How to Tolerate Typos in Strong Asymmetric PAKE

Ian McQuoid(✉), Mike Rosulek, and Jiayu Xu

Oregon State University, Corvallis, USA
{mcquoidi,rosulekm,xujiay}@oregonstate.edu

Abstract. Strong asymmetric password-authenticated key exchange (saPAKE) is the gold standard for password-based authentication. When authenticating using saPAKE, the client holds a cleartext password, and the server holds only a "digest" of the password. The two parties obtain a shared session key if and only if the client password matches the password encoded in the digest.

In this work we initiate the study of **strong asymmetric fuzzy PAKE (safPAKE)**, which allows the client and server to obtain a shared session key if the client's password is "close enough" to the password encoded in the digest, according to some policy. safPAKE can be used to tolerate incidental password typos in the PAKE setting, which is becoming a standard industry practice outside the PAKE setting. Our safPAKE functionality supports any "typo policy", and our protocol is practical when there are a small number of permissible mistypings of a password.

1 Introduction

Password-based authentication is delicate because passwords are low entropy. Cryptographic mechanisms that are safe when using high-entropy secrets can be unsuitable for passwords, allowing an attacker to *locally* check whether a password guess is correct after the fact. Such a mechanism will expose typical passwords after mounting an offline dictionary attack. Password-authenticated key exchange (PAKE) [7] was proposed as a cryptographic mechanism disallowing such offline attacks.

PAKE is an improvement over the password-over-TLS approach currently used on the Internet, in which a client sends (an encryption of) its password over a TLS channel, and the server, upon decryption, checks raw passwords against a password file stored in advance. The security of password-over-TLS critically relies on the assumption that the server's public key is correctly distributed to the client; however, numerous instances of "PKI failure" in recent years[1] clearly show that a password protocol without PKI—the setting of PAKE—is preferable.

Second author partially supported by NSF award 2150726.

[1] Amir Herzberg has previously provided a list of famous PKI failures [17].

Password-Authenticated Key Exchange (PAKE). PAKE is a family of protocols that allow two parties to agree on a common session key if and only if they have matching passwords. PAKE protocols are carefully designed to avoid exposing offline dictionary attacks as the shared strings (the passwords) are assumed to be low-entropy and therefore the space of secret stings is efficiently enumerable. Instead, they achieve the best possible security: the only way for an adversary to check a password guess is by participating in an interactive protocol session.

There are two main flavors of PAKE:

– **Symmetric PAKE** [4,6,7,9,11,23–26] requires both parties to hold the passwords in cleartext.
– **Asymmetric PAKE** [7,15,16,29,30] is more suitable for a client-server setting, where the server holds only a one-way digest of the password.

Asymmetric PAKE has the additional property that, even in the setting where the server's password file might be compromised, an adversary must perform an (offline) dictionary attack to recover any passwords. In other words, the server cannot store its passwords in the clear, and furthermore never sees the raw passwords *throughout the execution of the protocol*, which is another advantage over the password-over-TLS approach. The gold standard security definition for asymmetric PAKE is **strong asymmetric PAKE (saPAKE)** [10,21] where, in addition to the asymmetric PAKE security requirements, computation for the offline dictionary attack must be performed *after* compromising the server. This distinction is analogous to the server holding a digest $H(pw)$ for a public hash function H vs the server holding a digest $H(s; pw)$ for some uniform and secret "salt" s. In the first case (aPAKE), all possible digests can be computed before server compromise at which point the adversary immediately learns the correct password; while in the second case (saPAKE), the adversary must wait until server compromise before launching an offline attack.

The original security notion for PAKE is game-based [6]. In recent years, the standard has switched to the stronger notion in the Universal Composability (UC) framework [11,15,21], which supports arbitrary composition of the protocol (with itself or with other protocols). Variants of the original UC PAKE notion have been proposed, including relaxations of the UC functionality [1] and combining the security notion with idealized models [2].

Password Typos. Human-typed passwords are prone to typos, which prevent successful authentication. Some online services (in which the server sees the login password in the clear) are known to correct for some password typos. For example, Facebook accepts passwords with inverted upper/lowercase (e.g., erroneous Caps Lock) or an erroneously capitalized first letter (possible in many input methods on mobile phones) [28].

Chatterjee *et al.* [12] were the first to formally study the security implications of correcting minor typos—*i.e.*, allowing users to authenticate even if they make certain incidental typos. In their study of real-world login attempts to Dropbox, they found that 20% of users that experienced a failed login attempt would have saved at least 1 min of time if simple typo-correction had been enabled. They found that 3% of users that experienced a failed login attempt were never able to successfully log in, even though they used a password that would have been accepted under a simple typo-correction scheme.

Tolerating Typos in PAKE. It is relatively trivial to tolerate typos when the server sees the user's password attempt in the clear—*i.e.*, when authenticating with password-over-TLS. It is a challenge to tolerate typos in the PAKE setting, when neither party sees the other's password attempt. Previous constructions of fuzzy PAKE variants have achieved varying levels of security. Dupont et al. [13] constructed two fuzzy-input tolerant symmetric PAKE protocols, one from garbled circuits and oblivious transfer which can handle "closeness" for any efficiently computable circuit (in the language of our construction, any efficiently computable similarity predicate) and one from PAKE which can handle fuzzy inputs in a small hamming ball. The former is a feasibility result that is inefficient, whereas the latter was later found to be insecure [8]; [8] also presented another fuzzy symmetric PAKE protocol.

Moving the result for fuzzy PAKE to cover asymmetric security, Erwig et al. [14] constructed two fuzzy-input tolerant asymmetric PAKE protocols, one from error correcting codes and oblivious transfer which can handle fuzzy inputs in a hamming ball and the second which compiled aPAKE into a fuzzy protocol which achieves a weaker functionality but can handle fixed size closeness notions. Namely, the functionality has been weakened to allow for a fixed number of unstructured guesses in an online attack. Pongmorrakot and Chatterjee [27] similarly construct a fuzzy PAKE protocol which has security guarantees in the face of server compromise. However, their result similarly allows the server to make a fixed number of unstructured guesses by design, and allows the adversary to impersonate the user on compromise of the server—something which is not standard for aPAKE protocols.

Crucially, none of the results above achieves *strong* asymmetric security; even the fuzzy aPAKE protocols allow the adversary to perform some computation before server compromise, and recover the password immediately once the server is compromised. This is the starting point of our work:

> *Can we construct a typo-tolerant strong aPAKE protocol resilient to pre-computation attacks, preferably with reasonable efficiency?*

Naïve Approach. Let Σ_{pw} denote the set of allowable typos of the correct password pw. The server would like to authenticate any user who enters a password pw' from Σ_{pw}. The standard PAKE setting corresponds to the special case $\Sigma_{pw} = \{pw\}$.

A natural way to tolerate typos in saPAKE is to run t independent instances of saPAKE, where $t = |\Sigma_{pw}|$. In more detail: the server holds a password digest for each $\widetilde{pw} \in \Sigma_{pw}$. The client holds a password attempt pw' and tries to authenticate t times—once against each of these digests.

This naïve approach to typo tolerance has two main drawbacks:

– Consider what would be the best possible security for a single authentication attempt of typo-tolerant saPAKE: A corrupt client should learn whether

$pw' \in \Sigma_{pw}$, for *only one* guess pw'. A corrupt server should learn whether $pw' \in \Sigma_{pw}$ for *only one* guess pw.
The naïve protocol leaks more than this. Namely, a corrupt client can learn whether $pw'_i \in \Sigma_{pw}$ for t different and *arbitrary* guesses pw'_i. A corrupt server can learn whether $pw' \in S$ for a *completely unstructured* set S of cardinality t (rather than a "typo ball" Σ_{pw} around a single center pw). In other words, the naïve protocol makes online dictionary attacks t times faster (in terms of number of authentication attempts), for either party.
Fixing the issue for a corrupt client is relatively straight-forward. But it is non-trivial to prevent a corrupt server from making t *unstructured* guesses per authentication attempt.
- All costs of the protocol scale with the number t of allowed typos. For most password-typo policies (described above), t is indeed a very small constant, so this overhead is not prohibitive. But it is nontrivial to make any aspect of the protocol's complexity scale sublinearly with t.

1.1 Our Results

We show how to resolve both of the problems described above by giving an efficient protocol for **strong asymmetric fuzzy PAKE (safPAKE)**. Our protocol achieves best-possible security (with respect to password guesses) in the Universal Composability (UC) framework, with communication independent of t for a wide definition of "closeness" which includes a common subclass of the similarity predicates considered by Erwig et al. [14]. Our construction can be viewed as an extension of the saPAKE protocol by Jarecki, Krawczyk, and Xu [21], which is a compiler from any UC oblivious PRF (OPRF) and any UC aPAKE to UC saPAKE[2]; the main difference is that we add a one-round Private Information Retrieval (PIR) protocol between the OPRF and aPAKE stages. The saPAKE protocol in [21] can be viewed as a special case of our protocol with the equality check function for a similarity predicate and the usual PAKE leakage functions.

Additionally, our protocol up to the aPAKE stage can be seen independently as an input-normalization protocol where a client inputs a string x' and returns a "corrected" version of their string x. This normalization achieves the flavor of security needed for saPAKE—namely, resistance to online dictionary attacks and to precomputation for offline dictionary attacks. We believe that this conceptual contribution may find independent applications elsewhere.

A final contribution is to modify the 2-hash-Diffie-Hellman (2HDH) OPRF protocol of [20,21] to support a kind of key-extraction property that is necessary for our safPAKE protocol. As our OPRF functionality is key-aware and our OPRF protocol allows for key extraction, we believe that our safPAKE protocol is amenable to an interactive registration phase requiring only a single message

[2] The provided security proof in [21] has not been updated to their new saPAKE security model.

and without the honest server receiving the client's clear-text password. As a comparison, an interactive registration for OPAQUE [21] would require at least three messages as the OPRF would need to be evaluated in an online step. This is because OPAQUE's OPRF functionality does not have a notion of transferrable OPRF keys[3]. We leave modeling interactive registration in aPAKE, and its extensions, as future work.

2 Preliminaries

2.1 (Keyword) PIR

Our safPAKE protocol requires a (single-server) keyword PIR protocol. In keyword PIR, the server holds a key-value mapping $\mathrm{DB} : u_i \mapsto v_i$ and the client holds a key u. The outcome of the protocol is that the client learns (at least) $v := \mathrm{DB}[u]$, if it exists, while the server learns nothing about u.

We write our safPAKE protocol in terms of a 1-round PIR protocol for the sake of simplicity, although our protocol can be easily extended to use a multi-round PIR. Formally, a (1-round) keyword PIR protocol consists of the following algorithms:

- $\mathsf{PIR.Request}(u) \rightarrow (s, \mathrm{req})$
- $\mathsf{PIR.Respond}(\mathrm{req}, \mathrm{DB}) \rightarrow \mathrm{resp}$
- $\mathsf{PIR.Finish}(s, \mathrm{resp}) \rightarrow v$

It must satisfy the following properties:

- **Correctness:** For all key-value mappings DB, and for all keys u appearing in DB, the following returns true with overwhelming probability:

$$\boxed{\begin{array}{l} (s, \mathrm{req}) \leftarrow \mathsf{PIR.Request}(u) \\ \mathrm{resp} := \mathsf{PIR.Respond}(\mathrm{req}, \mathrm{DB}) \\ \mathrm{return}\ \mathrm{DB}[u] = \mathsf{PIR.Finish}(\mathrm{resp}, s) \end{array}}$$

  There is no correctness required when u is not a key appearing in DB.
- **Client Privacy:** For all u, u', the following distributions are indistinguishable:

$$\boxed{\begin{array}{l} (s, \mathrm{req}) \leftarrow \mathsf{PIR.Request}(u) \\ \mathrm{return\ req} \end{array}} \approx \boxed{\begin{array}{l} (s', \mathrm{req}) \leftarrow \mathsf{PIR.Request}(u') \\ \mathrm{return\ req} \end{array}}$$

[3] However, we provide evidence that an OPAQUE analog leveraging our OPRF functionality is secure and, therefore, OPAQUE would likely enjoy a single-message interactive registration phase.

Our protocol requires a **deterministic** $\mathsf{PIR.Respond}$ algorithm, which is standard since $\mathsf{PIR.Respond}$ is involved only in correctness, but not privacy.

Candidate Instantiations. In **trivial PIR**, req is empty and $\mathsf{PIR.Respond} := \mathrm{DB}$.

A classic keyword PIR protocol has req $:= \mathsf{Enc}(s, u)$, for a symmetric-key fully homomorphic encryption scheme. The server interpolates a polynomial P such that $P(u) = \mathrm{DB}[u]$ for all $u \in \mathrm{DB}$. It then homomorphically evaluates the polynomial on req to obtain an encryption of $P(u) = \mathrm{DB}[u]$.

2.2 Authenticated Encryption

Let $\mathcal{E} = (\mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme, with key space $\mathcal{K}$, and where plaintexts of length n are encrypted to ciphertexts in the set $\mathcal{C}(n)$.

Definition 1. *We say that $\mathcal{E}$ satisfies **authenticated encryption (AE\$) security** if the following two stateful oracles are computationally indistinguishable:*

```
$k \leftarrow \mathcal{K}$
$\mathtt{enc}(m \in \mathcal{M})$:
    return $\mathsf{Enc}(k, m)$
$\mathtt{dec}(c \in \mathcal{C})$:
    return $\mathsf{Dec}(k, c)$
```

$\approx$

```
// $\perp$ is the default value for undefined keys
$\mathtt{ptxts}[] := \perp$
$\mathtt{enc}(m \in \mathcal{M})$:
    $c \leftarrow \mathcal{C}(|m|)$
    $\mathtt{ptxts}[c] := m$
    return $c$
$\mathtt{dec}(c \in \mathcal{C})$:
    return $\mathtt{ptxts}[c]$
```

3 Oblivious PRF

Our protocol requires an Oblivious PRF (OPRF), but prior OPRF ideal functionalities [19–21] from related work are not suitable for our purposes. In our protocol, when an honest client authenticates using a valid password, she learns the server's OPRF key. This key gives her the ability to evaluate the OPRF without the server's help, which is necessary for some consistency checks in the protocol.

Existing OPRF functionalities do not explicitly have the notion of a "key" as a string that can be transferred or encrypted. In the OPAQUE protocol for instance, only an adversary can learn the server's OPRF key—when it compromises the server storage. This is modeled in the protocol by the adversary sending an explicit "compromise" command to the OPRF functionality, which grants the adversary the ability to evaluate the OPRF.

In this section, we describe an ideal OPRF functionality and corresponding protocol that supports transferrable PRF keys. The protocol is based on the 2-hash-Diffie-Hellman (2HDH) OPRF protocol of [20]. The main challenge in our setting is identifying which key a corrupt server is using in a specific OPRF interaction. This is a challenge because the key is a Diffie-Hellman exponent and thus difficult for the simulator to extract. We accomplish this by strategically adding another random oracle hash, resulting in what we call the **3-hash-Diffie-Hellman (3HDH)** OPRF.

3.1 The Functionality

The OPRF functionality is presented in Fig. 2. It mostly follows the OPRF functionality used in [21]. The crucial difference is that *our functionality has an explicit notion of server's PRF key*, resulting in the following concrete modifications:

- Upon initialization, the functionality generates a random string k as the key for the server.
- As mentioned above, there is no command to mark a server instance as "compromised" unlocking the adversary's ability to make offline PRF evaluation queries. Instead, anyone can attempt an offline evaluation by providing a candidate key. The correct key will give outputs consistent with the honest server, whereas each other key will correspond to an independent random function.
- The explicit PRF key can be used to index random functions. Since a man-in-the-middle adversary can replace the honest server's key with one of its own choosing, there might be multiple independent random functions, and the functionality needs to keep track of all of them. In the OPRF of [21], the simulator cannot extract those keys, but it can detect when the same key is used twice; therefore, the functionality maintains these random functions by letting the simulator supply the "index" of the chosen function.

  In our OPRF protocol, the simulator can often extract the correct key. In these cases, the index of the random function can be replaced by the actual key. However, there remain adversarial behaviors that result in a consistent random function (which the simulator can identify), but which are not consistent with any key. Although the simulator cannot extract, it turns out that neither can the adversary! More specifically, even the adversary cannot produce any string k that explains their behavior in terms of the honest PRF function. Hence, these random functions are *non-transferrable*, and we model them using the "index" approach of [21]—that is, the simulator can detect when the same index is being used and inform the functionality. functionality represents transferrable keys (*i.e.*, actual keys) as $(k, 1)$ and non-transferrable keys (*i.e.*, indices) as $(k, 0)$. Honest parties can make offline PRF queries *only* on a transferrable key, which models the fact that there is no key that can make the honest PRF function agree with this random function.

3.2 The 3-Hash Diffie-Hellman Protocol

The 2-hash Diffie-Hellman (2HDH) OPRF protocol [20] is an oblivious protocol for evaluating the following PRF:

$$F(k,x) = H_2(x, H_1(x)^k)$$

Because k is used as an exponent, it is not feasible for the simulator to extract k. The protocol is proven secure when H_1, H_2 are random oracles, and the range of H_1 is a group in which the One-More Gap Diffie-Hellman (OMGDH) assumption [5,20] holds.

Our modification is to simply apply a random oracle to k before running 2HDH. The result is an oblivious protocol for evaluating the PRF:

$$F(k,x) = H_2(x, H_1(x)^{H_3(k)})$$

The rest of the protocol is unchanged. We call the resulting protocol **3-hash Diffie-Hellman (3HDH)** for obvious reasons. The formal description is in Fig. 1.

Parameters:

- output length ℓ
- group $(\mathbb{G}, g, p)$
- hash functions $H_1 : \widetilde{D} \to \mathbb{G}$, $H_2 : \widetilde{D} \times \mathbb{G} \to \{0,1\}^\ell$, and $H_3 : \{0,1\}^\kappa \to \mathbb{Z}_p$ (modeled as random oracles)

Stealing Password Data

1. Upon receiving $(\mathsf{Init}, \mathrm{sid})$, S samples $k \leftarrow \{0,1\}^\kappa$ and records (sid, k).

Online Evaluation

1. Upon receiving $(\mathsf{PRFEval}, \mathrm{sid}, \mathrm{ssid}, \mathsf{S}, x)$, C:
   1. Samples $r \leftarrow \mathbb{Z}_p$ and records $(\mathrm{sid}, \mathrm{ssid}, r)$.
   2. Computes $a := H_1(x)^r$.
   3. Sends (ssid, a) to S and outputs $(\mathsf{Prefix}, \mathrm{sid}, \mathrm{ssid}, a)$.
2. Upon receiving $(\mathrm{ssid}, a \in \mathbb{G})$ from C and $(\mathsf{PRFSendComplete}, \mathrm{sid}, \mathrm{ssid})$, S:
   1. Retrieves record (sid, k) and computes $b := a^{H_3(k)}$.
   2. Sends (ssid, b) to C.
3. Upon receiving $(\mathrm{ssid}, b \in \mathbb{G})$ from S, C:
   1. Retrieves record $(\mathrm{sid}, \mathrm{ssid}, r)$ and computes $y := H_2(x, b^{1/r})$.
   2. Outputs $(\mathsf{PRFReceiveComplete}, \mathrm{sid}, \mathrm{ssid}, y)$.

Fig. 1. Protocol 3HDH

Security. Our 3HDH protocol (Fig. 1) realizes the OPRF functionality $\mathcal{F}_{\mathrm{OPRF}}$ (Fig. 2) under the $(N = Q_H + Q, Q)$-OMGDH assumption, where Q_H is the number of the adversary $\mathcal{A}$'s H_1 queries, and Q is the number of $\mathsf{PRFEval}$ messages issued by the environment $\mathcal{Z}$. We assume w.l.o.g. that $\mathcal{A}$ never repeats a random oracle query. Below we describe the simulator.

1. Sample $r_1, \ldots, r_N \leftarrow \mathbb{Z}_p$, and compute $g_1 := g^{r_1}, \ldots, g_N := g^{r_N}$. Initialize indices $J := 1$ (index of the current random group element g_J to be used) and $I := 1$ (the current function index).
2. On $\mathcal{A}$'s query $H_1(x)$,[4] set $\texttt{DL}[x] := r_J$, $H_1(x) := g_J$, and $J := J + 1$.
3. On $(\mathsf{PRFEval}, \text{sid}, \text{ssid}, \mathsf{S})$ from $\mathcal{F}_{\text{OPRF}}$, record $\texttt{ClientSession}[\text{ssid}] := (r_J, a := g_J)$ and send $(\text{sid}, \text{ssid}, a)$ from C to S and $(\text{sid}, \text{ssid}, a)$ to $\mathcal{F}_{\text{OPRF}}$. Set $J := J + 1$.
4. On $(\mathsf{PRFSendComplete}, \text{sid}, \text{ssid}, \mathsf{S})$ from $\mathcal{F}_{\text{OPRF}}$, and $(\text{sid}, \text{ssid}, a^*)$ from $\mathcal{A}$ sent to S, send $(\text{sid}, \text{ssid}, a^*)$ to $\mathcal{F}_{\text{OPRF}}$, and check if $\texttt{ServerKey}$ is defined. If not, sample $h \leftarrow \mathbb{Z}_p$ and record $\texttt{ServerKey} := (h, g^h)$. Either way, set $b := (a^*)^h$, and send $(\text{sid}, \text{ssid}, b)$ from C to S.
5. On $(\text{sid}, \text{ssid}, b^*)$ from $\mathcal{A}$ sent to C, retrieve $(r, a) := \texttt{ClientSession}[\text{ssid}]$ and compute $z := (b^*)^{1/r}$.
   A. If there exists a previous $H_3(k^*)$ query with $b^* = a^{H_3(k^*)}$, send $(\mathsf{PRFReceiveComplete}, \text{sid}, \text{ssid}, \mathsf{C}, (k^*, 1))$ to $\mathcal{F}_{\text{OPRF}}$.
   B. Otherwise, if $\texttt{ServerKey}$ is defined, send $(\mathsf{PRFReceiveComplete}, \text{sid}, \text{ssid}, \mathsf{C}, \mathsf{S})$ to $\mathcal{F}_{\text{OPRF}}$.
   C. Otherwise, if there exists a record $z = \texttt{index}[i]$, send $(\mathsf{PRFReceiveComplete}, \text{sid}, \text{ssid}, \mathsf{C}, (i, 0))$ to $\mathcal{F}_{\text{OPRF}}$.
   D. Otherwise, set $\texttt{index}[I] := z$ and send $(\mathsf{PRFReceiveComplete}, \text{sid}, \text{ssid}, \mathsf{C}, (I, 0))$ to $\mathcal{F}_{\text{OPRF}}$. Set $I := I + 1$.
6. On $\mathcal{A}$'s query $H_2(x, u)$, if $\texttt{DL}[x] = r$, set $H_2(x, u) := y$ where y is defined as follows:
   A. If there exists a previous $H_3(k)$ query with $g^{H_3(k)} = u^{1/r}$, send $(\mathsf{OfflineEval}, \text{sid}, (k, 1), x)$ to $\mathcal{F}_{\text{OPRF}}$ and obtain $\mathcal{F}_{\text{OPRF}}$'s response $(\mathsf{OfflineEval}, \text{sid}, y)$.
   B. Otherwise, if S is not compromised and $(h, z) := \texttt{ServerKey}$ is defined with $z = u^{1/r}$, choose a new sub-session id ssid* that has not appeared anywhere else in the game, send $(\mathsf{PRFEval}, \text{sid}, \text{ssid}^*, \perp, x)$ followed by $(\mathsf{PRFReceiveComplete}, \text{sid}, \text{ssid}^*, \mathsf{Sim}, \mathsf{S})$ to $\mathcal{F}_{\text{OPRF}}$, and obtain $\mathcal{F}_{\text{OPRF}}$'s response $(\mathsf{PRFReceiveComplete}, \text{sid}, \text{ssid}, y)$. If $\mathcal{F}_{\text{OPRF}}$ does not respond, abort.
   C. Otherwise, if there exists a record $z := \texttt{index}[I]$ such that $z = u^{1/r}$, send $(\mathsf{OfflineEval}, \text{sid}, (i, 0), x)$ to $\mathcal{F}_{\text{OPRF}}$ and obtain $\mathcal{F}_{\text{OPRF}}$'s response $(\mathsf{OfflineEval}, \text{sid}, y)$.
   D. Otherwise, set $\texttt{index}[I] = z$, send $(\mathsf{OfflineEval}, \text{sid}, (I, 0), x)$ to $\mathcal{F}_{\text{OPRF}}$ and obtain $\mathcal{F}_{\text{OPRF}}$'s response $(\mathsf{OfflineEval}, \text{sid}, y)$. Set $I := I + 1$.

   If $H_2(x, u)$ is still undefined, set $H_2(x, u) \leftarrow \{0, 1\}^\ell$.
7. On $\mathcal{A}$'s query $H_3(k)$, send $(\mathsf{TestKey}, \text{sid}, k)$ to $\mathcal{F}_{\text{OPRF}}$.
   A. If $(h, z) := \texttt{ServerKey}$ is defined, and $\mathcal{F}_{\text{OPRF}}$ responds with $(\mathsf{TestKey}, \text{sid}, \mathsf{S}, k)$, set $H_3(k) := h$.
   B. If $\texttt{ServerKey}$ is undefined, and $\mathcal{F}_{\text{OPRF}}$ responds with $(\mathsf{TestKey}, \text{sid}, \mathsf{S}, k)$, sample $h := H_3(k) \leftarrow \mathbb{Z}_p$ and set $\texttt{ServerKey} := (h, g^h)$.
   C. If $\mathcal{F}_{\text{OPRF}}$ responds with $(\mathsf{TestKey}, \text{sid}, \mathsf{S}, \perp)$, set $H_3(k) \leftarrow \mathbb{Z}_p$.

[4] The session id is part of the input of all RO queries and is omitted.

We briefly compare our simulator and the simulator for 2HDH [21, Appendix B]:

In the interactive OPRF protocol, the client sends $a := H_1(x)^r$, and the server raises this value to the exponent $H_3(k)$, resulting in $b := H_1(x)^{rH_3(k)}$. Following the logic of 2HDH, when the adversary is playing the role of a server, the simulator is able to detect when the same exponent is used. The 2HDH simulation strategy does not give any way to extract this exponent. In our 3HDH variant, the simulator has access to all the queries made by the adversary to H_3. Therefore, it can test all of them to see whether the server has raised the group element to that power. If so, the simulator has identified the *actual key* k (step 5A). Otherwise, the simulator is still able to associate an "index" of a random function, as in the simulator for 2HDH. Additionally, although the simulator does not know the effective exponent used by the server, no future (fresh) query to $H_3(k)$ will result in that exponent, except with negligible probability. This random function therefore is associated with a non-transferrable key (steps 5B and 5C).

Whenever an adversary queries $H_2(x, u)$, the simulator checks whether $u = H_1(x)^{H_3(k)}$ for any k that has been previously queried to H_3. If so, then this is an offline query for (k, x) (step 6A). If not, then this may represent the adversary completing an interactive query; the simulator detects these in the same manner as in 2HDH (steps 6B–6D).

Finally, our OPRF functionality gives to the adversary a mechanism TestKey to check whether it has successfully obtained an honest server's key. This is necessary because the simulator will simulate an honest server using a random exponent. Then later, the adversary may learn that server's key k through the expected mechanism of the protocol. When that adversary queries $H_3(k)$, the simulator will need to detect this k as a special value, and program H_3's output to be the exponent that it has been using (step 7A).

The security argument is very similar to that in [19], so we only provide a sketch. The only case where the simulator might fail to generate a view identical to the real-world view lies in step 6B, when $\mathcal{F}_{\mathrm{OPRF}}$ does not respond to the simulator's PRFReceiveComplete message. This event can be reduced to the one-more gap Diffie-Hellman assumption: the reduction generates $g_1, \ldots, g_N$ by querying the h-th power oracle, and in steps 5C and 5D, even without knowing $r_1, \ldots, r_N$, the reduction can use the DDH oracle to check if $DL(a, b^*)$ is equal to some other $DL(a', (b^*)')$ (where $a', (b^*)'$ are the values that appear in a previous client sub-session) and determine the index accordingly. Similarly, in step 7A the reduction can check if $DL(g^{H_3(k)}, u) = DL(x, g_j)$, and in steps 7C and 7D it can check if $DL(g_j, u)$ is equal to some $DL(a', (b^*)')$ in a previous client sub-session. In sum, the reduction can run the simulator by replacing any computation using $r_1, \ldots, r_N$ with DDH oracle queries, and when the "bad event" happens it can break the one-more gap Diffie-Hellman assumption.

Functionality $\mathcal{F}_{\text{OPRF}}$

Parameters: output length ℓ

Storage: four maps: key, output, tx, and $\mathsf{session}$

Initialization

Upon receiving $(\mathsf{Init}, \mathrm{sid})$ from honest P:

1. Sample $k \leftarrow \{0,1\}^{\kappa}$.
2. Record $\mathsf{key}[\mathsf{P}, \mathrm{sid}] := k$.
3. Send $(\mathsf{Init}, \mathrm{sid}, k)$ to P.

Offline Evaluation

Upon receiving $(\mathsf{OfflineEval}, \mathrm{sid}, (k, \mathtt{trans}), x)$ from P:

1. If P is honest, overwrite $\mathtt{trans} := 1$.
2. If $\mathsf{output}[k, \mathit{trans}, x]$ is undefined, sample $\mathsf{output}[k, \mathtt{trans}, x] \leftarrow \{0,1\}^{\ell}$.
3. Send $(\mathsf{OfflineEval}, \mathrm{sid}, \mathsf{output}[k, \mathtt{trans}, x])$ to P.

Upon receiving $(\mathsf{TestKey}, \mathrm{sid}, k)$ from $\mathcal{A}^*$:

1. If $\mathsf{key}[\mathsf{P}, \mathrm{sid}] = k$ for some party P, then send $(\mathsf{TestKey}, \mathrm{sid}, k, \mathsf{P})$ to $\mathcal{A}^*$.
2. Otherwise send $(\mathsf{TestKey}, \mathrm{sid}, k, \perp)$ to $\mathcal{A}^*$.

Online Evaluation

Upon receiving $(\mathsf{PRFEval}, \mathrm{sid}, \mathrm{ssid}, \mathsf{S}, x)$ from $\mathsf{P} \in \{\mathsf{C}, \mathcal{A}^*\}$:

1. Send $(\mathtt{PRFEval}, \mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{S})$ to $\mathcal{A}^*$.
2. Await response $(\mathrm{sid}, \mathrm{ssid}, \mathit{prfx})$ from $\mathcal{A}^*$, and if $\mathsf{P} = \mathsf{C}$, ignore this command if $(\mathsf{Prefix}, \mathrm{sid}, \mathrm{ssid}, \mathit{prfx})$ was delivered to C or S previously.
3. Otherwise, set $\mathsf{session}[(\mathrm{ssid}, \mathsf{P})] := (x, \mathit{prfx})$ and send $(\mathsf{Prefix}, \mathrm{sid}, \mathrm{ssid}, \mathit{prfx})$ to P.

Upon receiving $(\mathsf{PRFSendComplete}, \mathrm{sid}, \mathrm{ssid})$ from S:

1. Send $(\mathtt{PRFSendComplete}, \mathrm{sid}, \mathrm{ssid}, \mathsf{S})$ to $\mathcal{A}^*$.
2. Await response $(\mathrm{sid}, \mathrm{ssid}, \mathit{prfx}')$ from $\mathcal{A}^*$, and send $(\mathsf{Prefix}, \mathrm{sid}, \mathrm{ssid}, \mathit{prfx}')$ to S.
3. If $(x, \mathit{prfx}) := \mathsf{session}[(\mathrm{ssid}, \mathsf{C})]$ is defined with $\mathit{prfx} = \mathit{prfx}'$, store $\mathsf{session}[(\mathrm{ssid}, \mathsf{C})] := (x, \mathrm{OK})$.
4. Otherwise set $\mathsf{tx}[\mathsf{S}] := \mathsf{tx}[\mathsf{S}] + 1$.

Upon receiving $(\mathsf{PRFReceiveComplete}, \mathrm{sid}, \mathrm{ssid}, \mathsf{P}, i)$ from $\mathcal{A}^*$:

1. If either $(x, \mathit{prfx}) := \mathsf{session}[(\mathrm{ssid}, \mathsf{P})]$ is undefined or ($i = \mathsf{S}$, $\mathsf{tx}[\mathsf{S}] = 0$, and $\mathit{prfx} \neq \mathrm{OK}$), then ignore this command.
2. If $i = \mathsf{S}$, set $(k, \mathtt{trans}) := (\mathsf{key}[\mathsf{S}, \mathrm{sid}], 1)$. Otherwise, set $(k, \mathtt{trans}) := i$.
3. If $\mathsf{output}[k, \mathtt{trans}, x]$ is undefined, then sample $\mathsf{output}[k, \mathtt{trans}, x] \leftarrow \{0,1\}^{\ell}$.
4. Give $(\mathtt{PRFReceiveComplete}, \mathrm{sid}, \mathrm{ssid}, \mathsf{output}[k, \mathtt{trans}, x])$ to P.
5. If $i = \mathsf{S}$ and $\mathit{prfx} \neq \mathrm{OK}$ then set $\mathsf{tx}[\mathsf{S}] := \mathsf{tx}[\mathsf{S}] - 1$.

Fig. 2. Ideal functionality $\mathcal{F}_{\text{OPRF}}$

4 Security Model

In this section, we detail the security modeling for fuzzy strong asymmetric PAKE. Our functionality $\mathcal{F}_{\text{safPAKE}}$ (Fig. 3, Fig. 4) merges the fuzzy password testing interfaces of $\mathcal{F}_{\text{faPAKE}}$ [14] and the strong asymmetric security of $\mathcal{F}_{\text{saPAKE}}$ [21]. We note that this is not the standard saPAKE functionalty, and our use of a relaxed OPRF functionality (as [21] does) requires that we inherit their functionality as a base. The three changes we have made to $\mathcal{F}_{\text{saPAKE}}$ are

- We parameterize $\mathcal{F}_{\text{safPAKE}}$ to use any similarity predicate σ instead of the single definition of similarity $(\sigma(x, x') := x \overset{?}{=} x')$.
- We parameterize $\mathcal{F}_{\text{safPAKE}}$ to use two leakage functions $\ell_{\mathsf{C}}, \ell_{\mathsf{S}}$ instead the default leakage function ($\ell_{\mathsf{P}}(x, x') :=$"correct guess").
- We include Abdalla, Hasse, and Hesse's strengthening of $\mathcal{F}_{\text{PAKE}}$'s NewKey interface [3]. Namely, we remove the clause where if either of the parties are corrupted, the adversary may set the all parties' keys—even if the adversary hasn't successfully attacked the session.

We note that $\mathcal{F}_{\text{safPAKE}}$ can be parameterized with any predicate and leakage, however our protocol can support only certain predicates and leakage. We elaborate later when describing the protocol.

4.1 Leakage Function

Following previous formalizations of fuzzy PAKE [13,14] we provide a generalized OfflineTestPwd and TestPwd interface which can return leakage to the caller on a correct guess. Dupont et al. [13] define three leakage functions, based on what is leaked in the case of far/medium/close password guesses (under some metric).

We handle leakage slightly differently. We allow the functionality to be parameterized by two different leakage functions: One for the client ℓ_{C} and one for the server ℓ_{S}. Specifically, if the password test is against party P, we use leakage function ℓ_{P}. In the case where the passwords are similar, we give the adversary $\ell_{\mathsf{P}}(pw, pw')$ in an all-or-nothing situation. In any other case, the functionality simply replies "wrong password." Our final safPAKE protocol supports a range of σ predicates, and leakage $\ell_{\mathsf{S}}(pw, pw') = pw$, $\ell_{\mathsf{C}}(pw, pw') =$"correct guess". That is, the client learns the "correct" password and key agreement succeeds when the two passwords are similar while from the server's perspective, the key agreement simply succeeds or doesn't.

4.2 Functionality

In our functionality, we consider an asymmetric interaction where one party takes on the role of the client C who has a password as input pw' and the other takes on the role of the server S who holds a "password file" `file` which is a digest of some password pw. The goal of the interaction is to arrive at a shared key exactly when $\sigma(pw, pw') \neq \bot$.

Since the parties are assumed to supply low-entropy passwords, an integral security property of PAKE lies in allowing the adversary *exactly one password guess* per session. $\mathcal{F}_{\text{safPAKE}}$ provides this interface in the form of the TestPwd query. This query can be sent once per subsession and provides the adversary a single password guess. We note that a single password guess will encompass a potentially large "guessing surface" of passwords that satisfy the similarity predicate with the target password. This is inherent to any PAKE protocol that supports fuzzy matches, and it is important to note that this is different from allowing an adversary an arbitrary "guessing surface" of the same size. An adversary can guess only the "similarity neighborhood" Σ_{pw^*} of one password pw^*. This can be seen in the TestPwd interface which only allows a single password as input.

In (strong) asymmetric PAKE we must additionally account for the adversary learning the server's long-term secrets. To accomplish this, the adversary is capable of sending a corruption query StealPwdFile to the server and receiving the server's password file `file`. We distinguish such a query from the static *Byzantine* corruptions by not allowing the adversary to control the server or to modify the server's long-term storage. To ensure that precomputation does not allow the adversary to "instantly" learn passwords upon compromise, the OfflineTestPwd interface is only available after compromise, but allows the adversary an unlimited number of guesses without online interaction. This models the adversary's ability to perform a dictionary attack against the server's file, which is inevitable since the adversary learns enough to simulate login attempts against the server's password file. Aside from password guesses, the adversary may now impersonate the server through the Impersonate query which allows the server to use the password file `file` to connect with a client. No similar impersonation interface exists for the client. This implies that the server must identify the password underlying the password file to impersonate a client to the honest server.

5 Protocol Overview

In this section, we build up the intuition for our safPAKE protocol, step by step.
Step 0: JKX Compiler Jarecki, Krawczyk, and Xu, in their seminal paper on strong asymmetric PAKE (saPAKE) [21], provided a compiler from plain aPAKE to saPAKE. The server holds a PRF key k and the "aPAKE digest" of $rw = F(k, pw)$, where F is a PRF. To authenticate, the client with input pw' learns $F(k, pw')$ via an oblivious PRF (OPRF) protocol. The parties then perform aPAKE where the client uses $rw' := F(k, pw')$ as input, and the server uses its digest of rw. Authentication succeeds if $rw' = rw$, which happens only when $pw' = pw$.
Step 1: Run Many Instances. The intuitive starting point for safPAKE is to run t parallel instances of saPAKE, where t is the number of acceptable mistyped passwords. The client, who holds a password guess pw', should use it as input to all instances. The server holds an saPAKE password file for each possible mistyping of the correct password pw. In this way, pw' is compared against every mistyping of pw.

This naïve safPAKE protocol has several deficiencies, which we address next. First, both parties can make t guesses of the other's password in each authentication attempt. These t guesses can be completely unrelated—*i.e.*, they need not

Functionality $\mathcal{F}_{\text{safPAKE}}$

Parameters:

- security parameter κ
- client C, server S, and ideal adversary $\mathcal{A}^*$
- $\boxed{\text{symmetric similarity predicate } \sigma : \widetilde{D} \times \widetilde{D} \to \{\perp\} \cup \{0,1\}^\kappa}$
- $\boxed{\text{leakage functions } \ell_{\mathsf{C}}, \ell_{\mathsf{S}} : \widetilde{D} \times \widetilde{D} \to \{0,1\}^*}$

Storage:

- Five maps: $\mathsf{fileStatus}$, file, $\mathsf{sessionStatus}$, $\mathsf{session}$, and $\mathsf{delayedTests}$
- A server compromise flag flag

Password Registration

Upon receiving $(\mathsf{StorePwdFile}, \text{sid}, \mathsf{C}, pw)$ from S, if $\mathsf{file}[(\text{sid}, \mathsf{C}, \mathsf{S})]$ is undefined, set $\mathsf{file}[(\text{sid}, \mathsf{C}, \mathsf{S})] := pw$, set $\mathsf{fileStatus}[(\text{sid}, \mathsf{C}, \mathsf{S})] := \mathsf{uncompromised}$, and set $\mathsf{flag} := \mathsf{uncompromised}$.

Stealing Password Data

Upon receiving $(\mathsf{StealPwdFile}, \text{sid})$ from $\mathcal{A}^*$:

1. If $\mathsf{file}[(\text{sid}, \mathsf{C}, \mathsf{S})]$ is undefined, return "no password file" to $\mathcal{A}^*$.
2. Otherwise:
   1. If $\mathsf{fileStatus}[(\text{sid}, \mathsf{C}, \mathsf{S})] = \mathsf{uncompromised}$, set $\mathsf{fileStatus}[(\text{sid}, \mathsf{C}, \mathsf{S})] := \mathsf{stolen}$.
   2. Return "password file stolen" to $\mathcal{A}^*$.

Upon receiving $(\mathsf{OfflineTestPwd}, \text{sid}, pw^*)$ from $\mathcal{A}^*$:

1. If $\mathsf{fileStatus}[(\text{sid}, \mathsf{C}, \mathsf{S})] = \mathsf{stolen}$:
   1. Retrieve $pw := \mathsf{file}[(\text{sid}, \mathsf{C}, \mathsf{S})]$.
   2. If $\boxed{\sigma(pw, pw^*) \neq \perp\text{: return } \ell_{\mathsf{S}}(pw, pw^*)}$ to $\mathcal{A}^*$ and set $\mathsf{flag} := \mathsf{compromised}$.
   3. Otherwise, return "wrong guess" to $\mathcal{A}^*$.

Password Authentication

Upon receiving $(\mathsf{ClientSession}, \text{sid}, \text{ssid}, \mathsf{S}, pw')$ from C:

1. Send $(\mathtt{ClientSession}, \text{sid}, \text{ssid}, \mathsf{C}, \mathsf{S})$ to $\mathcal{A}^*$.
2. If $\mathsf{sessionStatus}[(\text{sid}, \text{ssid}, \mathsf{C}, \mathsf{S})]$ is undefined, set $\mathsf{session}[(\text{sid}, \text{ssid}, \mathsf{C}, \mathsf{S})] := pw'$ and set $\mathsf{sessionStatus}[(\text{sid}, \text{ssid}, \mathsf{C}, \mathsf{S})] := \mathsf{fresh}$.

Upon receiving $(\mathsf{ServerSession}, \text{sid}, \text{ssid})$ from S:

1. If $\mathsf{file}[(\text{sid}, \mathsf{C}, \mathsf{S})]$ is undefined, ignore this query.
2. Otherwise, retrieve $pw := \mathsf{file}[(\text{sid}, \mathsf{C}, \mathsf{S})]$ and send $(\mathtt{ServerSession}, \text{sid}, \text{ssid}, \mathsf{C}, \mathsf{S})$ to $\mathcal{A}^*$.
3. If $\mathsf{sessionStatus}[(\text{sid}, \text{ssid}, \mathsf{S}, \mathsf{C})]$ is undefined, set $\mathsf{session}[(\text{sid}, \text{ssid}, \mathsf{S}, \mathsf{C})] := pw$ and set $\mathsf{sessionStatus}[(\text{sid}, \text{ssid}, \mathsf{S}, \mathsf{C})] := \mathsf{fresh}$.

Fig. 3. Ideal functionality safPAKE (part 1). The $\boxed{\text{framed text}}$ are generalizations of $\mathcal{F}_{\text{saPAKE}}$ [21].

correspond to a "typo neighborhood" around a single correct password. Second, the protocol is a factor t more expensive than saPAKE.

Step 2: Reuse OPRF. The client is supposed to use the same pw' guess to each of the t saPAKE instances. It is easy to enforce this by using just a single OPRF

```
Active Session Attacks
Upon receiving $(\mathsf{Interrupt}, \mathrm{sid}, \mathrm{ssid}, \mathsf{S})$ from $\mathcal{A}^*$:
  1. If $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{S}, \mathsf{C})] = \mathsf{fresh}$, set $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{S}, \mathsf{C})] := \mathsf{interrupted}$ and set $\mathsf{delayedTests}[\mathrm{ssid}] := 1$.
Upon receiving $(\mathsf{TestPwd}, \mathrm{sid}, \mathrm{ssid}, \mathsf{P}, pw^*)$ from $\mathcal{A}^*$:
  1. If $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')]$ is undefined, ignore this query.
  2. Otherwise, retrieve $\widetilde{pw} := \mathsf{session}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')]$.
  $\boxed{\text{3. Let } s := \sigma(\widetilde{pw}, pw^*), l := \ell_{\mathsf{S}}(\widetilde{pw}, pw^*) \text{ if } \mathsf{P} = \mathsf{S} \text{ and let } s := \sigma(pw^*, \widetilde{pw}), l := \ell_{\mathsf{C}}(pw^*, \widetilde{pw}) \text{ if } \mathsf{P} = \mathsf{C}.}$
  4. If $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')] = \mathsf{fresh}$:
    1. If $s \neq \perp$, return $l$ to $\mathcal{A}^*$ and set $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')] := \mathsf{compromised}$.
    2. Otherwise, set $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')] := \mathsf{interrupted}$ and return "wrong guess" to $\mathcal{A}^*$.
  5. If $\mathsf{P} = \mathsf{S}$ and $\mathsf{delayedTests}[\mathrm{ssid}] = 1$: set $\mathsf{delayedTests}[\mathrm{ssid}] := 0$, if $s \neq \perp$, return $l$ to $\mathcal{A}^*$, otherwise return "wrong guess" to $\mathcal{A}^*$.
  6. In both cases, if $\mathsf{P} = \mathsf{S}$ and $s \neq \perp$, then set $\mathsf{flag} := \mathsf{compromised}$.

Upon receiving $(\mathsf{Impersonate}, \mathrm{sid}, \mathrm{ssid})$ from $\mathcal{A}^*$:
  1. If $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{C}, \mathsf{S})]$ is undefined or if $\mathsf{fileStatus}[(\mathrm{sid}, \mathsf{C}, \mathsf{S})] \neq \mathsf{stolen}$, ignore this query.
  2. Otherwise, retrieve $pw := \mathsf{file}[(\mathrm{sid}, \mathsf{C}, \mathsf{S})]$ and $pw' := \mathsf{session}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{C}, \mathsf{S})]$.
    1. If $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{C}, \mathsf{S})]$ is $\mathsf{fresh}$: if $\boxed{\sigma(pw, pw') \neq \perp,}$ set $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{C}, \mathsf{S})] := \mathsf{compromised}$ and return "correct guess" to $\mathcal{A}^*$; otherwise, set $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{C}, \mathsf{S})] := \mathsf{interrupted}$ and return "wrong guess" to $\mathcal{A}^*$.

Key Generation and Authentication
Upon receiving $(\mathsf{NewKey}, \mathrm{sid}, \mathrm{ssid}, \mathsf{P}, k^*)$ from $\mathcal{A}^*$ where $|k^*| = \kappa$:
  1. If $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')]$ is defined, but is not $\mathsf{completed}$:
    $\boxed{\text{1. Let } s := \sigma(\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{S}, \mathsf{C})], \mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{C}, \mathsf{S})])}$
    2. If the record is $\mathsf{compromised}$, or if ($\mathsf{P} = \mathsf{S}$, the record is $\mathsf{interrupted}$, and $\mathsf{flag} = \mathsf{compromised}$), set $k := k^*$.
    3. Else, if the record is $\mathsf{fresh}$, $(\mathrm{sid}, \mathrm{ssid}, k')$ was sent to $\mathsf{P}'$, $s \neq \perp$, and at the time $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}', \mathsf{P})]$ was $\mathsf{fresh}$, set $k := k'$.
    4. Otherwise, sample $k \leftarrow \{0,1\}^\kappa$.
  2. Finally, set $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')] := \mathsf{completed}$ and send $(\mathrm{sid}, \mathrm{ssid}, k)$ to $\mathsf{P}$.

Upon receiving $(\mathsf{TestAbort}, \mathrm{sid}, \mathrm{ssid}, \mathsf{P})$ from $\mathcal{A}^*$:
  1. If $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')]$ is defined, but is not $\mathsf{completed}$:
    $\boxed{\text{1. Let } s := \sigma(\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{S}, \mathsf{C})], \mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{C}, \mathsf{S})])}$
    2. If $\mathsf{session}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')]$ is $\mathsf{fresh}$ and $s \neq \perp$, send "success" to $\mathcal{A}^*$.
    3. Else, send "abort" to $\mathcal{A}^*$ and $(\mathsf{Abort}, \mathrm{sid}, \mathrm{ssid})$ to $\mathsf{P}$.
    4. In both cases, set $\mathsf{sessionStatus}[(\mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \mathsf{P}')] := \mathsf{completed}$.
```

Fig. 4. Ideal functionality safPAKE (part 2). The $\boxed{\text{framed text}}$ are generalizations of $\mathcal{F}_{\text{saPAKE}}$ [21].

instance of t of them. With just one OPRF instance, the client learns $F(k, pw')$ for its single password guess pw'. Now we need a way to check whether this OPRF output is consistent with one of the allowed mistypings of pw.

A natural way to do this is for the server to hold and send "correction ciphertexts" to the client. For every allowed mistyping $\widetilde{pw}$ of the correct password pw, there is a ciphertext that encrypts $rw = F(k, pw)$ under the key $F(k, \widetilde{pw})$. Suppose the client obtains $F(k, pw')$ from an OPRF instance, and also holds these ciphertexts. Intuitively, if the client's OPRF input is a valid mistyping of pw, then she can decrypt one of these ciphertexts and obtain $rw = F(k, pw)$. If her OPRF input is not a valid mistyping of pw then the ciphertexts are meaningless to her. Then the parties can perform a single aPAKE instance, where the server holds a digest of rw.

Step 3: Check All the Correction Ciphertexts. In step 2, we successfully restricted the client to a single password guess pw'. However, the server can still make t completely unrelated guesses of the client's password. It can encrypt rw, t times, under the keys $F(k, pw_1), \ldots, F(k, pw_t)$, where the pw_i's are completely unrelated instead of being the set of valid mistypings of a single password. As the server must be able to authenticate the client against t possible inputs, the server will always be allowed t guesses. However, in the best case scenario, the server must have all of these guesses (structurally) be known by the client upon connection.

We fix this problem with the following two ideas: First, the correction ciphertexts are encryptions of (k, pw)[5] instead of encryptions of rw. Second, the correction ciphertexts should be computed as a *deterministic function* of k and pw. Specifically, the randomness used for encryption should be derived from k and pw.

With these two ideas in place, whenever the client successfully decrypts a correction ciphertext, she obtains (k, pw). From (k, pw) she can not only compute $rw = F(k, pw)$ herself, but also compute the *entire set of correction ciphertexts.* Importantly, she can compute them knowing the set of valid mistypings of pw.[6] If she realizes that the server did not send this *exact* collection of ciphertexts, she will abort the interaction.

Intuitively, if the server sends correction ciphertexts that do not correctly correspond to the valid set of mistypings of a single password, then the client will *always* abort the protocol—either because the client couldn't open any ciphertext, or because the client opened a ciphertext but then detected that the set of ciphertexts was incorrect.

Attentive readers will notice that there is some circularity in the new correction ciphertexts, where its plaintext (k, pw) is hidden, because the encryption key $F(k, \widetilde{pw})$ is random, because the PRF seed k is hidden. We must indeed be careful about such circularity! Fortunately, saPAKE protocols like OPAQUE

[5] The server actually provides a way for the client to reconstruct pw instead of providing pw directly, but for simplicity in this section, we write the plaintexts as containing pw.

[6] This requires the password typo policy to be public.

already require a very strong OPRF where $F(k, x)$ can look random even to someone who knows k—provided that they haven't queried the PRF at $F(k, x)$ yet. Such a strong [O]PRF is enough to resolve the apparent circularity.
Step 4: Compress Communication with PIR. As a result of the previous step, we have a safPAKE protocol where both the client and server can make only a single password guess. However, the protocol's communication cost still scales with t, the number of valid mistypings. This is due to the t correction ciphertexts, which the server must send to the client.

We can compress this communication by using PIR. In our safPAKE protocol, there is only one ciphertext that the client can decrypt using its key $F(k, pw')$. If the client can find a way to "identify" which ciphertext to decrypt, then she can use (single-server) **keyword PIR** to fetch it from the server with low communication.

In more detail, let us interpret the output of $F(k, \widetilde{pw})$ in two pieces: $\mathrm{tag}_{\widetilde{pw}} \| \mathrm{ek}_{\widetilde{pw}} := F(k, \widetilde{pw})$, where ek is an encryption key and tag is a random identifier. The server can prepare a key-value mapping that maps $\mathrm{tag}_{\widetilde{pw}}$ to the correction ciphertext $\mathsf{Enc}(\mathrm{ek}_{\widetilde{pw}}; (k, pw))$.[7] The client learns $F(k, pw') = \mathrm{tag}_{pw'} \| \mathrm{ek}_{pw'}$, and then initiates a keyword PIR protocol on input $\mathrm{tag}_{pw'}$. If this is the tag corresponding to some ciphertext, then the client will obtain that ciphertext and be able to decrypt it as usual.

But whenever the client successfully decrypts the ciphertext, it must then recompute the *entire set* of correction ciphertexts and compare to what the server has sent. However, now the server is not actually sending the entire collection of ciphertexts, but is only running a PIR protocol. We resolve this problem by observing that typical PIR protocols can have deterministic server behavior (because PIR has no privacy guarantees for the server). So the client can recompute (in her head) the entire collection of correction ciphertexts, and check whether the server *followed the deterministic PIR protocol with this collection as its input.*

The consistency check in Step 3 (the client recomputes all the correction ciphertexts and compares to what the server previously sent) is in fact an instantiation of this deterministic-PIR consistency check, because sending the entire ciphertext database is a trivial PIR protocol.

5.1 Similarity Predicates

As we briefly introduced in the previous section, the client needs to know the set of valid mistypings of her input pw' in order to verify the server's computation. We define this set in terms of a **similarity predicate** σ. In traditional PAKE, we concern ourselves with a password dictionary D from which the parties can choose their inputs from. As we now must consider errors, we use an expanded dictionary $\widetilde{D}$ in our definitions.

[7] For simplicity, we are ignoring the encryption randomness which also should be derived from the output of $F(k, \widetilde{pw})$.

Definition 2 (Similarity Predicate). *A* **similarity predicate** *is an efficiently computable function* $\sigma : \widetilde{D} \times \widetilde{D} \to \{\perp\} \cup \{0,1\}^{\kappa}$ *along with an efficiently computable* **reconstruction function** $\rho : \widetilde{D} \times \{0,1\}^{\kappa} \to \{\perp\} \cup \widetilde{D}$ *such that*

- *correctness:* $\forall x, x' \in \widetilde{D}$ *such that* $y := \sigma(x, x')$*:* $y \neq \perp \implies \rho(x', y) = x$*.*
- *reconstruction bounded:* $\forall x \in \widetilde{D}, y \in \{0,1\}^{\kappa}$ *such that* $z := \rho(x, y)$*:* $z \neq \perp \implies x \in \Sigma_z$ *where* $\Sigma_z := \{x' \mid \sigma(z, x') \neq \perp\}$*.*

In other words, x is said to be "similar" to x' if and only if x can be "reconstructed" from x': $\rho(x', \sigma(x, x')) = x$. The set Σ_x is then said to be the set of all values "similar" to x.

We require four additional properties of σ:

- *self similarity*: $\forall x \in \widetilde{D}, \sigma(x, x) \neq \perp$.
- *input independent size*: $\forall x \in \widetilde{D}$, Σ_x is of some fixed cardinality n.
- *efficient enumerability*: $\forall x \in \widetilde{D}$, Σ_x can be enumerated in polynomial time.
- *predictable hints*: $\forall x \in \widetilde{D}, \sigma(x, \Sigma_x) = \{1, 2, 3, \ldots, n\}$.

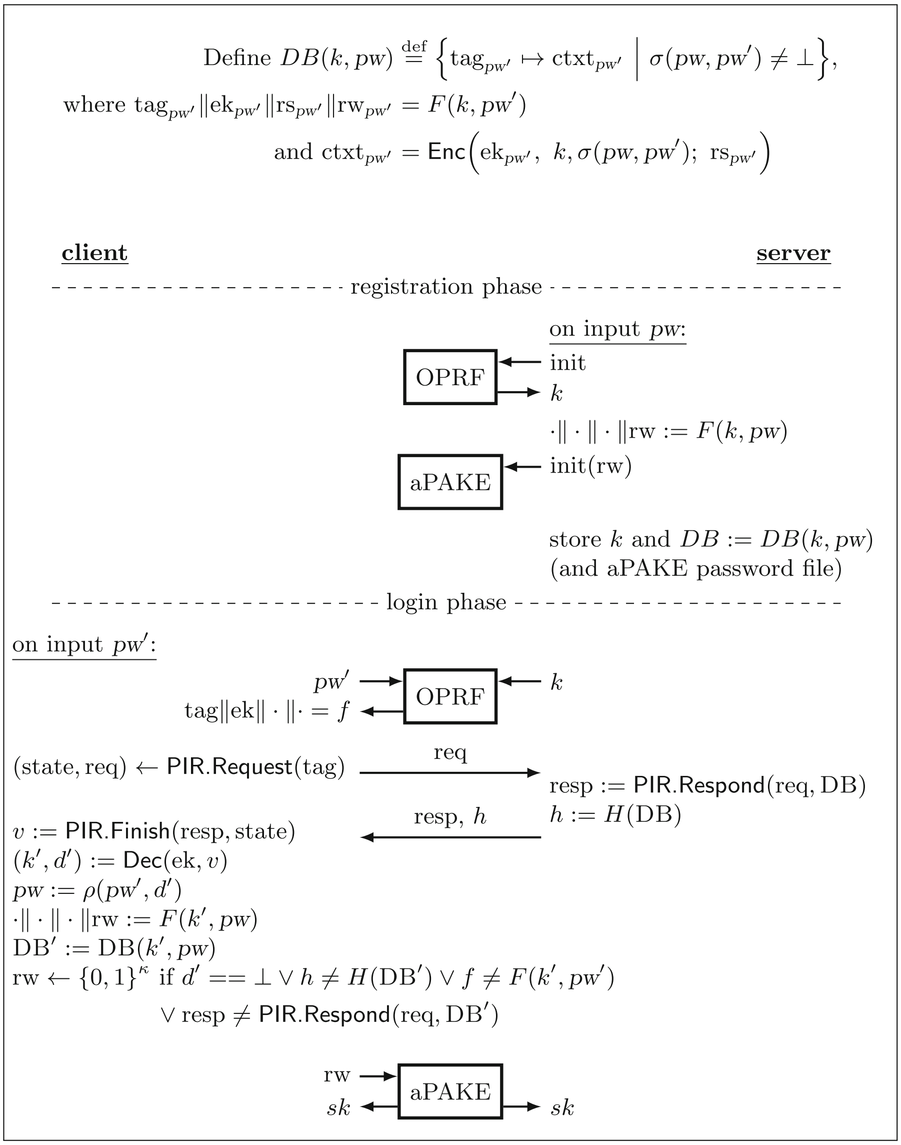
We assume that $\widetilde{D}$, and n are part of σ's description. It is important to note that our definition of predictable hints may seem stronger than necessary for our proof. Indeed, the set does not need to be exactly $\{1, 2, 3, \ldots, n\}$, but rather just needs to be some fixed set. We use $\{1, 2, 3, \ldots, n\}$, without loss of generality, to not bog down notation when we use the set's lexicographic ordering as an indexing set. We also conjecture that given a PIR protocol which hides the size of the server's database, we can drop the *input independent size* property for a weaker property which only requires an upper bound on the size of Σ_x.

The general definition covers many fuzzy distance functions such as repetition mistakes, capitalization errors, and hamming distance for fixed length passwords. This is a similar notion to the implicit requirements for the aPAKE to fuzzy aPAKE compiler due to Erwig et al. [14]. Additionally, for any *symmetric* similarity predicate which is also efficiently enumerable, we may generically implement $\sigma(x, x')$ by outputting the index of x in $\Sigma_{x'}$ (following some known ordering) and $\rho(x', y)$ can then output the yth element of $\Sigma_{x'}$.

6 A Strong Asymmetric Fuzzy PAKE Protocol

In this section we describe a compiler from an OPRF, PIR protocol, and an aPAKE protocol to a safPAKE protocol. The actual protocol isn't far from the intuition we have built so far and we present a graphical representation of the protocol in Fig. 5.

Registration In the registration phase, the server begins by interacting with the OPRF functionality to receive the user-specific key k and computes its salted password rw from $F(k, pw)$. The server then initializes the aPAKE server storage using rw and constructs its local PIR database centered around pw. This is accomplished by mapping password tags f_{pw_i}.tag to encryptions of the corresponding reconstruction hints $d_{pw_i} := \sigma(pw, pw_i)$.



$$\text{Define } DB(k, pw) \stackrel{\text{def}}{=} \left\{ \text{tag}_{pw'} \mapsto \text{ctxt}_{pw'} \;\middle|\; \sigma(pw, pw') \neq \bot \right\},$$

$$\text{where } \text{tag}_{pw'} \| \text{ek}_{pw'} \| \text{rs}_{pw'} \| \text{rw}_{pw'} = F(k, pw')$$

$$\text{and } \text{ctxt}_{pw'} = \mathsf{Enc}\Big(\text{ek}_{pw'},\ k, \sigma(pw, pw');\ \text{rs}_{pw'}\Big)$$

client
server
registration phase
on input pw:
OPRF
init
k
$\cdot \| \cdot \| \cdot \| \text{rw} := F(k, pw)$
aPAKE
init(rw)
store k and $DB := DB(k, pw)$
(and aPAKE password file)
login phase
on input pw':
pw'
OPRF
k
$\text{tag} \| \text{ek} \| \cdot \| \cdot = f$
$(\text{state}, \text{req}) \leftarrow \mathsf{PIR.Request}(\text{tag})$
req
$\text{resp} := \mathsf{PIR.Respond}(\text{req}, \text{DB})$
$h := H(\text{DB})$
resp, h
$v := \mathsf{PIR.Finish}(\text{resp}, \text{state})$
$(k', d') := \mathsf{Dec}(\text{ek}, v)$
$pw := \rho(pw', d')$
$\cdot \| \cdot \| \cdot \| \text{rw} := F(k', pw)$
$\text{DB}' := \text{DB}(k', pw)$
$\text{rw} \leftarrow \{0,1\}^\kappa$ if $d' == \bot \vee h \neq H(\text{DB}') \vee f \neq F(k', pw')$
$\vee\ \text{resp} \neq \mathsf{PIR.Respond}(\text{req}, \text{DB}')$
rw
aPAKE
sk
sk

Fig. 5. Graphical representation of our protocol. See text for omitted details.

1. Upon receiving $(\mathsf{StorePwdFile}, \text{sid}, \mathsf{C}, pw)$, S sends $(\mathsf{Init}, \text{sid})$ to $\mathcal{F}_{\text{OPRF}}$ and receives $(\mathsf{Init}, \text{sid}, k)$.
2. S then sends $(\mathsf{OfflineEval}, \text{sid}, (k, 1), \widetilde{pw})$ to $\mathcal{F}_{\text{OPRF}}$ for each $\widetilde{pw} \in \Sigma_{pw} := \{\widetilde{pw} \in \widetilde{D} \mid \sigma(pw, \widetilde{pw}) \neq \bot\}$
3. Upon receiving $(\mathsf{OfflineEval}, \text{sid}, f_{\widetilde{pw}})$, S parses $f_{\widetilde{pw}}$ as four equilength strings $f_{\widetilde{pw}}.\text{tag} \| f_{\widetilde{pw}}.\text{ek} \| f_{\widetilde{pw}}.\text{rs} \| f_{\widetilde{pw}}.rw$ each of length κ. We recall that as σ provides

predictable hints, we may use these hints as an ordering for passwords $\Sigma_{pw} = \{pw_1, pw_2, \ldots, pw_n\}$.

4. S then generates keyword PIR database $\mathrm{DB} := \{f_{pw_i}.\mathrm{tag} \mapsto \mathcal{E}.\mathsf{Enc}(f_{pw_i}.\mathrm{ek}; (k, d_{pw_i} := \sigma(pw, pw_i)); f_{pw_i}.\mathrm{rs})\}$ for $i \in \{1, 2, \ldots, n\}$.
5. Finally, S sends $(\mathsf{StorePwdFile}, \mathrm{sid}, \mathsf{C}, f_{pw}.rw)$ to $\mathcal{F}_{\mathrm{aPAKE}}$ and stores $\mathsf{file}[\mathrm{sid}] := (\mathrm{DB}, k)$.

Server Compromise

1. Upon receiving $(\mathsf{StealPwdFile}, \mathrm{sid})$ from $\mathcal{A}$, S retrieves $\mathsf{file}[\mathrm{sid}]$ and sends it to $\mathcal{A}$.

Login

During login, the client learns a PIR tag associated with their password pw' by evaluating the OPRF with the server. The client and server then perform a PIR exchange on the client's tag. Once the client receives the response, the client decrypts the reconstruction hint d' and reconstructs the server's password $pw := \rho(pw', d')$. The client then verifies the server's messages and runs an aPAKE subsession with the server on the salted password rw computed from $F(k, pw)$.

1. Upon receiving $(\mathsf{ClientSession}, \mathrm{sid}, \mathrm{ssid}, \mathsf{S}, pw')$, C sends $(\mathsf{PRFEval}, \mathrm{sid}, \mathrm{ssid}, \mathsf{S}, pw')$ to $\mathcal{F}_{\mathrm{OPRF}}$ receiving $(\mathsf{Prefix}, \mathrm{sid}, \mathrm{ssid}, prfx_{\mathsf{C}})$.
2. Upon receiving $(\mathsf{ServerSession}, \mathrm{sid}, \mathrm{ssid})$, S retrieves $(\mathrm{DB}, k) := \mathsf{file}[\mathrm{sid}]$ and sends $(\mathsf{PRFSendComplete}, \mathrm{sid}, \mathrm{ssid})$ to $\mathcal{F}_{\mathrm{OPRF}}$ receiving $(\mathsf{Prefix}, \mathrm{sid}, \mathrm{ssid}, prfx_{\mathsf{S}})$.
3. Upon receiving $(\mathtt{PRFReceiveComplete}, \mathrm{sid}, \mathrm{ssid}, f)$ from $\mathcal{F}_{\mathrm{OPRF}}$ C parses f as four equilength strings $f.\mathrm{tag}\|f.\mathrm{ek}\|f.\mathrm{rs}\|f.rw$ then C computes $(\mathrm{state}, \mathrm{req}) \leftarrow \mathsf{PIR.Request}(f.\mathrm{tag})$ and sends $(\mathrm{sid}, \mathrm{ssid}, \mathrm{req})$ to S.
4. Upon receiving $(\mathrm{sid}, \mathrm{ssid}, \mathrm{req})$ from C, S computes $\mathrm{resp} := \mathsf{PIR.Respond}(\mathrm{req}, \mathrm{DB})$, queries $\mathcal{F}_{\mathrm{RO}}$ on input DB receiving $h \in \{0,1\}^\kappa$ and sends $(\mathrm{sid}, \mathrm{ssid}, (\mathrm{resp}, h))$ to C.
5. Upon receiving $(\mathrm{sid}, \mathrm{ssid}, (\mathrm{resp}, h))$ from S, C computes $v := \mathsf{PIR.Finish}(\mathrm{resp}, \mathrm{state})$.
   1. C computes $z := \mathcal{E}.\mathsf{Dec}(f.\mathrm{ek}, v)$ and checks if z parses as $(k' \in \{0,1\}^\kappa, d' \in \{0,1\}^\kappa)$.
   2. If so, C then sets $pw := \rho(pw', d')$.
   3. C sends $(\mathsf{OfflineEval}, \mathrm{sid}, (k', 1), pw_i)$ to $\mathcal{F}_{\mathrm{OPRF}}$ for each $pw_i \in \Sigma_{pw}$ and catalogs the responses f_{pw_i}.
   4. C generates PIR database DB' using k' and the f_{pw_i} then computes $\mathrm{resp}' := \mathsf{PIR.Respond}(\mathrm{req}, \mathrm{DB}')$.
   5. C queries $\mathcal{F}_{\mathrm{RO}}$ on input DB' receiving $h' \in \{0,1\}^\kappa$.
   6. C checks if $\mathrm{resp}' = \mathrm{resp}$ and $h' = h$.
   7. If this check fails, parsing z failed, or $f_{pw'} \neq f$ C sets $f_{pw}.rw \leftarrow \{0,1\}^\kappa$.
   8. C sends $(\mathsf{ClientSession}, \mathrm{sid}, \mathrm{ssid}', \mathsf{S}, f_{pw}.rw)$ to $\mathcal{F}_{\mathrm{aPAKE}}$ where $\mathrm{ssid}' = \mathrm{ssid}\|prfx_{\mathsf{C}}\|\mathrm{req}\|\mathrm{resp}\|h$.
7. S sends $(\mathsf{ServerSession}, \mathrm{sid}, \mathrm{ssid}')$ to $\mathcal{F}_{\mathrm{aPAKE}}$ where $\mathrm{ssid}' = \mathrm{ssid}\|prfx_{\mathsf{S}}\|\mathrm{req}\|\mathrm{resp}\|h$.
8. Either party, upon receiving $(\mathrm{sid}, \mathrm{ssid}', \mathrm{sk})$ or $(\mathsf{Abort}, \mathrm{sid}, \mathrm{ssid}')$ from $\mathcal{F}_{\mathrm{aPAKE}}$, outputs the message received.

7 Security Analysis

We will now show that our protocol is a secure realization of the $\mathcal{F}_{\text{safPAKE}}$ functionality.

Theorem 1. *The protocol in Sect. 6 securely realizes the safPAKE functionality $\mathcal{F}_{safPAKE}$ (Fig. 3 and Fig. 4) in the $(\mathcal{F}_{RO}, \mathcal{F}_{aPAKE}, \mathcal{F}_{OPRF})$-hybrid model and the static Byzantine corruption setting, for any authenticated encryption scheme (Definition 1) $\mathcal{E}$; efficiently enumerable, input-independent-sized, self-similar, similarity predicate with predictable hints (Sect. 5.1) σ; secure keyword private information retrieval scheme ($\mathsf{PIR.Request}$, $\mathsf{PIR.Respond}$, $\mathsf{PIR.Finish}$) (Sect. 2.1); and leakage functions $\ell_{\mathsf{S}}(pw, pw') = pw$, $\ell_{\mathsf{C}}(pw, pw') =$ "correct password".*

7.1 Simulator

We construct the following simulator Sim for any PPT environment $\mathcal{Z}$. As standard in UC, we assume that the real adversary $\mathcal{A}$ is "dummy", *i.e.*, it merely passes messages to and from $\mathcal{Z}$. Without loss of generality, we also assume that all $\mathcal{F}_{\text{RO}}$ queries are made via $\mathcal{A}$, *i.e.*, $\mathcal{Z}$ does not make these queries on its own. In the following, the session ID is always included as part of a random oracle input and is omitted (*i.e.*, $H(\text{sid}, x)$ is simplified to $H(x)$).

Generating the Password File Upon receiving the first server message for a given sid, Sim generates a database on uniform values as follows. We recall that for any input $x \in \widetilde{D}$, the set of all similar inputs $\Sigma_x = \{\widetilde{x} \mid \sigma(x, \widetilde{x}) \neq \bot\}$ is of some fixed size n.

1. Sample n uniform strings $f_1, \ldots, f_n \leftarrow \{0,1\}^{3\kappa}$, sample two strings $k \leftarrow \{0,1\}^{\kappa}, rw \leftarrow \{0,1\}^{\kappa}$, and store these values $\texttt{params}[\text{sid}] := \{f_1, \ldots, f_n\}$, $\texttt{rw}[\text{sid}] := rw$.
2. For $i \in \{1, 2, 3, \ldots, n\}$, parse $f_i = f_i.\text{tag} \| f_i.\text{ek} \| f_i.\text{rs}$ each with length κ. If any tag, ek, or rs collide, abort. Then generate the database $\texttt{database}[\text{sid}] := \{f_i.\text{tag} \mapsto \mathcal{E}.\mathsf{Enc}(f_i.\text{ek}; (k, i); f_i.\text{rs})\}$.
3. Finally, store $\texttt{file}[\text{sid}] := (\texttt{database}[\text{sid}], k)$ and set $\texttt{password}[\text{sid}] := \bot$.

The difference between what the simulator does and a server's honest run of the protocol is that the honest server would receive a random OPRF key k from $\mathcal{F}_{\text{OPRF}}$, and would compute $f_i \| f_i.rw := F_{k,1}(pw_i)$ and $rw := F_{k,1}(pw).rw$ using $\mathsf{OfflineTestPwd}$ queries to $\mathcal{F}_{\text{OPRF}}$. *Programming the Password File* In many parts of the simulator, when $\mathcal{A}$ successfully guesses the session's password through $\mathsf{OfflineTestPwd}$ or through $\mathsf{TestPwd}$, Sim needs to equivocate or "explain" the password file it previously generated. To do so, we define a function the simulator can call to, $\mathsf{ProgramDatabase}(\text{sid}, pw, rw, k, \{f_1, \ldots, f_n\})$, as follows:

1. Compute Σ_{pw}. Recall that by the predictable hints property of σ, we may number all elements $\widetilde{pw} \in \Sigma_{pw}$ with $\widetilde{pw}$ being assigned $\sigma(pw, \widetilde{pw})$.
2. For each $pw_i \in \Sigma_{pw}$, if $F_{k,1}(pw_i)$ is undefined and $pw_i \neq pw$, sample $rw_i \leftarrow \{0,1\}^{\kappa}$ and set $F_{k,1}(pw_i) := f_i \| rw_i$. If any rw_i collide, abort.

3. If $F_{k,1}(pw)$ is undefined, set $F_{k,1}(pw) := f_j \| rw$ where j is the index of pw in Σ_{pw}.

Stealing the Password File and Offline Queries

4. Initialize $\texttt{tested}[\text{sid}] := \emptyset$.
5. Upon receiving $(\mathsf{StealPwdFile}, \text{sid})$ from $\mathcal{A}$ sent to $\mathcal{F}_{\text{aPAKE}}$, and $(\mathsf{StealPwdFile}, \text{sid})$ from $\mathcal{A}$ sent to S:
   - A. Send $(\mathsf{StealPwdFile}, \text{sid})$ to $\mathcal{F}_{\text{safPAKE}}$.
   - B. If $\mathcal{F}_{\text{safPAKE}}$ returns "password file stolen", then
     - I. Mark S $\mathsf{compromised}$.
     - II. For $rw := \texttt{rw}[\text{sid}]$, if $rw \in \texttt{tested}[\text{sid}]$, then return rw to $\mathcal{A}$ from $\mathcal{F}_{\text{aPAKE}}$. Otherwise, return "password file stolen" to $\mathcal{A}$ from $\mathcal{F}_{\text{aPAKE}}$.
     - III. Return $\texttt{file}[\text{sid}]$ to $\mathcal{A}$ from S.
   - C. Otherwise, return "no password file" to $\mathcal{A}$ as a message from $\mathcal{F}_{\text{aPAKE}}$.
6. Upon receiving $(\mathsf{OfflineEval}, \text{sid}, (k^*, \texttt{trans}^*), x)$ from $\mathcal{A}$ sent to $\mathcal{F}_{\text{OPRF}}$:
   - A. If S is marked $\mathsf{compromised}$, $k = k^*$, and $\texttt{trans} = 1$ where $(\text{DB}, k) := \texttt{file}[\text{sid}]$, send $(\mathsf{OfflineTestPwd}, \text{sid}, x)$ to $\mathcal{F}_{\text{safPAKE}}$.
   - B. If $\mathcal{F}_{\text{safPAKE}}$ returns pw, retrieve $rw := \texttt{rw}[\text{sid}]$, $(k, \{f_i\}) := \texttt{params}[\text{sid}]$, record $\texttt{password}[\text{sid}] := pw$, and run $\mathsf{ProgramDatabase}(\text{sid}, pw, rw, k, \{f_i\})$.
   - C. If S is not marked $\mathsf{compromised}$, $k^* = k$, and $F_{k,1}(\texttt{password}[\text{sid}])$ is undefined, abort.
   - D. In any other case:
     - I. If undefined, sample $F_{k^*,\texttt{trans}^*}(x) := f.\text{tag} \| f.\text{ek} \| f.\text{rs} \| f.rw \leftarrow \{0,1\}^{4\kappa}$ and abort if there exists another OPRF output with which collides on the tag, ek, rs, or rw.
     - II. Return $(\mathsf{OfflineEval}, \text{sid}, F_{k^*,\texttt{trans}^*}(x))$ to $\mathcal{A}$ from $\mathcal{F}_{\text{OPRF}}$.

7. Upon receiving $(\mathsf{TestKey}, \text{sid}, k^*)$ from $\mathcal{A}$ sent to $\mathcal{F}_{\text{OPRF}}$:
   - A. If $k = k^*$ where $(\text{DB}, k) := \texttt{file}[\text{sid}]$, set $\mathsf{P} := \mathsf{S}$.
   - B. Otherwise, set $\mathsf{P} := \perp$.
   - C. Return $(\mathsf{TestKey}, \text{sid}, k^*, \mathsf{P})$ to $\mathcal{A}$ from $\mathcal{F}_{\text{OPRF}}$.
8. Upon receiving $(\mathsf{OfflineTestPwd}, \text{sid}, rw^*)$ from $\mathcal{A}$ sent to $\mathcal{F}_{\text{aPAKE}}$:
   - A. Add rw^* to $\texttt{tested}[\text{sid}]$.
   - B. If S is marked $\mathsf{compromised}$, retrieve $rw := \texttt{rw}[\text{sid}]$ ($\texttt{rw}[\text{sid}]$ is defined when Sim generates $\texttt{file}[\text{sid}]$, which in turn is no later than S is compromised):
     - I. If $rw = rw^*$, send "correct guess" to $\mathcal{A}$ from $\mathcal{F}_{\text{aPAKE}}$.
     - II. Otherwise, send "wrong guess" to $\mathcal{A}$ from $\mathcal{F}_{\text{aPAKE}}$.

Password Authentication

9. Initialize maps `clientSession`, `serverSession`, `delayedTests`, and `delayedSessions`.

10. Upon receiving $(\mathsf{ClientSession}, \text{sid}, \text{ssid}, \mathsf{C}, \mathsf{S})$ from $\mathcal{F}_{\text{safPAKE}}$:
    A. If `clientSession`[ssid] is undefined, set `clientSession`[ssid] $:= (\mathsf{C}, \mathsf{S}, \bot, \bot, \bot, \bot, \bot)$; otherwise, ignore the query.
    B. Send $(\mathtt{PRFEval}, \text{sid}, \text{ssid}, \mathsf{C}, \mathsf{S})$ to $\mathcal{A}$ from $\mathcal{F}_{\text{OPRF}}$ and await $(\text{sid}, \text{ssid}, \mathit{prfx})$ from $\mathcal{A}$. Reject the response unless prfx is new.
    C. Update `clientSession`[ssid] $:= (\mathsf{C}, \mathsf{S}, \bot, \mathit{prfx}, \bot, \bot, \bot)$ and await $(\mathsf{PRFReceiveComplete}, \text{sid}, \text{ssid}, \mathsf{C}, i)$ from $\mathcal{A}$ sent to $\mathcal{F}_{\text{OPRF}}$.
    D. If $i = \mathsf{S}$ and (`serverSession`[ssid] is undefined or `serverSession`[ssid] $\neq (\mathsf{C}, \mathsf{S}, \cdot, \mathit{prfx}, \cdot, \cdot)$):
        I. If `delayedTests`[sid] $= 0$, ignore this message.
        II. Otherwise, set `delayedTests`[sid] $:=$ `delayedTests`[sid] $- 1$.
    E. If the request wasn't ignored, compute $(\text{state}, \text{req}) := \mathsf{PIR.Request}(0^\kappa)$ and send $(\text{sid}, \text{ssid}, \text{req})$ to $\mathcal{A}$ from C then await $(\text{sid}, \text{ssid}, (\text{resp}, h))$ from $\mathcal{A}$ sent to C.
    F. Upon receiving $(\text{sid}, \text{ssid}, (\text{resp}, h))$, send $(\mathsf{ClientSession}, \text{sid}, \text{ssid}\|\mathit{prfx}\|\text{req}\|\text{resp}\|h, \mathsf{C}, \mathsf{S})$ to $\mathcal{A}$ from $\mathcal{F}_{\text{aPAKE}}$, set `clientSession`[ssid] $:= (\mathsf{C}, \mathsf{S}, i, \mathit{prfx}, \text{req}, \text{resp}, h)$, and mark `clientSession`[ssid] as "aPAKE active".

11. Upon receiving $(\mathsf{ServerSession}, \text{sid}, \text{ssid}, \mathsf{C}, \mathsf{S})$ from $\mathcal{F}_{\text{safPAKE}}$
    A. If `serverSession`[ssid] is undefined, set `serverSession`[ssid] $:= (\mathsf{C}, \mathsf{S}, \bot, \bot, \bot, \bot, \bot)$; otherwise, ignore the query.
    B. Send $(\mathtt{PRFSendComplete}\text{sid}, \text{ssid}, \mathsf{S})$ to $\mathcal{A}$ from $\mathcal{F}_{\text{OPRF}}$, await $(\text{sid}, \text{ssid}, \mathit{prfx})$ from $\mathcal{A}$, and update `serverSession`[ssid] $:= (\mathsf{C}, \mathsf{S}, \bot, \mathit{prfx}, \bot, \bot, \bot)$.
    C. If `clientSession`[ssid] $\neq (\mathsf{C}, \mathsf{S}, \cdot, \mathit{prfx}, \cdot, \cdot, \cdot)$ or if `clientSession`[ssid] is undefined, send $(\mathsf{Interrupt}, \text{sid}, \text{ssid}, \mathsf{S})$ to $\mathcal{F}_{\text{safPAKE}}$, set `delayedTests`[sid] $:=$ `delayedTests`[sid] $+ 1$, and add ssid to `delayedSessions`[sid].
    D. Upon receiving $(\mathsf{PRFEval}, \text{sid}, \text{ssid}, \mathsf{S}, x)$ from $\mathcal{A}$ sent to $\mathcal{F}_{\text{OPRF}}$ send $(\mathtt{PRFEval}, \text{sid}, \text{ssid}, \mathcal{A}, \mathsf{S})$ to $\mathcal{A}$ from $\mathcal{F}_{\text{OPRF}}$.
        I. Await $(\mathsf{PRFReceiveComplete}, \text{sid}, \text{ssid}, \mathcal{A}, i)$ from $\mathcal{A}$ sent to $\mathcal{F}_{\text{OPRF}}$.
        II. If $i = \mathsf{S}$ and `delayedTests`[sid] > 0, set `delayedTests`[sid] $:=$ `delayedTests`[sid] $- 1$, choose any sub-session $\text{ssid}' \in$ `delayedSessions`[sid] send $(\mathsf{TestPwd}, \text{sid}, \text{ssid}', \mathsf{S}, x)$ to $\mathcal{F}_{\text{safPAKE}}$ and remove ssid' from `delayedSessions`[sid].
        III. If $i = (k, 1)$ where $(\text{DB}, k) :=$ `file`[sid]: If S is not compromised, and `password` has not been set, abort; otherwise, send $(\mathsf{OfflineTestPwd}, \text{sid}, x)$ to $\mathcal{F}_{\text{safPAKE}}$.
            a. If $\mathcal{F}_{\text{safPAKE}}$ responds with pw in either case, set `password`[sid] $:= \mathit{pw}$, retrieve $\mathit{rw} :=$ `rw`[sid], $(\text{DB}, k) :=$ `file`[sid], $\{f_j\} :=$ `params`[sid], and run $\mathsf{ProgramDatabase}(\text{sid}, \mathit{pw}, \mathit{rw}, k, \{f_j\})$.

b. In any case, if $F_{k,1}(x)$ is undefined, sample $F_{k,1}(x) := f.\text{tag}\|f.\text{ek}\|f.\text{rs}\|f.rw \leftarrow \{0,1\}^{4\kappa}$ and abort if there exists another OPRF output with which collides on the tag, ek, rs, or *rw*.

c. Finally, send $(\mathtt{PRFReceiveComplete}, \text{sid}, \text{ssid}, F_{k,1}(x))$ to $\mathcal{A}$.

IV. Otherwise, parse i as $(k^*, \mathtt{trans}^*)$ and if $F_{k^*,\mathtt{trans}^*}(x)$ is undefined; sample $F_{k^*,\mathtt{trans}^*}(x) := f.\text{tag}\|f.\text{ek}\|f.\text{rs}\|f.rw \leftarrow \{0,1\}^{4\kappa}$ and abort if there exists another OPRF output with which collides on the tag, ek, rs, or *rw*; and send $(\mathtt{PRFReceiveComplete}, \text{sid}, F_{k^*,\mathtt{trans}^*}(x))$ to $\mathcal{A}$.

E. Upon receiving $(\text{sid}, \text{ssid}, \text{req})$, compute $\text{resp} := \mathsf{PIR.Respond}(\text{req}, \mathtt{database}[\text{sid}])$ and $h := H(\mathtt{database}[\text{sid}])$, send $(\text{sid}, \text{ssid}, (\text{resp}, h))$ to $\mathcal{A}$ from S, and set $\mathtt{serverSession}[\text{ssid}] := (\mathsf{C}, \mathsf{S}, \mathsf{S}, \mathit{prfx}, \text{req}, \text{resp}, h)$.

F. Finally, send $(\mathsf{ServerSession}, \text{sid}, \text{ssid}\|\mathit{prfx}\|\text{req}\|\text{resp}\|h, \mathsf{C}, \mathsf{S})$ to $\mathcal{A}$ from $\mathcal{F}_{\text{aPAKE}}$ and mark $\mathtt{serverSession}[\text{ssid}]$ as "aPAKE active".

Active Session Attacks

12. Upon receiving $(\mathsf{TestPwd}, \text{sid}, \text{ssid}\|\mathit{prfx}\|\text{req}\|\text{resp}\|h, \mathsf{C}, rw^*)$ from $\mathcal{A}$ sent to $\mathcal{F}_{\text{aPAKE}}$, if $(\mathsf{C}, \mathsf{S}, i, \mathit{prfx}, \text{req}, \text{resp}, h) := \mathtt{clientSession}[\text{ssid}]$ is marked "aPAKE active":
    A. If $i = \mathsf{S}$, set $i := (k, 1)$ where $(\text{DB}, k) := \mathtt{file}[\text{sid}]$.
    B. If $\mathtt{trans}^* = 0$, set $pw^* := \perp$.
    C. If there exists an OPRF entry $F_{k^*,\mathtt{trans}^*}(pw^*) = s\|rw^*$ $(s \in \{0,1\}^{3\kappa})$, set $pw^* := x$, otherwise set $pw^* := \perp$.
    D. If $i = (k, 1)$, set $\text{DB}^* = \text{DB}$ and if either $\mathtt{password}[\text{sid}]$ is undefined or $pw^* \neq \mathtt{password}[\text{sid}]$, set $pw^* := \perp$.
    E. Otherwise, generate database $\text{DB}^* := \text{DB}(pw^*, (k^*, \mathtt{trans}^*))$ using $F_{k^*,\mathtt{trans}^*}(\widetilde{pw})$ for $\widetilde{pw} \in \Sigma_{pw^*}$. If any such value is undefined, set $pw^* := \perp$.
    F. If either $\text{resp} \neq \mathsf{PIR.Respond}(\text{DB}^*, \text{req})$ or $h \neq H(\text{DB}^*)$, set $pw^* := \perp$.
    G. Send $(\mathsf{TestPwd}, \text{sid}, \text{ssid}, \mathsf{C}, pw^*)$ to $\mathcal{F}_{\text{safPAKE}}$ and if $\mathcal{F}_{\text{safPAKE}}$ responds "wrong guess", Sim sets $pw^* := \perp$.
    H. If $\mathcal{F}_{\text{safPAKE}}$ responds with *pw*, set $\mathtt{password}[\text{sid}] := pw$, retrieve $rw := \mathtt{rw}[\text{sid}]$, $\{f_j\} := \mathtt{params}[\text{sid}]$, and run $\mathsf{ProgramDatabase}(\text{sid}, pw, rw, k, \{f_j\})$, mark $\mathtt{clientSession}[\text{ssid}]$ $\mathsf{compromised}$, and send "correct guess" to $\mathcal{A}$.
    I. Otherwise, send "wrong guess" to $\mathcal{A}$ from $\mathcal{F}_{\text{aPAKE}}$.
    J. In all cases, mark $\mathtt{clientSession}[\text{ssid}]$ "aPAKE key-ready".
13. Upon receiving $(\mathsf{TestPwd}, \text{sid}, \text{ssid}\|\mathit{prfx}\|\text{req}\|\text{resp}, \mathsf{S}, rw^*)$ from $\mathcal{A}$ sent to $\mathcal{F}_{\text{aPAKE}}$, if $(\mathsf{C}, \mathsf{S}, \mathsf{S}, \mathit{prfx}, \text{req}, \text{resp}, h) := \mathtt{serverSession}[\text{ssid}]$ is marked "aPAKE active".
    A. If $\mathtt{rw}[\text{sid}] = rw^*$, then send "correct guess" to $\mathcal{A}$.
    B. Otherwise, send "wrong guess" to $\mathcal{A}$.

C. In all cases, mark $\texttt{serverSession}[\mathrm{ssid}]$ "aPAKE key-ready".

14. Upon receiving $(\mathsf{Impersonate}, \mathrm{sid}, \mathrm{ssid}\|\mathit{prfx}\|\mathrm{req}\|\mathrm{resp}\|h)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathrm{aPAKE}}$, if $(\mathsf{C}, \mathsf{S}, i, \mathit{prfx}, \mathrm{req}, \mathrm{resp}, h) := \texttt{clientSession}[\mathrm{ssid}]$ is marked "aPAKE active":
    A. If S is not $\mathsf{compromised}$, ignore this query.
    B. If $(i = S$ or $i = (k, 1))$, $\mathrm{resp} = \mathsf{PIR.Respond}(\mathrm{req}, \mathrm{DB})$, and $h = H(\mathrm{DB})$, where $(\mathrm{DB}, k) := \texttt{file}[\mathrm{sid}]$, send $(\mathsf{Impersonate}, \mathrm{sid}, \mathrm{ssid})$ to $\mathcal{F}_{\mathrm{safPAKE}}$ and forward the response ("correct guess" or "wrong guess") to $\mathcal{A}$.
    C. If "correct guess" was sent, mark $\texttt{clientSession}[\mathrm{ssid}]$ $\mathsf{compromised}$.
    D. If "wrong guess" was sent, send $(\mathsf{TestPwd}, \mathrm{sid}, \mathrm{ssid}, \mathsf{C}, \perp)$ to $\mathcal{F}_{\mathrm{safPAKE}}$.
    E. In both cases, mark $\texttt{clientSession}[\mathrm{ssid}]$ "aPAKE key-ready".

Key Generation and Authentication

15. Upon receiving $(\mathsf{NewKey}, \mathrm{sid}, \mathrm{ssid}\|\mathit{prfx}_{\mathsf{C}}\|\mathrm{req}\|\mathrm{resp}\|h, \mathsf{C}, k^*)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathrm{aPAKE}}$, and $(\mathsf{C}, \mathsf{S}, i, \mathit{prfx}, \mathrm{req}, \mathrm{resp}, h) := \texttt{clientSession}[\mathrm{ssid}]$ is marked "aPAKE active" or "aPAKE key-ready":
    A. If there is a corresponding aPAKE session for the server (*i.e.*, $\texttt{serverSession}[\mathrm{ssid}] = (\mathsf{C}, \mathsf{S}, \mathsf{S}, \mathit{prfx}_{\mathsf{S}}, \mathrm{req}', \mathrm{resp}', h')$) and $i \neq \mathsf{S}$, $\mathit{prfx}_{\mathsf{C}} \neq \mathit{prfx}_{\mathsf{S}}$, $\mathrm{req} \neq \mathrm{req}'$, $\mathrm{resp} \neq \mathrm{resp}'$, or $h \neq h'$, send $(\mathsf{TestPwd}, \mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \perp)$ to $\mathcal{F}_{\mathrm{safPAKE}}$.
    B. Send $(\mathsf{NewKey}, \mathrm{sid}, \mathrm{ssid}, \mathsf{P}, k^*)$ to $\mathcal{F}_{\mathrm{safPAKE}}$ and mark $\texttt{clientSession}[\mathrm{ssid}]$ "aPAKE completed".
16. Upon receiving $(\mathsf{NewKey}, \mathrm{sid}, \mathrm{ssid}\|\mathit{prfx}_{\mathsf{C}}\|\mathrm{req}\|\mathrm{resp}\|h, \mathsf{S}, k^*)$ from $\mathcal{A}$ to $\mathcal{F}_{\mathrm{aPAKE}}$, and $(\mathsf{C}, \mathsf{S}, i, \mathit{prfx}, \mathrm{req}, \mathrm{resp}, h) := \texttt{serverSession}[\mathrm{ssid}]$ is marked "aPAKE active" or "aPAKE key-ready":
    A. If there is a corresponding aPAKE session for the client (*i.e.*, $\texttt{clientSession}[\mathrm{ssid}] = (\mathsf{C}, \mathsf{S}, \mathsf{S}, \mathit{prfx}_{\mathsf{C}}, \mathrm{req}', \mathrm{resp}', h')$) and $i \neq \mathsf{S}$, $\mathrm{req} \neq \mathrm{req}'$, $\mathrm{resp} \neq \mathrm{resp}'$, or $h \neq h'$, send $(\mathsf{TestPwd}, \mathrm{sid}, \mathrm{ssid}, \mathsf{S}, \perp)$ to $\mathcal{F}_{\mathrm{safPAKE}}$.
    B. Send $(\mathsf{NewKey}, \mathrm{sid}, \mathrm{ssid}, \mathsf{P}, k^*)$ to $\mathcal{F}_{\mathrm{safPAKE}}$ and mark $\texttt{serverSession}[\mathrm{ssid}]$ "aPAKE completed".
17. Upon receiving $(\mathsf{TestAbort}, \mathrm{sid}, \mathrm{ssid}\|\mathit{prfx}_{\mathsf{C}}\|\mathrm{req}\|\mathrm{resp}\|h, \mathsf{S})$ from $\mathcal{A}$ to $\mathcal{F}_{\mathrm{aPAKE}}$, and S has such an active aPAKE session:
    A. If there is a corresponding aPAKE session for the server (*i.e.*, $\texttt{serverSession}[\mathrm{ssid}] = (\mathsf{C}, \mathsf{S}, \mathsf{S}, \mathit{prfx}_{\mathsf{S}}, \mathrm{req}', \mathrm{resp}', h')$) and $i \neq \mathsf{S}$, $\mathit{prfx}_{\mathsf{C}} \neq \mathit{prfx}_{\mathsf{S}}$, $\mathrm{req} \neq \mathrm{req}'$, $\mathrm{resp} \neq \mathrm{resp}'$, or $h \neq h'$, send $(\mathsf{TestPwd}, \mathrm{sid}, \mathrm{ssid}, \mathsf{P}, \perp)$ to $\mathcal{F}_{\mathrm{safPAKE}}$.
    B. Send $(\mathsf{TestAbort}, \mathrm{sid}, \mathrm{ssid}, \mathsf{C})$ to $\mathcal{F}_{\mathrm{safPAKE}}$, relay the response ("success" or "abort") to $\mathcal{A}$ from $\mathcal{F}_{\mathrm{aPAKE}}$, and mark C's session record "aPAKE completed".
18. Upon receiving $(\mathsf{TestAbort}, \mathrm{sid}, \mathrm{ssid}\|\mathit{prfx}_{\mathsf{S}}\|\mathrm{req}\|\mathrm{resp}\|h, \mathsf{S})$ from $\mathcal{A}$ to $\mathcal{F}_{\mathrm{aPAKE}}$, and S has such an active aPAKE session:
    A. If there is a corresponding aPAKE session for the client (*i.e.*, $\texttt{clientSession}[\mathrm{ssid}] = (\mathsf{C}, \mathsf{S}, \mathsf{S}, \mathit{prfx}_{\mathsf{C}}, \mathrm{req}', \mathrm{resp}', h')$) and $i \neq \mathsf{S}$, $\mathrm{req} \neq \mathrm{req}'$, $\mathrm{resp} \neq \mathrm{resp}'$, or $h \neq h'$, send $(\mathsf{TestPwd}, \mathrm{sid}, \mathrm{ssid}, \mathsf{S}, \perp)$ to $\mathcal{F}_{\mathrm{safPAKE}}$.

B. Send $(\mathsf{TestAbort}, \mathrm{sid}, \mathrm{ssid}, \mathsf{C})$ to $\mathcal{F}_{\mathrm{safPAKE}}$, relay the response ("success" or "abort") to $\mathcal{A}$ from $\mathcal{F}_{\mathrm{aPAKE}}$, and mark S's session record "aPAKE completed".

Proof (Proof Outline). The proof follows a sequence of indistinguishable hybrids starting in the real world where the parties follow the protocol described in Sect. 6 and ending with the ideal world where interactions are simulated as per our simulator Sim in Sect. 7.1 which runs independently of the parties' passwords.

The first step of our hybrids introduces a dummy party for the client and the server which simply passes messages from the environment to their real counterparts. We then introduce a partial functionality $\mathcal{F}$ which sits between the dummy and real parties. This partial functionality begins by relaying all messages it receives to the simulator, and along the hybrids we build this partial functionality into $\mathcal{F}_{\mathrm{safPAKE}}$. In the beginning, the simulator receives $(\mathsf{StorePwdFile}, \mathrm{sid}, \mathsf{C}, pw)$ when the dummy server sends the message to $\mathcal{F}$ and receives $(\mathsf{ClientSession}, \mathrm{sid}, \mathrm{ssid}, \mathsf{S}, pw')$ when the dummy client sends the message to $\mathcal{F}$ and can run the real world protocol personally giving the keys to each of the parties. After setting up the conceptual functionality party, we give it all necessary interfaces, but provide it a partial NewKey interface which we eventually modify into the correct one found in $\mathcal{F}_{\mathrm{safPAKE}}$.

We then modify the client's code to run independently of the input password. This is accomplished by first replacing the client's PIR verification step. Instead of verifying the PIR response from the server using the client's PIR state, we extract the adversary's input to the aPAKE stage and compare it to the messages sent in the PIR stage (culminating in either a $\mathsf{TestPwd}$ query or an $\mathsf{Impersonate}$ query). Now, the client runs independently of its PIR state and by the client privacy property of PIR, we may replace the client's PIR request and OPRF request with a "dummy" request $(\mathrm{state}, \mathrm{req}) \leftarrow \mathsf{PIR.Request}(0^{\kappa})$. After this step, the client's code runs independently of the client password except for its preliminary query to $\mathcal{F}_{\mathrm{OPRF}}$.

We then modify the server's code to run independently of the input password unless it receives it from the functionality on a password testing query. To achieve this, we begin by distinguishing sub-sessions where the OPRF prefixes match from ones where they diverge. In the case of divergent sub-sessions, the simulator provides a single $\mathsf{TestPwd}$ ticket to the environment which can be redeemed to make a password guess. In such divergent sub-sessions, the simulator sends an $\mathsf{Interrupt}$ query and awaits a $\mathsf{PRFReceiveComplete}$ query which consumes a ticket. Upon consuming a ticket, the simulator extracts a guess from the OPRF query, sends a $\mathsf{TestPwd}$ query to the functionality, and aligns the aPAKE session accordingly. In the next set of hybrids, we replace the server's storage with a uniformly generated database independent of the server's password. We then align these databases whenever a ticket is redeemed, resulting in a correct password guess against the server, by programming the OPRF outputs of all passwords $\widetilde{pw} \in \Sigma_{pw}$ to match those used in the generation of the database.

Finally, we modify the partial functionality to align with $\mathcal{F}_{\text{safPAKE}}$. This is done, first, by replacing the intermediate NewKey interface in the partial simulator with the one found in $\mathcal{F}_{\text{safPAKE}}$. Once this is accomplished, we modify the functionality to no longer provide the simulator with the full ClientSession and ServerSession messages. As the simulator still relies on these messages, we make the final modifications to the simulator. First, we modify the server's execution to generate the dummy database upon receiving a successful StealPwdFile query or a ServerSession query. We then modify the client's code to no longer make its first OPRF query and for the simulator to instead send a PRFEval message to the environment without prompting. To finish the hybrids, we modify the partial functionality to no longer prompt the simulator by sending StorePwdFile.

We present a complete proof of theorem 1 in the full version of our paper. We additionally note, as a conjecture, that if the client leakage function $\ell_{\mathsf{C}}(pw, pw') =$"correct password" is replaced with the server's leakage function $\ell_{\mathsf{S}}(pw, pw') = pw$, then we may achieve a more efficient protocol which no longer requires the client to verify the server's PIR response. This also means that the PIR scheme would not need to have deterministic messages; however, this leakage function is no longer *minimal* in that it does not match what the honest parties learn at the end of the protocol.

8 Concrete Instantiations & Efficiency

We now discuss concrete instantiations for the components of our protocol (PIR scheme and aPAKE), and the resulting performance.
Choice of PIR Scheme. We consider two PIR schemes:

- **Trivial PIR:** The server sends the entire database to the client. Communication equals the size of the database.
- **FHE-PIR:** The client sends an FHE encryption of its query (tag) q to the server. The server homomorphically evaluates $P(q)$, where P is a polynomial obtained by interpolating $P(x) = DB[x]$. Communication is the size of one FHE-encrypted database element in each direction. Computation for the server is $O(|DB|)$ FHE operations.

Other PIR protocols between these extremes are possible. For example, it is possible to exponentially reduce the multiplicative depth of the FHE circuit by having the client send FHE encryptions of $q, q^2, q^4, q^8, \cdots$ instead of just an FHE encryption of q. This variant trades a $\log |DB|$ factor increase in communication for a significant reduction in FHE costs.

In our protocol, database items are symmetric-key encrypted ciphertexts, where the plaintext contains a PRF key and a small integer. Pessimistically, we consider each ciphertext to be 3κ bits, and we assume an additive overhead of $O(\kappa)$ to further encrypt the database item under FHE.
Round Complexity of safPAKE. Our OPRF protocol consists of two sequential rounds, with the client speaking first. Each OPRF message is a single group element. Afterwards, the rest of our protocol contains several sequential bottlenecks that affect the total round complexity:

- The client must obtain its OPRF output before making its PIR query. However, when using trivial-PIR, the client makes no query, so the server can just send its trivial-PIR message together with its final OPRF message.
- The client must obtain its PIR output before sending its first aPAKE message. However, if the server is first to speak in the aPAKE protocol, it may send the first aPAKE message along with its PIR response. Hence, our protocol favors aPAKE in which the server speaks first.

Overall, our protocol requires $2 + 2P + A - S$ rounds, where $P = 1$ for nontrivial PIR ($P = 0$ otherwise), the aPAKE has A sequential rounds, and $S = 1$ if the server speaks first in the aPAKE ($S = 0$ otherwise).

Among aPAKE protocols, the one of Hwang et al. [18] instantiated with the PAKE protocol of McQuoid et al. [26] is best suited to our application. It requires only one message from each party, and the server may be first to speak. Its communication cost is 7κ bits more than bare Diffie-Hellman, and it requires only one more exponentiation for the client than bare Diffie-Hellman.

Summary of safPAKE Costs. In Table 1 we show the concrete costs of our protocol when instantiated according to our recommendations here vs the naïve safPAKE described previously instantiated with OPAQUE [22] and reusing the OPRF stage. The reader should remember that the resultant protocol does not actually securely UC-realize the $\mathcal{F}_{\text{safPAKE}}$ functionality as there are no guarantees that the server uses the correct inputs. Importantly, when using a nontrivial PIR, the communication cost of our protocol is independent of the number of allowed password variants. When using a trivial PIR, communication is linear in the number of password variants, but the server's computation is constant.

Much of the computational cost comes from the client re-computing the PIR database. This involves $O(|DB|)$ PRF evaluations, each of which requires an exponentiation. In Table 1 we list our costs in terms of exponentiations E, hash-to-group operations H, and FHE multiplications F as these are our computational bottlenecks. Here, by n we mean the size of the typo-set Σ_{pw}—recall that this should be constant across all passwords.

Table 1. Cost evaluation of our safPAKE protocol. The "naïve" protocol is the insecure protocol described in Sect. 1.

	Naïve + [21]	Ours + Trivial PIR	Ours + FHE PIR
C Cost	$3n + 2$E, 1H	$(n+4)$E, $(n+3)$H	$(n+4)$E, $(n+3)$H, $O(n)$F
S Cost	$3n + 1$E	3E, 2H	3E, 2H, nF
Rounds	3	3	5
Communication	$(2n+2)\mathbb{G} + (5n+2)\kappa$	$(3n+9)\kappa + 4\mathbb{G}$	$O(\kappa) + 4\mathbb{G} + 9\kappa$
S Storage	$(5n+1)\kappa + 2n\mathbb{G}$	$(3n+1)\kappa$	$O(\kappa)n + (3n+1)\kappa$

Typo Policies: Our protocol is suitable for typo policies with a small or moderate number of allowable typos. Here we propose several such policies:

- **Facebook's policy**: Facebook allows the client to enter either a case-reversed password, a repeated first/last character password, or a case-reversed first letter password so, including the correct password, $\Sigma_{pw} = 5$ total password variants will be accepted[8].
- **Keyboard-adjacent character substitutions:** Suppose we allow at most k characters of the length-$\ell \geq k$ password to be substituted for characters that are adjacent on the keyboard, then $\Sigma_{pw} = \sum_{i \leq k} \binom{\ell}{i} 7^i$ total password variants will be accepted.
- **Consecutive characters transposed:** Suppose we allow at most k pairs of consecutive characters in the length-$\ell \geq k$ password to be transposed, then $\Sigma_{pw} = \sum_{i \leq k} \binom{\ell-1}{i}$ total password variants will be accepted.
- **Erroneous repeated or dropped characters:** Suppose we allow at most k characters in the length-$\ell \geq k$ password to be repeated or dropped, then $\Sigma_{pw} = \sum_{i \leq k} \binom{\ell}{i} 2^i$ total password variants will be accepted.

If we consider the cost of implementing Facebook's simple password-typo policy using our scheme, instantiated with the trivial PIR and a 256-bit elliptic curve group $\kappa = 128$, authentication would require 3 rounds, $((3 \cdot 5 + 9) \cdot 128 + 4 \cdot 256)/8 = 512$ bytes of communication, 12 exponentiations, 10 hash-to-curve operations, and 256 bytes of server storage.

9 Conclusion

Following in a line of study for fuzzy symmetric PAKE [13] and fuzzy asymmetric PAKE [14,27], we present a functionality for fuzzy *strong* asymmetric PAKE which affords the same strong guarantees of the saPAKE functionality from the literature [21] while allowing for arbitrary measures of similarity. We additionally provide a construction in the spirit of the aPAKE to saPAKE compiler due to Jarecki, Krawczyk, and Xu [21] and tPAKE due to Pongmorrakot and Chatterjee [27]. Our protocol realizing the $\mathcal{F}_{\text{safPAKE}}$ functionality for a common subclass of the similarity predicates considered by Erwig et al. [14] and can be built from a variant of the 2-hash-Diffie-Hellman OPRF herein called 3-hash-Diffie-Hellman, keyword private information retrieval, and existing aPAKE protocols. Additionally, we provide an incidental protocol for strong asymmetric normalization which we believe to be an independent interest for future study.

Our construction can achieve constant communication size in the size of the server's database, however has client computation equal to constructing the server's PIR database. For simple predicates such as first word capitalization or first-second letter transposition, this is reasonable, but for biometric applications may be too expensive for deployment. We believe that reducing the

[8] This count doesn't hold for all passwords, but will hold for the majority of passwords. *e.g.*, the empty password will have $\Sigma_{pw} = 1$.

computational burden for the client is possible and may pave the path to strong asymmetric biometric authentication. We leave this as an interesting line for future work.

References

1. Abdalla, M., Barbosa, M., Bradley, T., Jarecki, S., Katz, J., Xu, J.: Universally composable relaxed password authenticated key exchange. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12170, pp. 278–307. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56784-2_10
2. Abdalla, M., Barbosa, M., Katz, J., Loss, J., Xu, J.: Algebraic adversaries in the universal composability framework. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021. LNCS, vol. 13092, pp. 311–341. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92078-4_11
3. Abdalla, M., Haase, B., Hesse, J.: Security analysis of CPace. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021. LNCS, vol. 13093, pp. 711–741. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92068-5_24
4. Abdalla, M., Pointcheval, D.: Simple password-based encrypted key exchange protocols. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 191–208. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30574-3_14
5. Bellare, M., Namprempre, C., Pointcheval, D., Semanko, M.: The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. J. Cryptology **16**(3), 185–215 (2003)
6. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 139–155. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_11
7. Bellovin, S.M., Merritt, M.: Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In: Denning, D.E., Pyle, R., Ganesan, R., Sandhu, R.S., Ashby, V. (eds.) ACM CCS 93, pp. 244–250. ACM Press (1993)
8. Bootle, J., Faller, S., Hesse, J., Hostáková, K., Ottenhues, J.: Generalized fuzzy password-authenticated key exchange from error correcting codes. In: Guo, J., Steinfeld, R. (eds.) Advances in Cryptology – ASIACRYPT 2023. ASIACRYPT 2023. LNCS, vol. 14445, pp. 110–142. Springer, Singapore (2023). https://doi.org/10.1007/978-981-99-8742-9_4
9. Boyko, V., MacKenzie, P., Patel, S.: Provably secure password-authenticated key exchange using Diffie-Hellman. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 156–171. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_12
10. Bradley, T., Jarecki, S., Xu, J.: Strong asymmetric PAKE based on trapdoor CKEM. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 798–825. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_26
11. Canetti, R., Halevi, S., Katz, J., Lindell, Y., MacKenzie, P.: Universally composable password-based key exchange. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 404–421. Springer, Heidelberg (2005). https://doi.org/10.1007/11426639_24

12. Chatterjee, R., Athayle, A., Akhawe, D., Juels, A., Ristenpart, T.: pASSWORD tYPOS and how to correct them securely. In: 2016 IEEE Symposium on Security and Privacy, pp. 799–818. IEEE Computer Society Press (2016)
13. Dupont, P.-A., Hesse, J., Pointcheval, D., Reyzin, L., Yakoubov, S.: Fuzzy password-authenticated key exchange. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10822, pp. 393–424. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78372-7_13
14. Erwig, A., Hesse, J., Orlt, M., Riahi, S.: Fuzzy asymmetric password-authenticated key exchange. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020. LNCS, vol. 12492, pp. 761–784. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64834-3_26
15. Gentry, C., MacKenzie, P., Ramzan, Z.: A method for making password-based key exchange resilient to server compromise. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 142–159. Springer, Heidelberg (2006). https://doi.org/10.1007/11818175_9
16. Gu, Y., Jarecki, S., Krawczyk, H.: KHAPE: asymmetric PAKE from key-hiding key exchange. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12828, pp. 701–730. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84259-8_24
17. Herzberg, A.: Public-key infrastructure (PKI) (2022). https://uconn-my.sharepoint.com/:p:/g/personal/amir_herzberg_uconn_edu1/EX99yfXIWrtNqPo84-auJA4B-ZMLr4JHWZ-Zg83hvIv40w?e=fhOdGk
18. Hwang, J.Y., Jarecki, S., Kwon, T., Lee, J., Shin, J.S., Xu, J.: Round-reduced modular construction of asymmetric password-authenticated key exchange. In: Catalano, D., De Prisco, R. (eds.) SCN 2018. LNCS, vol. 11035, pp. 485–504. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98113-0_26
19. Jarecki, S., Kiayias, A., Krawczyk, H.: Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 233–253. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45608-8_13
20. Jarecki, S., Kiayias, A., Krawczyk, H., Xu, J.: TOPPSS: cost-minimal password-protected secret sharing based on threshold OPRF. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) ACNS 2017. LNCS, vol. 10355, pp. 39–58. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61204-1_3
21. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10822, pp. 456–486. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78372-7_15
22. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. Cryptology ePrint Archive, Report 2018/163 (2018)
23. Katz, J., Ostrovsky, R., Yung, M.: Efficient password-authenticated key exchange using human-memorable passwords. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 475–494. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44987-6_29
24. Katz, J., Vaikuntanathan, V.: Round-optimal password-based authenticated key exchange. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 293–310. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19571-6_18
25. Kaufman, C., Perlman, R.J.: PDM: A new strong password-based protocol. In: Wallach, D.S. (ed.) USENIX Security 2001. USENIX Association (2001)

26. McQuoid, I., Rosulek, M., Roy, L.: Minimal symmetric PAKE and 1-out-of-N OT from programmable-once public functions. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020, pp. 425–442. ACM Press (2020)
27. Pongmorrakot, T., Chatterjee, R.: tPAKE: typo-tolerant password-authenticated key exchange. In: Batina, L., Picek, S., Mondal, M. (eds.) SPACE 2020. LNCS, vol. 12586, pp. 3–24. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-66626-2_1
28. Protalinski, E.: Facebook passwords are not case sensitive (2011). https://www.zdnet.com/article/facebook-passwords-are-not-case-sensitive-update/
29. Santos, B.F.D., Gu, Y., Jarecki, S., Krawczyk, H.: Asymmetric PAKE with low computation and communication. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II, volume 13276 of LNCS, pp. 127–156. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-07085-3_5
30. Shoup, V.: Security analysis of *SPAKE2+*. In: Pass, R., Pietrzak, K. (eds.) TCC 2020. LNCS, vol. 12552, pp. 31–60. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64381-2_2