
   Else record $\langle ssid', \mathsf{act}\rangle$, set tx++, send (Interrupt, sid, $ssid'$, S) to $\mathcal{F}$.
   (b) Record $\langle ssid_{\Pi}, ssid', \mathsf{S}, \mathsf{U}\rangle$ and mark it fresh for $ssid_{\Pi} := ssid'|\mathsf{prfx}'$, and send (SvrSession, sid, $ssid_{\Pi}$, U, S) to $\mathsf{SIM}_{\mathsf{AKE}}$.
3. On (RcvComplete, sid, $ssid$, U, i^*) from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathsf{OPRF}}$, retrieve $\langle ssid, \mathsf{U}, \mathsf{prfx}\rangle$ (ignore the message if such record not found) and do in order:
   (a) If $i^* = \mathsf{S}$, S is not compromised, and there is no record $\langle ssid, \mathsf{U}, \mathsf{OK}\rangle$, then do: Ignore this message if $\mathsf{tx} = 0$, else set tx−−.
   (b) Augment record $\langle ssid, \mathsf{U}, \mathsf{prfx}\rangle$ to $\langle ssid, \mathsf{U}, \mathsf{prfx}, i^*\rangle$.
4. On (Eval, sid, $ssid$, S, x) followed by (RcvComplete, sid, $ssid$, $\mathcal{A}$, i^*) from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{OPRF}}$ (string prfx chosen by $\mathcal{A}$ for this Eval can be ignored), send (Eval, sid, $ssid$, $\mathcal{A}$, S) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{OPRF}}$ and do in order:
   (a) If $i^* \neq \mathsf{S}$ then send (Eval, sid, $ssid$, $F_{i^*}(x)$) to $\mathcal{A}$.
   (b) If $i^* = \mathsf{S}$ and $\mathsf{tx} > 0$, but there is no record $\langle ssid', \mathsf{act}\rangle$ then output halt.
   (c) If $i^* = \mathsf{S}$ and there are some records $\langle ssid', \mathsf{act}\rangle$ then do in order:
      i. If there is record $\langle ssid', \mathsf{act}\rangle$ which is *not* marked completed then choose $ssid'$ of any such record, but if all records $\langle ssid', \mathsf{act}\rangle$ are marked completed then choose $ssid'$ of any of those.
      ii. Ignore this message if $\mathsf{tx} = 0$, else set tx−− and send (TestPwd, sid, $ssid'$, S, x) to $\mathcal{F}$.
      iii. If $\mathcal{F}$ returns "correct guess," send (Compromise, sid, U) to $\mathsf{SIM}_{\mathsf{AKE}}$, and on response (p_u, P_u, P_s) set $\mathsf{rw} \leftarrow \mathsf{SIM}_{\mathsf{EQV}}((p_u, P_u, P_s))$ and $F_{\mathsf{S}}(x) := \mathsf{rw}$, record $\langle \text{file}, \mathsf{U}, \mathsf{S}, x\rangle$, and declare U compromised.
      iv. Send (Eval, sid, $ssid$, $F_{\mathsf{S}}(x)$) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{OPRF}}$, and modify the chosen record $\langle ssid', \mathsf{act}\rangle$ into $\langle ssid', \mathsf{used}\rangle$.

Fig. 10: Simulator SIM for SaPAKE of Fig. 8: OPRF Evaluation

AKE Simulation

1. For any $ssid$, as soon as $\langle ssid, \mathsf{U}, \mathsf{prfx}\rangle$ is augmented to $\langle ssid, \mathsf{U}, \mathsf{prfx}, i^*\rangle$ and $\mathcal{A}$ sends $(ssid, c^*)$ to U, retrieve $\mathsf{file}[sid] = (\cdot, \cdot, \cdot, c)$ and do *one* of the following:
   (a) If $(c^*, i^*) = (c, \mathsf{S})$, then send (TestAbort, sid, $ssid$, U) to $\mathcal{F}$.
   If $\mathcal{F}$ replies Succ, record $\langle ssid_\Pi, ssid, \mathsf{U}, \mathsf{S}\rangle$ marked fresh, and send (UsrSession, sid, $ssid_\Pi$, U, S) to $\mathsf{SIM}_{\mathsf{AKE}}$ for $ssid_\Pi := [ssid||\mathsf{prfx}]$.
   (b) Otherwise for every x s.t. $y = F_{i^*}(x)$ is defined, check if $\mathsf{AuthDec}_y(c^*)$ output parses as (p_u^*, P_u^*, P_s^*), and do *one* of the following:
      i. If there is no such x, send (TestPwd, sid, $ssid$, $\mathsf{U}, \perp$) followed by (TestAbort, sid, $ssid$, U) to $\mathcal{F}$.
      ii. If there are more than one such x's, output halt and abort.
      iii. If there is a unique such x, send (TestPwd, sid, $ssid$, U, x) to $\mathcal{F}$.
         - If $\mathcal{F}$ replies "wrong guess," send (TestAbort, sid, $ssid$, U) to $\mathcal{F}$.
         - If $\mathcal{F}$ replies "correct guess," do:
         (1) set $(p_u^*, P_u^*, P_s^*) := \mathsf{AuthDec}_y(c^*)$;
         (2) run Π_u on (p_u^*, P_u^*, P_s^*) and $ssid_\Pi = [ssid||\mathsf{prfx}]$;
         (3) when Π_u outputs SK^*, send (NewKey, sid, $ssid$, U, SK^*) to $\mathcal{F}$.
2. On all AKE-related interactions of $\mathcal{A}$ with all AKE sessions started by SIM's SvrSession and UsrSession queries to $\mathsf{SIM}_{\mathsf{AKE}}$ above, pass all messages between $\mathcal{A}$ and $\mathsf{SIM}_{\mathsf{AKE}}$, and react to messages sent by $\mathsf{SIM}_{\mathsf{AKE}}$'s interface with $\mathcal{F}_{\mathsf{AKE-KCI}}$ as follows:
   – On (Impersonate, sid, $ssid_\Pi, \mathsf{S}$), if S is declared compromised and there is record $\langle ssid_\Pi, ssid, \mathsf{U}, \mathsf{S}\rangle$ marked fresh, then mark it compromised and send (Impersonate, sid, $ssid$) to $\mathcal{F}$.
   – On (Impersonate, sid, $ssid_\Pi, \mathsf{U}$), if U is declared compromised and there is record $\langle ssid_\Pi, ssid, \mathsf{S}, \mathsf{U}\rangle$ marked fresh, then mark it compromised, retrieve $\langle$file, $\mathsf{U}, \mathsf{S}, \mathsf{pw}\rangle$ and send (TestPwd, sid, $ssid$, S, pw) to $\mathcal{F}$.
   – On (NewKey, sid, $ssid_\Pi, \mathsf{P}, SK^*$), if there is a record $\langle ssid_\Pi, ssid, \mathsf{P}, \mathsf{P}'\rangle$ not marked completed, do:
      • If the record is compromised, or P or P' is corrupted, set $SK := SK^*$.
      • If the record is fresh, and SIM sent (NewKey, sid, $ssid$, P', SK') to $\mathcal{F}$ while record $\langle ssid_\Pi, ssid, \mathsf{P}', \mathsf{P}\rangle$ was marked fresh, set $SK := SK'$.
      • Otherwise pick $SK \leftarrow_{\mathrm{R}} \{0,1\}^\tau$.
      Finally, mark $\langle ssid_\Pi, ssid, \mathsf{P}, \mathsf{P}'\rangle$ completed and send (NewKey, sid, $ssid$, P, SK) to $\mathcal{F}$.

Fig. 11: Simulator SIM for SaPAKE of Fig. 8: AKE Simulation

if there are $x_1 \neq x_2$ such that $\mathcal{A}$ queries both $y_1 = F_{i^*}(x_1)$ and $y_2 = F_{i^*}(x_2)$, and $\mathsf{AuthDec}_{y_1}(c^*) \neq \perp$ and $\mathsf{AuthDec}_{y_2}(c^*) \neq \perp$.

Here are throughout the proof below we say that "$\mathcal{A}$ queries $F_{i^*}(x)$," where index i^* may or may not be S, if (i) $\mathcal{A}$ sends $(\textsc{Eval}, sid, ssid, x)$ and then $(\textsc{RcvComplete}, sid, ssid, \mathcal{A}, i^*)$ to $\mathcal{F}_{\mathsf{OPRF}}$ *and* if $\mathcal{F}_{\mathsf{OPRF}}$ replies to this query with $F_{i^*}(x)$ (note that if $i^* = \mathsf{S}$ then $\mathcal{F}_{\mathsf{OPRF}}$ replies with $F_{\mathsf{S}}(x)$ if and only if $\mathsf{tx} > 0$, because $\mathcal{F}_{\mathsf{OPRF}}$'s record $\langle ssid, \mathcal{A}, x, \mathsf{prfx}\rangle$ corresponding to $\mathcal{A}$'s evaluation query can never satisfy $\mathsf{prfx} = \mathsf{OK}$), (ii) if $\mathcal{A}$ sends $(\textsc{OfflineEval}, sid, i^*, x)$ to $\mathcal{F}_{\mathsf{OPRF}}$ *and* if $\mathcal{F}_{\mathsf{OPRF}}$ replies to this query with $F_{i^*}(x)$ (note that if $i^* = \mathsf{S}$ then $\mathcal{F}_{\mathsf{OPRF}}$ replies with $F_{\mathsf{S}}(x)$ if and only if S is corrupted or COMPROMISED). This terminology is reused in subsequent games. Moreover, we refer to case (i) as "$\mathcal{A}$ queries $F_{i^*}(x)$ online," and to case (ii) as "$\mathcal{A}$ queries $F_{i^*}(x)$ offline."

Note that y_1 and y_2 are independent random strings in $\{0,1\}^{2\tau}$. Therefore, we can construct a reduction $\mathcal{R}_{\text{RBST2}}$ to the random-key robustness of AE where y_1 and y_2 are the challenge AE keys: $\mathcal{R}_{\text{RBST2}}$ picks a random pair (j_1, j_2) where $j_1, j_2 \in \{1, \ldots, q_{\text{F}}\}$ and $j_1 < j_2$[10] (a guess that y_1 and y_2 are the results of $\mathcal{A}$'s j_1-th and j_2-th queries), and runs the code of $\mathbf{G}_1$ except that it uses its input as the results of $\mathcal{A}$'s j_1-th and j_2-th queries. In every sub-session $\mathcal{R}_{\text{RBST1}}$ checks if $\mathsf{AuthDec}_{\mathsf{rw}}(c^*) \neq \perp$ and $\mathsf{AuthDec}_{\mathsf{rw}'}(c^*) \neq \perp$, and if so, it outputs c^* (and breaks the game). We have that

$$\mathbf{Dist}^{\mathbf{G}_1,\mathbf{G}_2}_{\mathcal{Z}} \leq \Pr[\text{HALT}] \leq \binom{q_{\text{F}}}{2} \cdot \mathbf{Adv}^{\text{RBST},\mathsf{AE}}_{\mathcal{R}_{\text{RBST2}}},$$

which is a negligible function of τ.

Game $\mathbf{G}_3$ ***(user aborts if $\mathcal{A}$ does not compute rw', password match, and $\mathcal{A}$ does not change the OPRF index but changes c)*:** In the case that $(c^* \neq c \wedge i^* = \mathsf{S})$ and $\mathsf{pw}' = \mathsf{pw}$, U outputs $(\textsc{abort}, sid, ssid)$ if $\mathcal{A}$ does not query $F_{i^*}(\mathsf{pw})$.

$\mathcal{Z}$'s views in $\mathbf{G}_2$ and $\mathbf{G}_3$ are identical unless $\mathcal{A}$ does not query $\mathsf{rw} = F_{\mathsf{S}}(\mathsf{pw})$ but on some U sub-session we have $c^* \neq c \wedge i^* = \mathsf{S} \wedge \mathsf{pw}' = \mathsf{pw}$ and $\mathsf{AuthDec}_{\mathsf{rw}}(c^*) \neq \perp$: In this case U outputs $(sid, ssid, SK)$ in $\mathbf{G}_2$ and $(\textsc{abort}, sid, ssid)$ in $\mathbf{G}_3$. Since $\mathcal{A}$ does not query $F_{\mathsf{S}}(\mathsf{pw})$, rw is a random string in $\{0,1\}^{2\tau}$ in $\mathcal{Z}$'s view. $\mathcal{Z}$ additionally learns $c \leftarrow \mathsf{AuthEnc}_{\mathsf{rw}}(p_u, P_u, P_s)$, but $\mathcal{A}$'s message is restricted to $c^* \neq c$. Therefore, we can construct a reduction $\mathcal{R}_{\text{AUTH1}}$ to the authenticity of AE where rw is the challenge AE key: $\mathcal{R}_{\text{AUTH1}}$ runs the code of $\mathbf{G}_2$, except that it uses its encryption oracle to compute $c \leftarrow \mathsf{AuthEnc}_{\mathsf{rw}}(p_u, P_u, P_s)$, and its decryption oracle to compute $\mathsf{AuthDec}_{\mathsf{rw}}(c^*)$ in every U sub-session (1) which runs on input $\mathsf{pw}' = \mathsf{pw}$ and (2) where the OPRF function index is $i^* = \mathsf{S}$. In each such sub-session $\mathcal{R}_{\text{AUTH1}}$ checks if $c^* \neq c$ and $\mathsf{AuthDec}_{\mathsf{rw}'}(c^*) \neq \perp$, and if so, it outputs c^* (and breaks the game). We have that

$$\mathbf{Dist}^{\mathbf{G}_2,\mathbf{G}_3}_{\mathcal{Z}} \leq \mathbf{Adv}^{\text{AUTH},\mathsf{AE}}_{\mathcal{R}_{\text{AUTH1}}},$$

[10] To be precise, $\mathcal{R}_{\text{RBST2}}$ picks $j'_1 \leftarrow_{\text{R}} \{1, \ldots, q_{\text{F}}\}$, $j'_2 \leftarrow_{\text{R}} \{1, \ldots, q_{\text{F}}\} \setminus \{j'_1\}$, and sets $j_1 := \min(j'_1, j'_2)$ and $j_2 := \max(j'_1, j'_2)$.

which is a negligible function of τ.

Game $\mathbf{G}_4$ ***(user aborts if $\mathcal{A}$ does not compute rw', and either passwords do not match or $\mathcal{A}$ changes the OPRF index)*:** In the case that $(c^* \neq c \wedge \mathsf{pw}' \neq \mathsf{pw}) \vee i^* \neq \mathsf{S}$, U outputs $(\text{ABORT}, sid, ssid)$ if $\mathcal{A}$ does not query $F_{i^*}(\mathsf{pw}')$.

$\mathcal{Z}$'s views in $\mathbf{G}_3$ and $\mathbf{G}_4$ are identical unless on some U sub-session $(c^* \neq c \wedge \mathsf{pw}' \neq \mathsf{pw}) \vee i^* \neq \mathsf{S}$, and $\mathcal{A}$ does not query $\mathsf{rw}' = F_{i^*}(\mathsf{pw}')$ but $\mathsf{AuthDec}_{\mathsf{rw}'}(c^*) \neq \perp$: In this case U outputs $(sid, ssid, SK)$ in $\mathbf{G}_3$ and $(\text{ABORT}, sid, ssid)$ in $\mathbf{G}_4$ on that sub-session. Call the event that such U sub-session exists E. For each $i \in \{1, \ldots, q_{\text{U}}\}$ define E_j as the event that on the j-th U sub-session, in the order determined by the initialization calls from $\mathcal{Z}$, it holds that (1) $(c^* \neq c \wedge \mathsf{pw}' \neq \mathsf{pw}) \vee i^* \neq \mathsf{S}$, (2) $\mathcal{A}$ does not query $F_{i^*}(\mathsf{pw}')$, (3) this is the first occurrence of pair (i^*, pw') on any U sub-session, and (4) $\mathsf{AuthDec}_{\mathsf{rw}'}(c^*) \neq \perp$. Note that E is the union of events E_j for $j = 1, \ldots, q_{\text{U}}$. Since for (i^*, pw') in the j-th U sub-session $\mathcal{A}$ does not query $F_{i^*}(\mathsf{pw}')$, rw' is not used anywhere else (in particular, $\mathcal{Z}$ learns $c \leftarrow \mathsf{AuthEnc}_{\mathsf{rw}}(p_u, P_u, P_s)$, but rw is independent of rw') and hence is a random string in $\{0,1\}^{2\tau}$ in $\mathcal{Z}$'s view. Therefore, for each E_j we can construct a reduction $\mathcal{R}_{\text{AUTH2},j}$ to the authenticity of AE where rw' in the j-th U sub-session is the challenge AE key: $\mathcal{R}_{\text{AUTH2},j}$ runs the code of $\mathbf{G}_3$. In the j-th U sub-session, $\mathcal{R}_{\text{AUTH2},j}$ uses the decryption oracle to check if E_j occurs, and if so, it outputs c^* (and breaks the game). We have that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_3,\mathbf{G}_4} \leq \Pr[E] \leq \sum_{j=1}^{q_{\text{U}}} \Pr[E_j] \leq q_{\text{U}} \cdot \mathbf{Adv}_{\mathcal{R}_{\text{AUTH2}}}^{\text{AUTH},\mathsf{AE}},$$

which is a negligible function of τ.

Note that the combined conditions introduced in $\mathbf{G}_3$ and $\mathbf{G}_4$ are equivalent to the following: In the case that $\neg(c^* = c \wedge i^* = \mathsf{S})$, U outputs $(\text{ABORT}, sid, ssid)$ if $\mathcal{A}$ does not query $F_{i^*}(\mathsf{pw}')$.

Game $\mathbf{G}_5$ ***(extract $\mathcal{A}$'s password guess on U interactions)*:** In $\mathbf{G}_4$, after U computes $\mathsf{rw}' = F_{i^*}(\mathsf{pw}')$ and receives $(ssid, c^*)$ and $(\mathsf{Prefix}, ssid, \mathsf{prfx})$, it tests if $\mathsf{AuthDec}_{\mathsf{rw}}(c^*)$ can be parsed as (p_u^*, P_u^*, P_s^*), and either runs Π_u on these decrypted AKE keys and $ssid_\Pi := [ssid||\mathsf{prfx}]$, or outputs $(\text{ABORT}, sid, ssid)$ if the parsing fails. Here we replace the above with the following ($ssid_\Pi$ does not change from $\mathbf{G}_4$ to $\mathbf{G}_5$, so we omit it in the description of $\mathbf{G}_5$ below):

1. If $c^* = c \wedge i^* = \mathsf{S}$, then do: (I) if $\mathsf{pw}' = \mathsf{pw}$ (which is case $(*)$), then U runs Π_u on inputs (p_u, P_u, P_s); (II) otherwise U outputs $(\text{ABORT}, sid, ssid)$.
2. If $\neg(c^* = c \wedge i^* = \mathsf{S})$, and there are $x_1 \neq x_2$ such that $\mathcal{A}$ queries both $y_1 = F_{i^*}(x_1)$ and $y_2 = F_{i^*}(x_2)$, and $\mathsf{AuthDec}_{y_1}(c^*) \neq \perp$ and $\mathsf{AuthDec}_{y_2}(c^*) \neq \perp$, output HALT and abort the entire game.
3. If $\neg(c^* = c \wedge i^* = \mathsf{S})$ and $\mathcal{A}$ queries $\mathsf{rw}' = F_{i^*}(x)$ for a *unique* x such that $\mathsf{AuthDec}_{\mathsf{rw}'}(c^*)$ can be parsed as (p_u^*, P_u^*, P_s^*), then do: (I) if $x = \mathsf{pw}'$ (which is case $(**)$), then U runs Π_u on inputs (p_u^*, P_u^*, P_s^*); (II) otherwise U outputs $(\text{ABORT}, sid, ssid)$.

4. Otherwise, i.e., $\neg(c^* = c \wedge i^* = \mathsf{S})$ but $\mathcal{A}$ makes no $F_{i^*}(x)$ query as in case 2 or 3 above, U outputs (ABORT, sid, $ssid$).

We argue that this modification does not change $\mathcal{Z}$'s view. First consider the case that $c^* = c \wedge i^* = \mathsf{S}$. In $\mathbf{G}_4$, if $\mathsf{pw}' = \mathsf{pw}$, then U runs Π_u on $\mathsf{AuthDec}_{\mathsf{rw}'}(c^*) = \mathsf{AuthDec}_{\mathsf{rw}}(c) = (p_u, P_u, P_s)$, which is replicated in case 1(I) of $\mathbf{G}_5$; otherwise U outputs (ABORT, sid, $ssid$) by the condition introduced in $\mathbf{G}_1$, which is replicated in case 1(II) of $\mathbf{G}_5$. Now consider the case that $\neg(c^* = c \wedge i^* = \mathsf{S})$. Then if case 2 occurs, i.e., $\mathcal{A}$ queries two distinct F_{i^*} outputs which both decrypt c^*, then $\mathbf{G}_4$ outputs HALT by the condition introduced in $\mathbf{G}_2$, which is the same as in $\mathbf{G}_5$. If $\mathcal{A}$ makes no $F_{i^*}(\mathsf{pw}')$ query, then U outputs (ABORT, sid, $ssid$) by the conditions introduced in $\mathbf{G}_3$ and $\mathbf{G}_4$, which is replicated in cases 3(II) and 4 of $\mathbf{G}_5$. The only remaining case is that $\mathcal{A}$ queries $F_{i^*}(\mathsf{pw}')$ and this is the unique query such that $\mathsf{AuthDec}'_{\mathsf{rw}}(c^*)$ can be parsed as (p_u^*, P_u^*, P_s^*), in which case in $\mathbf{G}_4$ U runs Π_u on (p_u^*, P_u^*, P_s^*), which is replicated in case 3(I) in $\mathbf{G}_5$. It follows that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_4,\mathbf{G}_5} = 0.$$

Comparison of $\mathbf{G}_5$ and the Simulated World. We argue that in $\mathbf{G}_5$, when $\mathcal{A}$ sends $(ssid, c^*)$ aimed at U and decides on the index i^* for which U computes F_{i^*}, the way that the game emulates U's response to (c^*, i^*) is the same as in the simulated world, except that $\mathbf{G}_5$ runs the AKE protocol Π_u on behalf of U in case this user instance encounters either case $(*)$ or $(**)$, while the simulator SIM executes Π_u only in case $(**)$, while in case $(*)$ the execution of Π_u is replaced by a simulation by $\mathsf{SIM}_{\mathsf{AKE}}$.

Disregarding the differences due to Π_u execution vs. Π_u simulation, the simulation of U instances acts based on the following two cases:

(i) If $c^* = c \wedge i^* = \mathsf{S}$ then SIM sends (TESTABORT, sid, $ssid$, U) to $\mathcal{F}$.
- If $\mathcal{F}$ returns SUCC, i.e., $\mathsf{pw}' = \mathsf{pw}$[11] then SIM proceeds to simulate Π_u. We call this case $(*)$, and it corresponds to case 1(I) in $\mathbf{G}_5$, while SIM handles it in step (1a) in Fig. 11.
- If $\mathcal{F}$ returns FAIL, i.e., $\mathsf{pw}' \neq \mathsf{pw}$, $\mathcal{F}$ sends (ABORT, sid, $ssid$) to U (who outputs this message). This corresponds to case 1(II) in $\mathbf{G}_5$, while SIM handles it in the same step as above.

(ii) If $\neg(c^* = c \wedge i^* = \mathsf{S})$, then for every x such that $y = F_{i^*}(x)$ was queried by $\mathcal{A}$, SIM checks if $\mathsf{AuthDec}_y(c^*)$ can be parsed as (p_u^*, P_u^*, P_s^*).
- If there are two or more such x's, i.e., $x_1 \neq x_2$ s.t. $\mathcal{A}$ queries both $y_1 = F_{i^*}(x_1)$ and $y_2 = F_{i^*}(x_2)$, and $\mathsf{AuthDec}_{y_1}(c^*) \neq \perp$ and $\mathsf{AuthDec}_{y_2}(c^*) \neq \perp$, then SIM outputs HALT and aborts. This corresponds to case 2 in $\mathbf{G}_5$, and SIM handles it in in step (1(b)ii) in Fig. 11.

[11] Note that our UC saPAKE functionality $\mathcal{F}$ does not check if S runs a session with a sub-session ID $ssid$ matching the tested session of U. Indeed, our protocol allows the adversary to test if $\mathsf{pw}' = \mathsf{pw}$ without regards to the $ssid$ on the U's tested session.

- If there is a unique such x, then SIM sends $(\textsc{TestPwd}, sid, ssid, \mathsf{U}, x)$ to $\mathcal{F}$. If $\mathcal{F}$ returns "correct guess," i.e., $x = \mathsf{pw}'$, SIM runs Π on the decrypted values $(p_u^*, P_u^*, P_s^*) \leftarrow \mathsf{AuthDec}_y(c^*)$. We call this case $(**)$, it corresponds to case 3(I) in $\mathbf{G}_5$, and SIM handles it in step (1(b)iii).
- If $\mathcal{F}$ returns "wrong guess," i.e., $x \neq \mathsf{pw}'$, then SIM sends $(\textsc{TestAbort}, sid, ssid, \mathsf{U})$ to $\mathcal{F}$, and $\mathcal{F}$ sends $(\textsc{abort}, sid, ssid)$ to U (who outputs this message). This corresponds to case 3(II) in $\mathbf{G}_5$, and SIM handles it in the same step as above.
- If there is no such x, then SIM sends $(\textsc{TestPwd}, sid, ssid, \mathsf{U}, \perp)$ and then $(\textsc{TestAbort}, sid, ssid, \mathsf{U})$ to $\mathcal{F}$, and $\mathcal{F}$ sends $(\textsc{abort}, sid, ssid)$ to U (who outputs this message). This corresponds to case 4 in $\mathbf{G}_5$, and SIM handles it in step (1(b)i).

We can see that if we omit the interaction between SIM and $\mathcal{F}$ above, and view SIM and $\mathcal{F}$ combined as the game challenger who interacts with $\mathcal{Z}$ and $\mathcal{A}$, then the behavior of this game challenger when $\mathcal{A}$ sends $(ssid, c^*)$ aimed at U is exactly the same with the behavior of $\mathbf{G}_5$, except for Π_u execution replaced by Π_u simulation in case $(*)$.

In the next four games ($\mathbf{G}_5$ – $\mathbf{G}_9$) we replace AKE credential generation and login protocol execution with the simulation by $\mathsf{SIM}_{\mathsf{AKE}}$.

Game $\mathbf{G}_6$ ***(outsource the generation of c and rw to $\mathsf{SIM}_{\mathsf{EQV}}$)*:** At the beginning of the game, let $\mathsf{SIM}_{\mathsf{EQV}}$ simulate c and leave rw undefined, and let $\mathsf{SIM}_{\mathsf{EQV}}$ "open" rw when $\mathcal{A}$ computes it. Concretely,

(1) At the beginning of the game, set $c \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(k_{pr} + 2k_{pb})$;

(2) When $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$, set $\mathsf{rw} \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(p_u, P_u, P_s)$.

Observe that in $\mathbf{G}_5$, $\mathcal{Z}$ sees $c \leftarrow \mathsf{AuthEnc}_{\mathsf{rw}}(p_u, P_u, P_s)$, and unless and until $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$ (and thus learns rw), rw is not used by any party except in generating c, hence is a random string in $\{0,1\}^{2\tau}$ independent of everything else (except for c) in $\mathcal{Z}$'s view. In particular, in $\mathbf{G}_5$ U does not evaluate F on any input, and all processing is based on whether $c^* = c \wedge i^* = \mathsf{S}$, whether $\mathsf{pw}' = \mathsf{pw}$, and on $\mathcal{A}$'s queries to F_{i^*}, which in the case $i^* = \mathsf{S}$ are $\mathcal{A}$'s queries to F_{S}. Therefore, in $\mathbf{G}_5$ c followed by rw in case $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$, are formed as in the "real game" in the encryption equivocability experiment for AE, where $\mathcal{A}$ sees the encryption c of (p_u, P_u, P_s) under key rw followed by the key rw (in case of $\mathcal{A}$'s query to $F_{\mathsf{S}}(\mathsf{pw})$). On the other hand, in $\mathbf{G}_6$ ciphertext c followed by key rw are formed as in the "ideal game" in the encryption equivocability experiment for AE. Therefore, we can construct a reduction $\mathcal{R}_{\mathrm{EQV}}$ to the equivocability of AE: $\mathcal{R}_{\mathrm{EQV}}$ runs the code of $\mathbf{G}_5$ except that it uses its input as c and rw, and copies $\mathcal{Z}$'s output. We have that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_5,\mathbf{G}_6} \leq \mathbf{Adv}_{\mathcal{R}_{\mathrm{EQV}}}^{\mathrm{EQV},\mathsf{AE}},$$

which is a negligible function of τ.

Game $\mathbf{G}_7$ ***(outsource the generation of p_u, P_u, p_s, P_s to $\mathsf{SIM}_{\mathsf{AKE}}$)*:** $\mathbf{G}_7$ lets $\mathsf{SIM}_{\mathsf{AKE}}$ generate the two parties' key pairs in the AKE protocol Π, p_u, P_u,

p_s, P_s, instead of generating them on its own. Concretely, *at the beginning of the game*, send (COMPROMISE, sid, S) and (COMPROMISE, sid, U) to $\mathsf{SIM}_{\mathsf{AKE}}$ and obtain p_u, P_u, p_s, P_s; *ignore all subsequent messages from* $\mathsf{SIM}_{\mathsf{AKE}}$.

Clearly, an environment distinguishing between $\mathbf{G}_6$ and $\mathbf{G}_7$ can be turned into an environment $\mathcal{Z}_{\text{AKE1}}$ distinguishing between the real execution of Π and the simulation of Π by $\mathsf{SIM}_{\mathsf{AKE}}$. Therefore,

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_6,\mathbf{G}_7} \leq \mathbf{Dist}_{\mathcal{Z}_{\text{AKE1}}}^{\Pi,\{\mathcal{F}_{\mathsf{AKE-KCI}},\mathsf{SIM}_{\mathsf{AKE}}\}},$$

where $\mathbf{Dist}_{\mathcal{Z}_{\text{AKE1}}}^{\Pi,\{\mathcal{F}_{\mathsf{AKE-KCI}},\mathsf{SIM}_{\mathsf{AKE}}\}}$ denotes the distinguishing advantage of $\mathcal{Z}_{\text{AKE1}}$ between the real execution of Π and the simulation of Π by $\mathsf{SIM}_{\mathsf{AKE}}$, and is a negligible function of τ.

Game $\mathbf{G}_8$ ***(leave p_u, P_u, p_s, P_s undefined until they are used):*** At the beginning of the game, do not send (COMPROMISE, sid, S) or (COMPROMISE, sid, U) to $\mathsf{SIM}_{\mathsf{AKE}}$, and leave p_u, P_u, p_s, P_s undefined. However,

(1) When $\mathcal{A}$ sends (STEALPWDFILE, sid) to S, send (COMPROMISE, sid, S) to $\mathsf{SIM}_{\mathsf{AKE}}$ to obtain (p_s, P_s, P_u);

(2) When $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$, send (COMPROMISE, sid, U) to $\mathsf{SIM}_{\mathsf{AKE}}$ to obtain (p_u, P_u, P_s).

Observe that in $\mathbf{G}_7$, p_s is not used unless and until $\mathcal{A}$ sends (STEALPWDFILE, sid) to S (at which time the game challenger must send p_s at part of its response $\mathsf{file}[sid]$); therefore, postponing generating p_s to the time when $\mathcal{A}$ sends (STEALPWDFILE, sid) to S does not change the game. Similarly, p_u is not used unless and until $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$ (at which time the game challenger must invoke $\mathsf{SIM}_{\mathsf{EQV}}(p_u, P_u, P_s)$ to generate rw as the response to $\mathcal{A}$); therefore, postponing generating p_u to the time when $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$ does not change the game. We have that

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_7,\mathbf{G}_8} = 0.$$

Comparison of $\mathbf{G}_8$ and the Simulated World. We argue that in $\mathbf{G}_8$, p_u, P_u, p_s, P_s, rw and c are generated in the same way as in the simulated world. In the simulated world SIM sets $c \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(k_{pr} + 2k_{pb})$ and p_u, P_u, p_s, P_s and rw are undefined until one of the two cases happen:

Case 1: When the adversary compromises the server, i.e. when $\mathcal{A}$ sends (STEALPWDFILE, sid) to S (step 1 of "Stealing Password Data and Offline Queries"), SIM sends (COMPROMISE, sid, S) to $\mathsf{SIM}_{\mathsf{AKE}}$ to obtain (p_s, P_s, P_u).

Case 2: When the adversary makes either a successful password test attack. This can happen in one of the following two ways. First, if $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$ *offline* (step 2 of "Stealing Password Data and Offline Queries"; note that $\mathcal{A}$ can query $F_{\mathsf{S}}(\mathsf{pw})$ offline only after compromising S), SIM sends (OFFLINETESTPWD, sid, pw) to $\mathcal{F}$, which replies "correct guess" (because pw is correct). Second, if $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$ *online* (step 4 of "OPRF Evaluation"), SIM checks if the $\mathcal{F}_{\mathsf{OPRF}}$ ticket counter tx is non-zero (recall that SIM emulates $\mathcal{F}_{\mathsf{OPRF}}$), and if so then SIM sends (TESTPWD, $sid, ssid', \mathsf{pw}$) to $\mathcal{F}$ where $ssid'$ is

a sub-session ID of some S session for which SIM holds record $\langle ssid', \mathsf{act} \rangle$. Since password pw is correct $\mathcal{F}$ will reply "correct guess" if $\mathcal{F}$ holds a server session record $\langle ssid', \mathsf{S}, \mathsf{U}, \mathsf{pw} \rangle$ s.t. $\mathsf{dPT}(ssid) = 1$. In either case, given $\mathcal{F}$'s response "correct guess", SIM declares U COMPROMISED, sends (COMPROMISE, sid, U) to $\mathsf{SIM}_{\mathsf{AKE}}$, and on $\mathsf{SIM}_{\mathsf{AKE}}$'s response (p_u, P_u, P_s), SIM gets $\mathsf{rw} \leftarrow \mathsf{SIM}_{\mathsf{EQV}}(p_u, P_u, P_s)$ and sets $F_{\mathsf{S}}(\mathsf{pw}) := \mathsf{rw}$.

Observe that this interaction creates the same view to $\mathcal{Z}$ and $\mathcal{A}$ as $\mathbf{G}_8$ does, at least with regards to $\mathcal{A}$'s view in case $\mathcal{A}$ evaluates $F_{\mathsf{S}}(\mathsf{pw})$, *assuming* that in the online query case SIM never encounters the case that $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$ online but either (1) $\mathsf{tx} = 0$ or (2) $\mathsf{tx} > 0$ but SIM holds no record $\langle ssid', \mathsf{act} \rangle$ or (3) SIM holds a record $\langle ssid', \mathsf{act} \rangle$ but $\mathcal{F}$ does not hold a record $\langle ssid', \mathsf{S}, \mathsf{U}, \mathsf{pw} \rangle$ s.t. $\mathsf{dPT}(ssid') = 1$. Below we argue that this event cannot happen, and consequently the simulator SIM interacting with functionality $\mathcal{F}$ creates exactly the view that $\mathbf{G}_8$ does in the case $\mathcal{A}$ evaluates $F_{\mathsf{S}}(\mathsf{pw})$.

Note that SIM emulates $\mathcal{F}_{\mathsf{OPRF}}$, and in particular it increments tx at each SNDRCOMPLETE with prfx' that does not match any U's evaluation record $\langle ssid, \mathsf{U}, x, \mathsf{prfx} \rangle$, and it decrements it whenever $\mathsf{tx} > 0$ and $\mathcal{A}$ sends (RCVCOMPLETE, $sid, ssid, \mathsf{U}, \mathsf{S}$) where $\langle ssid, \mathsf{U}, x, \mathsf{prfx} \rangle$ is one of such unmatched U records, or $\mathcal{A}$ sends (RCVCOMPLETE, $sid, ssid, \mathcal{A}, \mathsf{S}$) corresponding to some $\mathcal{A}$'s record $\langle ssid, \mathcal{A}, x, \mathsf{prfx} \rangle$. This is the same as $\mathcal{F}_{\mathsf{OPRF}}$ does, so tx in SIM's emulation of $\mathcal{F}_{\mathsf{OPRF}}$ has always the same value as in $\mathcal{F}_{\mathsf{OPRF}}$, and if $\mathcal{F}_{\mathsf{OPRF}}$ replies to $\mathcal{A}$'s online query $F_{\mathsf{S}}(x)$ then event (1) cannot happen in the simulation. Next, note that when SIM increments tx at some (SNDRCOMPLETE, $sid, ssid', \mathsf{S}$) query, see step (2a) in Fig. 10, it marks this S session as actively attacked by recording $\langle ssid', \mathsf{act} \rangle$, and the only way SIM can change this record to $\langle ssid', \mathsf{used} \rangle$, see step (4(c)iv), is when it sends $F_{\mathsf{S}}(x)$ and decrements tx. Therefore if $\mathsf{tx} > 0$ then there must be some S sessions whose status is act, thus event (2) cannot happen in the simulation. Finally, note that when SIM records $\langle ssid', \mathsf{act} \rangle$ it sends (INTERRUPT, $sid, ssid', \mathsf{S}$) to $\mathcal{F}$, see step (2a), at which point $\mathcal{F}$ sets $\mathsf{dPT}(ssid') := 1$, and the only way $\mathcal{F}$ sets $\mathsf{dPT}(ssid') := 0$ is if SIM sends (TESTPWD, $sid, ssid', \mathsf{S}, \cdot$) to $\mathcal{F}$. Since SIM sends such TESTPWD query, in step (4(c)ii), only if it holds record $\langle ssid', \mathsf{act} \rangle$, event (3) also cannot happen.

Now the only difference between $\mathbf{G}_8$ and the simulated world lies in the simulation of Π.

Game $\mathbf{G}_9$ *(outsource to $\mathsf{SIM}_{\mathsf{AKE}}$ the simulation of all Π_u instances of case $(*)$ and of all Π_s instances)*: Replace each execution of Π_s with their simulation by $\mathsf{SIM}_{\mathsf{AKE}}$, and replace the executions of Π_u with their simulation by $\mathsf{SIM}_{\mathsf{AKE}}$ for each all U sub-sessions which fall into case $(*)$, i.e., sub-sessions where $c^* = c \wedge i^* = \mathsf{S} \wedge \mathsf{pw}' = \mathsf{pw}$. Specifically, modify the game in case (*) as follows:

1. When U's OPRF sub-session is completed and $\mathcal{A}$ sends $c^* (= c)$ to U, send (USRSESSION, $sid, ssid_{\Pi}, \mathsf{U}, \mathsf{S}$) to $\mathsf{SIM}_{\mathsf{AKE}}$ for $ssid_{\Pi} = [ssid || \mathsf{prfx}]$ where prfx

was determined by $\mathcal{A}$ in the EVAL handling of this U sub-session, and record $\langle ssid_{\Pi}, ssid, \mathsf{U}, \mathsf{S}\rangle$ marked FRESH;

2. When $\mathcal{Z}$ inputs (SVRSESSION, sid, $ssid$) to S, send (SVRSESSION, sid, $ssid_{\Pi}, \mathsf{U}, \mathsf{S}$) to $\mathsf{SIM}_{\mathsf{AKE}}$ for $ssid_{\Pi} = [ssid||\mathsf{prfx}]$ where prfx was determined by $\mathcal{A}$ in the SNDRCOMPLETE handling of this S sub-session, and record $\langle ssid_{\Pi}, ssid, \mathsf{S}, \mathsf{U}\rangle$ marked FRESH;
3. On (IMPERSONATE, sid, $ssid_{\Pi}, \mathsf{P}$) from $\mathsf{SIM}_{\mathsf{AKE}}$, if there is a record $\langle ssid_{\Pi}, ssid, \mathsf{P}', \mathsf{P}\rangle$ marked FRESH and $\mathbf{G}_9$ sent (COMPROMISE, sid, P) to $\mathsf{SIM}_{\mathsf{AKE}}$ before, mark this record COMPROMISED;
4. While $\mathsf{SIM}_{\mathsf{AKE}}$ simulates Π instances, pass messages between $\mathsf{SIM}_{\mathsf{AKE}}$ and $\mathcal{A}$;
5. On (NEWKEY, sid, $ssid_{\Pi}, \mathsf{P}, SK^*$) from $\mathsf{SIM}_{\mathsf{AKE}}$, if there is a record $\langle ssid_{\Pi}, ssid, \mathsf{P}, \mathsf{P}'\rangle$ not marked COMPLETED, do:
   - If the record is COMPROMISED, or P or P' is corrupted, set $SK := SK^*$.
   - Else if the record is marked FRESH, a $(sid, ssid, SK')$ tuple was sent to P' while record $\langle ssid_{\Pi}, ssid, \mathsf{P}', \mathsf{P}\rangle$ was FRESH, set $SK := SK'$.
   - Else pick $SK \leftarrow_{\mathrm{R}} \{0,1\}^{\tau}$.

   Finally, mark $\langle ssid_{\Pi}, ssid, \mathsf{P}, \mathsf{P}'\rangle$ COMPLETED and send $(sid, ssid, SK)$ to P.

Clearly, an environment distinguishing between $\mathbf{G}_8$ and $\mathbf{G}_9$ can be turned into an environment $\mathcal{Z}_{\mathrm{AKE2}}$ distinguishing between the real execution of Π and the simulation of Π by $\mathsf{SIM}_{\mathsf{AKE}}$. Therefore,

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_8,\mathbf{G}_9} \leq \mathbf{Dist}_{\mathcal{Z}_{\mathrm{AKE2}}}^{\Pi,\{\mathcal{F}_{\mathsf{AKE-KCI}},\mathsf{SIM}_{\mathsf{AKE}}\}},$$

which is a negligible function of τ.

Comparison of $\mathbf{G}_9$ and the Simulated World. We argue that $\mathbf{G}_9$ is identical to the simulated world, i.e., to the ideal world interaction where the game challenger is split into the simulator SIM and the SaPAKE functionality $\mathcal{F}$. Note that $\mathbf{G}_9$ decides in the same way as $\mathbf{G}_5$ whether a U sub-session results in U outputting (ABORT, sid, $ssid$) or falls into cases $(*)$ and $(**)$, and it generates the AKE keys p_u, P_u, p_s, P_s and the AE ciphertext c in the same way as in $\mathbf{G}_8$, and we argued above that $\mathbf{G}_5$ and $\mathbf{G}_8$ execute these parts in the same way as SIM interacting with $\mathcal{F}$. The remaining part is to argue that $\mathbf{G}_9$ also emulates SIM interacting with $\mathcal{F}$ with respect to the session keys SK output by U and S.[12] The case that either U or S is corrupted are easiest to see because in that case $\mathcal{F}$ passes the key received by SIM to the corresponding party, thus below we assume that neither U nor S is corrupted.

Consider first U's output SK in case $(**)$. In $\mathbf{G}_9$, SK is determined by the output of protocol Π_u executed on behalf of U on inputs (p_u^*, P_u^*, P_s^*), which are in turn determined by (c^*, i^*) and $\mathcal{A}$'s queries to F_{i^*}, as described in $\mathbf{G}_5$. In the simulated world, as we argued in $\mathbf{G}_5$, SIM runs Π_u on the same inputs, hence SK computed by SIM is identically distributed. At the end of Π_u, SIM sends

[12] Recall that S's output is always of the form $(sid, ssid, SK)$, while U's output is $(sid, ssid, SK)$ in cases $(*)$ and $(**)$, and (ABORT, sid, $ssid$) otherwise; but we argued that these cases are handled in the simulated world in the same way as in game $\mathbf{G}_5$.

(NewKey, sid, $ssid$, U, SK) to $\mathcal{F}$, who will pass SK in message (sid, $ssid$, SK) to U because case ($**$) happens only if SIM sends (TestPwd, sid, $ssid$, U, x) to $\mathcal{F}$ (see step (1(b)iii) in Fig. 11) and $\mathcal{F}$ replies "correct guess," at which point $\mathcal{F}$ marked this SaPAKE-layer sub-session record $\langle ssid, \mathsf{S}, \mathsf{U}, \mathsf{pw}\rangle$ COMPROMISED.

Secondly, consider all Π executions which are outsourced to $\mathsf{SIM}_{\mathsf{AKE}}$ in $\mathbf{G}_9$, i.e., instances of Π_u which fall into case ($*$) and all instances of Π_s. In $\mathbf{G}_9$, SK output by party $\mathsf{P} \in \{\mathsf{U}, \mathsf{S}\}$ is determined by (1) the status of record $\langle ssid_\Pi, ssid, \mathsf{P}, \mathsf{P}'\rangle$ kept for this sub-session, (2) $\mathsf{SIM}_{\mathsf{AKE}}$'s message (NewKey, sid, $ssid_\Pi$, P, SK^*), and (3) whether (sid, $ssid$, SK') was sent to P' at the time there was a FRESH record $\langle ssid_\Pi, ssid, \mathsf{P}', \mathsf{P}\rangle$. Game $\mathbf{G}_9$ uses the same factors to decide on P's output by emulating functionality $\mathcal{F}_{\mathsf{AKE-KCI}}$. In the simulated world, SK determined by SIM is identically distributed, because SIM also emulates $\mathcal{F}_{\mathsf{AKE-KCI}}$ and uses the same rules to determine the status of each AKE-layer session, hence factors (1)-(3) play exactly the same role in the simulated world. However, similarly to case ($**$) discussed above, SIM does not output message (sid, $ssid$, SK) directly to P, but sends (NewKey, sid, $ssid$, P, SK) to $\mathcal{F}$, who then "post-processes" these keys, using its own records for these sub-sessions, resp. $\langle sid, \mathsf{P}, \mathsf{P}', \mathsf{pw}^\circ\rangle$ and $\langle ssid, \mathsf{P}', \mathsf{P}, \mathsf{pw}^{\circ\circ}\rangle$.

We argue that this post-processing by $\mathcal{F}$ always implements the same logic for determining SK on a given sub-session as SIM does. Specifically, we argue that the following three invariants hold:

1. If SIM passes $\mathsf{SIM}_{\mathsf{AKE}}$'s key SK^* to $\mathcal{F}$, i.e., if the AKE-layer sub-session record $\langle ssid_\Pi, ssid, \mathsf{P}, \mathsf{P}'\rangle$ is COMPROMISED, then the SaPAKE-layer sub-session record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}^\circ\rangle$ is either COMPROMISED, or, if $\mathsf{P} = \mathsf{S}$, it is INTERRUPTED but flag = COMPROMISED.
2. If there are two AKE-layer sub-session records with matching AKE-layer sub-session ID's, i.e., $\langle ssid_\Pi, ssid, \mathsf{P}, \mathsf{P}'\rangle$ and $\langle ssid'_\Pi, ssid', \mathsf{P}', \mathsf{P}\rangle$ such that $ssid_\Pi = ssid'_\Pi$, then it holds that (a) their SaPAKE-layer sub-session ID's match as well, i.e., $ssid = ssid'$, and (b) the passwords in the corresponding SaPAKE-layer sessions also match, i.e., $\mathcal{F}$ records for these sub-sessions, $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}^\circ\rangle$ and $\langle ssid', \mathsf{P}', \mathsf{P}, \mathsf{pw}^{\circ\circ}\rangle$, satisfy $\mathsf{pw}^\circ = \mathsf{pw}^{\circ\circ}$.
3. If two AKE-layer sub-sessions $\langle ssid_\Pi, ssid, \mathsf{P}, \mathsf{P}'\rangle$ and $\langle ssid'_\Pi, ssid', \mathsf{P}', \mathsf{P}\rangle$ are "connected" by the $\mathcal{F}_{\mathsf{AKE-KCI}}$ emulated by $\mathbf{G}_9$ (and by SIM), in the sense that step 5 in $\mathcal{F}_{\mathsf{AKE-KCI}}$ emulation in $\mathbf{G}_9$ (and step 2 in Fig. 11) output the same key SK to both sessions, then the corresponding SaPAKE-layer sub-sessions $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}^\circ\rangle$ and $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}^\circ\rangle$ are FRESH.

Invariant (1) implies that if AKE-layer sub-session record $\langle ssid_\Pi, ssid, \mathsf{P}, \mathsf{P}'\rangle$ is COMPROMISED then $\mathcal{F}$ will pass SK^* output by $\mathsf{SIM}_{\mathsf{AKE}}$ to P, hence key SK output by P in the simulated world is the same as in $\mathbf{G}_9$. Invariants (2) and (3) together imply that if AKE-layer sub-session records $\langle ssid_\Pi, ssid, \mathsf{P}, \mathsf{P}'\rangle$ and $\langle ssid_\Pi, ssid', \mathsf{P}', \mathsf{P}\rangle$ (assume w.l.o.g. that the latter sub-session completes first) output the same key SK, chosen at random either by $\mathbf{G}_9$ or by SIM in the $\mathcal{F}_{\mathsf{AKE-KCI}}$ emulation, then $\mathcal{F}$ will replicate this behavior: First, when $\langle ssid, \mathsf{P}', \mathsf{P}, \mathsf{pw}^{\circ\circ}\rangle$ completes, $\mathcal{F}$ picks a random key $SK' \leftarrow \{0,1\}^\tau$ as SK because, by invariant (3) this SaPAKE-layer sub-sessions

is FRESH. Second, when $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}^{\circ}\rangle$ completes, $\mathcal{F}$ will assign to it the same key SK' because, by invariant (3) that session will also be FRESH, and by invariant (2) since these two sub-sessions have the same AKE-layer ID's $ssid_{\Pi}$ (otherwise they wouldn't be connected on the AKE-layer), then their SaPAKE-layer ID's match too, i.e., $ssid = ssid'$, and so do their passwords, i.e., $\mathsf{pw}^{\circ} = \mathsf{pw}^{\circ\circ}$. In all other cases both $\mathbf{G}_9$ and SIM pick a random key SK for session $\langle ssid_{\Pi}, ssid, \mathsf{P}, \mathsf{P}'\rangle$, therefore in the simulated world regardless if $\mathcal{F}$ passes that key to P, or it replaces it with a new choice SK' of a random key, party P outputs $(sid, ssid, SK')$ for a random key SK', which matches the distribution of its outputs in $\mathbf{G}_9$

We argue that the three invariants indeed hold. We start from invariant (1). Note that a FRESH AKE-layer session $\langle ssid_{\Pi}, ssid, \mathsf{P}, \mathsf{P}'\rangle$ turns COMPROMISED if $\mathsf{SIM}_{\mathsf{AKE}}$ sends $(\text{IMPERSONATE}, sid, ssid_{\Pi}, \mathsf{P}')$ and P' is declared COMPROMISED. Consider case $\mathsf{P}' = \mathsf{S}$ first. S is declared COMPROMISED by SIM (and $\mathbf{G}_9$) only if $\mathcal{A}$ sends $(\text{STEALPWDFILE}, sid)$, see step 1 in Fig. 9, in which case $\mathcal{F}$ marks the password file COMPROMISED. If $\mathsf{SIM}_{\mathsf{AKE}}$ then sends $(\text{IMPERSONATE}, sid, ssid_{\Pi}, \mathsf{S})$ then SIM sends $(\text{IMPERSONATE}, sid, ssid)$ to $\mathcal{F}$, at which point $\mathcal{F}$ marks the SaPAKE-layer session $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}'\rangle$ as COMPROMISED *if* this SaPAKE-layer session is FRESH. However, note that U session in case $(*)$ does not start unless $\mathcal{F}$ replies SUCC to SIM's query $(\text{TESTABORT}, sid, ssid, \mathsf{U})$, see step (1a) in Fig. 11, which means that the SaPAKE-layer U session remained FRESH after SIM's TESTABORT query, and hence it became COMPROMISED after SIM's IMPERSONATE query.

Consider now the case when $\mathsf{P}' = \mathsf{U}$. If U is declared COMPROMISED then $\mathcal{A}$ queried $F_{\mathsf{S}}(\mathsf{pw})$, either offline or online, so SIM holds a record $\langle \text{FILE}, \mathsf{U}, \mathsf{S}, \mathsf{pw}\rangle$, hence if $\mathsf{SIM}_{\mathsf{AKE}}$ sends $(\text{IMPERSONATE}, sid, ssid_{\Pi}, \mathsf{U})$ then SIM sends $(\text{TESTPWD}, sid, ssid, \mathsf{S}, \mathsf{pw})$ to $\mathcal{F}$, and since the tested password is correct, $\mathcal{F}$ will process the SaPAKE-layer session $\langle ssid, \mathsf{S}, \mathsf{U}, \mathsf{pw}\rangle$ as follows: If this SaPAKE-layer session was FRESH then it will become COMPROMISED, and if it was INTERRUPTED then it will remain INTERRUPTED. However, note that if $\mathcal{A}$ queries $F_{\mathsf{S}}(\mathsf{pw})$ either offline or online, this means that either some OFFLINETESTPWD query to $\mathcal{F}$ in step 1, Fig. 9, or some TESTPWD query to $\mathcal{F}$ in step 4c, Fig. 10, received $\mathcal{F}$'s response "correct guess", but at this point $\mathcal{F}$ sets $\mathsf{flag} :=$ COMPROMISED. This means that SaPAKE-layer S session is either COMPROMISED or it is INTERRUPTED but $\mathsf{flag} =$ COMPROMISED, as claimed.

As for invariant (2), part (a) is immediate because $ssid_{\Pi}$ is formed as $[ssid||\mathsf{prfx}]$ on each session, so equality of $ssid_{\Pi}$'s implies equality of $ssid$'s. As for part (b), note that U session in case $(*)$ runs only if TESTABORT does not make it abort, see step 1a, Fig. 11, which means that $\mathsf{pw}' = \mathsf{pw}$.

We turn to invariant (3). Note that $\mathcal{F}_{\mathsf{AKE}-\mathsf{KCI}}$ emulation, by either $\mathbf{G}_9$ or SIM, connects these two AKE-layer session only if their AKE-layer ssid's match, i.e. $ssid_{\Pi} = ssid'_{\Pi}$. Note also that $ssid_{\Pi} == [ssid||\mathsf{prfx}]$ and $ssid_{\Pi} == [ssid'||\mathsf{prfx}']$, which implies that $ssid = ssid'$ and that the OPRF transcript prefixes of this U and S sessions matched as well, i.e. $\mathsf{prfx} = \mathsf{prfx}'$. Note that U's AKE-layer session $\langle ssid_{\Pi}, ssid, \mathsf{U}, \mathsf{S}\rangle$ starts FRESH in case $(*)$, and if that session remains FRESH

until NewKey message for it (as must be the case for $\mathcal{F}_{\mathsf{AKE-KCI}}$ to "connect" it to the S session), then SIM does not send to $\mathcal{F}$ any queries which would change the status of the SaPAKE-layer session, i.e. the corresponding SaPAKE-layer session stays FRESH. Regarding S's AKE-layer session $\langle ssid'_{\Pi}, ssid', \mathsf{S}, \mathsf{U} \rangle$, note that when this session starts, in step 2a in Fig. 10, if $\mathsf{prfx} = \mathsf{prfx}'$ then SIM does not write record $\langle ssid, \mathsf{act} \rangle$. Consequently SIM does not send Interrupt for that session to $\mathcal{F}$, and also SIM will never choose that session in step 4(c)i, Fig. 10, and hence will not send TestPwd for that session to $\mathcal{F}$ in step 4(c)ii. (These are all consequences of the fact that if OPRF transcript prefixes match then SIM cannot, and does not, use this S session to evaluate S's random function F_{S}.) Consequently, the corresponding SaPAKE-layer session stays FRESH as well, as we claimed.

Summing up all results above, we conclude that $\mathcal{Z}$'s distinguishing advantage between the real world and the simulated world is a negligible function of the security parameter τ, which completes the proof.

6 OPAQUE: A Strong Asymmetric PAKE Instantiation

Fig. 12 shows OPAQUE, a concrete instantiation of the generic OPRF+AKE protocol from Fig. 8. The OPRF is instantiated with the DH-OPRF scheme from [31] recalled in Appendix B, while the AKE protocol can be instantiated with any AKE protocol that realizes the $\mathcal{F}_{\mathsf{AKE-KCI}}$ functionality from Fig. 7. In Fig. 12 this is illustrated with HMQV [38]. Note that the two messages of DH-OPRF and the first two messages from HMQV run "in parallel" hence the OPRF does not add to the total number of messages exchanged in the protocol. In this case, the number of messages is three as needed for HMQV to instantiate the $\mathcal{F}_{\mathsf{AKE-KCI}}$ functionality (see Section 5.1). The third component of the protocol is a random-key robust authenticated encryption scheme AE which can accommodate multiple instantiations as discussed below.

By Theorem 2 on the security of the generic OPRF+AKE construction, by Lemma 1 in Appendix B on the security of DH-OPRF, the security of HMQV as AKE-KCI, and assuming a random-key robust and equivocable instantiation of AE, we get that protocol OPAQUE realizes functionality $\mathcal{F}_{\mathsf{saPAKE}}$. Hence it is a provably-secure Strong aPAKE protocol, under the One-More Diffie-Hellman assumption [5, 31] in the random oracle model. A similar result can be argued on the basis of the SIGMA protocol from [37] – see Section 5.1.

6.1 Protocol Details and Properties

We expand on the specification of OPAQUE and the protocol's properties.

• *Password registration.* Password registration is the only part of the protocol assumed to run over secure channels where parties can authenticate each other. We note that while OPAQUE is presented with S doing all the registration operations, in practice one may want to avoid that. Instead, we can let S choose an OPRF key k_s and U choose pw, and then run the OPRF protocol between

Public Parameters and Components

- Security parameter τ
- Group G of prime order $q, |q| = 2\tau$ and generator g (G^* denotes $G \setminus \{1\}$).
- Hash functions $H(\cdot,\cdot)$, $H'(\cdot)$ with ranges $\{0,1\}^{2\tau}$ and G, respectively.
- Pseudorandom function (PRF) $f(\cdot)$ with range $\{0,1\}^{2\tau}$.
- OPRF function defined as $F_k(x) = H(x, (H'(x))^k)$ for key $k \in \mathbb{Z}_q$.
- Random-key robust authenticated encryption scheme ($\mathsf{AuthEnc}, \mathsf{AuthDec}$).
- Key exchange formula KE defined below.

Password Registration

1. ($\textsc{StorePwdFile}, sid, \mathsf{U}, \mathsf{pw}$): S computes $k_s \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, $\mathsf{rw} := F_{k_s}(\mathsf{pw})$, $p_s \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, $p_u \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, $P_s := g^{p_s}$, $P_u := g^{p_u}$, $c \leftarrow \mathsf{AuthEnc}_{\mathsf{rw}}(p_u, P_u, P_s)$; it records $\mathsf{file}[sid] := \langle k_s, p_s, P_s, P_u, c\rangle$.

Login

1. ($\textsc{UsrSession}, sid, ssid, \mathsf{S}, \mathsf{pw}$): U picks $r, x_u \leftarrow_{\mathrm{R}} \mathbb{Z}_q$; sets $\alpha := (H'(\mathsf{pw}))^r$ and $X_u := g^{x_u}$; sends α and X_u to S.
2. ($\textsc{SvrSession}, sid, ssid$): On input α from U, S proceeds as follows:
   (a) Checks that $\alpha \in G^*$. If not, outputs ($\textsc{abort}, sid, ssid$) and halts;
   (b) Retrieves $\mathsf{file}[sid] = \langle k_s, p_s, P_s, P_u, c\rangle$;
   (c) Picks $x_s \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and computes $\beta := \alpha^{k_s}$ and $X_s := g^{x_s}$;
   (d) Computes $K := \mathsf{KE}(p_s, x_s, P_u, X_u)$ and sets: $ssid' := H(sid, ssid, \alpha)$, $SK := f_K(0, ssid'), A_s = f_K(1, ssid')$;
   (e) Sends β, X_s, c and A_s to U;
3. On input β, X_s, c and A_s from S, U proceeds as follows:
   (a) Checks that $\beta \in G^*$. If not, outputs ($\textsc{abort}, sid, ssid$) and halts;
   (b) Computes $\mathsf{rw} := H(\mathsf{pw}, \beta^{1/r})$;
   (c) Computes $\mathsf{AuthDec}_{\mathsf{rw}}(c)$. If the result is $\perp$, outputs ($\textsc{abort}, sid, ssid$) and halts. Otherwise sets $(p_u, P_u, P_s) := \mathsf{AuthDec}_{\mathsf{rw}}(c)$;
   (d) Computes $K := \mathsf{KE}(p_u, x_u, P_s, X_s)$ and sets: $ssid' := H(sid, ssid, \alpha)$, $SK := f_K(0, ssid'), A_s = f_K(1, ssid'), A_u = f_K(2, ssid')$;
   (e) Verifies that A_s is same as received from S. If not, it outputs ($\textsc{abort}, sid, ssid$) and halts.
   (f) Sends A_u to S and outputs ($sid, ssid, SK$).
4. On input A_u from U, S verifies that $A_u = f_K(2, ssid')$. If not, it outputs ($\textsc{abort}, sid, ssid$) and halts; else it outputs ($sid, ssid, SK$).

Key exchange formula KE with HMQV instantiation (if any of $X_u, P_u, X_s, P_s \notin G^*$ the receiving party outputs ($\textsc{abort}, sid, ssid$) and halts)

$$\text{For } \mathsf{S}\text{: } \mathsf{KE}(p_s, x_s, P_u, X_u) = H\left((X_u P_u^{e_u})^{x_s + e_s p_s}\right)$$
$$\text{For } \mathsf{U}\text{: } \mathsf{KE}(p_u, x_u, P_S, X_S) = H\left((X_s P_s^{e_s})^{x_u + e_u p_u}\right)$$

where $e_u = H(X_u, \mathsf{S}, ssid') \bmod q, e_s = H(X_s, \mathsf{U}, ssid') \bmod q$.

Fig. 12: Protocol OPAQUE with DH-OPRF and HMQV

U and S so only U learns its secrets $(\mathsf{pw}, \mathsf{rw}, p_u)$. The server chooses its pair (p_s, P_s) and provides P_s to U who builds the ciphertext c and sends to S. In this way the server *never* sees the user's password, a *major* benefit, for example to avoid accidental storage of plaintext passwords that has affected also security-conscious companies [2, 3]. Note that this prevents S from checking password rules, an operation that can be moved to the client side (restricted server-side checks such as preventing the repeat of a recent password can be implemented).

- *Authenticated encryption.* As specified in Section 5.2, the authenticated encryption scheme AE used in the protocol needs to satisfy the random-key robustness property defined there. In practice, using an encrypt-then-mac scheme with HMAC-256 (or larger) as the MAC provides this property (in particular, any AE scheme can be made robust by computing HMAC on top of the ciphertext output by $\mathsf{AuthEnc}$). We note that the standard GCM mode for authenticated encryption is not random-key robust but it can be adapted to achieve this property.

- *Key exchange and forward secrecy.* The generic AKE representation in Fig. 12 via the KE formula is done for simplicity and since it applies to HMQV and, more generally, to protocols that follow the implicit authentication approach. Other protocols may require additional operations, such as signatures in the case of the SIGMA as mentioned above. It follows from our analysis that any KE protocol used with OPAQUE must resist KCI attacks and enjoy full forward secrecy (against active attacks). The latter condition implies that OPAQUE must have at least three messages[13].

- *User iterated hashing.* OPAQUE can be strengthened by increasing the cost of a dictionary attack in case of server compromise. This is done by changing the computation of rw to $\mathsf{rw} = H^n(F_k(\mathsf{pw}))$, that is, the client applies n iterations of the function H on top of the result of the OPRF value $F_k(\mathsf{pw})$. In practice, the iterations H^n would be replaced with one of the standard password-based key derivation functions, such as PBKDF2 [36] or bcrypt [46], or by more modern memory-hard functions such as Argon2 [10] or Scrypt [44]. This forces an attacker that compromises the password file at the server to compute for *each* candidate password pw' the function $F_k(\mathsf{pw}')$ as well as the additional n hash iterations. Note that n needs not be remembered by the user; it can be sent from S to U in the server's message. Furthermore, one can follow Boyen's design and apply the probabilistic Halting KDF function [11] as used in [12] so that the iterations count is hidden from the attacker and even from the server. An additional benefit of client-side hardening is that not only it slows down offline attacks upon server compromise but also online password-guessing attacks. On the other hand, clients running on weak machines are limited in the amount of hardening they can apply.

[13] To achieve full forward secrecy one of the client messages must depend on the user's private key [38]. So at the minimum one needs a first message from the client with user account information, followed by a message from the server with the user's envelope, and a third from the client that depends on the user's private key.

• *Performance.* OPAQUE takes three messages, one exponentiation for S, two and a hashing-into-G for U, plus the cost of KE. With HMQV, the latter cost is one offline fixed-base exponentiation and one multi-exponentiation (at the cost of 1.17 regular exponentiations) per party (about three exponentiations in total for the server and four for the user). All exponentiations are in regular DH groups, hence accommodating the fastest elliptic curves (e.g., no pairings). It is common in PAKE protocols to count number of group elements transmitted between the parties. In OPAQUE, U sends two while S sends three (one, P_u, can be omitted at the cost of one fixed-based exponentiation at the client). See also Section 6.3.

• *Performance comparison.* The introduction presents background on OPAQUE and other password protocols. Here we provide a comparison with the more efficient among these protocols, particularly those that are being, or have been, considered for standardization. Clearly, OPAQUE is superior security-wise as the only one not subject to pre-computation attacks, but it also fares well in terms of performance.

AugPAKE [49, 50], is computationally very efficient with only 2.17 exponentiations per party; however, it uses 4 messages and does not provide forward secrecy. In addition, the protocol has only been analyzed as a PAKE protocol, not aPAKE [50]. Another proposed aPAKE protocol, SPAKE2+ [4, 19], uses two messages only and 3 multi-exponentiations (or about 3.5 exponentiations) per party which is similar to OPAQUE cost. The security of the protocol has only been informally argued in [19] and to the best of our knowledge no formal analysis has appeared. We also mention SRP which has been included in TLS ciphersuites in the past but is considered outdated as it does not have an instantiation that works over elliptic curves (the protocol is defined over rings and uses both addition and multiplication). Its implementations over RSA moduli is therefore less efficient than those over elliptic curve; it also takes 4 messages.

Recently, a few protocols have been presented with proofs of aPAKE security but, as the rest, they are vulnerable to pre-computation. The protocol VTBPEKE in [45] uses 3 messages and 4 exponentiations per party and was proven secure in the non-UC aPAKE model of [9], while AuCPace [28] requires 4 messages and 4 (resp. 3) exponentiations for the server (resp. client) and is proven in the UC aPAKE model of [26]. Also proven in this model is [34], a *simultaneous* one-round scheme that works over bilinear groups and requires 4 exponentiations and 3 pairing per party. We note that the above protocols require an initial message from server to user in order to transmit salt, which may result in one or two added messages to the above message counts (except for VTBPEKE which already includes the salt transmission in its 3 messages). All these protocols, like OPAQUE, work in the RO model.

• *Threshold implementation.* We comment on a simple extension of OPAQUE that can be very valuable in large deployments, namely, the ability to implement the OPRF phase as a Threshold OPRF [32]. In this case, an attacker needs to break into a threshold of servers to be able to impersonate the servers to the user or to run an offline dictionary attack. Such an implementation requires no user-

side changes, i.e., the user does not need to know if the system is implemented with one or multiple servers.

• *OPAQUE as a general secret retrieval mechanism.* An important feature of OPAQUE is that it can serve not only as an aPAKE protocol but more generally as a means for retrieving a secret or credential from a server (such a secret is protected under ciphertext c stored at the server). In this functionality, OPAQUE acts as a 1-out-of-1 implementation of the PPSS scheme from [32]. The retrieved secret can be used to protect information such as a bitcoin wallet, serve as a user-controlled encryption key for a backup or other information repository (e.g., a password manager), used as an authentication or signing key, and more. This offers a far more secure alternative to the practice of deriving low-entropy secrets directly from a user's password.

6.2 OPAQUE and TLS: Client authentication and hedging against PKI failures

As discussed earlier, OPAQUE offers a much more secure alternative to password-authenticated key exchange than the current practice of transmitting passwords over TLS. Yet, OPAQUE (as any other aPAKE) still requires additional mechanisms for negotiating cryptographic parameters (such as crypto algorithms) and for establishing the means needed to encrypt and authenticate communications using the keys generated by OPAQUE. Thus, it is natural to compose OPAQUE with the TLS protocol to offer strong password security while leveraging the standardized negotiation and record-layer security of TLS. Moreover, TLS can offer an initial server-authenticated channel to protect the privacy of account information, such as user name, transmitted between client and server. Here we discuss possible schemes for composing OPAQUE and TLS. We consider TLS 1.3 [47] as the upcoming and more secure version of TLS although some of the mechanisms can be implemented via prior versions of TLS.

The simplest TLS-OPAQUE combination is one where U's private key p_U stored by OPAQUE at S is used as a signature key for TLS client authentication. In this case, the OPAQUE-extended handshake protocol includes the following sequential steps (for a total of 5 messages): (i) a 1-RTT run of TLS 1.3 handshake protocol that produces a session key authenticated by S's TLS certificate; (ii) the first two OPAQUE messages exchanged between client and server excluding the KE values g^x, g^y (these were already exchanged as part of the TLS 1-RTT run); (iii) TLS 1.3 client authentication using U's private signature key p_U retrieved from S in step (ii).

These steps result in mutual authentication where server's authentication is accomplished based on a TLS certificate. The client can either trust such a certificate or it can verify equality of the certificate's public key against P_S as retrieved by OPAQUE. In case of a mismatch the client can request a signature

of S using P_S which is computed on the TLS transcript[14]. In the latter case, the protocol does not rely on PKI certificates except for protecting account information. *In all cases, the security of passwords and password authentication does not rely on PKI but on OPAQUE only.*

Variants of the above scheme include the use of a TLS 1.3 0-RTT exchange for sending the first OPAQUE message (including protected account information) in which case steps (i) and (ii) are executed concurrently for a total of three messages (flights in TLS jargon) as in regular TLS. This variant, while more efficient, relies on 0-RTT which is available only to clients and servers that have previously shared a key (negotiated in a previous handshake). A 0-RTT variant independent of pre-shared keys and based instead on a server's public key is possible (e.g., [40]) but it is not standardized by TLS 1.3. Finally, if protecting the secrecy of user's account information is not considered necessary then steps (i) and (ii) can run concurrently (without using the 0-RTT scheme); in this case server's authentication is based on OPAQUE's server key P_S. This setting also allows for a more efficient scheme using HMQV as illustrated in Fig. 12 (with additional key derivation and record layer processing based on TLS).

We note that the security of the above variants and composition rely on the modularity of OPAQUE that can compose the OPRF steps with arbitrary key-exchange protocols (with KCI and forward security). We remark that the security of TLS 1.3 has been analyzed in multiple works (cf. [20–22, 24, 35, 41]) with client authentication via exported authentication (or "post-handshake authentication") studied in [39].

6.3 An OPAQUE variant: Multiplicative blinding

A variant of OPAQUE is obtained by replacing the user's exponential blinding operation $\alpha := H'(\mathsf{pw})^r$ in DH-OPRF with $\alpha := H'(\mathsf{pw}) \cdot g^r$. The server responds as before with $\beta = \alpha^{k_s}$. Assuming that U knows the value $y = g^{k_s}$ (previously stored or received from S), it can compute the same "hashed Diffie-Hellman" value $H'(\mathsf{pw})^{k_s}$ as β / y^r. The advantage of this variant is that while the number of client exponentiations remains the same, one is fixed-base (g^r) and the other (y^r) can also be fixed-base if U caches y, a realistic possibility for accounts where the user logs in frequently (e.g., a personal email or social network). Computing y^r can also be done while waiting for the server's response to reduce latency. Moreover, both exponentiations can be done offline although only short-term storage is recommended as the leakage of r exposes $H'(\mathsf{pw})$ (hence opens pw to a dictionary attack). If U does not store y, it needs to be transmitted to U by S together with the response β. This still allows for fixed-base optimization for computing g^r but not for y^r. Note that the two OPAQUE variants (with exponential or multiplicative blinding) compute the same value rw, hence an implementation can support both and leave it up to the client to choose one (and request y from S if needed).

[14] Such additional server authentication and the client authentication in step (iii) can be implemented using TLS exported authenticators as defined in [51] (client authentication in this case corresponds to post-handshake authentication in [47]).

However, it turns out that this multiplicative mechanism results in an OPRF protocol that does *not* realize our OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$. Thus, our analysis here does not imply the security of the multiplicative OPAQUE variant in general. If rw is redefined as $\mathsf{rw} := H(\mathsf{pw}, y, H'(\mathsf{pw})^{k_s})$, i.e., if y is included under the hash, then the resulting OPRF does realize our functionality, and OPAQUE remains secure as SaPAKE under both blinding variants. This change, however, introduces a (slight) overhead of having to transmit y even if the client implements the exponential blinding operation. An alternative approach would be to replace the OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ with a weaker form $\mathcal{F}'_{\mathsf{OPRF}}$ and to show that (i) $\mathcal{F}'_{\mathsf{OPRF}}$ is realized by the multiplicative variant (even without hashing y) and (ii) $\mathcal{F}'_{\mathsf{OPRF}}$ is sufficient for proving Theorem 2 hence implying the security of OPAQUE as SaPAKE. We intend to investigate this weakening of $\mathcal{F}_{\mathsf{OPRF}}$.

Acknowledgments. We thank Julia Hesse and Björn Tackmann for their invaluable comments that helped improve our presentation and formal treatment, and to Clemens Hlauschek, Kevin Lewi and Payman Mohassel for helpful discussions.

References

1. CFRG, Crypto Forum Research Group, `https://datatracker.ietf.org/rg/cfrg/documents/`.
2. Facebook stored hundreds of millions of passwords in plain text, `https://www.theverge.com/2019/3/21/18275837/facebook-plain-text-password-storage-hundreds-millions-users`.
3. Google stored some passwords in plain text for fourteen years, `https://www.theverge.com/2019/5/21/18634842/google-passwords-plain-text-g-suite-fourteen-years`.
4. M. Abdalla and D. Pointcheval. Simple password-based encrypted key exchange protocols. In *Topics in Cryptology – CT-RSA 2005*, pages 191–208. Springer, 2005.
5. M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3), 2003.
6. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology – EUROCRYPT 2000*, pages 139–155. Springer, 2000.
7. S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Computer Society Symposium on Research in Security and Privacy – S&P 1992*, pages 72–84. IEEE, 1992.
8. S. M. Bellovin and M. Merritt. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *ACM Conference on Computer and Communications Security – CCS 1993*, pages 244–250. ACM, 1993.
9. F. Benhamouda and D. Pointcheval. Verifier-based password-authenticated key exchange: New models and constructions. *IACR Cryptology ePrint Archive*, 2013:833, 2013.

10. A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: new generation of memory-hard functions for password and other applications. In *IEEE European Symposium on Security and Privacy – EuroS&P 2016*, pages 292–302. IEEE, 2016.
11. X. Boyen. Halting password puzzles. In *Usenix Security Symposium – SECURITY 2007*, pages 119–134. The USENIX Association, 2007.
12. X. Boyen. HPAKE: Password authentication secure against cross-site user impersonation. In *Cryptology and Network Security – CANS 2009*, pages 279–298. Springer, 2009.
13. V. Boyko, P. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *Advances in Cryptology – EUROCRYPT 2000*, pages 156–171. Springer, 2000.
14. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science – FOCS 2001*, pages 136–145. IEEE, 2001.
15. R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. MacKenzie. Universally composable password-based key exchange. In *Advances in Cryptology – EUROCRYPT 2005*, pages 404–421. Springer, 2005.
16. R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology – EUROCRYPT 2001*, pages 453–474. Springer, 2001.
17. R. Canetti and H. Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In *Advances in Cryptology – CRYPTO 2002*, pages 143–161. Springer, 2002. Also Cryptology ePrint Archive, Report 2002/120.
18. R. Canetti and H. Krawczyk. Universally composable notions of key exchange and secure channels. In *Advances in Cryptology – EUROCRYPT 2002*, pages 337–351. Springer, 2002.
19. D. Cash, E. Kiltz, and V. Shoup. The twin Diffie-Hellman problem and applications. In *Advances in Cryptology – EUROCRYPT 2008*, pages 127–145. Springer, 2008.
20. C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *ACM CCS 17*, 2017.
21. C. Cremers, M. Horvat, S. Scott, and T. van der Merwe. Automated verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *IEEE S&P 2016.*, 2016.
22. B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM CCS*, 2015. Also, Cryptology ePrint Archive, Report 2015/914.
23. P. Farshim, C. Orlandi, and R. Rosie. Security of symmetric primitives under incorrect usage of keys. *IACR Transactions on Symmetric Cryptology*, 2017(1):449–473, 2017.
24. M. Fischlin and F. Günther. Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. In *IEEE S&P 2017*, 2017.
25. M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography – TCC 2005*, pages 303–324. Springer, 2005.
26. C. Gentry, P. MacKenzie, and Z. Ramzan. A method for making password-based key exchange resilient to server compromise. In *Advances in Cryptology – CRYPTO 2006*, pages 142–159. Springer, 2006.
27. L. Gong, M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, 1993.

28. B. Haase and B. Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. In *CHES*, 2019. Cryptology ePrint Archive, Report 2018/286.
29. S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):230–268, 1999.
30. S. Jarecki, A. Kiayias, and H. Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *Advances in Cryptology – ASIACRYPT 2014*, pages 233–253. Springer, 2014.
31. S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. Highly-efficient and composable password-protected secret sharing (or: how to protect your bitcoin wallet online). In *IEEE European Symposium on Security and Privacy – EuroS&P 2016*, pages 276–291. IEEE, 2016.
32. S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In *Applied Cryptology and Network Security – ACNS 2017*, pages 39–58. Springer, 2017.
33. S. Jarecki, H. Krawczyk, and J. Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *Advances in Cryptology - EUROCRYPT 2018*, pages 456–486. Springer, 2018.
34. C. S. Jutla and A. Roy. Smooth NIZK arguments. In *Theory of Cryptography – TCC 2018*, pages 235–262. Springer, 2018.
35. B. B. K. Bhargavan and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE S&P 2017*, 2017.
36. B. Kaliski. PKCS# 5: Password-based cryptography specification version 2.0. 2000.
37. H. Krawczyk. SIGMA: The "SIGn-and-MAc" approach to authenticated Diffie-Hellman and its use in the IKE protocols. In *CRYPTO*, pages 400–425, 2003.
38. H. Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol (extended abstract). In *Advances in Cryptology – CRYPTO 2005*, page 546. Springer, 2005.
39. H. Krawczyk. Unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in TLS 1.3). In *ACM CCS 2016*, 2016.
40. H. Krawczyk and H. Wee. The OPTLS protocol and TLS 1.3. In *EuroS&P*, 2016.
41. X. Li, J. Xu, Z. Zhang, D. Feng, , and H. Hu. Multiple handshakes security of TLS 1.3 candidates. In *IEEE S&P 2016.*, 2016.
42. P. Mackenzie. More efficient password-authenticated key exchange. In *Topics in Cryptology – CT-RSA 2001*, pages 361–377. Springer, 2001.
43. P. MacKenzie, S. Patel, and R. Swaminathan. Password-authenticated key exchange based on RSA. In *Advances in Cryptology – ASIACRYPT 2000*, pages 599–613. Springer, 2000.
44. C. Percival. Stronger key derivation via sequential memory-hard functions, `http://www.tarsnap.com/scrypt/scrypt.pdf`,.
45. D. Pointcheval and G. Wang. VTB-peke: Verifier-based two-basis password exponential key exchange. In *ACM Asia Conference on Computer and Communications Security – AsiaCCS 2017*, pages 301–312. ACM, 2017.
46. N. Provos and D. Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.
47. E. Rescorla. The transport layer security (TLS) protocol version 1.3 (draft 24), Feb. 2018.
48. J. Schmidt. Requirements for password-authenticated key agreement (PAKE) schemes. Technical report, 2017.
49. S. Shin and K. Kobara. Augmented password-authenticated key exchange (AugPAKE). *draft-irtf-cfrg-augpake-08*.

50. S. Shin, K. Kobara, and H. Imai. Security proof of AugPAKE. *IACR Cryptology ePrint Archive*, 2010:334, 2010.
51. N. Sullivan. Exported authenticators in TLS (draft 4), Oct. 2018.

A UC OPRF Definition: Discussion of Revisions

In Section 3 we showed a definition of the adaptive UC OPRF functionality, $\mathcal{F}_{\mathsf{OPRF}}$, shown in Fig. 3. The definition of $\mathcal{F}_{\mathsf{OPRF}}$ in Fig. 3 diverges from the definition of the same functionality we gave in the proceedings version of this paper [33], and the definition there was itself a revision of the UC OPRF definition given in [31]. Below we first explain the modifications which both the UC OPRF of Fig. 3 and the UC OPRF of [33] share vis-a-vis the UC OPRF version of [31], and then we explain the revisions made by the UC OPRF of Fig. 3 in comparison to the UC OPRF definition in [33].

Changes from OPRF Functionality of [31]. To use UC OPRF in our application(s) we need to make some changes to the way functionality $\mathcal{F}_{\mathsf{OPRF}}$ was defined in [31], as described below. Changes (3) and (4) are essentially syntactic and require only cosmetic changes in the security argument. Change (2) makes the functionality weaker. Change (1) is the only one which influences the security argument in a more essential way. Fortunately, the DH-OPRF protocol that we use for OPRF instantiation in our protocols, shown in [31] to realize their version of the OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$, also realizes our modified $\mathcal{F}_{\mathsf{OPRF}}$ functionality. We recall the DH-OPRF protocol in Fig. 13 in Appendix B, adapting its syntax to our changes in $\mathcal{F}_{\mathsf{OPRF}}$, and we argue that the security proof of [31] which shows that it realizes $\mathcal{F}_{\mathsf{OPRF}}$ defined by [31] extends to the modified functionality $\mathcal{F}_{\mathsf{OPRF}}$ presented here.

(1) We extend the OPRF functionality to allow the *adaptive compromise* of a server holding the PRF key via a Compromise message. Such action is needed in the aPAKE setting where the attacker can compromise a server's password file that contains an OPRF key. After the compromise, $\mathcal{A}$ is allowed to compute that server's PRF function on any value of its choice using command OfflineEval. We note that functionality $\mathcal{F}_{\mathsf{OPRF}}$ distinguishes between (statically) *corrupted servers* and (adaptively) *compromised sessions* (the latter representing different OPRF keys at the same server), in consistency with the aPAKE functionality from Fig. 2 that distinguishes between an entirely corrupted server and particular aPAKE instances that can be adaptively compromised by an adversary.

(2) We eliminate the condition that $\mathcal{F}_{\mathsf{OPRF}}$ aborts on message $(\text{RcvComplete}, ssid, i)$, denoted $(\text{RcvComplete}, ssid, \mathsf{S}^*)$ in [31], if (i) server S is honest, (ii) $i \neq \mathsf{S}$, and (iii) this OPRF sub-session was initialized by U via command $(\text{Eval}, ssid, \mathsf{S}', x)$ for $\mathsf{S}' = \mathsf{S}$, where S is the server which initialized the OPRF instance with session ID *sid*. The OPRF protocol of [31] makes use of an authenticated channel, hence the user U aborts if the adversary does not relay the messages from the server S if U ran this protocol with S and S is honest. In contrast, we implement UC OPRF without relying

on authenticated channels, so such clause must be deleted. This does not affect the security of our Strong aPAKE protocols, which are password-only AKE's and are thus intended to be used over insecure channels.

(3) We change the SndrComplete message so that it is sent from S if S is uncompromised, and from of $\mathcal{A}$ only if S is compromised. This allows an honest aPAKE server to enforce a single OPRF execution, and thus e.g. a single password guess per aPAKE sub-session, which is crucial for aPAKE security.

(4) We add an Initialization phase to the functionality, which models a server picking an OPRF key. This interface simplifies the usage of OPRF in aPAKE application where teh server picks an OPRF key for each new user. This modeling differs from [31] who framed OPRF initialization as an interactive procedure through an Eval call while here it is performed locally by the server.

Syntactic Differences from Adaptive OPRF Functionality in [33]. In the definition of the Adaptive OPRF functionality in Fig. 3 we introduce several modifications to the way this functionality was defined in the proceedings version of this paper [33]. The purpose of these changes is to simplify some functionality interfaces and to clarify some ambiguities in the definition of $\mathcal{F}_{\mathsf{OPRF}}$ in [33]. In the explanation below we will refer to the OPRF functionality as defined in Fig. 3 as a "revised" functionality $\mathcal{F}_{\mathsf{OPRF}}$, and to the same functionality as defined in [33] as a "proceedings-version" functionality $\mathcal{F}_{\mathsf{OPRF}}$.

The most important difference introduced by the revised functionality $\mathcal{F}_{\mathsf{OPRF}}$ is that it uses variable i to index random function instances, denoted $F_{sid,i}(x)$, whereas the proceedings-version of $\mathcal{F}_{\mathsf{OPRF}}$ used variable S to index these functions, denoted $F_{sid,\mathsf{S}}$ therein.[15] Indeed, a sequence of works on UC OPRF [30–33] used the same convention of indexing random functions kept by $\mathcal{F}_{\mathsf{OPRF}}$ by variable S. However, this notation for function indices has an unfortunate effect of using the same variable to denote data, i.e. an index of a random function, and to denote real-world entities. For example, in the proceedings-version $\mathcal{F}_{\mathsf{OPRF}}$ the adversary can make an honest client U output $F_{sid,\mathsf{S}^*}(x)$ for any *bitstring* S^* which does not correspond to an honest server S, but the proceedings-version $\mathcal{F}_{\mathsf{OPRF}}$ models this with SndrComplete message sent by a *corrupt entity* S^* to $\mathcal{F}_{\mathsf{OPRF}}$, followed by message (RcvComplete, ..., S^*) sent by $\mathcal{A}$. This seemingly requires the adversary to create a "virtual corrupt entity" for every function it creates, and is not compliant with UC conventions [14]. We stress that this new notation for the OPRF functionality requires only syntactic changes in the security arguments given for the UC OPRF schemes in [31, 33]. Indeed, the UC OPRF notation proposed above is a better fit for these security arguments.[16]

[15] In Fig. 3 we use notation F_i instead of $F_{sid,i}$, but all records of functionality $\mathcal{F}_{\mathsf{OPRF}}$ are implicitly indexed by the session identifier *sid*.

[16] In particular, the simulators shown for the UC OPRF protocol in [31] treats variables S' and S^* as data, not "virtual corrupt entities", and the messages (SndrComplete, *sid*, *ssid*) pertaining to $\mathsf{S}' \neq \mathsf{S}$ were sent by the simulator, i.e. the ideal-world adversary, as required by the revised $\mathcal{F}_{\mathsf{OPRF}}$ syntax.

This change of function-indexing variable is reflected in the modified syntax of functionality commands OfflineEval, SndrComplete, and RcvComplete. We explain this new syntax below, and we note that all these commands use variable S to denote the identifier of a unique entity which initializes an OPRF instance with session id *sid* by sending (Init, *sid*) to $\mathcal{F}_{\mathsf{OPRF}}$.

(1) Off-line evaluation of function $F_{sid,i}$ on argument x is executed via command (OfflineEval, sid, i, x). This command can be issued by server S, in which case if S is honest then it is allowed only if $i = \mathsf{S}$, since an honest server S evaluates only its own function $F_{sid,\mathsf{S}}$, while a corrupt server S can evaluate an arbitrary function instance. The adversary $\mathcal{A}$ can also issue this command, but if S is not compromised (or corrupted) then $\mathcal{A}$ can send it only for $i \neq \mathsf{S}$, since an adversary cannot evaluate function $F_{sid,\mathsf{S}}$ without compromising server S.

(2) Command (SndrComplete, $sid, ssid$) models the execution of the server-side OPRF protocol using S's keys, and its effect is to increment counter tx which counts the server-side OPRF executions for the "honest server function" $F_{sid,\mathsf{S}}$. This command can be issued by S, or by $\mathcal{A}$ if S is compromised because S compromise allows $\mathcal{A}$ to run the server-side OPRF protocol using S's keys.

A key difference from the proceedings-version $\mathcal{F}_{\mathsf{OPRF}}$ model is that in the revised $\mathcal{F}_{\mathsf{OPRF}}$ command SndrComplete is used to model the sender-side OPRF evaluation *only* if it is executed on the keys held by S, hence (1) the revised $\mathcal{F}_{\mathsf{OPRF}}$ does not keep counters $\mathsf{tx}(sid, i)$ for adversarial functions $F_{sid,i}$, $i \neq \mathsf{S}$,[17] and (2) neither the adversary nor any "virtual corrupt server" need to send a SndrComplete message to represent the server-side execution of the OPRF protocol on adversarially-chosen keys. (See also item 4 below.)

(3) Command (RcvComplete, $sid, ssid, i$), issued by $\mathcal{A}$, models the adversary letting server-side OPRF messages pass to the party P who runs the user-side OPRF protocol, in which case P outputs $F_{sid,i}(x)$ for the argument x which P entered via command (Eval, $sid, ssid, \mathsf{S}', x$). The adversary can make P output $F_{sid,i}$ for any $i \neq \mathsf{S}$ because in the real-world a network adversary interacting with P can run the server-side OPRF protocol using arbitrary (O)PRF keys. Adversary $\mathcal{A}$ can also set $i = \mathsf{S}$, which represents letting the real-world P receive the server-side OPRF messages executed on S's keys, but then the protocol succeeds only if $\mathsf{tx} \geq 0$, i.e. if there is an "unclaimed" S's session which $\mathcal{A}$ can pair with P's session. In this case P outputs $F_{sid,\mathsf{S}}(x)$ and the functionality decrements counter tx.

Other Syntactic Differences from UC OPRF of [33] Other syntactic differences between our revised OPRF functionality of Fig. 3 and the proceedings-version OPRF functionality of [33] include the following:

(4) The revised $\mathcal{F}_{\mathsf{OPRF}}$ keeps a counter only for the "honest" function F_{S}, whereas the proceedings-version $\mathcal{F}_{\mathsf{OPRF}}$ kept it for each adversarial function. We eliminate the counters for adversarially-controlled functions from the revised model because they appear not to play a meaningful role in the

[17] In the proceedings-version functionality these counters were indexed by "virtual corrupt server" identities denoted "S" therein.

proceedings-version OPRF functionality since the functionality allows the adversary to increment such counters at will via command SndrComplete.

(5) The revised $\mathcal{F}_{\mathsf{OPRF}}$ models offline computation of any adversarial function $F_{sid,i}$, for $i \neq \mathsf{S}$, via call OfflineEval, whereas the proceedings-version $\mathcal{F}_{\mathsf{OPRF}}$ modeled it using a sequence of adversarial calls Eval, SndrComplete, RcvComplete, i.e. as an instance of on-line protocol running "in the head".[18]

(6) The revised functionality $\mathcal{F}_{\mathsf{OPRF}}$ splits the initialization of random function $F_{sid,\mathsf{S}}$ via command Init from offline evaluation of $F_{sid,\mathsf{S}}$ on some argument x via command $(\text{OfflineEval}, sid, i, x)$ for $i = \mathsf{S}$, whereas the proceedings-version $\mathcal{F}_{\mathsf{OPRF}}$ combined initialization with one offline evaluation call.

B The DH-OPRF Protocol Realizing Revised $\mathcal{F}_{\mathsf{OPRF}}$

Fig. 13 shows the DH-OPRF protocol of [31], called 2HashDH therein, syntactically modified to realize the *adaptive* OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ defined in Fig. 3 in Section 3. Recall that the $\mathcal{F}_{\mathsf{OPRF}}$ functionality we show in Section 3 is a revision of the (static) OPRF functionality defined in [31], but it is also a revision of the earlier version of the adaptive OPRF functionality which appeared in the conference version of this paper [33]. The protocol shown below is essentially the same as in [31] and requires the same One-More Diffie-Hellman assumption [5, 31] for security. The only differences between 2HashDH in Fig. 13 and in [31] are syntactic: First, we eliminate the user's "input-output caching" mechanism used in [31]. Second, the protocol in Fig. 13 outputs the OPRF protocol prefix, namely the U-to-S message a, to both U and S instances. As explained in Section 3 outputting these protocol transcript prefixes provides a better "glue" which a higher-level protocol can use to compose OPRF with some other protocol, as protocol OPRF-AKE of Section 5 does by composing OPRF with AKE.

Modifications in the Proof of [31]. The proof of Lemma 1 is very similar to the proof of security given in [31], so we only briefly discuss how our modifications to $\mathcal{F}_{\mathsf{OPRF}}$ influence the security proof. The ideal-world adversary, i.e., simulator SIM, is shown in Fig. 14. Fig. 14 denotes functionality $\mathcal{F}_{\mathsf{OPRF}}$ as $\mathcal{F}$ for brevity, and it makes the following notational assumptions: (1) $\mathcal{F}$'s initialization message $(\text{Init}, \mathsf{S}, sid)$ fixes the identifier S and session ID sid for the rest of the simulation, and all messages to and from $\mathcal{F}$ and all its internal records are implicitly tagged with sid; (2) If S is corrupted then SIM acts as if S was compromised from the very beginning; (3) The identifier S of the server for which $\mathcal{F}$ sends the initialization message $(\text{Init}, \mathsf{S}, sid)$ is encoded as a different binary string than any integer value; (4) There is integer N s.t. the number of hash function H'

[18] For example, the simulator shown in [31] reacts to the real-world adversary's local computation of hash function $H_2(x, v)$, $v \neq H_1(x)^k$ where k is the key of server S, with three messages to the functionality: $(\text{Eval}, sid, \mathsf{S}', x)$ for arbitrary S', $(\text{SndrComplete}, sid, p)$ for a unique index p associated with an adversarial "public key" y_p s.t. $(g, y_p, H_1(x), v)$ is a DDH tuple, and $(\text{RcvComplete}, sid, p)$.

Components: Hash functions $H(\cdot,\cdot)$, $H'(\cdot)$ with ranges $\{0,1\}^\ell$ and G, respectively. Functions H, H' are specific to the OPRF instance initialized for a *unique* session ID *sid*, and they should be implemented by folding *sid* into their inputs.

Initialization: On input (INIT, *sid*), S picks $k \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and stores (sid, k).

Server Compromise: On (COMPROMISE, *sid*, S) from the adversary, reveal key k.

Offline Evaluation

- On input (OFFLINEEVAL, *sid*, S, x) for *sid* matching record (sid, k), S outputs (OFFLINEEVAL, *sid*, y) for $y = H(x, (H'(x))^k)$.

Online Evaluation

- On input (EVAL, *sid*, *ssid*, S', x), U picks $r \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, records $(sid, ssid, r)$, sends $(ssid, a)$ for $a = H'(x)^r$ to S', and outputs (Prefix, *ssid*, a).
- On input (SNDRCOMPLETE, *sid*, *ssid*′) and message $(ssid, a)$ from U s.t. $a \in \mathbb{G}$, S retrieves pair (sid, k) with matching *sid*, aborts if such pair is not found, else sends $(ssid, b)$ for $b = a^k$ to U and outputs (Prefix, *ssid*′, a).
- On message $(ssid, b)$ s.t. $b \in \mathbb{G}$, U retrieves tuple $(sid, ssid, r)$, aborts if tuple not found, else outputs (EVAL, *sid*, *ssid*, y) for $y = H(x, b^{1/r})$.

Fig. 13: Adaptive OPRF Protocol DH-OPRF

queries made by the real-world adversary $\mathcal{A}$ is upper-bounded by $N/2$, and the number of online OPRF evaluation sub-sessions started by $\mathcal{Z}$ via command EVAL to some honest user U is also upper-bounded by $N/2$.

Lemma 1. *The DH-OPRF protocol shown in Fig. 13 realizes functionality $\mathcal{F}_{\mathsf{OPRF}}$ of Fig. 3 under the One-More Diffie Hellman assumption in ROM.*

Using these assumptions, the simulator acts in a similar way as the one shown in [31]: SIM picks a random key k as S does, and uses it by computing $b = a^k$ for every incoming message $a \in \mathbb{G}$ in SNDRCOMPLETE. SIM embeds a discrete-log trapdoor in every H' output, setting $H'(x) := g^r$ for random r, and recording this choice as $\langle H', x, r\rangle$. SIM similarly embeds a discrete-log trapdoor in OPRF messages a sent on behalf of any honest U session $(sid, ssid, \mathsf{U})$, by setting $a \leftarrow g^r$ for random r, and recording this choice as $\langle ssid, \mathsf{U}, r\rangle$. SIM also keeps track of all Random Function indexes which are evaluted by adversary $\mathcal{A}$ either offline, through H queries, or online, through $\mathcal{A}$'s responses b to user U's message a. Each function is equated with its "public key" $z = g^k$. First, SIM records the honest S's function this way as $\langle F, 0, k, z\rangle$ for $z = g^k$, identifying this function with index "0". Secondly, every time $\mathcal{A}$ queries H on new point (x, u), SIM checks if there is record $\langle F, i, \cdot, z'\rangle$ and $\langle H', x, r\rangle$ s.t. $z' = u^{1/r}$, because this is equivalent to $\mathsf{DL}(H'(x), u) = \mathsf{DL}(g, z')$. If this holds for $i \neq 0$ then $\mathcal{A}$ offline evaluates some adversarial function of its choice, hence in that

case SIM sends $(\textsc{OfflineEval}, sid, i, x)$ to $\mathcal{F}$ and embeds value $F_{sid,i}(x)$ returned by $\mathcal{F}$ into $H(x,u)$. If this holds for $i=0$ and S is not compromised then $\mathcal{A}$ must be completing some OPRF instance as the user, hence in that case SIM sends $(\textsc{Eval}, sid, ssid', \perp, x)$ and $(\textsc{RcvComplete}, sid, ssid', \mathsf{SIM}, \mathsf{S})$ to $\mathcal{F}$ for some fresh $ssid'$ value. If $\mathcal{F}$ does not return any answer this means that $\mathcal{F}$ ticket counter is 0, and that this local computation of $F_{sid,\mathsf{S}}(x)$ by $\mathcal{A}$ violated the security properties of $\mathcal{F}_{\mathsf{OPRF}}$, in which case SIM halts, and the simulation obviously diverges from the real world execution. Otherwise SIM embeds value $F_{sid,\mathsf{S}}(x)$ returned by $\mathcal{F}$ into $H(x,u)$.

1. Pick $r_1, \ldots, r_N \leftarrow_{\mathrm{R}} \mathbb{Z}_q$. Set $g_1 := g^{r_1}, \ldots, g_N := g^{r_N}$, and $J := 1$ and $I := 1$.
2. On $(\textsc{Init}, \mathsf{S}, sid)$ from $\mathcal{F}$ pick $k \leftarrow \mathbb{Z}_q$ and record $\langle F, 0, k, z = g^k \rangle$.
3. On $(\textsc{Compromise}, sid)$ from $\mathcal{A}$, declare S as COMPROMISED, retrieve tuple $\langle F, 0, k, z \rangle$, send $(\textsc{Compromise}, sid)$ to $\mathcal{F}$, and send (sid, k) to $\mathcal{A}$.
4. On $\mathcal{A}$'s fresh query x to H', set $H'(x) \leftarrow g_J$, record $\langle H', x, r_J \rangle$, and set J++.
5. On $(\textsc{Eval}, sid, ssid, \mathsf{U}, \mathsf{S}')$ from $\mathcal{F}$, set $a \leftarrow g_J$, respond with $\mathsf{prfx} = a$ to $\mathcal{F}$, send $(sid, ssid, a)$ to $\mathcal{A}$ as U's message to S', record $\langle ssid, \mathsf{U}, r_J \rangle$, and set J++.
6. On $(\textsc{SndrComplete}, sid, \mathsf{S})$ from $\mathcal{F}$ and message $(sid, ssid, a)$ (where $a \in \mathbb{G}$) from $\mathcal{A}$ sent on behalf of some user to server S, respond with $\mathsf{prfx} = a$ to $\mathcal{F}$, retrieve $\langle F, 0, k, z \rangle$, and send $(sid, ssid, b)$ for $b = a^k$ to $\mathcal{A}$ as S's response.
7. On message $(sid, ssid, b)$ (where $b \in \mathbb{G}$) from $\mathcal{A}$ sent on behalf of some server to user U, retrieve records $\langle ssid, \mathsf{U}, r \rangle$ and $\langle F, i, \cdot, z' \rangle$ for $z' = b^{1/r}$.
   If there is no record $\langle F, i, \cdot, z' \rangle$, set $i := I$, record $\langle F, i, \perp, z' \rangle$, and set I++.
   In either case send $(\textsc{RcvComplete}, sid, ssid, \mathsf{U}, i)$ to $\mathcal{F}$.
8. On $\mathcal{A}$'s fresh query (x,u) to H, retrieve record $\langle H', x, r \rangle$. If there is no such record, then pick $H(x,u) \leftarrow_{\mathrm{R}} \{0,1\}^\ell$. Otherwise do the following:
   (1) If record $\langle F, 0, k, z \rangle$ satisfies that $z = u^{1/r}$ then
      i. If S is COMPROMISED, send $(\textsc{OfflineEval}, sid, \mathsf{S}, x)$ to $\mathcal{F}$, and on $\mathcal{F}$'s response $(\textsc{OfflineEval}, sid, y)$ set $H(x,u) := y$.
      ii. If S is *not* COMPROMISED, pick a fresh identifier $ssid^*$ and send $(\textsc{Eval}, sid, ssid^*, \perp, x)$ and $(\textsc{RcvComplete}, sid, ssid^*, \mathsf{SIM}, \mathsf{S})$ to $\mathcal{F}$.
      If $\mathcal{F}$ ignores the last message then output HALT and abort.
      Else on $\mathcal{F}$'s response $(\textsc{Eval}, sid, ssid^*, y)$ set $H(x,u) := y$.
   (2) Else, if there is tuple $\langle F, i, \perp, u^{1/r} \rangle$ for $i \neq 0$ then send $(\textsc{OfflineEval}, sid, i, x)$ to $\mathcal{F}$, and on $\mathcal{F}$'s response $(\textsc{OfflineEval}, sid, y)$ set $H(x,u) := y$.
   (3) Else, record $\langle F, i, \perp, u^{1/r} \rangle$ for $i = I$, send $(\textsc{OfflineEval}, sid, i, x)$ to $\mathcal{F}$, and on $\mathcal{F}$'s response $(\textsc{OfflineEval}, sid, y)$ set $H(x,u) := y$ and set I++.

Fig. 14: Simulator SIM for Protocol DH-OPRF (with $\mathcal{F}_{\mathsf{OPRF}}$ denoted as $\mathcal{F}$)

Finally, if (x,u) query to H' does not match any recorded function $\langle F, i, \cdot, z' \rangle$ s.t. $z' = u^{1/r}$, then SIM defines a new function $\langle F, i', \perp, z' \rangle$ for fresh index i' and $z' = u^{1/r}$. SIM interprets $\mathcal{A}$'s OPRF responses b to messages $a = g^r$ which SIM sends on behalf of some honest user U in a similar way: Note that if $a = g^r$ then $z' = b^{1/r} = g^{\mathsf{DL}(a,b)}$). Therefore SIM can identify the function computed by

U on this OPRF interaction with the public key $z' = b^{1/r}$. As in the case of responding to H queries, SIM first checks if there exists record $\langle F, i, \cdot, z' \rangle$, and otherwise it creates a new record $\langle F, i', \perp, z' \rangle$ for fresh i'.

The only non-syntactic changes in the argument that under OMDH assumption this simulation presents the same view as in the real execution, compared to the simimlar argument given in [31], is that (1) $\mathcal{A}$ may at any time compromise server S for a specific *sid* and learn key k; and (2) that if some user session $(sid, ssid_u, \mathsf{U})$ and some server session $(sid, ssid_s, \mathsf{S})$ output the same prefixes $\mathsf{prfx} = a$, then this interaction does not increase the ticket counter, and does not count to the pool of OPRF interactions which $\mathcal{A}$ can use to compute $F_{sid,\mathsf{S}}(x)$ on some input x.

Regarding (1), note that after server compromise $\mathcal{A}$ can compute the server's function on any argument, but SIM can detect that by catching H query on (x, v) for $v = (H'(x))^k$, and can simulate this by sending $(\textsc{OfflineEval}, sid, \mathsf{S}, x)$ to $\mathcal{F}_{\mathsf{OPRF}}$. Furthermore, note that event HALT may only occur if server S is not marked COMPROMISED at that time; hence the argument which upper-bounds Pr[HALT] given in [31] is not affected by this change because it assumes that S is not compromised at the time. Regarding (2), it is easy to see that if $\mathcal{A}$ who forwards to some server session the message $a = g^r$ sent by SIM on behalf of some honest user U session, the security reduction can respond to such message with $b = z^r$, where z is the OMDH *challenge* public key $z = g^k$. Therefore the reduction will not need to query the One-More Diffie-Hellman oracle $(\cdot)^k$ on all such S sessions, and thus these sessions will not increase the number of arguments x on which adversary $\mathcal{A}$ can compute $u = (H'(x))^k$, by querying H on (x, u), without breaking the OMDH assumption.