Fuzzy Asymmetric Password-Authenticated Key Exchange

Andreas Erwig[1], Julia Hesse[2], Maximilian Orlt[1], and Siavash Riahi[1]

[1]Technische Universität Darmstadt, Germany
[2]IBM Research - Zurich, Switzerland

`{andreas.erwig, maximilian.orlt, siavash.riahi}@tu-darmstadt.de, jhs@zurich.ibm.com`

August 16, 2020

Abstract

Password-Authenticated Key Exchange (PAKE) lets users with passwords exchange a cryptographic key. There have been two variants of PAKE which make it more applicable to real-world scenarios:

- *Asymmetric* PAKE (aPAKE), which aims at protecting a client's password even if the authentication server is untrusted, and
- *Fuzzy* PAKE (fPAKE), which enables key agreement even if passwords of users are noisy, but "close enough".

Supporting fuzzy password matches eases the use of higher entropy passwords and enables using biometrics and environmental readings (both of which are naturally noisy).

Until now, both variants of PAKE have been considered only in separation. In this paper, we consider both of them simultaneously. We introduce the notion of *Fuzzy Asymmetric PAKE* (fuzzy aPAKE), which protects against untrusted servers *and* supports noisy passwords. We formulate our new notion in the Universal Composability framework of Canetti (FOCS'01), which is the preferred model for password-based primitives. We then show that fuzzy aPAKE can be obtained from oblivious transfer and some variant of robust secret sharing (Cramer et al, EC'15). We achieve security against malicious parties while avoiding expensive tools such as non-interactive zero-knowledge proofs. Our construction is round-optimal, with message and password file sizes that are independent of the schemes error tolerance.

1 Introduction

In a world of watches interacting with smartphones and our water kettle negotiating with the blinds in our house, communicating devices are ubiquitous. Developments in user-centric technology are rapid, and they call for authentication methods that conveniently work with, e.g., biometric scans, human-memorable passwords or fingerprints derived from environmental readings.

Password-authenticated Key Exchange (PAKE) protocols [BM92, BPR00, BMP00, KOY01, GL03, KV11, CDVW12, BBC+13] are the cryptographic answer to this need. They solve the problem of establishing a secure communication channel between two users who share nothing but a low-entropy string, often simply called *password*. Two interesting variants of PAKE protocols that are known from the literature are *asymmetric* PAKE [BM93, GMR06, JKX18, BJX19] which aims at protecting the user's password even if his password file at some server is stolen, and *fuzzy* PAKE [DHP+18] which can tolerate some errors in the password. The former is useful in settings where authentication servers store thousands of user accounts and the server cannot be fully trusted. The latter introduces a usability aspect to PAKE protocols used by humans trying to remember passwords exactly. Furthermore, fuzzy PAKE broadens applicability of PAKE to the fuzzy setting and thereby allows using environmental readings or biometrics as passwords.

This work is the first to consider a combination of both PAKE variants. Namely, we introduce the notion of *fuzzy asymmetric PAKE* (fuzzy aPAKE). This new primitive allows a client and an untrusted server to authenticate to each other using a password, and both parties are guaranteed to derive the same cryptographic key as long as their passwords are within some predefined distance (in some predefined metric). Consider a client authenticating to a server using his fingerprint scan. In this setting, asymmetric PAKE protocols would not work since subsequent scans do not match exactly. Fuzzy PAKE, on the other hand, would require the server to store the fingerprint (or at least some template of it that uniquely identifies the person) in the clear, which is unacceptable for sensitive and ephemeral personal data that is biometrics. Fuzzy asymmetric PAKE, as introduced in this paper, is the only known cryptographic solution that applies to this setting: it works with fuzzy authentication data *and* does not reveal this authentication data to the server.

Why is this hard? Given that there is a lot of literature about both asymmetric PAKE and fuzzy cryptography, one could ask whether existing techniques could be used to obtain fuzzy aPAKE. As explained already in [DHP$^+$18], techniques from fuzzy cryptography such as information reconciliation [BBR88] or fuzzy extractors [DRS04] cannot be used with passwords of low entropy. Essentially, these techniques lose several bits of their inputs, which is acceptable when inputs have high entropy, but devastating in case of passwords.

Looking at techniques for asymmetric PAKE, all of them require some kind of password hardening such as hashing [GMR06, HL19, PW17], applying a PRF [JKX18] or a hash proof system [BJX19]. Unfortunately, such functions destroy all notions of closeness of their inputs by design. Further, it is unclear how to define a fuzzy version of, e.g., an oblivious PRF as used in [JKX18] that is not simply a constant function. While such definitions exist for "fuzzy" cryptographic hashing (e.g., robust property-preserving hashing [BLV19]), these functions either do not provide useful error correction or already their description leaks too much information about the password of the client. Overall, there seems to be no candidate asymmetric PAKE which can be made fuzzy.

Regarding more naive approaches, it is tempting to try to apply generic techniques for multi-party computation to obtain a fuzzy PAKE such as garbled circuits [Yao86]. The circuit would be created w.r.t some function of the password $h \leftarrow H(\mathsf{pw})$. The user's input would be pw'. Now the circuit finds all passwords close enough to pw' and outputs the shared key if one of these passwords yield h. Despite the inefficiency of this approach, it is unclear how to actually write down the circuit. As shown in [Hes19], h needs to be the output of some idealized assumption such as a programmable random oracle, and thus has no representation as a circuit.

Our contributions In this paper, we give the first formal definition of fuzzy asymmetric PAKE. Our definition is in the Universal Composability framework of Canetti [Can01], which is the preferred model for PAKE protocols (cf., e.g., [JKX18] for reasons). Essentially, we take the aPAKE functionality from [GMR06] (in a revised version due to [Hes19]) and equip it with fuzzy password matching (taken from the fuzzy PAKE functionality $\mathcal{F}_{\mathsf{fPAKE}}$ from [DHP$^+$18]). Our resulting functionality $\mathcal{F}_{\mathsf{faPAKE}}$ is flexible in two ways: it can be optionally equipped with a mutual key confirmation (often called *explicit authentication*), and, just as $\mathcal{F}_{\mathsf{fPAKE}}$, $\mathcal{F}_{\mathsf{faPAKE}}$ can be parametrized with arbitrary metrics for distance, arbitrary thresholds and arbitrary adversarial leakage. Thus, our model is suitable to analyze protocols for a wide range of applications, from tolerating only few language-specific typos in passwords [CWP$^+$17] to usage of noisy biometric scans of few thousand bits length.

We then give two constructions for fuzzy asymmetric PAKE. Our first construction Π_{faPAKE} uses error-correcting codes (ECC)[1] and oblivious transfer (OT) as efficient building blocks. Π_{faPAKE} works for Hamming

[1]More precisely, we use a variant of *Robust Secret Sharing*, which can be instantiated with some class of error-correcting codes. However, since most readers are presumably more familiar with the latter, we describe our constructions in terms of codes.

distance and can correct $\mathcal{O}(\log(n))$ errors in n-bit passwords. Let us now give more details on Π_{faPAKE}.

The idea of our protocol is to first encode a cryptographic key and store it at the server, in a file together with random values to hide the codeword. The exact position of the codeword in the file is dictated by the password. A client holding a close enough password is thus able to retrieve almost the whole codeword correctly and can thus decode the session key given the error correction capabilities of the encoding. An attacker stealing the password file, however, cannot simply decode since the file contains too much randomness. To remove this randomness, he is bound to decode subsets of the file until he finds two subsets which decode to the same session key. Since decoding can be assumed to be as expensive as hashing, the effort of an off-line dictionary attack on the password file follows from a purely combinatorial argument on the parameters of the scheme (i.e., password size and error correction threshold).

To bound the client to one password guess per run of the protocol (which is the common security requirement for PAKE), we employ an n-times 1-out-of-2 OT scheme. Each OT lets the client choose either the true or the random part of the codeword for each of the n password bits (here we assume that the codeword is from $\mathbb{F}^n$ for some large field $\mathbb{F}$). Further, we apply randomization techniques to keep a client from collecting parts of the password file over several runs of the protocol.

A plus of our protocol is that it elegantly circumvents usage of expensive techniques such as non-interactive zero-knowledge proofs to ensure security against a malicious server. Indeed, a malicious server could make the client reconstruct the session key regardless of her password by entering only the true codeword in the OT. Such attacks would be devastating in applications where the client uses the session key to encrypt her secrets and sends them to the bogus server. Thus, the client needs a means to check correct behavior of the server. We achieve this by letting the server send his transcript of the current protocol run (e.g., the full password file) to the client, symmetrically encrypted with the session key. The client decrypts and checks whether the server executed the protocol with a password close enough to his own. Crucially, a corrupted client can only decrypt (and thus learn the server's secrets) if he holds a close enough password, since otherwise he will not know the encryption key.

Our proof of security is in the UC model and thus our protocol features composability guarantees and security even in the presence of adversarially-chosen passwords. As shown in [Hes19], strong idealized assumptions are necessary in order to achieve security in the UC model in case of asymmetric PAKE protocols. The reason lies in the adaptive nature of a server compromise attack (an adversary stealing the password file), against which our fuzzy version of asymmetric PAKE should also provide some protection. And indeed, our proof is in the generic group model and additionally requires encryption to be modeled as an ideal cipher. Both assumptions provide our simulator with the power to monitor off-line password guesses (*observability*) of the environment as well as to adjust a password file to contain a specific password even after having revealed the file (*programmability*)[2]. As a technicality, usage of the generic group model requires the client to perform decoding *in the exponent*. We give an example of a code that is decodable in the exponent.

Our second construction Π_{transf} is a "naive" approach of building fuzzy aPAKE from aPAKE. Namely, for a given pw, a server could simply store a list of, say, k hashes $H(\mathsf{pw}')$ for all pw' close enough to pw. Then, client and server execute k times an aPAKE protocol, with the client entering the same password every time and the server entering all hashes one by one. The fully secure protocol would need to protect against malicious behavior, e.g., by having both parties prove correct behavior. Unfortunately, this approach has two drawbacks. First, it does not scale asymptotically and has huge password files and communication overhead depending not only on the fuzziness threshold but also on the size of the password. Second, we show that Π_{transf} cannot be considered a secure fuzzy aPAKE, but has slightly weaker security guarantees.

On the plus side, Π_{transf} is already practical (and sufficiently secure) for applications where only few passwords should let the client pass. Facebook's authentication protocol, for example, is reported to correct capitalization of the first letter [Ale15], resulting in only two hashes to be stored in the password file. As analyzed in [CAA+16, CWP+17], correcting few common typographical mistakes as, e.g., accidental caps lock, increases usability significantly more than it decreases security. For such applications, our protocol

[2]We mention that already the fuzzy PAKE construction for Hamming distance from [DHP+18] relies on both the ideal cipher and random oracle model. Usage of the generic group model (together with a random oracle) has been recently shown useful in constructing strongly secure aPAKEs [BJX19].

Π_{transf} is a good choice.

1.1 Roadmap

In Section 2 we give a definition of our main building blocks, error-correcting codes which are decodable in the exponent. In Section 3, we provide the formal definition of fuzzy aPAKE and discuss the design of our functionality. Our fuzzy aPAKE protocol can be found in Section 4. Our naive approach of building faPAKE from aPAKE can be found in Section 5. Efficiency is considered in Section 6.

2 Preliminaries

2.1 Robust Secret Sharing in the exponent

An l-out-of-n secret sharing scheme allows to share a secret value s into n shares $(s_1, \cdots, s_n)$ in such a way that given at least l of these shares, the secret can be reconstructed. Simultaneously, any tuple of shares smaller than l is distributed independently of s. *Robust secret sharing* (RSS) [CDD+15] improves upon secret sharing schemes in the presence of malicious shares. Intuitively, an $(n, l-1, r)_q$-RSS is an l-out-of-n secret sharing scheme which allows the presence of up to $n-r$ corrupted shares. In detail the reconstruction of the secret is reliable for an n-tuple input $(\hat{s}_1, \cdots, \hat{s}_n)$ of r different secret shares s_i and $n-r$ random values a_i even if the positions of the correct shares are unknown.

We recall the definition of RSS as stated in [DHP+18]. For a vector $c \in \mathbb{F}_q^n$ and a set $A \subseteq [n]$, we denote with c_A the projection $\mathbb{F}_q^n \to \mathbb{F}_q^{|A|}$, i.e., the sub-vector $(c_i)_{i \in A}$.

Definition 2.1. *Let $\lambda \in \mathbb{N}$, q a λ-bit prime, $\mathbb{F}_q$ a finite field and $n, l, r \in \mathbb{N}$ with $l < r \leq n$. An $(n, l, r)_q$ robust secret sharing scheme (RSS) consists of two probabilistic algorithms* $\mathsf{Share} : \mathbb{F}_q \to \mathbb{F}_q^n$ *and* $\mathsf{Rec} : \mathbb{F}_q^n \to \mathbb{F}_q$ *with the following properties:*

- l-privacy*: for any $s, s' \in \mathbb{F}_q, A \subset [n]$ with $|A| \leq l$, the projections c_A of $c \xleftarrow{\$} \mathsf{Share}(s)$ and c'_A of $c' \xleftarrow{\$} \mathsf{Share}(s')$ are identically distributed.*
- r-robustness*: for any $s \in \mathbb{F}_q, A \subset [n]$ with $|A| \geq r$, any c output by $\mathsf{Share}(s)$, and any $\tilde{c}$ such that $c_A = \tilde{c}_A$, it holds that $\mathsf{Rec}(\tilde{c}) = s$.*

We now introduce a variant of RSS which produces shares that are hidden in the exponent of some group G, and which features a reconstruction algorithm that can handle shares in the exponent. At the same time we sacrifice absolute correctness of Rec and allow for a negligible error in the definition of robustness.

Definition 2.2 (Robust Secret Sharing in the Exponent)**.** *Let $\lambda \in \mathbb{N}$, q a λ-bit prime, $\mathbb{F}_q$ a finite field and $n, l, r \in \mathbb{N}$ with $l < r \leq n$. Let $RSS = (\mathsf{Share}', \mathsf{Rec}')$ be a $(n, l, r)_q$ robust secret sharing scheme and let $G = \langle g \rangle$ be a cyclic group of prime order q. An $(n, l, r)_q$ robust secret sharing scheme in the exponent (RSSExp) with respect to G consists of two probabilistic algorithms* $\mathsf{Share} : \mathbb{F}_q \to G^n$ *and* $\mathsf{Rec} : G^n \to G$ *which are defined as follows:*

- $\mathsf{Share}(s)$: *On input a secret value $s \leftarrow \mathbb{F}_q$, obtain secret shares $(s_1, \cdots, s_n) \leftarrow \mathsf{Share}'(s)$ and output $(g^{s_1}, \cdots, g^{s_n})$.*
- $\mathsf{Rec}(g^{\hat{s}_1}, \cdots, g^{\hat{s}_n})$: *On input n group elements, this algorithm outputs $g^{\hat{s}}$, where $\hat{s} \leftarrow \mathsf{Rec}'(\hat{s}_1, \cdots, \hat{s}_n)$.*

Further, an (n, l, r)-RSSExp scheme fulfills the following properties:

- *l-privacy: as in Definition 2.1.*
- *r-robustness: for any $s \in \mathbb{F}_q, A \subset [n]$ with $|A| \geq r$, any c output by* $\mathsf{Share}(s)$, *and any $\tilde{c}$ such that $c_A = \tilde{c}_A$, it holds that* $\mathsf{Rec}(\tilde{c}) = g^s$ *with overwhelming probability in n.*

Note that any (n, l, r)-RSSExp scheme trivially fulfills the l-privacy property. In the next part of this section we show how to achieve r-robustness.

Instantiations of RSSExp In [DHP⁺18], it is shown how to construct an RSS scheme from any *maximum distance separable* (MDS) code. An $(n+1,k)_q$ MDS code is a linear q-ary code of length n and rank k, which can correct up to $\lfloor (n-k+1)/2 \rfloor$ errors. We refer to [Rot06] for a more in depth introduction to linear codes.

Concretely, [DHP⁺18] propose to use Reed-Solomon codes, which are closely related to Shamir's secret sharing scheme [MS81]. In general, we are not aware of any RSS scheme that is not also an MDS code. For this reason, we focus now on decoding algorithms of linear codes.

Which decoding alorithm works also in the exponent? In the following Lemma we show that it is possible to build an $(n, l-1, l+t, g)$-RSSExp scheme from an l-out-of-$(l+2t)$ Shamir's secret sharing scheme.

Lemma 2.3. *Let* $n, l \in \mathbb{N}$ *and* $(\mathsf{Share}', \mathsf{Rec}')$ *be an* l*-out-of-*n *Shamir's secret sharing scheme with* $n = l + 2t$ *for some* t *and* $t \cdot l = \mathcal{O}(n \log n)$*,* $G = \langle g \rangle$ *a cyclic group of order* q*. Further let* Share *be the algorithm that outputs* $g^{\mathsf{Share}'(s)}$ *on input* $s \in \mathbb{F}_q$*. Then there exists an algorithm* Rec *using* $poly(n) \cdot \mathcal{O}(\log q)$ *group operations such that* $(\mathsf{Share}, \mathsf{Rec})$ *is an* $(n, l-1, l+t)$*-RSSExp scheme with respect to* G*.*

Proof. $(l-1)$-privacy of l-out-of-n Shamir's secret sharing scheme is shown in [DHP⁺18], Lemma 5, and can be directly applied to the case where shares are lifted to the exponent of some group. Let Rec be the "unique decoding by randomized enumeration" algorithm defined by Canetti and Goldwasser [CG99] (essentially, the algorithm decodes random subsets of shares until it finds redundancy), but applied to shares in the exponent using, e.g., Lagrange interpolation. Peikert [Pei06] shows in his Proposition 2.1 that, if $t < (n+1-l)/2$ (i.e., the number of errors allows for unique decoding) and $t \cdot l = \mathcal{O}(n \log n)$, then Rec succeeds with overwhelming probability in n and requires $poly(n) \cdot \mathcal{O}(\log q)$ group operations. Since $n = l+2t$, it holds that $t < (n+1-l)/2$ and hence $(l+t)$-robustness is achieved. □

3 Security Model

We now present our security definition for asymmetric fuzzy password authenticated key exchange (Π_{faPAKE}). Our functionality combines the fuzzy PAKE functionality $\mathcal{F}_{\mathsf{fPAKE}}$ from [DHP⁺18] with the asymmetric PAKE functionality $\mathcal{F}_{\mathsf{apwKE}}$ [GMR06] (with revisions due to [Hes19]). In order to capture the notion of fuzziness in our model, we say that a key exchange using passwords pw and pw' is successful if $d(pw, pw') \leq \delta$, where d is an arbitrary distance function and δ a fixed threshold. $\mathcal{F}_{\mathsf{fPAKE}}$ can be parametrized with arbitrary functions $hdist()$ such as Hamming distance or edit distance.

Roles: In this work we consider an asymmetric setting, namely a client $\mathcal{P_C}$ and a server $\mathcal{P_S}$. Each party executes different code. In this setting $\mathcal{P_C}$ uses a password pw while $\mathcal{P_S}$ has access to some value denoted by FILE, which is generated from a password pw' but does not immediately reveal pw'. The goal of $\mathcal{P_C}$ is convincing $\mathcal{P_S}$ that $d(pw, pw') \leq \delta$, while $\mathcal{P_S}$ only has access to FILE (and does not have access to pw').

Modeling Adversarial Capabilities: The standard security requirement for PAKE is that an attacker is bound to one password guessing attempt per run of the protocol. This resistance to off-line dictionary attacks is also featured by our functionality $\mathcal{F}_{\mathsf{faPAKE}}$ via the TestPwd interface that can be called by the adversary only once per session. Since we are in the setting of asymmetric PAKE, however, the adversary can also gain access to the password file FILE by compromising the server. Such a compromise is essentially a corruption query with the effect that a part of the internal state of the server is leaked to the adversary. However, opposed to standard corruption, the adversary is not allowed to control the party or modify its internal state. $\mathcal{F}_{\mathsf{faPAKE}}$ provides an interface for server compromise named StealPwdFile. As a consequence of such a query (which, as natural for corruption queries, can only be asked by the adversary upon getting instructions from the environment), a dictionary attack becomes possible. Such an attack is reflected in $\mathcal{F}_{\mathsf{faPAKE}}$ by the OfflineTestPwd interface, which allows an unbounded number of password guesses. Accounting for protocols that allow precomputation of, e.g., hash tables of the form $H(\mathsf{pw})$, $\mathcal{F}_{\mathsf{faPAKE}}$ accepts

OfflineTestPwd queries already *before* StealPwdFile was issued. $\mathcal{F}_{\mathsf{faPAKE}}$ silently stores these guesses in the form of (offline, pw) records. Upon StealPwdFile, $\mathcal{F}_{\mathsf{faPAKE}}$ sends the client's pw_C to the adversary in case a record (offline, pw_C) exists. This models the fact that the adversary learns the client's password from his precomputated values only upon learning the password file, i.e., compromising the server[3]. Besides offline password guesses, the adversary can use file of the compromised server to run a key exchange session with the user. This is captured within the Impersonate interface.

All these interfaces were already present in aPAKE functionalities in the literature. The key difference of $\mathcal{F}_{\mathsf{faPAKE}}$ is now that all these interfaces apply fuzzy matching when it comes to comparing passwords. Namely, $\mathcal{F}_{\mathsf{faPAKE}}$ is parametrized with two thresholds δ and γ. δ is the "success threshold", for which it is guaranteed that passwords within distance δ enable a successful key exchange. On the other hand, γ can be seen as the "security threshold", with $\gamma \geq \delta$. Guessing a password within range γ does not enable the adversary to successfully exchange a key, but it might provide him with more information than just "wrong guess". Following [DHP+18], we enable weakenings of $\mathcal{F}_{\mathsf{faPAKE}}$ in terms of leakage from adversarial interfaces (cf. Figure 2). Here, the adversary, in addition to learning whether or not his password guess was close enough, is provided with the output of different leakage functions L_c, L_m and L_f. Essentially, he learns $L_c(\mathsf{pw}, \mathsf{pw}')$ if his guess was within range δ of the other password, L_m if it was within range $\gamma > \delta$ and L_f if it was further away than γ. $\mathcal{F}_{\mathsf{faPAKE}}$ can be instantiated with any thresholds γ, δ and arbitrary functions L_c, L_m, L_f. Looking ahead, the additional threshold γ enables us to prove security of constructions using building blocks such as error-correcting codes, which come with a "gray zone" where reliable error correction is not possible, but also the encoded secret is not information-theoretically hidden. While guessing a password in this gray zone does not enable an attacker to reliably compute the same password as the client, security is still considered to be compromised since some information about the honest party's password (and thus her key) might be leaked. To keep the notion flexible, we allow describing the amount of leakage with $L_m(\cdot, \cdot)$ and mark the record `compromised` to model partial leakage of the key.

Naturally, one would aim for δ and γ to be close, where $\delta = \gamma$ offers optimal security guarantees in terms of no special adversarial leakage if passwords are only $\delta + 1$ apart (an equivalent formulation would be to set $L_m = L_f$). $\mathcal{F}_{\mathsf{faPAKE}}$ is strongest if $L_f = L_m = L_c = \perp$. Below we provide examples of nontrivial leakage functions, verbatim taken from [DHP+18].

Since in a fuzzy aPAKE protocol the password file stored at the server needs to allow for fuzzy matching, files are required to store the password in a structured or algebraic form. An adversary stealing the file could now attempt to alter the file to contain a different (still unknown) password. This kind of attack does not seem to constitute a real threat, since the attacker basically just destroyed the file and cannot use it anymore to impersonate the server towards the corresponding client. To allow for efficient protocols, we therefore choose to incorporate malleability of password files into our functionality $\mathcal{F}_{\mathsf{faPAKE}}$ by allowing the adversary to present a function f within an Impersonate query. The impersonation attack is then carried out with $f(pw)$ instead of pw, where pw denotes the server's password.

Figure 1 depicts $\mathcal{F}_{\mathsf{faPAKE}}$ with the set of leakage functions from the second example below, namely leaking whether the password is close enough to derive a common cryptographic key.

[3]Recent PAKE protocols [JKX18, BJX19] have offered resistance against so-called precomputation attacks, where an attacker should not be able to pre-compute any values that can be used in the dictionary attack. Our protocols do not offer such guarantees.

The functionality $\mathcal{F}_{\mathsf{faPAKE}}$ is parameterized by a security parameter λ and tolerances $\delta \leq \gamma$. It interacts with an adversary $\mathcal{S}$ and a client and a server party $\mathcal{P} \in \{\mathcal{P}_C, \mathcal{P}_S\}$ via the following queries:

Password Registration

- On $(\textsc{StorePwdFile}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw})$ from $\mathcal{P}_S$, if this is the first $\textsc{StorePwdFile}$ message, record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ and mark it `uncompromised`.

Stealing Password Data

- On $\boxed{(\textsc{StealPwdFile}, \mathsf{sid})}$ from $\mathcal{S}$, if there is no record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$, return "no password file" to $\mathcal{S}$. Otherwise, if the record is marked `uncompromised`, mark it `compromised`; regardless, for all records $(\textsc{offline}, \mathsf{pw}')$ set $d \leftarrow d(\mathsf{pw}, \mathsf{pw}')$ and do:
  - If $d \leq \delta$, send ("correct guess", pw') to $\mathcal{S}$;

  If no such pw' is recorded, return "password file stolen" to $\mathcal{S}$.

- On $(\textsc{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}')$ from $\mathcal{S}$, do:
  - If there is a record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ marked `compromised`, then set $d \leftarrow d(\mathsf{pw}, \mathsf{pw}')$ and do:
    * If $d \leq \delta$, mark record `compromised` and send "correct guess" to $\mathcal{S}$;
    * If $d > \delta$, mark record `interrupted` and send "wrong guess" to $\mathcal{S}$.
  - Else, record $(\textsc{offline}, \mathsf{pw}')$

Password Authentication

- On $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}')$ from $\mathcal{P}_C$, send $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S)$ to $\mathcal{S}$. Also, if this is the first $\textsc{UsrSession}$ message for ssid, record $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$ and mark it `fresh`.

- On $(\textsc{SrvSession}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{P}_S$, retrieve $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ and send $(\textsc{SrvSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S)$ to $\mathcal{S}$. Also, if this is the first $\textsc{SrvSession}$ message for ssid, record $(\mathsf{ssid}, \mathcal{P}_S, \mathcal{P}_C, \mathsf{pw})$ and mark it `fresh`.

Active Session Attacks

- On $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{pw}')$ from $\mathcal{S}$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ marked `fresh`, then set $d \leftarrow d(\mathsf{pw}, \mathsf{pw}')$ and do:
  - If $d \leq \delta$, mark record `compromised` and send "correct guess" to $\mathcal{S}$;
  - If $d > \delta$, mark record `interrupted` and send "wrong guess" to $\mathcal{S}$.

- On $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid}, f)$ from $\mathcal{S}$, if there is a record $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ marked `fresh` and a record $(\textsc{file}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw}')$ marked `compromised`, then set $d \leftarrow d(\mathsf{pw}, f(\mathsf{pw}'))$ and do:
  - If $d \leq \delta$, mark record `compromised` and send "correct guess" to $\mathcal{S}$;
  - If $d > \delta$, mark record `interrupted` and send "wrong guess" to $\mathcal{S}$.

Key Generation and Implicit Authentication

- On $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{k})$ from $\mathcal{S}$ where $|\mathsf{k}| = \lambda$ or $\mathsf{k} = \perp$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ not marked `completed`, do:
  - If the record is marked `compromised`, or either $\mathcal{P}$ or $\mathcal{P}'$ is corrupted, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k})$ to $\mathcal{P}$.
  - Else if the record is marked `fresh`, $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k}')$ was sent to $\mathcal{P}'$, and at that time there was a record $(\mathsf{ssid}, \mathcal{P}', \mathcal{P}, \mathsf{pw})$ with $d(\mathsf{pw}, \mathsf{pw}') \leq \delta$ marked `fresh`, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k}')$ to $\mathcal{P}$.
  - Else if $\mathsf{k} \neq \perp$, the record is marked `fresh`, $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k}')$ was sent to $\mathcal{P}'$, and at that time there was a record $(\mathsf{ssid}, \mathcal{P}', \mathcal{P}, \mathsf{pw})$ with $d(\mathsf{pw}, \mathsf{pw}') \leq \delta$ marked `fresh`, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k}')$ to $\mathcal{P}$.
  - Else, pick $\mathsf{k}'' \xleftarrow{\$} \{0,1\}^\lambda$ and send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k}'')$ to $\mathcal{P}$.

  Finally, mark $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ `completed`.

Figure 1: Ideal functionality $\mathcal{F}_{\mathsf{faPAKE}}$. Framed queries can only be asked upon getting instructions from $\mathcal{Z}$.

- If $d \leq \delta$, mark the record `compromised` and reply to $\mathcal{S}$ with $L_c(\mathsf{pw}, \mathsf{pw}')$;
- If $\delta < d \leq \gamma$, mark the record `compromised` and reply to $\mathcal{S}$ with $L_m(\mathsf{pw}, \mathsf{pw}')$;
- If $\gamma < d$, mark the record `interrupted` and reply to $\mathcal{S}$ with $L_f(\mathsf{pw}, \mathsf{pw}')$.

Figure 2: Modified distance checks to allow for different leakage to be used in TestPwd, OfflineTestPwd, Impersonate and StealPwdFile. In StealPwdFile, record marking is skipped.

Examples of leakage functions.

1. *No leakage.* The strongest option is to provide no feedback at all to the adversary. We define $\mathcal{F}^N_{\mathsf{faPAKE}}$ to be the functionality described in Figure 1, except that TestPwd, Impersonate, OfflineTestPwd and StealPwdFile use the check depicted in Figure 2 with
$$L^N_c(\mathsf{pw}, \mathsf{pw}') = L^N_m(\mathsf{pw}, \mathsf{pw}') = L^N_f(\mathsf{pw}, \mathsf{pw}') = \perp .$$

2. *Correctness of guess.* The basic functionality $\mathcal{F}_{\mathsf{faPAKE}}$, described in Figure 1, leaks the correctness of the adversary's guess. That is, in the language of Figure 2,
$$\begin{aligned} & L_c(\mathsf{pw}, \mathsf{pw}') = \text{"correct guess"}, \\ \text{and} \qquad & L_m(\mathsf{pw}, \mathsf{pw}') = L_f(\mathsf{pw}, \mathsf{pw}') = \text{"wrong guess"}. \end{aligned}$$

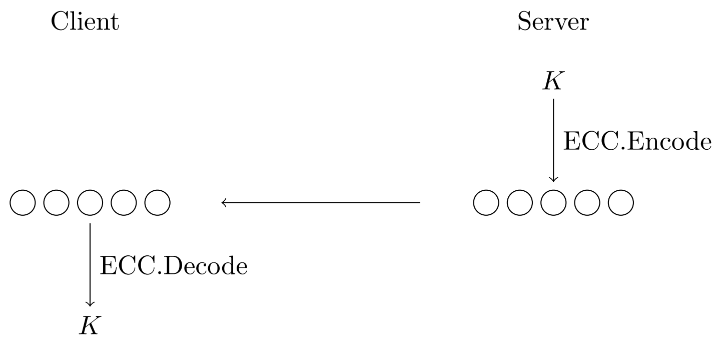
3. *Matching positions ("mask").* Assume the two passwords are strings of length n over some finite alphabet, with the jth character of the string pw denoted by $\mathsf{pw}[j]$. We define $\mathcal{F}^M_{\mathsf{faPAKE}}$ to be the functionality described in Figure 1, except that TestPwd, Impersonate, OfflineTestPwd and StealPwdFile use the check depicted in Figure 2, with L_c and L_m that leak the indices at which the guessed password differs from the actual one when the guess is close enough (we will call this leakage the *mask* of the passwords). That is,
$$\begin{aligned} & L^M_c(\mathsf{pw}, \mathsf{pw}') = (\{j \text{ s.t. } \mathsf{pw}[j] = \mathsf{pw}'[j]\}, \text{"correct guess"}), \\ & L^M_m(\mathsf{pw}, \mathsf{pw}') = (\{j \text{ s.t. } \mathsf{pw}[j] = \mathsf{pw}'[j]\}, \text{"wrong guess"}) \\ \text{and} \qquad & L^M_f(\mathsf{pw}, \mathsf{pw}') = \text{"wrong guess"}. \end{aligned}$$

4. *Full password.* The weakest definition — or the strongest leakage — reveals the entire actual password to the adversary *if the password guess is close enough.* We define $\mathcal{F}^P_{\mathsf{faPAKE}}$ to be the functionality described in Figure 1, except that TestPwd, Impersonate, OfflineTestPwd and StealPwdFile use the check depicted in Figure 2, with
$$L^P_c(\mathsf{pw}, \mathsf{pw}') = L^P_m(\mathsf{pw}, \mathsf{pw}') = \mathsf{pw} \text{ and } L^P_f(\mathsf{pw}, \mathsf{pw}') = \text{"wrong guess"}.$$

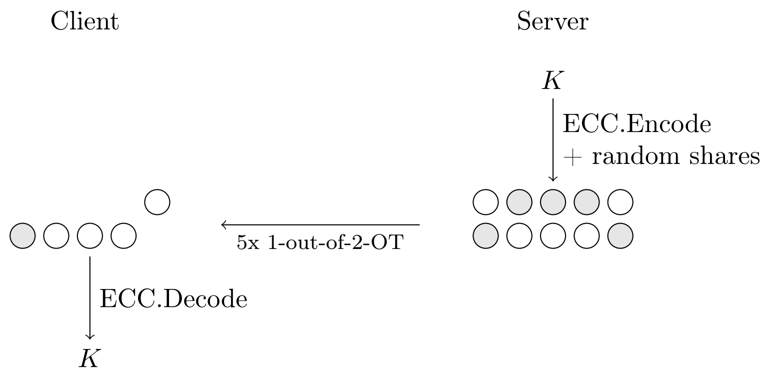
4 Fuzzy aPAKE from Secret Sharing

We now describe our protocol for fuzzy aPAKE with Hamming distance as metric for closeness of passwords. The very basic structure of our protocol is as follows: we let the server encode a cryptographic key K using an error-correcting code[4]. The resulting codeword (different parts of codeword are depicted as white circles in the illustration below) is then transmitted to the client, who decodes to obtain the key.

[4]Formally, we will define our scheme using the more general concept of robust secret sharing. However, for this overview it will be convenient to use the terminology of error-correcting codes.

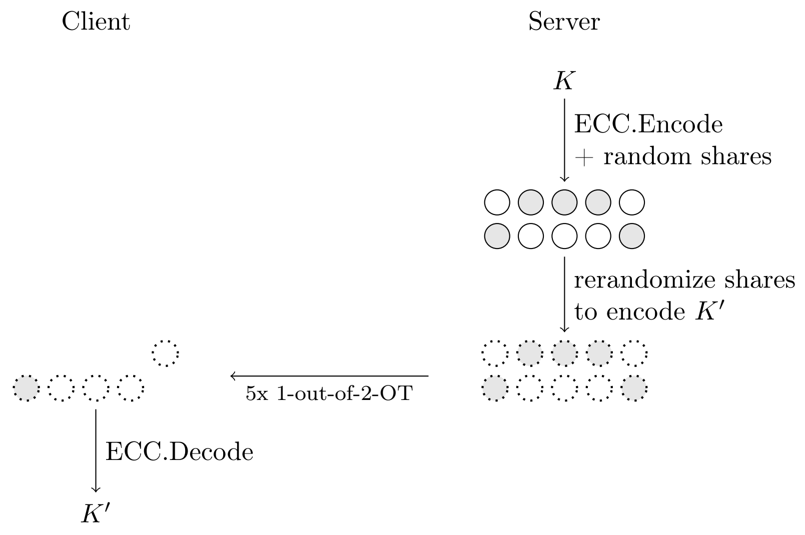


To make the retrieval of the cryptographic key password-dependent, the server stores the codeword together with randomness (depicted as grey circles below) in a password file. The position of the true codeword values in the file are dictated by the password bits. For example, in the illustration below, the server uses the password 01110. For this, we require the encoding algorithm to output codewords whose dimension matches the number of password bits. Now instead of getting the full password file, the client can choose to see only one value per column (either a part of the codeword or a random value). Technically, this is realized by employing a n-time 1-out-of-2 oblivious transfer (OT) protocol [5], where $n = 5$ is the password size of our toy example. The oblivious part is crucial to keep the server from learning the client's password. With this approach, passwords within the error correction threshold of the password used by the server are sufficient to let the client decode the cryptographic key. In the illustration below, the client uses password 11110, letting him obtain 4/5 of the codeword correctly. Furthermore, an adversary stealing the password file is now faced with the computationally expensive task of finding the codeword within the file. Generalized to an $(n-2t)$-out-of-n RSS, the naive approach of finding $n-2t$ shares of the codeword by taking random subsets succeeds with probability $1/2^{n-2t}$ (as there are $\binom{n}{2t}$ "good" choices containing shares only, and $\binom{n}{2t} \cdot 2^{n-2t}$ choices overall). Here, n is the password size and t the number of errors that the fuzzy aPAKE protocol allows in passwords.



The above protocol can only be used to derive a single cryptographic key. Further, it is prone to a malicious client who could send pw and $\mathsf{pw} \oplus 1^n$ in two subsequent runs and obtain the full password file. The solution is randomization of the password file in each run of the protocol. This is straightforward for linear secret sharing.

[5]The protocol is not restricted by 1-out-of-2 OT, but can use 1-out-of-n OT for any $n \in \mathbb{N}$. In this work we consider $n = 2$, but in practice $n > 2$ might be useful to reduce the number of wrong shares (e.g. $n = 2^7$ in case of ASCII encoding).



Unfortunately, the above protocol cannot be proven UC secure. As already mentioned before, UC-secure asymmetric PAKE protocols require an idealized assumption to reveal password guesses against the file to the adversary [Hes19]. Furthermore, we need to require that a password file does not fix the password that is contained in it, in order to prove security in the presence of adaptive server compromise attacks. To remedy the situation, we let the server store the password file in the exponent of a publicly known large group and prove security of our construction in the generic group model [Sho97]. As a consequence, the client now needs to perform decoding in the exponent. We summarize in Section 2 which known decoding techniqes work also in the exponent, and detail in Section 6 how this affects the parameter choices of our scheme.

To complete our high-level protocol description, we now consider malicious behavior of client and server in the above protocol. Firstly, we observe that the client cannot cheat apart from using a different password in the OT (which does not constitute an attack) or outputting a wrong cryptographic key (which also does not constitute an attack). Things look differently when we consider a malicious server. The server could, e.g., deviate from the protocol by entering only correct codeword parts in the OT, making the key exchange succeed regardless of the password the client is using. To prevent such attacks, we let the server prove correct behavior by encrypting his view of the protocol run under the symmetric key K'. The view consists of the randomized password file as well as g^{pw}. A client being able to derive K' can now check whether the server indeed holds a password pw close enough to his own, and whether the transmitted password file parts match the password file created with pw. The formal description of our protocol can be found in Figure 3.

It is worth noting the similarity of our protocol to the fuzzy PAKE from RSS/ECC of [DHP+18]. Namely, the overall idea is the same (server choosing and encoding K, sending it to the client who can decode if and only if his password is close enough). Essentially, both protocols transmit the codeword *encrypted* with the password, using a symmetric cipher that tolerates errors in the password - let us call this a *fuzzy symmetric cipher*. [DHP+18] uses the following fuzzy symmetric cipher: XOR the codeword (the message) with cryptographic keys derived from the individual password bits. These cryptographic keys are exchanged using PAKE on individual password bits. Unfortunately, this approach does not work in the asymmetric setting, since the server would have to store the password in the clear to access its individual bits. For the asymmetric case, one has to come up with a fuzzy cipher that works with a key that is some function of the password. This function needs to have two properties: hide the password sufficiently, and still allow to evaluate distance of its input.

4.1 Security

Theorem 4.1. *Let $n, l, t \in \mathbb{N}$ with $n = l + 2t$ and* $(\mathsf{Share}, \mathsf{Rec})$ *be an $(n, l-1, l+t)$-RSSExp scheme with respect to a generic group G. Then the protocol depicted in Figure 3 UC-emulates $\mathcal{F}^{P}_{\mathsf{faPAKE}}$ in the $\mathcal{F}_{\mathsf{IC}}, \mathcal{F}^{n}_{\mathsf{OT}}$-*

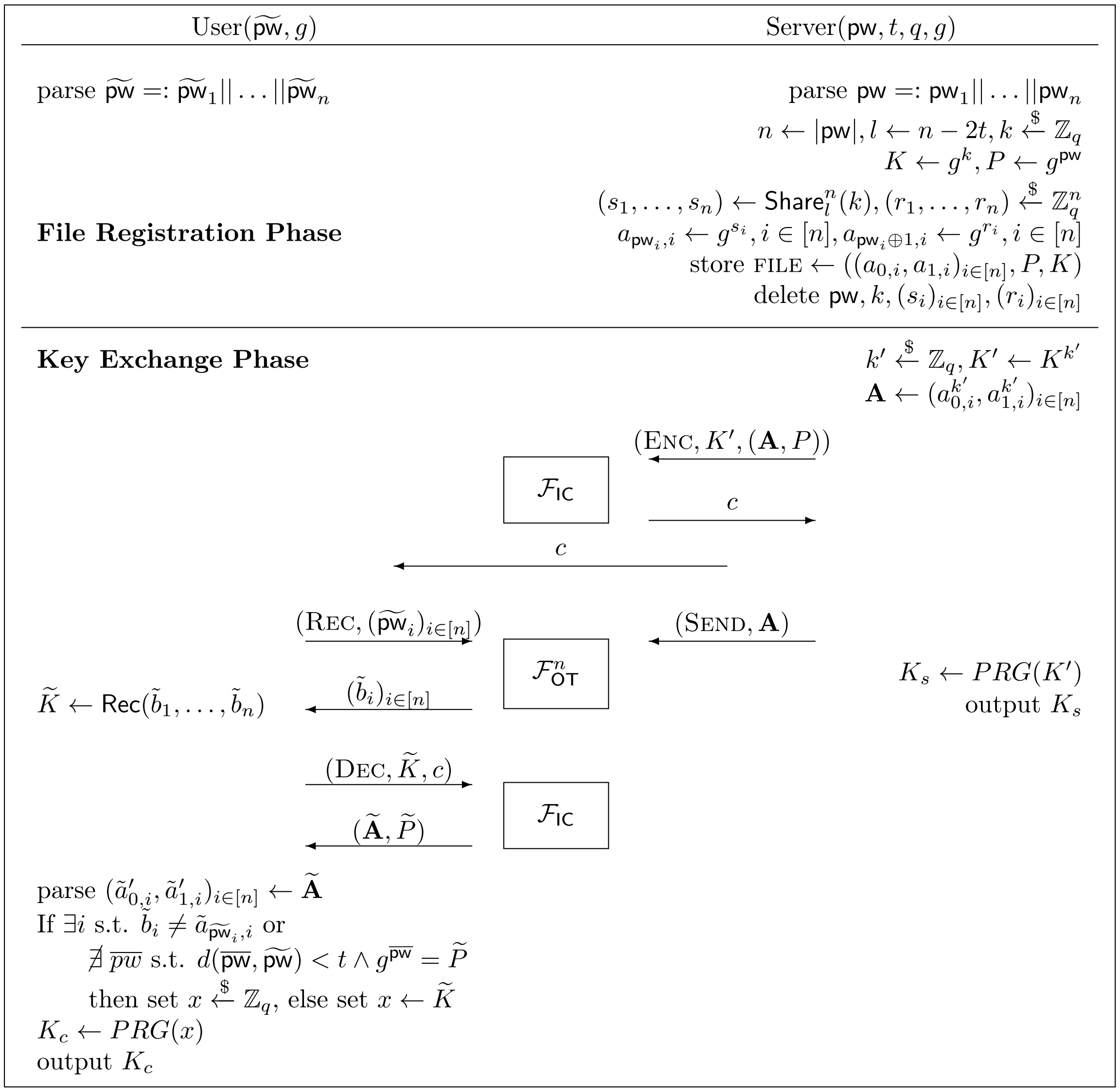

User$(\widetilde{\mathsf{pw}}, g)$
Server(pw, t, q, g)
parse $\widetilde{\mathsf{pw}} =: \widetilde{\mathsf{pw}}_1 || \dots || \widetilde{\mathsf{pw}}_n$
parse $\mathsf{pw} =: \mathsf{pw}_1 || \dots || \mathsf{pw}_n$
$n \leftarrow |\mathsf{pw}|, l \leftarrow n - 2t, k \xleftarrow{\$} \mathbb{Z}_q$
$K \leftarrow g^k, P \leftarrow g^{\mathsf{pw}}$
$(s_1, \dots, s_n) \leftarrow \mathsf{Share}_l^n(k), (r_1, \dots, r_n) \xleftarrow{\$} \mathbb{Z}_q^n$
File Registration Phase
$a_{\mathsf{pw}_i, i} \leftarrow g^{s_i}, i \in [n], a_{\mathsf{pw}_i \oplus 1, i} \leftarrow g^{r_i}, i \in [n]$
store FILE $\leftarrow ((a_{0,i}, a_{1,i})_{i \in [n]}, P, K)$
delete $\mathsf{pw}, k, (s_i)_{i \in [n]}, (r_i)_{i \in [n]}$
Key Exchange Phase
$k' \xleftarrow{\$} \mathbb{Z}_q, K' \leftarrow K^{k'}$
$\mathbf{A} \leftarrow (a_{0,i}^{k'}, a_{1,i}^{k'})_{i \in [n]}$
$(\text{Enc}, K', (\mathbf{A}, P))$
$\mathcal{F}_{\mathsf{IC}}$
c
c
$(\text{Rec}, (\widetilde{\mathsf{pw}}_i)_{i \in [n]})$
$(\text{Send}, \mathbf{A})$
$\mathcal{F}_{\mathsf{OT}}^n$
$K_s \leftarrow PRG(K')$
output K_s
$\widetilde{K} \leftarrow \mathsf{Rec}(\tilde{b}_1, \dots, \tilde{b}_n)$
$(\tilde{b}_i)_{i \in [n]}$
$(\text{Dec}, \widetilde{K}, c)$
$\mathcal{F}_{\mathsf{IC}}$
$(\widetilde{\mathbf{A}}, \widetilde{P})$
parse $(\tilde{a}'_{0,i}, \tilde{a}'_{1,i})_{i \in [n]} \leftarrow \widetilde{\mathbf{A}}$
If $\exists i$ s.t. $\tilde{b}_i \neq \tilde{a}_{\widetilde{\mathsf{pw}}_i, i}$ or
$\nexists\ \overline{pw}$ s.t. $d(\overline{\mathsf{pw}}, \widetilde{\mathsf{pw}}) < t \wedge g^{\overline{\mathsf{pw}}} = \widetilde{P}$
then set $x \xleftarrow{\$} \mathbb{Z}_q$, else set $x \leftarrow \widetilde{K}$
$K_c \leftarrow PRG(x)$
output K_c

Figure 3: Protocol Π_{faPAKE} for asymmetric fuzzy PAKE using an n times 1-out-of-2 Oblivious Transfer.

hybrid model, with $\gamma = 2t$, $\delta = t$, Hamming distance $d()$ and with respect to static byzantine corruptions and adaptive server compromise.

Before giving a detailed proof of Theorem 4.1, we present a high level proof sketch in which we consider the different cases of corruption.

Proof sketch: The overall proof strategy is to give a simulated transcript and output of the protocol that is indistinguishable from a real protocol execution and runs independently of the parties' passwords. The simulator is allowed to make one password guess per execution (in case of compromised server the simulator can run several offline password guesses). In the following, we describe the different cases of corruption that have to be considered.

- **Honest session:** Apart from the interaction between client and server through the UC-secure OT, the only message that needs to be simulated is one ideal cipher output which is sent from the server to the client and serves as a commitment to the servers values. Since the ideal cipher generates a uniformly random ciphertext from the ciphertext space, the simulator can replace the $\mathcal{F}_{\mathsf{IC}}$ output by a random value as long as the key is unknown. Hence, the simulator runs independently from the passwords of the parties.

- **Corrupted client:** In case of corrupted client, it is crucial to bind the client to submitting all n password bits at once such that the client is not able to adaptively change the password bits based on previous OT outputs. We achieve this by using non-adaptive n times 1-out-of-2 OT executions. Hence, $\mathcal{S}$ is able to query TestPwd on the submitted password bits before it needs to simulate the OT outputs for the client. In case TestPwd returns the server's password, $\mathcal{S}$ can simulate valid OT outputs. Otherwise, $\mathcal{S}$ chooses random outputs which is indistinguishable from the real execution due to the privacy property of the RSSExp scheme.

- **Corrupted server:** Whenever the corrupted server sends the ciphertext that contains the OT inputs and g^{pw}, $\mathcal{S}$ reconstructs pw from the inputs to the ideal cipher and the generic group operations requestes by the environment. $\mathcal{S}$ then checks whether pw is close to the client's password using the TestPwd interface. If so the simulator gets the client's password and can simulate the client. Otherwise the client's behavior is independent of its password. Hence, $\mathcal{S}$ can simulate the client with an arbitrary password that is not close to the server's.

- **Server compromise:** (1) Simulating the password file. $\mathcal{S}$ assembles a table with random group element handles as password file, and a random handle corresponding to g^{k}. As soon as $\mathcal{Z}$ starts decoding with some subset of these elements by querying the GGM, $\mathcal{S}$ learns these queries. As soon as this subset of elements corresponds to a password, the simulator submits this password to OfflineTestPwd. If the answer includes the server's password, then $\mathcal{S}$ programs the GGM such that the decoding results in the handle of g^{k}.

  (2) Impersonation attacks. The environment could use a file (e.g., the one obtained from $\mathcal{S}$ or a randomized variant of it) to impersonate the server. For this, the environment has to modify the ciphertext c to encrypt the file. Upon the environment sending an encryption query to $\mathcal{F}_{\mathsf{IC}}$ including an element P at the end of the message to be encrypted, the simulator checks if the GGM contains a tuple (pw, P). If so, $\mathcal{S}$ runs a TestPwd query on pw and learns the client's password $\widetilde{\mathsf{pw}}$ in case pw and $\widetilde{\mathsf{pw}}$ are close[6]. If there is no tuple (pw, P) in the GGM, $\mathcal{S}$ checks whether P was computed from the file (A', P') by the environment sending $f(P')$ to the GGM (and the simulator replying with P). If such a query happened, $\mathcal{S}$ issues an Impersonate query using the same function f.

- **MITM attack on honest session:** Apart from the interaction between client and server through the UC-secure OT, the only message that is sent is one ideal cipher output from the server to the client. Any attempt by $\mathcal{Z}$ to tamper with this message can be detected and hence $\mathcal{S}$ can simulate accordingly.

[6]We could alternatively let $\mathcal{S}$ issue an Impersonate query, but since the password is known issueing TestPwd works just as well.

The detailed proof can be found in Appendix 4.1.

Password Salting. In the UC modeling each protocol session has access to a fresh instantiation of the ideal functionalities. Consequently each protocol session invokes a fresh instantiation of RO or GGM, which return different values when queried on the same input in different sessions. Therefore the password files generated for two users with the same password are different. In practice however the passwords must be salted, i.e. instead of storing the g^{pw}, the server stores $g^{(\mathsf{sid}||\mathsf{pw})}$ where sid is the respective session identifier. By applying this standard technique of salting in practice, the password files for two clients who use the same password would be different.

Use Cases for Hamming Distance metric. Although hamming distance is not the most optimal way to measure the distance of two passwords, it is quite suitable for biometric applications. As an example, a server can derive the password file from a client's iris scan or fingerprint such that the client can use this biometric data for authentication. Another example would be wearable or IoT devices. Such devices can measure unique characteristics of the user or environment, such as heart beat patterns and use these measurements for authentication. Our next construction is more suitable for password matching applications where users authenticate themselves with a human memorable password, but might input some characters of the password incorrectly.

5 Fuzzy aPAKE from standard aPAKE

We now show how to construct a fuzzy aPAKE from asymmetric PAKE. Essentially, the idea is to let the server run an aPAKE protocol with the client multiple times, entering all the passwords that are close to the password he originally registered. For formally defining the protocol, it will be convenient to assume a (possibly probabilistic) function $\mathsf{close}(pw) := \{pw_i | d(\mathsf{pw}, \mathsf{pw}_i) < \delta\}$ that produces a set of all authenticating passwords. For example, for $d(), \delta$ accepting passwords where the first letter's case should be ignored, we would get $\mathsf{close}(\text{holy–moly!}) = \{\text{Holy–moly!, holy–moly!}\}$. When asking to register a password file containing pw, the server stores $\textsc{file} := \{H(\mathsf{pw}_i) | \mathsf{pw}_i \in \mathsf{close}(\mathsf{pw}) \; \forall i = 1, ..., |\mathsf{close}(\mathsf{pw})|\}$ as arbitrarily ordered list of hash values of all authenticating passwords. Let $k := |\textsc{file}|$ be the number of such passwords. Now client and server execute the aPAKE protocol k times, where the client *always* enters his password, and the server enters all values from the password file (in an order determined by a random permutation τ). Then, similar to our protocol Π_{faPAKE}, the server proves honest behavior by encrypting the (permuted) password file under all k keys generated by the aPAKE protocol. The client decrypts and looks for a password file that was generated from a password that is close to his own password. If he finds such a file, he uses the corresponding decryption key (generated from aPAKE) to perform an explicit authentication step with the server. Note that this extra round of explicit authentication cannot be skipped, since otherwise the server would not know which key to output. While the computation on the client side sounds heavy at first sight, if both parties follow the protocol, all but one decryption attempts on the client side will fail. The client can efficiently recognized a failed decryption attempt by searching the decrypted message for the hash of his own password. The protocol is depicted in Figure 4.

Π_{transf} does not scale asymptotically, neither in the size of the password nor the number of errors. As an example, for correcting only one arbitrary error in an n-bit password, the password file size is already $k = n + 1$. For correcting up to t errors, we get $k := 1 + \sum_{i=1}^{t} \binom{n}{i}$. Note that k determines not only the size of the password file but also the number of aPAKE executions. On the plus side, the construction works with arbitrary metric and distances, does not have a "security gap" between δ and γ and has reasonable computational complexity on both the client and server side.

Unfortunately Π_{transf} cannot be proven secure given the original ideal functionality $\mathcal{F}_{\mathsf{faPAKE}}$, or rather its variant with explicit authentication (see Figure 9). In a nutshell, an attacker tampering with the single aPAKE executions can issue k password guesses using arbitrary passwords from the dictionary. A fuzzy aPAKE as defined within $\mathcal{F}_{\mathsf{faPAKE}}$, however, needs to bound the attacker to use k *close* passwords. To remedy the situation we modify the $\textsc{TestPwd}$ interface of our $\mathcal{F}_{\mathsf{faPAKE}}$ functionality such that it allows

n single password guesses (see Figure 8). By single guess we mean that, instead of comparing a guess to all passwords within some threshold of the password of the attacked party (as it is done by $\mathcal{F}_{\mathsf{faPAKE}}$), it is compared to just one password. In case the client is attacked, the functionality compares with the client's password (and allows k such comparisons). In case the server is attacked, comparison is against a randomly chosen password close to the server's password[7]. Overall, the amount of information that the attacker obtains from both TestPwd interfaces in Figures 1 and 8 is comparable: they both allow the attacker to exclude k passwords from being "close enough" to authenticate towards an honest party. Stated differently, to go through the whole dictionary D of passwords, with both TestPwd interfaces an attacker would need to tamper with $|D|/k$ key exchange sessions.

We let $\mathcal{F}'_{\mathsf{faPAKE}}$ denote the ideal functionality $\mathcal{F}^{P}_{\mathsf{faPAKE}}$ with interfaces TestPwd and NewKey taken from Figures 8 and 9.

Theorem 5.1. *Protocol Π_{transf} UC-emulates $\mathcal{F}'_{\mathsf{faPAKE}}$ with arbitrary distance function $d()$ and arbitrary threshold $\delta = \gamma$ in the $(\mathcal{F}_{\mathsf{aPAKE}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{IC}})$-hybrid model w.r.t static corruptions and adaptive server compromise and $H()$ denoting calls to $\mathcal{F}_{\mathsf{RO}}$.*

Proof sketch. We need to consider the following attack scenarios:

- *Passive attacks*: The environment $\mathcal{Z}$ tries to distinguish uncorrupted real and ideal execution by merely observing transcript and outputs of the protocol, while providing the inputs of both honest parties. Since the outputs of the protocol are random oracle outputs and the transcript consists of a random ciphertext vector $\overrightarrow{e}$ output by the ideal cipher, $\mathcal{Z}$ cannot distinguish real outputs from simulated random values unless it queries either the ideal cipher functionality $\mathcal{F}_{\mathsf{IC}}$ or the random oracle $\mathcal{F}_{\mathsf{RO}}$ with the corresponding inputs. This can be excluded with overwhelming probability since these inputs are uniformly random values of high entropy chosen by honest parties.

- *Active message tampering*: We consider $\mathcal{Z}$ injecting a message into a protocol execution between two honest parties. The only messages being sent in unauthenticated channels are the encryption vector $\overrightarrow{e}$ and the explicit authentication message h. Replacing the message h would simply result in two different keys as output for the parties, simulatable by sending $\perp$ via NewKey. Tampering with $\overrightarrow{e}$ is a bit more tricky. Namely, we have to consider $\mathcal{Z}$ modifying only single components of $\overrightarrow{e}$. Tampering with each element of the vector $\overrightarrow{e}$ lowers the probability for the parties to output the same key. Hence, the simulator needs to adjust the probability for the parties to output the same key by forcing the functionality to only output the same session key with this exact probability, i.e., the simulator sends $\perp$ via NewKey with the inverse probability.

- *(Static) Byzantine corruption*: We consider the case where $\mathcal{Z}$ corrupts one of the parties.
  - In case of corrupted server, given an adversarially computed $\overrightarrow{e}$, the simulator extracts all k passwords used by $\mathcal{Z}$ from the server's inputs to $\mathcal{F}_{\mathsf{IC}}$ and $\mathcal{F}_{\mathsf{RO}}$ and submits them as password guess to $\mathcal{F}'_{\mathsf{faPAKE}}$ (via TestPwd). $\mathcal{S}$ then uses the answers (either "wrong guess" or the client's true password) to continue the simulation faithfully. In case the corrupted server deviates from the protocol (e.g., $\overrightarrow{e}$ does not encrypt a set of passwords generated by $\mathsf{close}()$, or sends garbage to the $\mathcal{F}_{\mathsf{aPAKE}}$ instance in which the server uses the client's password), the simulator sends $\perp$ via the NewKey interface to simulate failure of the key exchange.
  - The case of a corrupted client is handled similarly using the freedom of k individual TestPwd queries.

- *Server compromise*: The password file is simulated without knowledge of the password by sampling random hash values. The simulator now exploits observability and programmability of the random oracle (that models the hash function) as follows: as soon as $\mathcal{Z}$ wants to compute $H(\mathsf{pw})$, $\mathcal{S}$ submits

[7]Programming this randomized behavior into the functionality greatly simplifies proving security of Π_{transf} and does not seem to weaken the functionality compared to one using non-randomized equality checks.

pw to its OfflineTestPwd interface. Upon learning the server's true password, $\mathcal{S}$ programs the random oracle such that the password file contains hash values of all passwords close to pw.

- *Attacking* $\mathcal{F}_{\mathsf{aPAKE}}$: While using $\mathcal{F}_{\mathsf{aPAKE}}$ as hybrid functionality helps the parties to exchange the key, it gives us a hard time when simulating. Essentially, the simulator has to simulate answers to all adversarial interfaces of each instance of $\mathcal{F}_{\mathsf{aPAKE}}$ since $\mathcal{Z}$ is allowed to query them. And $\mathcal{F}_{\mathsf{aPAKE}}$ has a lot of them: StealPwdFile, TestPwd, OfflineTestPwd and Impersonate. In a nutshell, OfflineTestPwd queries can be answered by querying the corresponding interface at $\mathcal{F}'_{\mathsf{faPAKE}}$. The same holds for StealPwdFile and Impersonate, only that they can be queried only once in $\mathcal{F}'_{\mathsf{faPAKE}}$. Our proof thus needs to argue that the one answer provided by $\mathcal{F}_{\mathsf{faPAKE}}$ includes already enough information to simulate answers to all k. The most annoying interface, namely TestPwd is handled by forwarding each individual TestPwd guess to $\mathcal{F}'_{\mathsf{faPAKE}}$. This explains why $\mathcal{F}'_{\mathsf{faPAKE}}$ needs to allow k individual password guesses instead of one fuzzy one (as provided by $\mathcal{F}_{\mathsf{faPAKE}}$).

6 Efficiency

Efficiency of Π_{faPAKE}. When instantiated with the statically secure OT from [BDD+17], Π_{faPAKE} is round-optimal and requires each party to send only one message. While 2 consecutive messages are in any case required for the OT, we can conveniently merge the ciphertext send by the server with his message send within the OT. As detailed in Section A, this results in a total message size of $8\lambda n + |c| = 8\lambda n + (2n+1)\lambda \approx 10\lambda n$ bits. For each login attempt of a client, the server needs to perform $2n+1$ group exponentiations in order to refresh the values in the password file, as well as an encryption of $2n+1$ group elements. Finally, the server has to perform one PRG execution. Note that the server has to do some additional computations during the initial setup phase of the protocol, however since this phase is only run once, we do not consider its complexity in this section. The client's computation is where our protocol lacks efficiency. Namely, with the naive decoding technique from [CG99], client's computation is only polynomial in $|\mathsf{pw}|$ if the error correction capability δ is not larger than $\log|\mathsf{pw}|$. And still for such δ, going beyond password sizes of, say, 40 bits does not seem feasible.

Efficiency of Π_{transf}. In order to achieve the fuzzy password matching in Π_{transf}, the server is required to store one hash value for each password that lies within distance δ of the original password. As a consequence, the password file size is highly dependent on these threshold parameters. If we consider Hamming distance as done in our first construction, for $\delta = 1$ the password file is of size $\mathcal{O}(n)$. However for $\delta = 2$ it grows to $\mathcal{O}(n^2)$ and for $\delta = 3$ to $\mathcal{O}(n^3)$. Hence, such error tolerance can only be achieved in Π_{transf} at the cost of huge password files. The same correlation to the error tolerance holds for the amount of aPAKE executions in Π_{transf}.

In order to determine the computational complexity of Π_{transf} in terms of required group operations, we chose an instantiation of an aPAKE protocol, OPAQUE [JKX18], that requires a constant number of group exponentiations. As previously discussed, Π_{transf} requires k aPAKE executions with k being the size of the password file.

Despite its shortcomings when used with Hamming distance, Π_{transf} serves as a good illustration for how to construct a general purpose faPAKE protocol that already has practical relevance. Instantiated with distance and threshold suitable to correct, e.g., capitalization of first letters or transposition of certain digits, we obtain an efficient "almost secure" fuzzy aPAKE scheme.

We present a comparison of the two schemes in Table 1. Π_{transf} is listed twice. First it is compared to Π_{faPAKE} when using Hamming distance. The last row indicates its efficiency for parameters resulting in k authenticating passwords, where k can be as small as 2.

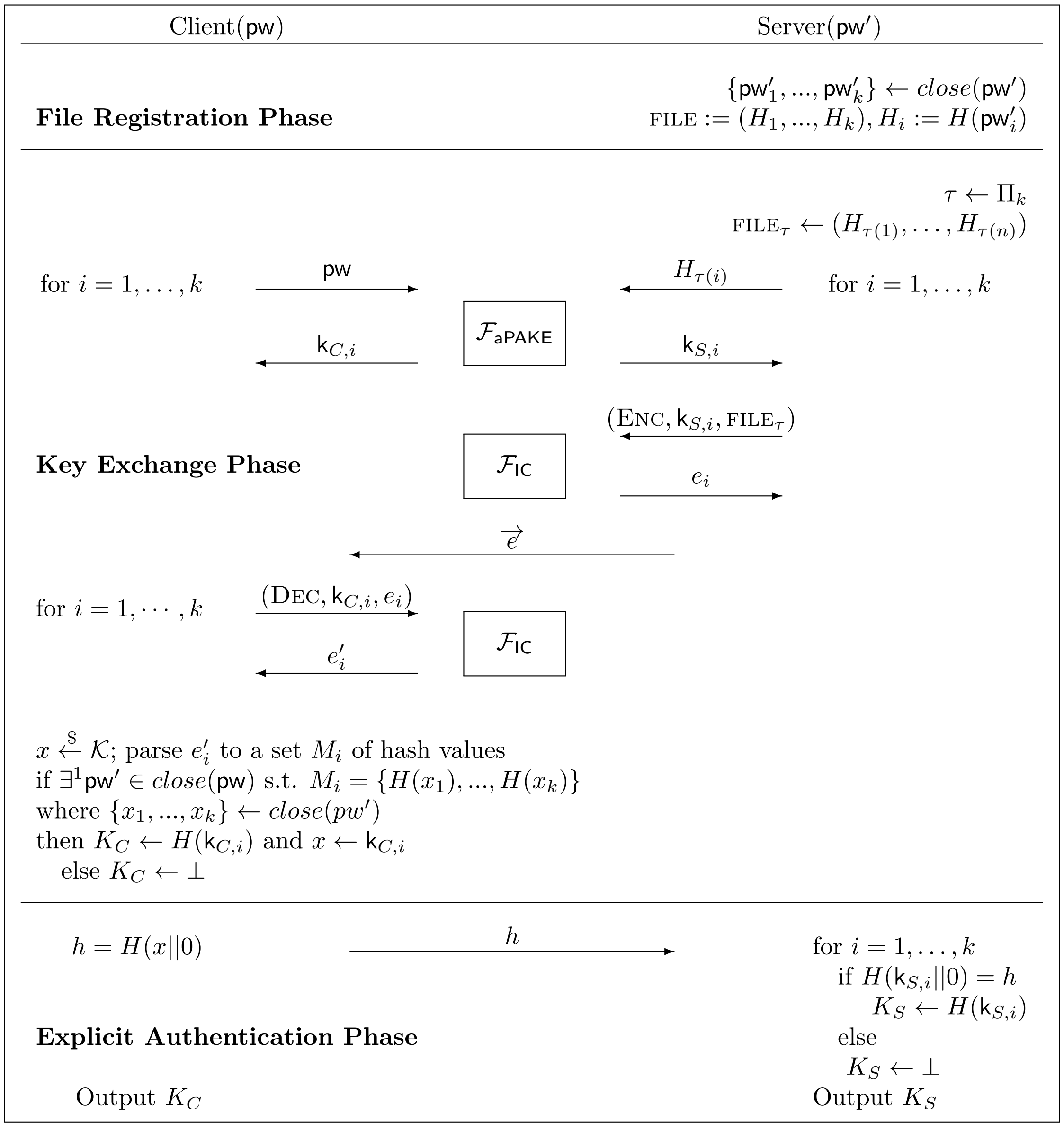

Client(pw)
Server(pw')
File Registration Phase
$\{\mathsf{pw}'_1, ..., \mathsf{pw}'_k\} \leftarrow close(\mathsf{pw}')$
$\text{FILE} := (H_1, ..., H_k), H_i := H(\mathsf{pw}'_i)$
$\tau \leftarrow \Pi_k$
$\text{FILE}_\tau \leftarrow (H_{\tau(1)}, \dots, H_{\tau(n)})$
for $i = 1, \dots, k$
pw
$H_{\tau(i)}$
for $i = 1, \dots, k$
$\mathcal{F}_{\mathsf{aPAKE}}$
$\mathsf{k}_{C,i}$
$\mathsf{k}_{S,i}$
$(\text{Enc}, \mathsf{k}_{S,i}, \text{FILE}_\tau)$
Key Exchange Phase
$\mathcal{F}_{\mathsf{IC}}$
e_i
$\overrightarrow{e}$
for $i = 1, \cdots, k$
$(\text{Dec}, \mathsf{k}_{C,i}, e_i)$
$\mathcal{F}_{\mathsf{IC}}$
e'_i
$x \xleftarrow{\$} \mathcal{K}$; parse e'_i to a set M_i of hash values
if $\exists^1 \mathsf{pw}' \in close(\mathsf{pw})$ s.t. $M_i = \{H(x_1), ..., H(x_k)\}$
where $\{x_1, ..., x_k\} \leftarrow close(pw')$
then $K_C \leftarrow H(\mathsf{k}_{C,i})$ and $x \leftarrow \mathsf{k}_{C,i}$
else $K_C \leftarrow \perp$
$h = H(x||0)$
h
for $i = 1, \dots, k$
if $H(\mathsf{k}_{S,i}||0) = h$
$K_S \leftarrow H(\mathsf{k}_{S,i})$
else
$K_S \leftarrow \perp$
Explicit Authentication Phase
Output K_C
Output K_S

Figure 4: Protocol Π_{transf} for fuzzy asymmetric PAKE. The parties participate in k executions of the aPAKE protocol. Afterwards they verify if at least one of the produced k keys match and agree on it. We denote $\Pi_n := perm(1, ..., k)$ the set of permutations $[k] \rightarrow [k]$. $close(\mathsf{pw})$ is a function outputting a list of all authenticating passwords (see text for a formal description).

	File size	Message size	Thresholds	Metric	Client	Server	Assumption
Π_{faPAKE}	$(2n+2)\lambda$	$10\lambda n$	$2\delta = \gamma$	Hamming	$poly(n) \cdot \mathcal{O}(\log q)$	$\mathcal{O}(n \log q)$	IC, GGM
Π_{transf}	$\mathcal{O}(n^\delta)$	$\mathcal{O}(n^\delta)$	$\delta = \gamma$	Hamming	$\mathcal{O}(n^\delta \log q)$	$\mathcal{O}(n^\delta \log q)$	IC, ROM
Π_{transf}	λk	$\mathcal{O}(k)$	$\delta = \gamma$	arbitrary	$\mathcal{O}(k)$	$\mathcal{O}(k)$	IC, ROM

Table 1: Comparison of Π_{faPAKE} and Π_{transf}. We assume n-bit passwords in case of Hamming distance. File size and communication complexity are in bits. The Client and Server column indicate the number of group operations.

7 Conclusion

In this paper, we initiated the study of *fuzzy asymmetric PAKE*. Our security notion in the UC framework results from a natural combination of existing functionalities. Protocols fulfilling our definition enjoy strong security guarantees common to all UC-secure PAKE protocols such as protection against off-line attacks and simulatability even when run with adversarially-chosen passwords.

We demonstrate that UC-secure fuzzy aPAKE can be build from OT and Error-Correcting Codes, where fuzziness of passwords is measured in terms of their Hamming distance. Our protocol is inspired by the ideas of [DHP+18] for building a fuzzy *symmetric* PAKE. We also show how to build a (mildly less secure) fuzzy aPAKE from (non-fuzzy) aPAKE. Our construction allows for arbitrary notions of fuzziness and yields efficient, strongly secure and practical protocols for use cases such as, e.g., correction of typical orthographic errors in typed passwords.

Our two constructions nicely show the trade-offs that one can have for fuzzy aPAKE. The "naive" construction from aPAKE has large password file size when used with Hamming distance, but also works for arbitrary closeness notions possibly leading to small password files and practical efficiency. The construction using Error-Correcting Codes is restricted to Hamming distance and $\log(|\mathsf{pw}|)$ error correction threshold. I comes with a computational overhead on the client side, but has only little communication and small password file size. It is worth noting that, for this construction, all efficiency drawbacks could be remedied by finding a more efficient decoding method that works in the exponent. We leave this as well as finding more fuzzy aPAKE constructions as future work. Specifically, no fuzzy aPAKE scheme with *strong* compromise security (as defined in [JKX18]) is known.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) Emmy Noether Program *FA 1320/1-1*, by the *DFG CRC 1119 CROSSING* (project S7), by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*, by the VeriSec project 16KIS0634 from the Federal Ministry of Education and Research (BMBF), and by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786725 – OLYMPUS.

We would like to thank Sophia Yakoubov for helpful discussions on earlier versions of this work.

References

[Ale15] Alec Muffet. Facebook: Password hashing & authentication, presentation at real world crypto, 2015.

[BBC+13] Fabrice Benhamouda, Olivier Blazy, Céline Chevalier, David Pointcheval, and Damien Vergnaud. New techniques for SPHFs and efficient one-round PAKE protocols. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 449–475. Springer, Heidelberg, August 2013.

[BBR88] Charles H. Bennett, Gilles Brassard, and Jean-Marc Robert. Privacy amplification by public discussion. *SIAM J. Comput.*, 17(2):210–229, 1988.

[BDD+17] Paulo S. L. M. Barreto, Bernardo David, Rafael Dowsley, Kirill Morozov, and Anderson C. A. Nascimento. A framework for efficient adaptively secure composable oblivious transfer in the ROM. *CoRR*, abs/1710.08256, 2017.

[BJX19] Tatiana Bradley, Stanislaw Jarecki, and Jiayu Xu. Strong asymmetric PAKE based on trapdoor CKEM. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 798–825. Springer, Heidelberg, August 2019.

[BLV19] Elette Boyle, Rio LaVigne, and Vinod Vaikuntanathan. Adversarially robust property-preserving hash functions. In Avrim Blum, editor, *ITCS 2019*, volume 124, pages 16:1–16:20. LIPIcs, January 2019.

[BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, May 1992.

[BM93] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 244–250. ACM Press, November 1993.

[BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 156–171. Springer, Heidelberg, May 2000.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000.

[CAA+16] Rahul Chatterjee, Anish Athayle, Devdatta Akhawe, Ari Juels, and Thomas Ristenpart. pASSWORD tYPOS and how to correct them securely. In *2016 IEEE Symposium on Security and Privacy*, pages 799–818. IEEE Computer Society Press, May 2016.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CDD+15] Ronald Cramer, Ivan Bjerre Damgård, Nico Döttling, Serge Fehr, and Gabriele Spini. Linear secret sharing schemes from error correcting codes and universal hash functions. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 313–336. Springer, Heidelberg, April 2015.

[CDVW12] Ran Canetti, Dana Dachman-Soled, Vinod Vaikuntanathan, and Hoeteck Wee. Efficient password authenticated key exchange via oblivious transfer. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 449–466. Springer, Heidelberg, May 2012.

[CG99] Ran Canetti and Shafi Goldwasser. An efficient threshold public key cryptosystem secure against adaptive chosen ciphertext attack. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 90–106. Springer, Heidelberg, May 1999.

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 404–421. Springer, Heidelberg, May 2005.

[CWP+17] Rahul Chatterjee, Joanne Woodage, Yuval Pnueli, Anusha Chowdhury, and Thomas Ristenpart. The TypTop system: Personalized typo-tolerant password checking. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 329–346. ACM Press, October / November 2017.

[DHP+18] Pierre-Alain Dupont, Julia Hesse, David Pointcheval, Leonid Reyzin, and Sophia Yakoubov. Fuzzy password-authenticated key exchange. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 393–424. Springer, Heidelberg, April / May 2018.

[DRS04] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 523–540. Springer, Heidelberg, May 2004.

[GL03] Rosario Gennaro and Yehuda Lindell. A framework for password-based authenticated key exchange. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 524–543. Springer, Heidelberg, May 2003. `http://eprint.iacr.org/2003/032.ps.gz`.

[GMR06] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 142–159. Springer, Heidelberg, August 2006.

[Hes19] Julia Hesse. Separating standard and asymmetric password-authenticated key exchange. Cryptology ePrint Archive, Report 2019/1064, 2019. `https://eprint.iacr.org/2019/1064`.

[HL19] Björn Haase and Benoît Labrique. Aucpace: Efficient verifier-based PAKE protocol tailored for the iiot. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):1–48, 2019.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Heidelberg, April / May 2018.

[KOY01] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient password-authenticated key exchange using human-memorable passwords. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 475–494. Springer, Heidelberg, May 2001.

[KV11] Jonathan Katz and Vinod Vaikuntanathan. Round-optimal password-based authenticated key exchange. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 293–310. Springer, Heidelberg, March 2011.

[MS81] Robert J. McEliece and Dilip V. Sarwate. On sharing secrets and Reed-Solomon codes. *Commun. ACM*, 24(9):583–584, 1981.

[Pei06] Chris Peikert. On error correction in the exponent. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 167–183. Springer, Heidelberg, March 2006.

[PW17] David Pointcheval and Guilin Wang. VTBPEKE: Verifier-based two-basis password exponential key exchange. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *ASIACCS 17*, pages 301–312. ACM Press, April 2017.

[Rot06] Ron Roth. *Introduction to Coding Theory.* Cambridge University Press, New York, NY, USA, 2006.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

A Ideal Functionalities

In this Section we recall ideal functionalities from the literature, sometimes slightly adapted to our purposes.

The functionality $\mathcal{F}_{\mathsf{IC}}$ takes as input the security parameter k, and interacts with an adversary $\mathcal{S}$ and with a set of (dummy) parties $P_1, \ldots, P_n$ by means of these queries:

- $\mathcal{F}_{\mathsf{IC}}$ keeps a (initially empty) list L containing 3−tuples of bit strings and two (initially empty) sets C_{sk} and M_{sk} for every sk. (The sets are not created until sk is first used, thus avoiding the need to instantiate exponentially many sets.)
- **Upon receiving a query $(\textsc{Enc}, \mathsf{sid}, \mathsf{sk}, m)$ (with $m \in \{0,1\}^k$) from some party P_i or $\mathcal{S}$, do:**
  - If there is a 3−tuple $(\mathsf{sk}, m, \tilde{c})$ for some $\tilde{c} \in \{0,1\}^k$ in the list L, set $c := \tilde{c}$.
  - If there is no such record, choose uniformly $c \in \{0,1\}^k \backslash C_{\mathsf{sk}}$ which is the set consisting of ciphertexts not already used with sk. Next, it stores the 3−tuple $(\mathsf{sk}, m, c) \in L$ and sets both $M_{\mathsf{sk}} \leftarrow M_{\mathsf{sk}} \cup \{m\}$ and $C_{\mathsf{sk}} \leftarrow C_{\mathsf{sk}} \cup \{c\}$.

  Once c is set, reply to the activating machine with $(\textsc{Enc}, \mathsf{sid}, c)$.
- **Upon receiving a query $(\textsc{Dec}, \mathsf{sid}, \mathsf{sk}, c)$ (with $c \in \{0,1\}^k$) from some party P_i or $\mathcal{S}$, do:**
  - If there is a 3−tuple $(\mathsf{sk}, \tilde{m}, c)$ for some $\tilde{m} \in \{0,1\}^k$ in L, set $m := \tilde{m}$.
  - If there is no such record, choose uniformly $m \in \{0,1\}^k \backslash M_{\mathsf{sk}}$ which is the set consisting of plaintexts not already used with sk. Next, it stores the 3−tuple $(\mathsf{sk}, m, c) \in L$ and sets both $M_{\mathsf{sk}} \leftarrow M_{\mathsf{sk}} \cup \{m\}$ and $C_{\mathsf{sk}} \leftarrow C_{\mathsf{sk}} \cup \{c\}$.

  Once m is set, reply to the activating machine with $(\textsc{Dec}, \mathsf{sid}, m)$.

Figure 5: Functionality $\mathcal{F}_{\mathsf{IC}}$

Ideal Cipher. An ideal cipher [BPR00] is a block cipher that takes a plaintext or a ciphertext as input. We describe the ideal cipher functionality $\mathcal{F}_{\mathsf{IC}}$ in Figure 5. $\mathcal{F}_{\mathsf{IC}}$ a perfectly random permutation for each key by ensuring injectivity for each query simulation: to this aim, it uses a list L and projections M_{sk} and C_{sk}, that are global, independently of the sid.

Ideal Oblivious Transfer (OT). The functionality $\mathcal{F}^n_{\mathsf{OT}}$ describes a bundled version of a 1-out-of-2 OT, where output is only generated if n inputs have been provided. The functionality maintains subsession identifiers ssid and can thus be used to exchange an n-fold 1-out-of-2 OT multiple times. For simplicity, we require both the sender and the receiver to input all n inputs with one message.

$\mathcal{F}^n_{\mathsf{OT}}$ can be realized by any UC-secure protocol for 1-out-of-2 OT with the slight modification that the sender only continues the protocol after having received n input-dependent messages of the client (in UC-secure protocol, the client is usually committed to his input when sending his first message). E.g., one could modify the round-efficient statically secure OT protocol from [BDD$^+$17], Figure 3, to let the sender Alice wait for receiver Bob to complete the first step of the protocol n times. The protocol requires one round of communication. In total, 3 strings, 1 public key and 2 ciphertexts are send around per 1-out-of-2 OT. For sender inputs from $\mathbb{F}_q^2$ and security parameter λ with $q = 2^\lambda$, the communication complexity of the n-fold 1-out-of-2 OT is then $8\lambda n$ bits.

B UC execution of Π_{faPAKE}

- $\mathcal{P}_C$ upon receiving $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}')$ from $\mathcal{Z}$ does the following:
  1. Parse pw as $\mathsf{pw} = \mathsf{pw}_1 || \cdots || \mathsf{pw}_n$ and wait to receive (sid, c) from $\mathcal{P}_S$.
  2. Send $(\textsc{Rec}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, (\mathsf{pw}_i)_{i \in [n]})$ to $\mathcal{F}^n_{\mathsf{OT}}$.

The functionality $\mathcal{F}_{\mathsf{OT}}^n$ is parameterized with a security parameter λ. It interacts with an adversary $\mathcal{S}$ and a client and a server party $\mathcal{P} \in \{\mathcal{P}_C, \mathcal{P}_S\}$ via the following queries.

- On $(\textsc{Send}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, (v_{0,i}, v_{1,i})_{i\in[n]})$ from $\mathcal{P}_S$ where $(v_{0,i}, v_{1,i}) \in \mathbb{F}_q^2\ \forall i \in [n]$, store $(\mathsf{ssid}, (v_{0,i}, v_{1,i})_{i\in[n]})$ and send $(\textsc{Send}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathcal{P}_C)$ to $\mathcal{S}$. Ignore further Send messages with the same ssid from $\mathcal{P}_S$.
- On $(\textsc{Rec}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, (x_i)_{i\in[n]})$ from $\mathcal{P}_C$ where $x_i \in \mathbb{F}_2\ \forall i \in [n]$, store $(\mathsf{ssid}, (x_i)_{i\in[n]})$ and send $(\textsc{Rec}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S)$ to $\mathcal{S}$. Ignore further Rec messages with the same ssid from $\mathcal{P}_C$.
- On $(\textsc{Continue}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{S}$,
  - if there are records $(\mathsf{ssid}, (v_{0,i}, v_{1,i})_{i\in[n]})$ and $(\mathsf{ssid}, (x_i)_{i\in[n]})$ output $(\mathsf{ssid}, (v_{x_i,i})_{i\in[n]})$ to $\mathcal{P}_C$.
  - else ignore this message.

Figure 6: Ideal functionality $\mathcal{F}_{\mathsf{OT}}^n$ for an Oblivious Transfer.

3. Upon receiving $(\mathsf{ssid}, (\tilde{b}_i)_{i\in[n]})$ execute the the algorithm $\widetilde{K} \leftarrow \mathsf{Rec}(\tilde{b}_1, \cdots, \tilde{b}_n)$
4. Send $(\textsc{Dec}, \mathsf{sid}, \widetilde{K}, c))$ to $\mathcal{F}_{\mathsf{IC}}$ and receive $(\textsc{Dec}, \mathsf{sid}, (\widetilde{\mathbf{A}}, \widetilde{P}))$.
5. Check if

$$(\exists i \text{ s.t. } \tilde{b}_i \neq \tilde{a}_{\widetilde{\mathsf{pw}}_i, i}) \vee (\nexists\ \overline{pw} \text{ s.t. } d(\overline{\mathsf{pw}}, \widetilde{\mathsf{pw}}) < t \wedge g^{\overline{\mathsf{pw}}} = \widetilde{P})$$

   If so choose x uniformly random from $\mathbb{Z}_q$, otherwise set x as $\widetilde{K}$
6. Output $(\mathsf{sid}, \mathsf{ssid}, PRG(x))$ to $\mathcal{Z}$.

- $\mathcal{P}_S$ upon receiving $(\textsc{StorePwdFile}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw})$ from $\mathcal{Z}$ does the following:
  1. Parse $\mathsf{pw} := \mathsf{pw}_1 || \ldots || \mathsf{pw}_n$
  2. Compute the following:

$$\begin{aligned}
&n \leftarrow |\mathsf{pw}|, l \leftarrow n - 2t, k \xleftarrow{\$} \mathbb{Z}_q \\
&(s_1, \ldots, s_n) \leftarrow \mathsf{Share}_l^n(k) \\
&(r_1, \ldots, r_n) \xleftarrow{\$} \mathbb{Z}_q^n \\
&(a_{0,i}, a_{1,i})_{i\in[n]} \text{ where } a_{\mathsf{pw}_i, i} := g^{s_i} \\
&\text{store FILE} \leftarrow ((a_{0,i}, a_{1,i})_{i\in[n]}, g^{\mathsf{pw}}, g^k) \\
&\text{delete } \mathsf{pw}, k, (s_i)_{i\in[n]}, (r_i)_{i\in[n]}
\end{aligned}$$

- $\mathcal{P}_S$ upon receiving $(\textsc{SrvSession}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{Z}$ retrieves $\text{FILE} := ((a_{0,i}, a_{1,i})_{i\in[n]}, P, K)$ and does the following:
  1. Compute $k' \xleftarrow{\$} \mathbb{Z}_q, \mathbf{A}' \leftarrow (a_{0,i}^{k'}, a_{1,i}^{k'})_{i\in[n]}, K' \leftarrow K^{k'}$
  2. Send $(\textsc{Enc}, \mathsf{sid}, K', (\mathbf{A}', P)))$ to $\mathcal{F}_{\mathsf{IC}}$ and receive $(\textsc{Enc}, \mathsf{sid}, c)$.
  3. Send (sid, c) to $\mathcal{P}_C$
  4. Send $(\textsc{Send}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, (A))$ to $\mathcal{F}_{\mathsf{OT}}^n$.
  5. Output $(\mathsf{sid}, \mathsf{ssid}, PRG(K'))$ to $\mathcal{Z}$.

C Proof of Theorem 4.1

Proof. We start with the real execution of protocol Π_{faPAKE} as detailed in Appendix B, where parties are running with the dummy adversary $\mathcal{A}$. We then modify the execution in a way indistinguishable for the environment, ending up in the ideal execution where dummy parties run with $\mathcal{F}_{\mathsf{faPAKE}}^M$ and a simulator $\mathcal{S}$.

First, we note that in the case of Hamming distance as a metric of closeness, $\mathcal{F}^{M}_{\mathsf{faPAKE}} = \mathcal{F}^{P}_{\mathsf{faPAKE}}$ (i.e., knowing the wrong bit positions of one's password reveals already the full password) and we thus are allowed to assume that $\mathcal{F}^{M}_{\mathsf{faPAKE}}$ replies with the password whenever a guess contained less than γ errors.

Our proof works in the generic group model (GGM). This means that the simulator is in charge of performing all group operations. Formally, upon input an exponent a to the GGM, $\mathcal{S}$ creates a fresh group element identifier id_a and replies with id_a. $\mathcal{S}$ internally stores a table with entries of the form (a, id_a) (where essentially id_a represents the element g^a). Further, $\mathcal{S}$ also answers queries $id_a * id_b$ to the GGM, where $*$ is symbolic for the group operation. If there exists entries $(a, id_a), (b, id_b), (c, id_c)$ with $c = a + b$, then $\mathcal{S}$ replies with id_c, otherwise $\mathcal{S}$ picks a fresh identifier as answer. Similarly, $\mathcal{S}$ answers queries $(id_a)^b$ with id_c if there exist entries $(a, id_a), (c, id_c)$ with $c = ab$. Further $\mathcal{S}$ keeps track of functions submitted to the GGM. If $\mathcal{S}$ simulates group elements for unknown exponent e.g. $id_{\mathsf{pw}} = g^{\mathsf{pw}}$ and $id_{f(\mathsf{pw})} = g^{f(\mathsf{pw})}$, the elements are also added to the table with functions x, $f(x)$ as placeholder for the exponent (x, id_{pw}), $(f(x), id_{f(\mathsf{pw})})$.

Game $\mathbf{G}_0$: Real execution
This is the real execution of Π_{faPAKE} where the environment $\mathcal{Z}$ runs the protocol Fig. 3 (UC execution is given in Appendix B) with parties $\mathcal{P}_C$ and $\mathcal{P}_S$, both having access to ideal functionalities $\mathcal{F}^{n}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{IC}}$, and the dummy adversary $\mathcal{A}$.

Game $\mathbf{G}_1$: Fully simulated
We now make purely conceptual changes without modifying the interfaces of $\mathcal{Z}$. First we add a relay ITI between each wire of the parties and $\mathcal{Z}$. These relays are in fact the dummy parties from [Can01]. In addition another relay, denoted by $\mathcal{F}$, is added covering all of the wires between the dummy and the real parties (all wires first enter $\mathcal{F}$ and then are routed according to the original wires to the real parties). Lastly we group all existing instances from the previous game (this includes all the real parties, ideal functionalities and $\mathcal{A}$) in one machine and call it the simulator, or $\mathcal{S}$ for short. We emphasize that $\mathcal{S}$ is now responsible for executing $\mathcal{F}_{\mathsf{IC}}$ and $\mathcal{F}^{n}_{\mathsf{OT}}$. The differences from this game to the previous one are illustrated in Figure 7 .

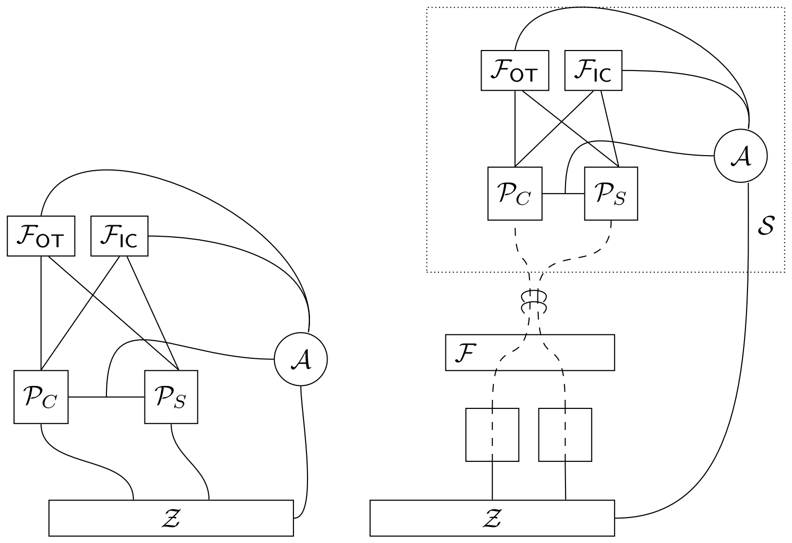


Figure 7: Transition from game $\mathbf{G}_0$ (left) to game $\mathbf{G}_1$ (right), showing a setting where both parties are honest.

Game $\mathbf{G}_2$: Password File Attacks/Server Compromise
In this game we change the simulation to simulate the password file attacks (stealing the password file)

without receiving the password file. In addition the simulation does not require the password of the server in case it is queried to the GGM and the file is stolen.

Modifications to $\mathcal{F}$: Add StorePwdFile, StealPwdFile, TestPwd and OfflineTestPwd interfaces to $\mathcal{F}$ as in Figures 1 and 2 with leakage functions $L_c^P(\mathsf{pw}, \mathsf{pw}') = L_m^P(\mathsf{pw}, \mathsf{pw}') = \mathsf{pw}$ and $L_f^P(\mathsf{pw}, \mathsf{pw}') =$ "wrong guess". $\mathcal{F}$ still forwards the server's password to $\mathcal{S}$ upon the registration of the password file.

Modifications to $\mathcal{S}$: First we note that, opposed to the previous game where all inputs were relayed to $\mathcal{S}$, due to $\mathcal{F}$ now implementing interface StorePwdFile, $\mathcal{S}$ will not receive the trigger $(\textsc{StorePwdFile}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw})$ to simulate the server's password file registration. Instead, we detail below when (and how) $\mathcal{S}$ will produce a file.

- Upon v from $\mathcal{Z}$ for the GGM, $\mathcal{S}$ sends $(\textsc{OfflineTestPwd}, \mathsf{sid}, v)$ to $\mathcal{F}$. If $\mathcal{F}$ returns an answer including the server's password pw, $\mathcal{S}$ simulates the password registration phase of the server using pw.
- Upon receiving a query from $\mathcal{Z}$ to the GGM that constitutes a linear combination of $n-2t$ shares $a_{b_i,i}$, $\mathcal{S}$ sets $\mathsf{pw}_i := b_i$ for $n-2t$ values of i. $\mathcal{S}$ sets all $2t$ missing bits of $\mathsf{pw} := \mathsf{pw}_1 ... \mathsf{pw}_n$ to 0 and sends $(\textsc{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw})$ to $\mathcal{F}$.
  - If $\mathcal{F}$ returns an answer including the server's password pw', $\mathcal{S}$ simulates the password registration phase of the server using pw'. After having generated the values $g^{s_1}, ..., g^{s_n}$, he programs the GGM to contain pairs $(s_j, id_{j,\mathsf{pw}'_j}), j \in [n]$ where pw'_j denotes the j-th bit of the password pw'.
  - If $\mathcal{F}$ returns only "wrong guess" without any password, $\mathcal{S}$ does nothing (i.e., gives back activation to $\mathcal{Z}$).
- Upon receiving $(\textsc{StealPwdFile}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{S}$ sends $(\textsc{StealPwdFile}, \mathsf{sid})$ to $\mathcal{F}$.
  - If $\mathcal{F}$ replies "no password file", $\mathcal{S}$ forwards "no password file" to $\mathcal{Z}$.
  - If $\mathcal{F}$ replies "password file stolen", $\mathcal{S}$ creates a password file containing fresh group element identifiers $((id_{i,1}, id_{i,0})_{i\in[n]}, id_P, id_K)$ and sends it to $\mathcal{Z}$.
  - If $\mathcal{F}$ replies with a password pw', then $\mathcal{S}$ simulates the password registration phase of the server using pw'. For every password pw with $d(\mathsf{pw}, \mathsf{pw}') \le \gamma$, if $\mathcal{S}$ submitted $(\textsc{OfflineTestPwd}, \mathsf{pw})$ due to $\mathcal{Z}$ querying the GGM with a linear combination of $n-2t$ shares as described above, $\mathcal{S}$ additionally programs the GGM as described above.

The main difference between this game and the previous game is that the password file is not forwarded to $\mathcal{S}$ and upon the server being compromised by $\mathcal{Z}$, $\mathcal{S}$ generates the password file by choosing fresh group elements. $\mathcal{Z}$ can only test a password against the file if it either queries pw to the GGM, or reconstructs k by sending a linear combination of at least $n-2t = l$ identifiers of the password file to the GGM (which includes randomization attacks). Note that less identifiers do not reveal anything about k due to $(l-1)$-privacy of the RSSExp. In both cases $\mathcal{S}$ sends an $(\textsc{OfflineTestPwd}, \mathsf{pw})$ to $\mathcal{F}$ and learns if the password queried by the $\mathcal{Z}$ is close to server's password. If so $\mathcal{S}$ can program the GGM. Note that due to the fact that GGM outputs elements at random querying far away passwords or bits of the password would not revile any information to the environment.

If the simulator learns pw from the OfflineTestPwd query, he can perfectly simulate the server's password file. Otherwise the environment has not yet queried pw to the GGM and thus $\mathcal{Z}$'s view of the password file is fresh identifiers in this as well as the previous game. Since the $\mathcal{F}_{\mathsf{IC}}$ generates ciphertexts randomly and the output of the GGM is random as well, $\mathcal{Z}$ cannot distinguish the ciphertext generated in this game and the previous game. In addition $\mathcal{S}$ outputs $K_s \leftarrow PRG(K')$ where $k' \xleftarrow{\$} \mathbb{Z}_q$ and $K' \leftarrow id_K^{k'}$ as the server's key which is indistinguishable from the $\mathbf{G}_1$. Hence $\mathbf{G}_2$ is indistinguishable from $\mathbf{G}_1$.

Game $\mathbf{G}_3$: Functionality Produces and Forwards The Key

In this game the simulator does not generate the keys on behalf of the honest parties and uses $\mathcal{F}$ in order to send keys to the parties. In addition in case of two honest parties $\mathcal{S}$ does not use the password of the parties.

Modifications to $\mathcal{S}$:

- **Two Honest Parties:**
  - Upon receiving a request from $\mathcal{Z}$ to replace the ciphertext c with c' that is being sent by the Server to the Client, $\mathcal{S}$ checks:
    - If $c \neq c'$ then send $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \perp)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \perp)$ to $\mathcal{F}$
    - Otherwise ($\mathcal{Z}$ does not interrupt) Ignore the keys produced by $\mathcal{P}_C$ and $\mathcal{P}_S$, produce the keys K_c and K_s uniformly at random from the key space and send $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, K_c)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_s)$ to $\mathcal{F}$
- **Impersonation Attack:**
  - Upon $\mathcal{Z}$ impersonating the server and sending a ciphertext c which is not obtained by querying $\mathcal{F}_{\mathsf{IC}}$, $\mathcal{S}$ chooses x from $\mathbb{Z}_q$ uniformly at random and sends $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \perp)$ to $\mathcal{F}$.
  - Upon $\mathcal{Z}$ impersonating the server and sending the input values $(\textsc{Enc}, K', (\mathbf{A}, P))$ to $\mathcal{F}_{\mathsf{IC}}$, $\mathcal{S}$ checks if the GGM contains the pair (pw, P).
    - If there is no tuple (pw, P) in GGM:
      - If there was query $f(id_{P'})$ to the GGM for $id_{P'}$ sent by $\mathcal{S}$ as second-to-last value in the password file and $\mathcal{S}$ answered this query with P, send $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid}, f)$ to $\mathcal{F}$.
      - Else $\mathcal{S}$ simulates that $\mathcal{P}_C$ chooses x from $\mathbb{Z}_q$ uniformly at random, set $K_c \leftarrow PRG(x)$ and sends $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \perp)$ to $\mathcal{F}$.
    - Else $\mathcal{S}$ sends $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw})$ to $\mathcal{F}$.
    - After sending $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid}, f)$ or $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw})$ to $\mathcal{F}$ check:
      - Upon receiving "wrong guess" from $\mathcal{F}$, $\mathcal{S}$ chooses a random $\mathsf{pw}_{\mathcal{S}}$ and continues the simulation of the client with $\mathsf{pw}_{\mathcal{S}}$ but $\mathcal{S}$ chooses x uniformly at random and sets $K_c \leftarrow PRG(x)$.
      - Otherwise upon receiving "correct guess" and the client's password $\widetilde{\mathsf{pw}}$, $\mathcal{S}$ continues the simulation of the client with $\widetilde{\mathsf{pw}}$.
      - Regardless of the $\mathcal{F}$'s response, upon calculating K_c, $\mathcal{S}$ sends $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_c)$ to $\mathcal{F}$.
- **One Corrupt Party:**
  - Upon the calculation of K_C and K_S by the simulated parties, $\mathcal{S}$ sends $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, K_c)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_c)$ to $\mathcal{F}$.

Modifications to $\mathcal{F}$: $\mathcal{F}$ stores all relayed values in form of records as done by $\mathcal{F}^M_{\mathsf{faPAKE}}$ in Figure 1. Additionally, the $\textsc{UsrSession}$, $\textsc{SrvSession}$ and $\textsc{NewKey}$ interfaces of $\mathcal{F}^M_{\mathsf{faPAKE}}$ are added to $\mathcal{F}$, with the difference that $\mathcal{F}$ still includes passwords in the messages to $\mathcal{S}$.

In case both parties are honest, the environment has no information about the pre-image of K_c and K_s, namely x and K'. In case of $d(\mathsf{pw}, \widetilde{\mathsf{pw}}) \leq \delta$, in $\mathbf{G}_3$, $\mathcal{F}$ outputs the same key to both parties. In $\mathbf{G}_2$, in case $d(\mathsf{pw}, \widetilde{\mathsf{pw}}) \leq \delta$, both parties outputted the same key due to $(l+t)$-robustness of the RSSExp scheme. This is because an honest client with close password would get at least $l + t = n - \delta$ correct shares and at most $t = \delta$ wrong shares from the server. According to $(l+t)$-robustness the client can reconstruct the same $\widetilde{K}$ from these n shares except with negligible probability.

In case of $d(\mathsf{pw}, \widetilde{\mathsf{pw}}) > \delta$, without any information about the secret key given to $\mathcal{F}_{\mathsf{IC}}$, the output of $\mathcal{F}_{\mathsf{IC}}$ is indistinguishable from a random string of the same length. Since the environment has no information about the uniformly distributed K', it can only learn the encrypted message by guessing K', which succeeds with negligible probability. Since x is either $\widetilde{K}$ or chosen uniformly at random, the keys $K_c = PRG(x)$ and $K_s = PRG(K')$ are indistinguishable from a random value except with negligible probability by the PRG assumption. Hence if $\mathcal{Z}$ does not interrupt the protocol execution between two honest parties, $\mathcal{S}$ can ignore the passwords, c and keys of the two parties and submits $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, K_c)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_s)$ to $\mathcal{F}$ for randomly generated K_c and K_s

and $\mathbf{G}_3$ would be indistinguishable from $\mathbf{G}_2$ except with the negligible probability of distinguishing between the random keys generated by $\mathcal{F}$ in $\mathbf{G}_3$ and the pseudorandom keys generated by the parties in $\mathbf{G}_2$.

Finally, without knowing the value K' used by the honest server, the environment cannot produce a new valid ciphertext c'. Therefore, the Client can **not** decrypt c' to a valid massage $((a'_{0,i}, a'_{1,i})_{i\in[n]}, g^{\mathsf{pw}})$ and hence the Client will produce a random key. Hence, $\mathcal{S}$ can ignore the passwords, c and keys of the two parties and submits $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \bot)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \bot)$ to $\mathcal{F}$. Therefore in case both parties are honest $\mathbf{G}_3$ is indistinguishable from $\mathbf{G}_2$ except with negligible probability.

In case the environment impersonates the server, i.e. the server is not participating in the protocol execution while $\mathcal{Z}$ communicates with the client, if $\mathcal{Z}$ does not query $\mathcal{F}_{\mathsf{IC}}$ and sends a random ciphertext c to the client, the honest client can **not** decrypt c' to a valid massage $((a'_{0,i}, a'_{1,i})_{i\in[n]}, g^{\mathsf{pw}})$. If $\mathcal{Z}$ queries $\mathcal{F}_{\mathsf{IC}}$ but uses a value P which is not produced by the GGM, the client cannot find a $\overline{\mathsf{pw}}$ such that $g^{\overline{\mathsf{pw}}} = P$. Hence in both of these cases the honest client will generate x uniformly at random. There are two possibilities in case c is generated by $\mathcal{F}_{\mathsf{IC}}$ and P is produced by GGM. Either P is generated upon $\mathcal{Z}$ querying the value pw to GGM, or P is generated by GGM upon $\mathcal{Z}$ querying $f(id_{P'})$ to GGM where $id_{P'}$ is part if the password file (representing g^{pw}) generated by $\mathcal{S}$ at random and given to $\mathcal{Z}$. If there is a tuple (pw, P) in the GGM, $\mathcal{S}$ sends $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw})$ to $\mathcal{F}$ and checks if this password is close to the client's password. Otherwise if P is generated by applying f to $id_{P'}$, $\mathcal{S}$ sends $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid}, f)$ to $\mathcal{F}$ and checks if after applying f to server's password the passwords are still close. Regardless, after the $\textsc{TestPwd}$ or $\textsc{Impersonate}$ interface is used the password record will either be marked `compromised` or `interrupted`, in the first cases $\mathcal{F}$ outputs the keys that it gets as input from $\mathcal{S}$ and a randomly chosen key otherwise. If $\textsc{TestPwd}$ or $\textsc{Impersonate}$ interface return "correct guess" (and the record is marked `compromised`) then $\mathcal{S}$ learns the client's password and can simulate the execution perfectly. Otherwise $\mathcal{Z}$ has queried $\mathcal{F}_{\mathsf{IC}}$ using a password which is not close to client's password. In this case the simulator can simulate the client with a random $\mathsf{pw}_{\mathcal{S}}$ and since the honest client does not have a password $\overline{\mathsf{pw}}$ such that that $g^{\overline{\mathsf{pw}}} = P$ it will choose x randomly. Hence in both cases $\mathcal{F}$ produces keys for the client which are indistinguishable to the keys produced in $\mathbf{G}_2$ except with negligible probability. Therefore in case $\mathcal{Z}$ impersonates the server, this case $\mathbf{G}_3$ is indistinguishable from $\mathbf{G}_2$ except with negligible probability. We recall that the case where the $\mathcal{Z}$ re-randomizes the password file after stealing it is handled in $\mathbf{G}_2$.

If one of the parties is corrupted, $\mathcal{F}$ outputs the keys that it gets as input from $\mathcal{S}$. Since the protocol simulation still uses the password of the honest party, simulation of transcript, $\mathcal{F}_{\mathsf{IC}}$ and $\mathcal{F}_{\mathsf{OT}}$ is as in the game before, and thus $\mathbf{G}_3$ is indistinguishable from $\mathbf{G}_2$ game except with negligible probability.

Game $\mathbf{G}_4$: Replacing Ciphertext With Random Ciphertext and Ignoring the Password of the Parties In Case of Two Honest Parties:

In this game $\mathcal{S}$ does not generate a ciphertext c on behalf of the server and chooses a random ciphertext in case both parties are honest and participating in the protocol (this does **not** include the impersonation case).

Modifications to $\mathcal{S}$: In case of both parties being honest, the ciphertext c is replaced by a value chosen uniformly at random from the ciphertext space.

Since the functionality $\mathcal{F}_{\mathsf{IC}}$ generates ciphertexts that are chosen uniformly at random, the environment can only distinguish the ciphertext from a random string if it has some information about the key K'. Since K' is chosen uniformly at random by the honest server, the environment can only succeed in distinguishing the ciphertext in $\mathbf{G}_4$ and $\mathbf{G}_3$ with negligible probability by guessing the correct key. The Simulator's calculations in $\mathbf{G}_3$ in the case where both parties are honest and participating in the protocol execution are independent from the client's and server's password. We recall that in case both parties are honest and participating in the protocol execution, no $\textsc{TestPwd}$ or $\textsc{Impersonate}$ queries are made to $\mathcal{F}$. Therefore $\mathbf{G}_4$ and $\mathbf{G}_3$ are indistinguishable.

Game $\mathbf{G}_5$: Ignoring Client's Password in case of Corrupted Server

In this game $\mathcal{S}$ does not use the client's and server's password in order to simulate the honest client and instead $\mathcal{S}$ uses the TESTPWD interface.

Modifications to $\mathcal{S}$: $\mathcal{S}$ ignores the password of the client, waits until the server sends the ciphertext c to the client. If $\mathcal{Z}$ has not received c from querying $\mathcal{F}_{\mathsf{IC}}$ or $\mathcal{Z}$ queries $\mathcal{F}_{\mathsf{IC}}$ on input values $(\text{ENC}, K', (\mathbf{A}, P))$ where GGM does not contain the pair (pw, P), $\mathcal{S}$ simulates that $\mathcal{P}_C$ chooses x from $\mathbb{Z}_q$ uniformly at random. Else $\mathcal{S}$ sends $(\text{TESTPWD}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw})$ to $\mathcal{F}$. Upon receiving "wrong guess" from $\mathcal{F}$, $\mathcal{S}$ chooses a random $\mathsf{pw}_{\mathcal{S}}$ with $d(\mathsf{pw}, \mathsf{pw}_{\mathcal{S}}) > \gamma$ and continues the simulation of the client with $\mathsf{pw}_{\mathcal{S}}$. Otherwise upon receiving "correct guess" and the client's password $\widetilde{\mathsf{pw}}$, $\mathcal{S}$ continues the simulation of the client with $\widetilde{\mathsf{pw}}$.

We now show that the key outputted by the honest client in $\mathbf{G}_5$ is indistinguishable from the key that the client outputs in $\mathbf{G}_4$. In case $\mathcal{S}$ receives a password from $\mathcal{F}$, the simulation in $\mathbf{G}_5$ and $\mathbf{G}_4$ are perfectly indistinguishable. In case $\mathcal{S}$ receives "wrong guess" from $\mathcal{F}$ it holds that $d(\widetilde{\mathsf{pw}}, \mathsf{pw}) > \gamma$, where $\widetilde{pw}$ denotes the password of the honest client. This means that the honest client obtains at most $n - \gamma - 1$ correct shares. Since $(l-1) = (n - \gamma - 1)$, $(l-1)$-privacy of the RSSExp guarantees that the client in the previous game as well as in this game, cannot reconstruct the password and will output $PRG(x)$ for some randomly chosen x. If $\mathcal{Z}$ chose c without querying $\mathcal{F}_{\mathsf{IC}}$, the client cannot decrypt c to a correct value and will receive a randomly chosen plain-text. Finally if GGM does not contain the pair (pw, P) the honest client cannot reconstruct a password $\overline{\mathsf{pw}}$ such that $g^{\overline{\mathsf{pw}}} = P$, hence the client will output $PRG(x)$ for some randomly chosen x. Thus all three cases are indistinguishable from the view of $\mathcal{Z}$.

Game $\mathbf{G}_6$: Ignoring Servers's Password in case of Corrupted Client

In this game $\mathcal{S}$ does not use the server's and client's password in order to simulate the honest server.

Modifications to $\mathcal{S}$: In case of a corrupted client, $\mathcal{S}$ chooses a string c uniformly at random from the ciphertext space of $\mathcal{F}_{\mathsf{IC}}$ and sends it to the client. Upon the client sending the inputs $(\text{REC}, (\widetilde{\mathsf{pw}}_i)_{i\in[n]})$ to $\mathcal{F}_{\mathsf{OT}}$, $\mathcal{S}$ sets $\widetilde{\mathsf{pw}} = \widetilde{\mathsf{pw}}_1 || \cdots || \widetilde{\mathsf{pw}}_n$ and sends $(\text{TESTPWD}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \widetilde{\mathsf{pw}})$ to $\mathcal{F}$.

- If $\mathcal{F}$ returns "correct guess" alongside a password pw, $\mathcal{S}$ computes the correct password file $\text{FILE} \leftarrow ((a_{0,i}, a_{1,i})_{i\in[n]}, g^{\mathsf{pw}}, K)$ by using pw. Afterwards $\mathcal{S}$ computes $K' \leftarrow K^{k'}$ where k' is chosen uniformly at random from $\mathbb{Z}_q$, and uses the values $(a_{0,i}^{k'}, a_{1,i}^{k'})_{i\in[n]}$ to simulate $\mathcal{F}_{\mathsf{OT}}$. Upon the client querying $\mathcal{F}_{\mathsf{IC}}$ with the values (DEC, K', c), $\mathcal{S}$ returns the tuple $((a_{0,i}^{k'}, a_{1,i}^{k'})_{i\in[n]}, g^{\mathsf{pw}})$.
- If $\mathcal{F}$ returns "wrong guess", $\mathcal{S}$ chooses the values $((a_{0,i}, a_{1,i})_{i\in[n]}, g^{\mathsf{pw}}, K)$ uniformly at random and uses the values $(a_{0,i}^{k'}, a_{1,i}^{k'})_{i\in[n]}$ to simulate $\mathcal{F}_{\mathsf{OT}}$ where k' is chosen uniformly at random from $\mathbb{Z}_q$. Upon the query $(\text{DEC}, \widetilde{K}, c)$ to $\mathcal{F}_{\mathsf{IC}}$, $\mathcal{S}$ returns m which is chosen uniformly at random from the message space.

Modifications to $\mathcal{F}$: $\mathcal{F}$ does not forward the server's and client's password of the parties to $\mathcal{S}$.

$\mathcal{F}_{\mathsf{IC}}$ produces random ciphertexts and since $\mathcal{Z}$ does not know the key K' used and chosen uniformly at random by the server, it cannot distinguish the real cipher text and a randomly chosen one except with negligible probability. Hence $\mathcal{S}$ can replace the output of $\mathcal{F}_{\mathsf{IC}}$ by a uniform random value.

Upon making the TESTPWD query, if the server uses a password within distance γ from $\widetilde{\mathsf{pw}}$, the simulator receives "correct guess", learns the server's password pw. In this case $\mathcal{S}$ can perfectly simulate the server and decrypt the randomly chosen ciphertext to a valid message of the form $(\mathbf{A}, P)$, hence in this case $\mathbf{G}_6$ is indistinguishable from $\mathbf{G}_5$. If the password used by the server is not close to the one used by the corrupted client, namely $d(\mathsf{pw}, \widetilde{\mathsf{pw}}) > \gamma$, all encrypted values can be replaced by random values due to the properties of $\mathcal{F}_{\mathsf{IC}}$ and the fact that $(l-1) = (n - \gamma - 1)$-privacy of the RSSExp ensures that the client cannot reconstruct the correct encryption key K' and will only reconstruct a random key. This decrypted message is therefore indistinguishable from a random value, again due to the definition of $\mathcal{F}_{\mathsf{IC}}$.

Finally in $\mathbf{G}_6$, $\mathcal{S}$ does not uses the client's and server's password in any of the cases (both parties honest or one party corrupted). Therefore $\mathcal{F}$ does not forward them to $\mathcal{S}$ and $\mathbf{G}_6$ would be indistinguishable from $\mathbf{G}_5$ except with negligible probability.

In the last game, it holds that $\mathcal{F} = \mathcal{F}^{P}_{\mathsf{faPAKE}}$. For this, it is crucial to observe that $\mathcal{F}$ is not forwarding the passwords of client and server to $\mathcal{S}$ anymore, and indeed our simulation in $\mathbf{G}_6$ is independent of the true passwords of both parties. Thus, $\mathbf{G}_6$ is equal to the ideal execution with the simulator given below. This concludes our proof of the theorem. ☐

Simulator code

1. Upon v from $\mathcal{Z}$ for the GGM, $\mathcal{S}$ sends $(\textsc{OfflineTestPwd}, \mathsf{sid}, v)$ to $\mathcal{F}$. If $\mathcal{F}$ returns an answer including the server's password pw, $\mathcal{S}$ simulates the password registration phase of the server using pw.

2. Upon receiving a query from $\mathcal{Z}$ to the GGM that constitutes a linear combination of $n-2t$ shares $a_{b_i,i}$, $\mathcal{S}$ sets $\mathsf{pw}_i := b_i$ for $n-2t$ values of i. $\mathcal{S}$ sets all $2t$ missing bits of $\mathsf{pw} := \mathsf{pw}_1 ... \mathsf{pw}_n$ to 0 and sends $(\textsc{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw})$ to $\mathcal{F}$.
   - If $\mathcal{F}$ returns an answer including the server's password pw', $\mathcal{S}$ simulates the password registration phase of the server using pw'. After having generated the values $g^{s_1}, ..., g^{s_n}$, he programs the GGM to contain pairs $(s_j, id_{j,\mathsf{pw}'_j}), j \in [n]$ where pw'_j denotes the j-th bit of the password pw'.
   - If $\mathcal{F}$ returns only "wrong guess" without any password, $\mathcal{S}$ does nothing (i.e., gives back activation to $\mathcal{Z}$).

3. Upon receiving $(\textsc{StealPwdFile}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{S}$ sends $(\textsc{StealPwdFile}, \mathsf{sid})$ to $\mathcal{F}$.
   - If $\mathcal{F}$ replies "no password file", $\mathcal{S}$ forwards "no password file" to $\mathcal{Z}$.
   - If $\mathcal{F}$ replies "password file stolen", $\mathcal{S}$ creates a password file containing fresh group element identifiers $((id_{i,1}, id_{i,0})_{i\in[n]}, id_P, id_K)$ and sends it to $\mathcal{Z}$.
   - If $\mathcal{F}$ replies with a password pw', then $\mathcal{S}$ simulates the password registration phase of the server using pw'. For every password pw with $d(\mathsf{pw}, \mathsf{pw}') \leq \gamma$, if $\mathcal{S}$ submitted $(\textsc{OfflineTestPwd}, \mathsf{pw})$ due to $\mathcal{Z}$ querying the GGM with a linear combination of $n-2t$ shares as described above, $\mathcal{S}$ additionally programs the GGM as described above.
   - In case of two honest parties upon receiving a request from $\mathcal{Z}$ to replace the ciphertext c with c' that is being sent by the Server to the Client, $\mathcal{S}$ checks:
     - If $c \neq c'$ then choose x uniformly at random from $\mathbb{Z}_q$ and set $K_c \leftarrow PRG(x)$. In addition, send $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \perp)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \perp)$ to $\mathcal{F}$
     - Otherwise ($\mathcal{Z}$ does not interrupt) Ignore the keys produced by $\mathcal{P}_C$ and $\mathcal{P}_S$, produce the keys K_c and K_s uniformly at random from the key space and send $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, K_c)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_s)$ to $\mathcal{F}$
   - Upon $\mathcal{Z}$ impersonating the server and sending a ciphertext c which is not obtained by querying $\mathcal{F}_{\mathsf{IC}}$, $\mathcal{S}$ chooses x from $\mathbb{Z}_q$ uniformly at random and sends sends $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \perp)$.
   - Upon $\mathcal{Z}$ impersonating the server and sending the input values $(\textsc{Enc}, K', (\mathbf{A}, P))$ to $\mathcal{F}_{\mathsf{IC}}$, $\mathcal{S}$ checks if the GGM contains the pair (pw, P).
     - If there is no tuple (pw, P) in GGM:
       - If there was query $f(id_{P'})$ to the GGM for $id_{P'}$ sent by $\mathcal{S}$ as second-to-last value in the password file and $\mathcal{S}$ answered this query with P, send $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid}, f)$ to $\mathcal{F}$.
       - Else $\mathcal{S}$ simulates that $\mathcal{P}_C$ chooses x from $\mathbb{Z}_q$ uniformly at random, set $K_c \leftarrow PRG(x)$ and sends $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \perp)$ to $\mathcal{F}$.
     - Else $\mathcal{S}$ sends $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw})$ to $\mathcal{F}$.

- After sending ($\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid}, f$) or ($\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw}$) to $\mathcal{F}$ check:
  * Upon receiving "wrong guess" from $\mathcal{F}$, $\mathcal{S}$ chooses a random $\mathsf{pw}_\mathcal{S}$ and continues the simulation of the client with $\mathsf{pw}_\mathcal{S}$ but $\mathcal{S}$ chooses x uniformly at random and sets $K_c \leftarrow PRG(x)$.
  * Otherwise upon receiving "correct guess" and the client's password $\widetilde{\mathsf{pw}}$, $\mathcal{S}$ continues the simulation of the client with $\widetilde{\mathsf{pw}}$.
  * Regardless of the $\mathcal{F}$'s response, upon calculating K_c, $\mathcal{S}$ sends ($\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_c$) to $\mathcal{F}$.

- In case of one corrupted party, upon the calculation of K_C and K_S by the simulated parties, $\mathcal{S}$ sends ($\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, K_c$) and ($\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_c$) to $\mathcal{F}$.

4. In case of both parties being honest, the ciphertext c is replaced by a value chosen uniformly at random from the ciphertext space.

5. In case of corrupted server, $\mathcal{S}$ ignores the password of the client, waits until the server sends the ciphertext c to the client. If $\mathcal{Z}$ has not received c from querying $\mathcal{F}_{\mathsf{IC}}$ or $\mathcal{Z}$ queries $\mathcal{F}_{\mathsf{IC}}$ on input values ($\textsc{Enc}, K', (\mathbf{A}, P)$) where GGM does not contain the pair (pw, P), $\mathcal{S}$ simulates that $\mathcal{P}_C$ chooses x from $\mathbb{Z}_q$ uniformly at random. Else $\mathcal{S}$ sends ($\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw}$) to $\mathcal{F}$. Upon receiving "wrong guess" from $\mathcal{F}$, $\mathcal{S}$ chooses a random $\mathsf{pw}_\mathcal{S}$ with $d(\mathsf{pw}, \mathsf{pw}_\mathcal{S}) > \gamma$ and continues the simulation of the client with $\mathsf{pw}_\mathcal{S}$. Otherwise upon receiving "correct guess" and the client's password $\widetilde{\mathsf{pw}}$, $\mathcal{S}$ continues the simulation of the client with $\widetilde{\mathsf{pw}}$.

6. In case of a corrupted client, $\mathcal{S}$ chooses a string c uniformly at random from the ciphertext space of $\mathcal{F}_{\mathsf{IC}}$ and sends it to the client. Upon the client sending the inputs ($\textsc{Rec}, (\widetilde{\mathsf{pw}}_i)_{i \in [n]}$) to $\mathcal{F}_{\mathsf{OT}}$, $\mathcal{S}$ sets $\widetilde{\mathsf{pw}} = \widetilde{\mathsf{pw}}_1 || \cdots || \widetilde{\mathsf{pw}}_n$ and sends ($\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \widetilde{\mathsf{pw}}$) to $\mathcal{F}$.
   - If $\mathcal{F}$ returns "correct guess" alongside a password pw, $\mathcal{S}$ computes the correct password file $\textsc{file} \leftarrow ((a_{0,i}, a_{1,i})_{i \in [n]}, g^{\mathsf{pw}}, K)$ by using pw. Afterwards $\mathcal{S}$ computes $K' \leftarrow K^{k'}$ where k' is chosen uniformly at random from $\mathbb{Z}_q$, and uses the values $(a_{0,i}^{k'}, a_{1,i}^{k'})_{i \in [n]}$ to simulate $\mathcal{F}_{\mathsf{OT}}$. Upon the client querying $\mathcal{F}_{\mathsf{IC}}$ with the values ($\textsc{Dec}, K', c$), $\mathcal{S}$ returns $((a_{0,i}^{k'}, a_{1,i}^{k'})_{i \in [n]}, g^{\mathsf{pw}})$.
   - If $\mathcal{F}$ returns "wrong guess", $\mathcal{S}$ chooses the values $((a_{0,i}, a_{1,i})_{i \in [n]}, g^{\mathsf{pw}}, K)$ uniformly at random and uses the values $(a_{0,i}^{k'}, a_{1,i}^{k'})_{i \in [n]}$ to simulate $\mathcal{F}_{\mathsf{OT}}$ where k' is chosen uniformly at random from $\mathbb{Z}_q$. Upon the query ($\textsc{Dec}, \widetilde{K}, c$) to $\mathcal{F}_{\mathsf{IC}}$, $\mathcal{S}$ returns $((a_{0,i}^{k'}, a_{1,i}^{k'})_{i \in [n]}, g^{\mathsf{pw}})$ which were chosen uniformly at random.

D Proving Theorem 5.1

We first give a modified $\textsc{TestPwd}$ interface that allows proving security of Π_{transf} in Figure 8. Additionally, we give a $\textsc{NewKey}$ interface which requires explicit authentication, taken from, e.g., [CHK+05, Hes19].

Game $\mathbf{G}_0$: Real execution

This is the real execution of Π_{transf} where the environment $\mathcal{Z}$ runs the protocol 4 with parties $\mathcal{P}_C$ and $\mathcal{P}_S$, both having access to an ideal aPAKE functionality $\mathcal{F}_{\mathsf{aPAKE}}$ and $\mathcal{F}_{\mathsf{IC}}$, and an adversary Adv that, w.l.o.g., can be assumed to be the dummy adversary as shown in [Can01].

Game $\mathbf{G}_1$: Fully simulated

We now make purely conceptual changes without modifying the interfaces of $\mathcal{Z}$. First we add a relay between each wire of the parties and $\mathcal{Z}$. These relays are in fact the dummy parties from [Can01]. In addition another relay, denoted by $\mathcal{F}$, is added covering all of the wires between the dummy and the real parties (all wires first enter $\mathcal{F}$ and then are relayed according to the original wires to the real parties). Lastly we group all existing instances from the previous game in $\mathcal{Z}$ (this includes all the real parties ideal functionalities etc.) in one machine and call it $\mathcal{S}$. We emphasize that $\mathcal{S}$ is now responsible for executing the core of $\mathcal{F}_{\mathsf{IC}}$, $\mathcal{F}_{\mathsf{RO}}$ and $\mathcal{F}_{\mathsf{aPAKE}}$.

On $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}'\})$ from $\mathcal{S}$, if there is a record $(\mathsf{ssid}, \mathcal{P}_S, \mathcal{P}_C, \mathsf{pw})$ marked `fresh` do:

- If less than $n+1$ TestPwd queries are sent by $\mathcal{S}$ and if $d(\mathsf{pw}', \mathsf{pw}) \leq \delta$, choose a password denoted by pw'' uniformly at random such that pw'' is not marked *tested* and $d(\mathsf{pw}'', \mathsf{pw}) \leq \delta$, mark pw'' as *tested* and do:
  - If $\mathsf{pw}'' = \mathsf{pw}'$, mark record `compromised` and send "correct guess" to $\mathcal{S}$;
  - If $\mathsf{pw}'' \neq \mathsf{pw}'$ send "wrong guess" to $\mathcal{S}$ and check:
    - if this is the $n+1$th TestPwd query and the record is not marked `compromised`, mark record `interrupted`.

On $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw}'\})$ from $\mathcal{S}$, if there is a record $(\mathsf{ssid}, \mathcal{P}_C, \mathcal{P}_S, \mathsf{pw})$ marked `fresh` do:

- If less than $n+1$ TestPwd queries are sent by $\mathcal{S}$, do:
  - If $\mathsf{pw}' = \mathsf{pw}$, mark record `compromised` and send "correct guess" to $\mathcal{S}$;
  - If $\mathsf{pw}' \neq \mathsf{pw}$ send "wrong guess" to $\mathcal{S}$ and check:
    - if this is the $n+1$th TestPwd query and the record is not marked `compromised`, mark record `interrupted`.

Figure 8: n-times exact TestPwd functionality. In case of a TestPwd query against the client's password, the functionality checks if the queried password and the client's password are equal. In case of a TestPwd query against the server's password, the functionality checks if the queried password is close to the server's password and if so randomly chooses an untested password close to server's password and checks if this password is equal to the queried password.

Game $\mathbf{G}_2$: Functionality Produces and Forwards The Key

Modifications to $\mathcal{S}$: Upon the calculation of K_C and K_S by the simulated parties, $\mathcal{S}$ sends $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, K_C)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_S)$ to $\mathcal{F}$.

Modifications to $\mathcal{F}$: $\mathcal{F}$ stores all relayed values in form of records as done by $\mathcal{F}'_{\mathsf{faPAKE}}$. Additionally, the UsrSession and SrvSession interfaces from Figure 1 and the NewKey interface from Figure 9 are added to $\mathcal{F}$, with the difference that $\mathcal{F}$ still includes passwords in the messages to $\mathcal{S}$.

From the point of view of $\mathcal{Z}$, the only difference between this and the previous game is that $\mathcal{F}$ produces the key in case of an honest session. In case one of the parties is corrupted the functionality outputs the keys that it gets as input from $\mathcal{S}$, which are still produced from the true passwords (the one of the client is learned via UsrSession queries from $\mathcal{F}$, the one of the server still gets relayed by $\mathcal{F}$ within the StorePwdFile input).

In case both parties are honest and the passwords are close, the environment has no information about the pre-image of K_C and K_S, namely $\mathsf{k}_{C,i}$ and $\mathsf{k}_{S,i}$. Therefore it cannot distinguish K_C and K_S generated by $\mathcal{F}_{\mathsf{RO}}$ from the uniform random session keys generated by $\mathcal{F}$. If the passwords are not close the output of the parties is $\perp$ in both games.

Game $\mathbf{G}_3$: Simulating the Server when there are no Byzantine corruptions and Handling the queries OfflineTestPwd, Impersonate and TestPwd to $\mathcal{F}_{\mathsf{aPAKE}}$ For Corrupted Client and Honest Server

In this game we show how to simulate a password file and the server's part in the $\mathcal{F}_{\mathsf{aPAKE}}$ instances without knowing the true password, but only in case there are no Byzantine corruptions. Since we assume static Byzantine corruption, we can let the simulation depend on the Byzantine corruption status. The simulator keeps track of the passwords used in the internal aPAKEs by the server. To this end the simulator produces a permuted list L which contains the password list used by the server. The

Key Generation and Explicit Authentication

- On $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathsf{k})$ from $\mathcal{S}$ where $|\mathsf{k}| = \lambda$ or $\mathsf{k} = \perp$, if there is a record $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ not marked `completed`, do:
  - If the record is marked `compromised`, or either $\mathcal{P}$ or $\mathcal{P}'$ is corrupted, or $\mathsf{k} = \perp$ and the record is `fresh`, then send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k})$ to $\mathcal{P}$.
  - Else if the record is marked `fresh`, a $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k}')$ tuple was sent to $\mathcal{P}'$, and at that time there was a record $(\mathsf{ssid}, \mathcal{P}', \mathcal{P}, \mathsf{pw}')$ marked `fresh` with $d(\mathsf{pw}, \mathsf{pw}') \leq \delta$, send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k}')$ to $\mathcal{P}$.
  - Else if the record is `interrupted` or if it is `fresh` and there is a record $(\mathsf{sid}, \mathcal{P}', \mathcal{P}, \mathsf{pw}')$ with $d(\mathsf{pw}, \mathsf{pw}') \leq \delta$, then send $(\mathsf{sid}, \mathsf{ssid}, \perp)$ to $\mathcal{P}$.
  - Else, pick $\mathsf{k}'' \overset{\$}{\leftarrow} \{0,1\}^\lambda$ and send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{k}'')$ to $\mathcal{P}$.

  Finally, mark $(\mathsf{ssid}, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ as `completed`.

Figure 9: NewKey interface for explicit authentication.

ordering in the list represents the order in which passwords are input in the $\mathcal{F}_{\mathsf{aPAKE}}$ instances, i.e., the i-th entry is input to the i-th instance of $\mathcal{F}_{\mathsf{aPAKE}}$.

Since we can assume PID-wise corruption, $\mathcal{S}$ is allowed to issue StealPwdFile to all instances of $\mathcal{F}_{\mathsf{aPAKE}}$ and to $\mathcal{F}_{\mathsf{faPAKE}}$ as soon as $\mathcal{Z}$ compromised the server in any protocol instance.

Modifications to $\mathcal{S}$: We modify the simulation in case of no Byzantine corruptions or corrupted client as follows.

Simulating the password file. Upon receiving $(\textsc{StealPwdFile}, \mathsf{sid})$ from $\mathcal{Z}$, $\mathcal{S}$ sends the message $(\textsc{StealPwdFile}, \mathsf{sid})$ to $\mathcal{F}$.

- If $\mathcal{F}$ responds with “no password file”, forward “no password file” to $\mathcal{Z}$.
- Else if $\mathcal{F}$ responds with “password file stolen”, for all values v queried already to the $\mathcal{F}_{\mathsf{RO}}$, $\mathcal{S}$ sends a message $(\textsc{OfflineTestPwd}, \mathsf{sid}, v)$ to $\mathcal{F}$.
  - If $\mathcal{F}$ returns an answer including a password pw, $\mathcal{S}$ executes the password registration phase of the server using pw.
  - Else $\mathcal{S}$ creates a password file containing fresh random elements $(H_i)_{i\in[k]}$ not yet used within the simulation of $\mathcal{F}_{\mathsf{RO}}$.
  - $\mathcal{S}$ uses the password file in the simulation of all $\mathcal{F}_{\mathsf{aPAKE}}$ instances (i.e., $\mathcal{S}$ does not let the simulated $\mathcal{P}_S$ input passwords into any $\mathcal{F}_{\mathsf{aPAKE}}$ anymore).
  - $\mathcal{S}$ sends the resulting password file to the environment.
- Upon $\mathcal{Z}$ sending x to $\mathcal{F}_{\mathsf{RO}}$, $\mathcal{S}$ sends $(\textsc{OfflineTestPwd}, \mathsf{sid}, x)$ to $\mathcal{F}$:
  - If $\mathcal{F}$ returns an answer including a password pw, $\mathcal{S}$ executes the password registration phase of the server using pw. After having generated the values $\mathsf{pw}'_1, ..., \mathsf{pw}'_k$ that are close to the password pw, he programs $\mathcal{F}_{\mathsf{RO}}$ to contain pairs $(\mathsf{pw}'_j, H_j)_{j\in[k]}$ where $(H_j)_{j\in[k]}$ is the password file given to $\mathcal{Z}$ previously.
  - If $\mathcal{F}$ returns only “wrong guess”, $\mathcal{S}$ does nothing (i.e., gives back activation to $\mathcal{Z}$).

Simulating server's participation in $\mathcal{F}_{\mathsf{aPAKE}}$ instances.

- In case $\mathcal{S}$ receives a password pw from $\mathcal{F}$, $\mathcal{S}$ generates a list L as follows: given a password pw, compute all close passwords to pw, store them in a list L and apply a random permutation to the list such that from these close passwords, password's marked as “position i” are at position i and password's marked as “not position i” are not in position i of the list L.
- Upon $\mathcal{Z}$ sending $(\textsc{StealPwdFile}, \mathsf{sid})$ to the i-th $\mathcal{F}_{\mathsf{aPAKE}}$, if the i-th entry of L contains a password pw' then send (“correct guess”, pw') to $\mathcal{Z}$. Otherwise, if not already done $\mathcal{S}$ issues StealPwdFile to $\mathcal{F}$ and relays the $\mathcal{F}$'s answer to $\mathcal{Z}$.

- Upon $\mathcal{Z}$ sending $(\textsc{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}')$ to the i-th $\mathcal{F}_{\mathsf{aPAKE}}$ after sending a $\textsc{StealPwdFile}$ request, if $\mathcal{S}$ has not yet received "correct guess" as reply to $\textsc{StealPwdFile}$ from $\mathcal{F}$ and no password list L was generated before, $\mathcal{S}$ sends $(\textsc{OfflineTestPwd}, \mathsf{sid}, \mathsf{pw}')$ to $\mathcal{F}$. If $\mathcal{F}$ returns "correct guess" and a password pw, $\mathcal{S}$ computes the list L. Regardless, if the list was created in this case or before, check if the i-th element of L is equal to pw' return "correct guess" to $\mathcal{Z}$. Otherwise if no list exists or the previous check fails return "wrong guess" to $\mathcal{Z}$.
- Upon $\mathcal{Z}$ sending $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid})$ to the i-th $\mathcal{F}_{\mathsf{aPAKE}}$ distingish the following cases:
  - **Case both parties are honest (no byzantine corruptions):**
    1. If this is $\mathcal{Z}$'s first impersonate request and a password for the i-th $\mathcal{F}_{\mathsf{aPAKE}}$ is already registered and the password file stolen, aPAKE instance is not marked *non-fresh* and no password list L was generated before, send the message $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{F}$. If $\mathcal{F}$ responds with "correct guess" and a password pw, compute the list L. Otherwise return "wrong guess".
    2. If there exists a list L send $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw}_i)$ to $\mathcal{F}$, where pw_i is the i-th password in the list L. Forward $\mathcal{F}$'s response to $\mathcal{Z}$ and if the response is "correct guess" mark pw_i as "client's password", otherwise return "wrong guess".

    Regardless mark the i-th aPAKE as *non-fresh.*
  - **Case client is corrupted:**
    1. If both client and server have successfully made $\textsc{UsrSession}$ and $\textsc{SrvSession}$ queries to the i-th $\mathcal{F}_{\mathsf{aPAKE}}$, the password file is stolen, the i-th aPAKE is not marked *non-fresh* and no list L is stored, send the message $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}'_i)$, where pw'_i is the password that client registered in this $\mathcal{F}_{\mathsf{aPAKE}}$, to $\mathcal{F}$. If $\mathcal{F}$ responds with "correct guess" return "correct guess". Otherwise return "wrong guess" and mark pw'_i as "not position i".
    2. If there is a list L stored check if $\mathsf{pw}'_i = \mathsf{pw}_i$ where pw_i is the i-th password in the list L and pw'_i is the password that is used by the client in the i-th instance of aPAKE. If so return "correct guess" and "wrong guess" otherwise to $\mathcal{Z}$.

    Regardless mark the i-th aPAKE as *non-fresh.*
- Upon the query $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}')$ from $\mathcal{Z}$ for the i-th $\mathcal{F}_{\mathsf{aPAKE}}$ and this aPAKE is not marked *non-fresh*:
  - If no list L is stored, send a query $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}')$ to $\mathcal{F}$. If $\mathcal{F}$ returned "correct guess" mark pw' as "position i" and otherwise "not position i" and forward $\mathcal{F}$'s response to $\mathcal{Z}$.
  - If there is a list L stored check if $\mathsf{pw}' = \mathsf{pw}_i$ where pw_i is the i-th password in the list L. If so return "correct guess" and "wrong guess" otherwise to $\mathcal{Z}$.

  Regardless mark the i-th aPAKE as *non-fresh.*
- Upon the query $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw}')$ from $\mathcal{Z}$ for the i-th $\mathcal{F}_{\mathsf{aPAKE}}$ and this aPAKE is not marked *non-fresh* distinguish between the following cases:
  - **Case both parties are honest (no byzantine corruptions):**
    * If no password is stored and marked as "client's password", send a query $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw}')$ to $\mathcal{F}$ and forward $\mathcal{F}$'s response to $\mathcal{Z}$. If the response is "correct guess" mark pw' as "client's password".
    * If there is a password pw marked as "client's password" check if $\mathsf{pw} = \mathsf{pw}'$. If so return "correct guess" and "wrong guess" otherwise to $\mathcal{Z}$.

    Regardless mark the i-th aPAKE as *non-fresh.*
  - **Case client is corrupted:**
    * If the corrupted client has sent a $\textsc{UsrSession}$ and the password pw to the i-th $\mathcal{F}_{\mathsf{aPAKE}}$ check if $\mathsf{pw} = \mathsf{pw}'$. If so return "correct guess" and if $\mathsf{pw} \neq \mathsf{pw}'$ return "wrong guess" to $\mathcal{Z}$.

    Regardless mark the i-th aPAKE as *non-fresh.*

Modifications to $\mathcal{F}$: Add StorePwdFile, StealPwdFile, Impersonate and OfflineTestPwd interfaces to $\mathcal{F}$ as in Figure 1 and TestPwd as in Figure 8. In case of any Byzantine corruptions, $\mathcal{F}$ informs the simulator about the server's password when receiving a StorePwdFile input.

We note that by adding interface StorePwdFile to $\mathcal{F}$, $\mathcal{S}$ does not use the server's password to simulate the added interfaces and also does not receive any trigger from $\mathcal{F}$ anymore to simulate the password registration phase. Therefore, the difference between $\mathbf{G}_3$ and $\mathbf{G}_2$ is that $\mathcal{S}$ now fully simulates the server.

First, $\mathcal{S}$ now generates the password file only on demand (of $\mathcal{Z}$) and by programming $\mathcal{F}_{\mathsf{RO}}$ instead of (as in $\mathbf{G}_2$) executing $\mathcal{P}_S$'s code with the true password. $\mathcal{Z}$ can only notice if it inputs a password close to the server to $\mathcal{F}_{\mathsf{RO}}$. But then, our $\mathcal{S}$ submits this password as OfflineTestPwd guess and programs $\mathcal{F}_{\mathsf{RO}}$ consistently in case of a positive answer including the server's true password. Otherwise, no password close to server's password has been queried to $\mathcal{F}_{\mathsf{RO}}$ and the simulated password file contains, from the point of view of $\mathcal{Z}$, uniformly random values just as in $\mathbf{G}_2$.

Second, $\mathcal{S}$ simulates the server's part in the $\mathcal{F}_{\mathsf{aPAKE}}$ instances to be able to answer adversarial queries to these instances issued by $\mathcal{Z}$. For OfflineTestPwd, Impersonate (in case of no byzantine corruptions) and StealPwdFile it is straightforward to verify that the interfaces in $\mathcal{F}'_{\mathsf{faPAKE}}$ provide $\mathcal{S}$ with enough information to simulate $\mathcal{F}_{\mathsf{aPAKE}}$'s responses indistinguishable to a setting where all instances of $\mathcal{F}_{\mathsf{aPAKE}}$ obtain the correct passwords. For a TestPwd guess against the client where the client is honest, querying any instance of $\mathcal{F}_{\mathsf{aPAKE}}$ with a (previously untested) password close to the server's true password will result in "correct guess" in $\mathbf{G}_2$ with probability $1/k$ due to the honest server applying a random permutation to its password file, while a password that is too far away results in "wrong guess" in both games. TestPwd in $\mathcal{F}'_{\mathsf{faPAKE}}$ produces "correct guess" with the exact same probability. On the other hand, TestPwd in $\mathcal{F}'_{\mathsf{faPAKE}}$ allows k "exact" guesses against the client's password, which is sufficient to answer all TestPwd queries that $\mathcal{Z}$ can send to the k $\mathcal{F}_{\mathsf{aPAKE}}$ instances.

In case the client is corrupted, when $\mathcal{Z}$ requests a TestPwd query against the client's password, $\mathcal{S}$ can check and respond according to the password registered by the client in this instance of aPAKE. In addition in order to simulate the Impersonate query of the internal aPAKEs, $\mathcal{S}$ checks if the password registered by the client in this instance of aPAKE is equal to the serve's password. This is either done by making the query (TestPwd, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}'_i$) to $\mathcal{F}$ or checking if the i-th element of the list L is equal to pw'_i (where pw'_i is the password that the client registered in the i-th aPAKE). Hence $\mathbf{G}_3$ and $\mathbf{G}_2$ are indistinguishable.

Game $\mathbf{G}_4$: Handling the queries OfflineTestPwd, Impersonate and TestPwd to $\mathcal{F}_{\mathsf{aPAKE}}$ For Corrupted Server

Modifications to $\mathcal{S}$:

- Upon a query from $\mathcal{Z}$ to send a message of the form (Impersonate, $\mathsf{sid}, \mathsf{ssid}$) to the i-th $\mathcal{F}_{\mathsf{aPAKE}}$:
  1. If both client and server have successfully made UsrSession and SrvSession queries to the i-th $\mathcal{F}_{\mathsf{aPAKE}}$, the password file is stolen and the i-th aPAKE is not marked *non-fresh*, send the message (TestPwd, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw}_i$), where pw_i is the password that server registered in this $\mathcal{F}_{\mathsf{aPAKE}}$, to $\mathcal{F}$. If $\mathcal{F}$ responds with "correct guess" return "correct guess" and mark pw_i as "client's password". Otherwise return "wrong guess".

  Regardless mark the i-th aPAKE as *non-fresh.*
- Upon the query (TestPwd, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, \mathsf{pw}'$) from $\mathcal{Z}$ for the i-th $\mathcal{F}_{\mathsf{aPAKE}}$ and this aPAKE is not marked *non-fresh* check if the password registered by the server in this aPAKE is equal to pw'. If so return "corect guess" and otherwise return "wrong guess". Regardless mark the i-th aPAKE as *non-fresh.*
- Upon the query (TestPwd, $\mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw}'$) from $\mathcal{Z}$ for the i-th $\mathcal{F}_{\mathsf{aPAKE}}$ and this aPAKE is not marked *non-fresh*, distinguish the following cases:

– If there is a password pw marked as "client's password", check if $\mathsf{pw} = \mathsf{pw}'$, if so return "correct guess" and "wrong guess" otherwise.

– If there are no passwords marked as "client's password", send the message send the message $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_C, \mathsf{pw}')$ to $\mathcal{F}$. If $\mathcal{F}$ responds with "correct guess" return "correct guess" and mark pw' as "client's password". Otherwise return "wrong guess".

Regardless mark the i-th aPAKE as *non-fresh*.

In this game $\mathcal{S}$ simulates the queries to the internal $\mathcal{F}_{\mathsf{aPAKE}}$ in case only the client is honest. This case is very similar to the situation where only the server is honest except the passwords that the server uses for each of the aPAKEs is known to $\mathcal{S}$, hence there is no need to produce the list L. Similar to the $\mathbf{G}_3$, TestPwd queries against the server's password can be simulated exactly using the passwords that the sever has registered in each of the aPAKEs and impersonate and TestPwd queries against the client's password are simulated by submitting TestPwd to $\mathcal{F}$ using the password that the server has registered in this instance of aPAKE and the password that $\mathcal{Z}$ has queried respectively. Therefore $\mathbf{G}_4$ and $\mathbf{G}_3$ are indistinguishable.

Game $\mathbf{G}_5$: Generating aPAKE keys at random For Two Honest Parties

Modifications to $\mathcal{S}$: For all aPAKEs that are not marked *non-fresh* or $\mathcal{S}$ did not return "correct guess" to $\mathcal{Z}$ produce $\mathsf{k}_{C,i}$ and $\mathsf{k}_{S,i}$ at random. For such keys that are chosen randomly, $\mathcal{S}$ creates ciphertexts using random string as input of $\mathcal{F}_{\mathsf{IC}}$. Furthermore choose h uniformly at random. In addition let l' be the number of non-fresh aPAKE instances and do:

- Upon receiving a request from $\mathcal{Z}$ to replace h with h', check if the pre-image of h' is of the form $\mathsf{k}_{S,i}||0$ and $\mathsf{k}_{S,i}$ is the key produced by an $\mathcal{F}_{\mathsf{aPAKE}}$ that for which $\mathcal{Z}$ has guessed the server's password correctly and received "correct guess". If so $\mathcal{S}$ sets $K_S \leftarrow H(\mathsf{k}_{S,i})$, otherwise set $K_S \leftarrow \perp$.
- Upon receiving a request from $\mathcal{Z}$ to replace $\overrightarrow{e}$ with $\overrightarrow{e}'$, the simulator computes the number of changed ciphertexts in $\overrightarrow{e}$ where corresponding aPAKE is not marked *non-fresh* and demoted by l. Otherwise if no such command is sent by $\mathcal{Z}$ set $l = 0$.
- Regardless with probability $(l + l')/k$, set $K_C \leftarrow \perp$ and x to a uniformly chosen from $\mathcal{K}$ at the end of the key exchange phase and $K_S \leftarrow \perp$ at the end of the Explicit Authentication phase.

Upon calculation of K_C and K_S send $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_S)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_C)$ to $\mathcal{F}$.

Modifications to $\mathcal{F}$: $\mathcal{F}$ does not forward the password of the parties to the simulator.

In this game $\mathcal{S}$ stops using the passwords of the client and server and fully simulates the execution. Since the environment does not know $\mathsf{k}_{S,i}$ for any $\mathcal{F}_{\mathsf{aPAKE}}$ that is fresh, if h' is not equal to $H(\mathsf{k}_{S,i}||0)$ where $\mathsf{k}_{S,i}$ is outputted by a non-fresh $\mathcal{F}_{\mathsf{aPAKE}}$, $\mathcal{Z}$ can only produce a correct hash value h' with negligible probability of guessing a $\mathsf{k}_{S,i}$ correctly. Hence, in this case the parties produce different keys. In addition $\mathcal{Z}$ knows $\mathsf{k}_{S,i}$ if it has correctly guessed the password of the server in which case $\mathcal{S}$ can decide what key would be sent to the server by $\mathcal{F}$.

Without the knowledge of the keys $\mathsf{k}_{C,i}$ with $i \in [k]$, the environment cannot produce new valid ciphertexts or decrypt the ciphertexts in $\overrightarrow{e}$. In addition for all non-fresh aPAKEs, the keys that are received by the client and server different (according to the functionality both records must be fresh in order for the parties to get the same keys). Let $l + l'$ be the total number of non-fresh aPAKEs and the number of modified ciphertexts, even if the client and server uses a close password, the probability for both parties producing the same key is now $1 - \frac{l+l'}{k}$, due to the random permutation π used by the Server. In other words both parties use the same password in one of the tampered locations with probability $\frac{l+l'}{k}$. Therefore the simulator sets both the client's and server's key to $\perp$ with this probability. Finally $\mathcal{Z}$ does not have any information about the pre-image of h. As mentioned earlier if $\mathcal{Z}$ has guessed the client's password for one of these aPAKE instances correctly $\mathcal{S}$ can simulate the protocol execution as in the real world. Otherwise the aPAKEs also produce keys unknown to $\mathcal{Z}$ uniformly at random. Hence $\mathbf{G}_5$ is indistinguishable from $\mathbf{G}_4$ except with negligible probability.

Game $\mathbf{G}_6$: Removing Client's Keys for Honest Client and Corrupted Server

Modifications to $\mathcal{S}$: $\mathcal{S}$ ignores $\mathsf{k}_{C,i}$s for all aPAKE indexes that are not marked *non-fresh* and only uses the corresponding $\mathsf{k}_{S,i}$s to open the encrypted openings in the simulated run. More precisely, upon C receiving $\overrightarrow{e}$ where $|\overrightarrow{e}| = n$, $\mathcal{S}$ decrypts every $e_i \in \overrightarrow{e}$ for which i-th aPAKE is not marked *non-fresh* using $\mathsf{k}_{S,i}$ and if it is marked *non-fresh* using $\mathsf{k}_{C,i}$ and gets $H'_1, \cdots, H'_n$. The simulator sends a the request $(\textsc{TestPwd}, \mathsf{sid}, \mathcal{P}_C, \mathsf{pw}'_i)$ to $\mathcal{F}$ for all aPAKE indexes that are not marked *non-fresh* where pw'_i is the pre-image of H'_i stored in the random oracle and do the following at the end of the Key exchange phase:

1. If $\mathcal{F}$ returns "correct guess" and pw'_i or if there already exists a password pw_i marked "client's password", check if the hash values are well formed and there exists only one key that can decrypt the encrypted values, set $K_C \leftarrow H(\mathsf{k}_{C,i})$ and $x \leftarrow \mathsf{k}_{C,i}$ if the i-th aPAKE is marked *non-fresh* or set $K_C \leftarrow H(\mathsf{k}_{S,i})$ and $x \leftarrow \mathsf{k}_{S,i}$ if the i-th aPAKE is not marked *non-fresh.*
2. Otherwise set $K_C \leftarrow \perp$ and $x \xleftarrow{\$} \mathcal{K}$.

Upon calculation of K_C and K_S send $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_S)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_C)$ to $\mathcal{F}$.

In this game the distribution of K_C is indistinguishable to the previous game, since if the client and server do have a close password and the server sends a correct password file the client produces the same key K_C and x as in the previous game. We note that in case the aPAKE for which the client's password is used is marked *non-fresh*, $\mathcal{S}$ knows the key generated by this aPAKE and hence must set $K_C \leftarrow H(\mathsf{k}_{C,i})$ and $x \leftarrow \mathsf{k}_{C,i}$ otherwise (and since the correct password is used in this aPAKE) $\mathcal{S}$ must set $K_C \leftarrow H(\mathsf{k}_{S,i})$ and $x \leftarrow \mathsf{k}_{S,i}$.

Finally if the parties do not have a close password (as in the previous game), K_C is set to $\perp$ and x is chosen randomly. Hence $\mathbf{G}_6$ is indistinguishable from $\mathbf{G}_5$.

Game $\mathbf{G}_7$: Removing the aPAKE's and the Client's Password for an Honest Client and a Corrupted Server

Modifications to $\mathcal{S}$: The simulator chooses uniform random $\{\mathsf{k}_{S,i}\}_{i=1,\dots,k}$ instead of using the simulated aPAKE protocols for aPAKE's that are not marked *non-fresh* and the server did not get "correct guess" when querying the $\textsc{TestPwd}$ interface and guessing the server's password.

Modifications to $\mathcal{F}$: The Client's password is not forwarded to $\mathcal{S}$.

If the environment was able distinguish this game from the previous game, it can distinguish the random keys from the keys produced by the aPAKE which contradicts the security of aPAKE. In addition based on the $\mathbf{G}_7$, $\mathcal{S}$ does not use the Client's password in case of honest Client and corrupted Server. Hence the environment cannot distinguish $\mathbf{G}_7$ from $\mathbf{G}_6$.

Game $\mathbf{G}_8$: Removing the aPAKE's In the Protocol for Corrupted Client and Honest Server

Modifications to $\mathcal{S}$: For all aPAKEs that are not marked *non-fresh*, if the list L is not generated yet $\mathcal{S}$ sends the query $(\textsc{TestPwd}, \mathsf{sid}, \mathcal{P}_S, \mathsf{pw}_{C,i})$ to $\mathcal{F}$ where $\mathsf{pw}_{C,i}$ is the password that the corrupted client used in this aPAKE. If $\mathcal{F}$ returns "correct guess" mark it "position i" and "not position i" otherwise. Furthermore if $\mathcal{F}$ returned "correct guess" produce the same key $\mathsf{k}_{C,i} = \mathsf{k}_{S,i}$ uniformly at random for both parties and otherwise produce different keys. If there exists a list L, for all aPAKEs that are not marked *non-fresh* check if $\mathsf{pw}_{C,i} = \mathsf{pw}_i$ where pw_i is the i-th element of L and $\mathsf{pw}_{C,i}$ is the password that the corrupted client used in this aPAKE, if so produce the same key $\mathsf{k}_{C,i} = \mathsf{k}_{S,i}$ uniformly at random for both parties and otherwise produce different keys.

For all aPAKEs in which $\mathcal{S}$ has learned server's password (from the $\textsc{TestPwd}$ interface) and both the server and client receive the same key, $\mathcal{S}$ uses this password and key in order to create the encryption. Otherwise $\mathcal{S}$ uses randomly chosen ciphertext.

Upon calculation of K_C and K_S send $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_S)$ and $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_S, K_C)$ to $\mathcal{F}$.

Modifications to $\mathcal{F}$: The Server's pass-string is not forwarded to $\mathcal{S}$.

According to the security of aPAKE, a key produced by the aPAKE is indistinguishable from a random key. If both parties use the same password (and the aPAKE outputs the same key for both parties), $\mathcal{S}$ must produce a valid encryption since the corrupted client should be able to decrypt these elements in $\overrightarrow{e}$. Otherwise the client can only decrypt other elements by guessing the correct key produced by the aPAKE protocol which happens with negligible probability. In addition in $\mathbf{G}_8$, $\mathcal{S}$ does not use the clients password in case of corrupted client and honest server, hence $\mathbf{G}_8$ is indistinguishable from $\mathbf{G}_7$ except with negligible probability.