Generalized Fuzzy Password-Authenticated Key Exchange from Error Correcting Codes

Jonathan Bootle[1], Sebastian Faller[1,2], Julia Hesse[1⋆], Kristina Hostáková[2]
Johannes Ottenhues[3⋆⋆], and

[1] IBM Research Europe – Zurich
[2] ETH Zurich
[3] University St. Gallen

Abstract. Fuzzy Password-Authenticated Key Exchange (fuzzy PAKE) allows cryptographic keys to be generated from authentication data that is both fuzzy and of low entropy. The strong protection against offline attacks offered by fuzzy PAKE opens an interesting avenue towards secure biometric authentication, typo-tolerant password authentication, and automated IoT device pairing. Previous constructions of fuzzy PAKE are either based on Error Correcting Codes (ECC) or generic multi-party computation techniques such as Garbled Circuits. While ECC-based constructions are significantly more efficient, they rely on multiple special properties of error correcting codes such as maximum distance separability and smoothness.
We contribute to the line of research on fuzzy PAKE in two ways. First, we identify a subtle but devastating gap in the security analysis of the currently most efficient fuzzy PAKE construction (Dupont et al., Eurocrypt 2018), allowing a man-in-the-middle attacker to test individual password characters. Second, we provide a new fuzzy PAKE scheme based on ECC and PAKE that provides a built-in protection against individual password character guesses and requires fewer, more standard properties of the underlying ECC. Additionally, our construction offers better error correction capabilities than previous ECC-based fuzzy PAKEs.

Keywords: Attacks on Public-Key Constructions · Key Exchange Protocols · Password-Based Cryptography · UC Framework.

1 Introduction

Password-authenticated key exchange (PAKE) protocols allow two users to exchange symmetric keys from plaintext passwords only. PAKEs are a useful tool in unlocking smartcards (e.g., German ID card [BFK09,BDFK12,BFK13], FIDO2

⋆ Author supported by the Swiss National Science Foundation (SNSF) under the AMBIZIONE grant "Cryptographic Protocols for Human Authentication and the IoT"
⋆⋆ Author partially funded by the EU-funded Marie Curie ITN TReSPAsS-ETN project under the grant agreement 860813

[BBCW21]), securing wireless networks [All22,CNPR22], and pairing IoT devices [HL19]. First formalized by Bellovin and Merrit [BM92], PAKEs provide optimal protection of potentially low-entropy passwords, a feature that is called *resistance against offline dictionary attacks*. Essentially, a PAKE does not leak any information about the password used by the counterparty in case of password mismatch. This guarantee needs to hold against network adversaries observing the protocol execution as well as active attackers playing one user in the protocol. Such strong protection is vital for use cases of PAKE. Without it, any typing of, e.g., the 6-digit PIN restricting access to a German national ID card would already expose that PIN to brute-force attacks mounted through malicious card reader hardware or software. Building secure and efficient PAKEs that even feature strong universal composability guarantees is fairly simple. The general idea is to run a Diffie-Hellman key exchange, and either encrypt the Diffie-Hellman public keys with the password [BM92,BPR00], or derive the group generator from the password [Jab96,BMP00,Mac01,HS14,HL19][4].

Unfortunately, in many application scenarios of PAKE protocols, usability can hinder their adoption. Manually entering PINs in stressful situations, such as unlocking one's smart ID card next to a waiting officer, or pairing a wearable IoT device while walking, is prone to repeated mistyping. As another example, consider IoT devices that want to automatically exchange a key with devices located in their close proximity. These devices can automatically derive passwords from sensor readings of their direct environment, but they likely end up with similar but not exactly matching bitstrings. In these situations, key exchange through PAKE fails even though the legitimate owner of the ID card was present, and even though the IoT devices were actually sitting next to each other.

In 2018, Dupont et al. [DHP+18] proposed a new cryptographic primitive called *fuzzy* PAKE (fPAKE), which allows two users to exchange a symmetric key from *similar* passwords. In addition to being a candidate to resolve the usability issues with password mistyping/mismatching in the aforementioned scenarios, fuzzy PAKE opens an interesting avenue towards secure biometric authentication, as it enables secure comparison of consecutive biometric readings *without leaking any other information about the biometric* than "match" or "no match". To date, fuzzy PAKE is the only cryptographic primitive that can provide such optimal protection of data that is both noisy *and* of low entropy.

Dupont et al. [DHP+18] gave two constructions of fuzzy PAKE, one relying on Garbled Circuits and one relying on error-correcting codes (ECC)[5] and PAKE, using Hamming distance as a metric for similarity of passwords. The ECC based construction is to the best of our knowledge the most efficient fuzzy PAKE construction in the literature [DHP+18] so far. At a high level, the proto-

[4] [Hv22] provides an excellent overview of PAKEs in the literature, also mentioning other approaches than building PAKEs through Diffie-Hellman.

[5] Dupont et al. [DHP+18] use the terminology of *robust secret sharing* (RSS) instead of error-correcting codes, and show how to instantiate RSS with ECC. In this work, we state their construction in terms of an ECC, as it enables better comparison with our protocol.

col works as follows. One party takes the role of the sender, chooses a symmetric key s, encodes it with the error-correcting code to get $C \leftarrow \mathsf{Enc}(s)$, XORs the resulting codeword with the password to get $E \leftarrow C \oplus \mathsf{pw}$, and sends E to the receiver. The receiver XORs the received E with their own password to get $C' \leftarrow E \oplus \mathsf{pw}'$, and attempts to decode C' to retrieve s. Depending on whether pw and pw' were similar enough, the codeword C' is close enough to C to uniquely decode to s. The error correction capability of the fuzzy PAKE is hence directly related to the error correction threshold of the code. However, the protocol as described above would be prone to offline attacks: the receiver can repeatedly decode with many password guesses pw', yielding a list of candidate keys. To prevent such a dictionary attack, [DHP+18] introduce a "password expansion" step prior to the encoding of the key. Here, the two parties run a PAKE on *every character* of their passwords pw and pw', yielding key vectors K and K' that match in exactly the same entries as pw and pw', while the other entries are uniformly distributed. Because this expansion step requires active participation of both parties and the vector entries are PAKE keys that are hard to guess, one party alone cannot turn a password into a key vector. Hence, parties – malicious or honest – are limited to only one password guess per protocol run.

In fact, to prevent C from leaking information about the key s in case the receiver is malicious, C is computed by encoding s concatenated with an extra random input r. The error correcting code must satisfy multiple special properties, such as a *uniformity* property, whereby small subsets of codeword entries appear uniformly random, independently of s, and a *smoothness* property, whereby decoding an extremely noisy codeword results in a random key. Uniformity is a well-known property, satisfied by the popular Reed-Solomon codes, which has been extensively studied [CCG+07,CDBN15,Wei16,BCL22], and can be obtained via various transformations including the method presented in [DHP+18]. Smoothness is less standard, and [DHP+18] rely on maximum distance separable codes in order to achieve it.

1.1 Our contribution

First, we identify a subtle but devastating issue in the ECC-based fuzzy PAKE construction of [DHP+18]. In a bit more detail, we give an attack that allows the adversary to make guesses on individual password bits (or characters), allowing them to extract a user's, e.g., n-bit password from only n executions of the protocol with that user. Our attack demonstrates that the measures taken by [DHP+18] to protect against such individual guesses, namely ensuring that an attacker either guesses "all-or-zero" bits of the password, subtly fails due to incorrect binding of the different protocol parts. Here, binding refers to ensuring that all messages come from the same sender. Although there exist generic techniques for binding together messages sent over unauthenticated channels [BCL+05], they cannot be applied to the construction of [DHP+18] in black-box way due to its modular layout.[6]

[6] More precisely, and for the reader who is familiar with the Universal Composability framework [Can01]: In Section 3.2, we argue that the split transformation of Barak

The effect of our attack is that an active attacker learns one bit of an honest user's password by simply messing with an honest protocol run between two users. While the security notion of fuzzy PAKE [DHP+18] does leak information about the password, i.e., it leaks the exact password of an honest user to an attacker, we stress that this attacker must be actively running a session with the honest user *and* use close password to the one of the attacked honest user. Our attack requires much less knowledge from the adversary. In more detail, it can be executed by a network adversary essentially flipping some bits of the exchanged messages between honest clients. The attacker only needs to make an assumption about the distance of the passwords used in the execution. For example, in settings where fuzzy PAKE is usually executed without the proper passwords, e.g, mistyping happens only rarely, every honest protocol run would reveal one bit about the secret password of an honest party to a network attacker. Deriving such information from honest protocol runs is prohibited by the security model of [DHP+18].

Reference	error corr. capability	rounds	msg size	Sender	Receiver	model
[DHP+18]	$\lfloor \frac{n-k}{2} \rfloor$	1	$\mathcal{O}(n)$	$2n$ exp	$\mathsf{Dec} + 2n$ exp	RO, IC, CRS
Our work, unique dec.	$\lfloor \frac{n-k}{2} \rfloor$	1	$\mathcal{O}(n)$	$2n$ exp	$\mathsf{Dec} + 2n$ exp	RO, IC, CRS, SA
Our work, list dec.	$n - 1 - \lfloor \sqrt{(k-1)n} \rfloor$	1	$\mathcal{O}(n)$	$2n$ exp	$\mathsf{LDec} + 2n$ exp	RO, IC, CRS, SA

Fig. 1. Comparison of ECC-based fuzzy PAKE constructions, using a Reed-Solomon code of rank k, and block length n equal to the size of the password. Sender and Receiver columns indicate computation complexity. Both protocols are instantiated with EKE2 [DHP+18] to derive concrete performances. RO, IC, and CRS are required for the security of EKE2. SA is split authentication. We stress that the ECC-based [DHP+18] cannot be considered a secure fuzzy PAKE (Section 3.2).

Our second contribution is a new fuzzy PAKE protocol based on error-correcting codes, which directly builds upon the ideas of [DHP+18] but (i) fixes the previous insecurities, (ii) provides better error-correction capabilities, and (iii) relies on fewer properties of the underlying code. These improvements come at the cost of less efficient computation and communication. We provide a high-level comparison in Figure 1. For (i), the idea is to ensure that the transformation of Barak et al. [BCL+05] applies to the whole protocol, simply by instantiating all building blocks that require interaction between the parties *before* applying the transformation. For (ii) we investigate how the (fixed) fuzzy PAKE protocol of [DHP+18], which applies unique decoding of the ECC, benefits from applying a non-unique decoding technique. Namely, we make use of *list decoding* which is a decoding technique that takes a codeword with errors as input, and that can produce a list of candidate codewords including the original one. When the

et al. [BCL+05] cannot be meaningfully applied to transform a *hybrid* protocol that assumes authenticated channels, to a version that is secure with unauthenticated channels. In a nutshell, the reason is that the hybrid building blocks are unaffected by the transformation and do not carry any authentication guarantees.

ECC threshold $n-k+1$ (as % of n)	Password Length n									
	$n=8$		$n=32$		$n=64$		$n=128$		$n=256$	
	[DHP+18]	**Ours**	[DHP+18]	**Ours**	[DHP+18]	**Ours**	[DHP+18]	**Ours**	[DHP+18]	**Ours**
10%			1	1	3	3	6	6	12	12
25%	1	1	4	4	8	8	16	17	32	34
30%	1	1	4	4	9	10	19	20	38	41
50%	2	2	8	9	16	18	32	37	64	74

Fig. 2. Error correction capability for the ECC-based PAKE constructions of [DHP+18] and in this work using list decoding, for various password lengths n and an $[n, k, d]$-code $\mathcal{C}$. The table shows the number of password errors below which key exchange will succeed. The ECC threshold refers to the percentage of errors above which key exchange will fail without leaking information about the password. Gray colored cells show parameter settings for which our construction improves upon previous works.

number of errors is too large for unique decoding to work, list decoding can still produce reasonable-sized lists of codewords that still contain the original one. We give examples of the improved error correction capability of our scheme over previous works in Figure 2, For (iii) we leverage new ingredients in our construction in order to bypass the smoothness property completely.

Intuitively, such non-unique decoding introduces a "correctness error" into the fuzzy PAKE protocol of [DHP+18], since the receiver decodes a list of key candidates and has to guess which one is the key of the sender. Moreover, it is unclear how to generalize the smoothness property and its proof in [DHP+18] to the list decoding regime, due to the higher number of errors.

Our idea is to restore perfect correctness by letting the sender give a *hint* about the correct key to the receiver. This hint, however, needs to be carefully crafted so that it benefits the correctness of the key exchange without revealing too much information about the password or the key of the sender. For example, simply using a hash of the sender's key is not possible. This is because during security proofs, the hint may have to be simulated, without knowledge of the key, against an adversary who does know the key and is able to check that whether the hint was computed correctly or not. Instead, the sender hashes the encoding C of s. Based on the uniformity of the error-correcting code, this hint contains sufficient entropy to prevent information leakage on the key. Furthermore, the fact that the hint uniquely determines the correct codeword actually allows us to dispense with the smoothness property completely. Since smoothness, which relied on maximum distance separability, is no longer needed, the space of possible codes which can be used in our construction is much larger, opening the door to codes with even better list-decoding capabilities and fuzzy PAKEs with even higher error tolerance in future.

Altogether, we are able to give a construction with better error correction capability than previous schemes in the literature. More formally, we prove the following in Theorem 2.

Our fuzzy PAKE using a Reed-Solomon code of rank k, block length n, *list* decoding, a hint as described above and signing under ephemeral public keys to

achieve binding, is secure and has an error correction capability of
$$\delta = n - 1 - \sqrt{(k-1)n}.$$

We can also apply our results to obtain a fixed version of the fuzzy PAKE of [DHP+18] et al. by using unique decoding instead of list decoding, offering a trade-off between runtime (unique decoding is generally faster than list decoding) and error correction capability (see Figure 2 for numerical examples of the error correction terms).

Corollary 1. *Our fuzzy PAKE using a Reed-Solomon code of rank k, block length n, **unique** decoding, a hint as described above and signing under ephemeral public keys to achieve binding, is secure and has an error correction capability of $\lfloor \frac{n-k}{2} \rfloor$. The computational overhead over [DHP+18] is one signature per message, and one hash by the sender.*

Roadmap. In Section 2 we provide preliminaries on error-correcting codes and list decoding, and on implicit PAKE which is a building block of our and previous ECC-based fuzzy PAKE protocols. Section 3 recaps fuzzy PAKE including its security model in the UC framework, explains the fuzzy PAKE of [DHP+18] and presents an attack that allows to test password bits. In Section 4 we give our improved fuzzy PAKE protocol and prove its security.

1.2 Related Work

Prior to the introduction of fuzzy PAKE by Dupont et al. [DHP+18], several attempts to base cryptography on low-entropy or noisy shared data have been made. Information reconciliation [BBR88] and fuzzy extractors [DORS08,RW04] let two parties identify common randomness in shared noisy bitstrings. However, the identification comes at the price of leakage, such that these techniques cannot be used when the shared data is of low entropy, such as in the case of passwords. Canetti et al. [CFP+16] construct special-purpose schemes for securely comparing the Hamming distance of, e.g., biometric readings. However, the security of their construction again relies on the data having a certain min-entropy. The PAKE-based construction for comparison of biometric readings of Boyen et al. [BDK+05] does not protect against offline attacks on the biometric data and hence cannot reach the security level of fuzzy PAKEs.

Biometric authentication is a very active area of research. In the following we list several works that aim at providing secure biometric authentication by introducing, e.g., additional trust anchors, or by leveraging the entropy in the biometric scans. BETA [ABM+21] generates a token to be used for authenticating a client to a server, by letting three client devices communicate over authenticated channels. In the fuzzy PAKE setting, each entity only needs one device and also a fuzzy PAKE is secure even in unauthenticated channels. Agrawal et al. [ABMR20] present a protocol for authenticating a user with their biometric by letting the user's phone, an external terminal and a service provider communicate to compute the authentication result. Differences to fuzzy PAKE are

that [ABMR20] has three parties instead of two, and their protocol uses cosine similarity instead of Hamming distance to compare the biometrics. Moreover, [ABMR20] focuses on authentication whereas a fuzzy PAKE achieves key exchange. Wang et al. [WHC+21] construct authenticated key change for secure messaging in which the users do not need to store their secret keys but instead derive them from their biometrics. [WHC+21] uses the biometric as an error term for LWE samples, which places very strong requirements on the biometric distribution. A fuzzy PAKE is stronger; its security cannot rely on the distribution of the authentication data, and hence our protocol is secure even when used with low entropy passwords (or biometrics). Jiang et al. [JLHG22] build key exchange from fuzzy data, which is also the goal of a fuzzy PAKE. However, similar to [WHC+21], [JLHG22] relies on high entropy of that fuzzy data. A fuzzy PAKE however does not make any assumptions about the entropy of the authentication data (=passwords). The tools used in [JLHG22] such as secure sketches are known to be inherently insecure when used on low entropy data. Erwig et al. [EHOR20] leverages robust secret sharing, which previous fuzzy PAKEs and our construction also rely on. Contrary to us, [EHOR20] builds an *asymmetric* PAKE, with a client and a server party where the latter is not allowed to store the password in the clear. While this is an extremely relevant setting in practise, their solutions are computationally expensive and the authors note that it seems infeasible to go beyond passwords of 40 bits. The works of Chatterjee et al. [CAA+16], [CWP+17] propose typo-tolerant password authentication systems. These password authentication systems tolerate mistyping on the user side, i.e., the user can successfully authenticate to some server even when using a password slightly different from the one they registered with. While [CAA+16] corrects a predefined set of most common typos (e.g., capitalization of first letter), the TypTop system [CWP+17] offers personalized adoption to frequent typos of users, meaning that the system will learn from past typos and apply certain rules whether to accept these errors in the future. The crucial difference to fuzzy PAKE is that both these typo-tolerant systems require the server party to learn the *cleartext* password of the user, as the password needs to be fed into the decision procedure as well as into the evolving "accepted typos" cache. The main goal of fuzzy PAKE is to prevent such transmission of clear-text passwords among the two parties. The concept of typo tolerance was subsequently carried over to PAKE [PC20]. Typo-tolerant PAKE is a system to log in to a server despite of typos in the password. The protocol in [PC20] requires storage and communication proportional to the number of close passwords, which limits the potential applications of the protocol. The storage and communication cost of our protocol only depends on the length of the password but not on the number of close passwords, which also makes it possible to use our protocol with biometrics instead of passwords.

In [RX23] it is demonstrated that the common notion of PAKE in the UC-framework does not automatically guarantee correctness. More concretely, the ideal functionality $\mathcal{F}_{\mathsf{PAKE}}$ [CHK+05] does not guarantee that two honest parties that are not maliciously attacked do output the same key when they run on

the same password. Several ways to overcome these definitional issues were proposed in [RX23]. The most simple way is to demand correctness as a separate property from the protocol. This style of listing several properties is usually not desirable in UC-protocols as the additionally demanded properties might not be preserved under composition. However, [RX23, Thm. 3] shows that any PAKE protocol that is correct *and* UC-realizes $\mathcal{F}_{\mathsf{PAKE}}$ can be interpreted as a protocol that is secure in an enhanced UC PAKE formalization that ensures correctness (by modeling the man-in-the-middle adversary as a third protocol party). The observations of [RX23] rely on the way output is produced in $\mathcal{F}_{\mathsf{PAKE}}$ and not on the way how the passwords are checked. Therefore, they also apply to our $\mathcal{F}_{\mathsf{fPAKE}}$ (and $\mathcal{F}_{\mathsf{iPAKE}}$) functionality and we have to assert correctness separately.

2 Preliminaries

Notation. We denote vectors in boldface, e.g., $\boldsymbol{V} \in \mathbb{F}^n$. Let $Q \subseteq [n] := \{1, \ldots, n\}$. Then, we denote with $\boldsymbol{V}|_Q := (\boldsymbol{V}_i)_{i \in Q} \in \mathbb{F}^{|Q|}$ the restriction of $\boldsymbol{V}$ to Q. We write PPT for probabilistic polynomial time.

2.1 Universal Composability (UC)

To formally define and prove security of fuzzy PAKEs, we use the UC framework of Canetti [Can01] as is standard in the PAKE literature. Our goal here is to introduce the basic terminology and notation of the UC framework at a very high level and only to the extent needed for the understanding of later sections. For a more accurate and formal explanation of the UC framework, we refer the reader to the work of Canetti [Can01].

In our case, we always consider protocols between two parties (which we typically denote $\mathcal{P}_0$ and $\mathcal{P}_1$) that are executed in the presence of a PPT adversary $\mathcal{A}$ who can corrupt any party at the beginning of the protocol execution (i.e., we consider so-called static corruption). By corruption we mean that the adversary gets a full control over the corrupt parties and learns their internal state. Parties and the adversary $\mathcal{A}$ get their inputs from a special entity, called the *environment* $\mathcal{Z}$, which represents everything external to the protocol execution. $\mathcal{Z}$ also receives outputs from both parties and the adversary. The execution of a protocol in presence of an adversary $\mathcal{A}$ is typically referred to as the "real world".

In the UC framework, security requirements of a protocol are defined via *ideal functionalities*. At a very high level, an ideal functionality $\mathcal{F}$ defines the intended input/output behaviour of parties, the allowed leakage of the protocol and the influence of an adversary on the protocol. $\mathcal{F}$ communicates with the environment through *dummy parties* $\mathcal{P}_0, \mathcal{P}_1$ that simply forward messages between $\mathcal{Z}$ and $\mathcal{F}$. The functionality additionally communicates with an adversary which is typically called the *simulator* and denoted $\mathcal{S}$. The execution of $\mathcal{F}$ in the presence of a simulator $\mathcal{S}$ is typically referred to as the "ideal world". We say that a protocol π UC-realizes (or UC-emulates) an ideal functionality $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that for any

PPT environment $\mathcal{Z}$ the ideal and real worlds are indistinguishable except for negligible probability. In other words, any attack possible on the protocol π can be simulated as an attack on the ideal functionality.

One of the main benefits of the UC framework is that it natively supports protocol composition and hence allows for natural modularization of protocol designs. Concretely, parties in a protocol can communicate with *hybrid* ideal functionalities $\mathcal{H}_1, \mathcal{H}_2, \ldots$. In such a case, we say that the protocol is a defined in the "$(\mathcal{H}_1, \mathcal{H}_2, \ldots)$-hybrid world". The UC composition theorem then allows to securely replace the hybrid ideal functionalities with concrete protocols that UC-realize those functionalities.

2.2 Error Correcting Codes (ECC)

A *linear error-correcting code* $\mathcal{C}$ over a finite field $\mathbb{F}_q$ of order q with rank k and block length n is a linear subspace $\mathcal{C} \subseteq \mathbb{F}_q^n$ of dimension k. We can associate $\mathcal{C}$ with an injective linear *encoding function* $\mathsf{Enc}\colon \mathbb{F}_q^k \to \mathbb{F}_q^n$, where $\mathrm{Im}(\mathsf{Enc}) = \mathcal{C}$.

- The *Hamming distance* $d(\boldsymbol{V}, \boldsymbol{V}')$ between vectors $\boldsymbol{V}, \boldsymbol{V}' \in \mathbb{F}_q^n$ is defined by
$$d(\boldsymbol{V}, \boldsymbol{V}') := |\{i \in [n] \colon \boldsymbol{V}_i \neq \boldsymbol{V}'_i\}| \ .$$
- The *minimum distance* $d(\mathcal{C})$ of a linear code $\mathcal{C}$ is defined as
$$d(\mathcal{C}) := \min_{\boldsymbol{C}, \boldsymbol{C}' \in \mathcal{C}, \boldsymbol{C} \neq \boldsymbol{C}'} d(\boldsymbol{C}, \boldsymbol{C}') \ .$$

A linear code over $\mathbb{F}_q$ with rank k, blocklength n, and minimum distance d is referred to as an $[n, k, d]_q$ code. The "q" will usually be omitted.

Error Correction. Let $e \in \mathbb{Z}$ with $0 \leq e < d/2$. A $[n, k, d]$-code $\mathcal{C}$ is said to be *e-error-correcting* if there exists a function $\mathsf{Dec}\colon \mathbb{F}_q^n \to \mathbb{F}_q^k$ such that for all $\boldsymbol{V} \in \mathbb{F}_q^n$, $\boldsymbol{M} \in \mathbb{F}_q^k$ with $d(\boldsymbol{V}, \mathsf{Enc}(\boldsymbol{M})) \leq e$, we have $\mathsf{Dec}(\boldsymbol{V}) = \boldsymbol{M}$. It is well-known that an $[n, k, d]$-code is $\lfloor \frac{d-1}{2} \rfloor$-error correcting.

Further, we say that a code is *efficiently* e-error-correcting if Dec can be computed in polynomial time in n. Note that *all* linear codes are efficiently 0-error correcting, as the message associated with an error-free codeword can simply be recovered via Gaussian elimination.

List Decoding. Let $e \in \mathbb{Z}$ with $0 \leq e \leq n$. Let $\ell \in \mathbb{N}$ with $\ell \geq 1$. An $[n, k, d]$-code $\mathcal{C}$ is said to be (e, ℓ)*-list-decodable* if for all $\boldsymbol{V} \in \mathbb{F}_q^n$, the list L of codewords $\boldsymbol{C} \in \mathcal{C}$ with $d(\boldsymbol{V}, \boldsymbol{C}) \leq e$ contains at most ℓ codewords.

Further, we say that a code is *efficiently* (e, ℓ)-list-decodable if there is an algorithm LDec which computes the list of codewords in polynomial time in n. This necessarily implies that ℓ is polynomial in n. In this case, we will often simply say that $\mathcal{C}$ is efficiently e-list-decodable.

It will be convenient to use shorthand such as "$\boldsymbol{C} \in \mathsf{LDec}(\boldsymbol{V})$" when a codeword $\boldsymbol{C}$ is part of the output list when LDec is run on input $\boldsymbol{V}$.

2.3 Randomized Codes

We also consider codes with hiding properties, using terminology from [BCL22]. Let $\mathcal{C}$ be a $[n, k, d]$ code. Let $k_M, k_R \in \mathbb{N}$ with $k_M + k_R = k$. We may consider a *randomized encoding function* $\overline{\mathsf{Enc}}\colon \mathbb{F}_q^{k_M} \times \mathbb{F}_q^{k_R} \to \mathbb{F}_q^n$ defined by $\overline{\mathsf{Enc}}(\boldsymbol{M}, \boldsymbol{R}) := \mathsf{Enc}(\boldsymbol{M} \| \boldsymbol{R})$. When considering $\overline{\mathsf{Enc}}$, we refer to $\mathcal{C}$ as a *randomized linear code*.

Uniform Codes. A randomized linear code $\mathcal{C}$ is said to be B-*query uniform* if for any set $Q \subseteq [n]$ with $|Q| \leq B$, the distribution of $\{\overline{\mathsf{Enc}}(\boldsymbol{M}, \boldsymbol{R})|_Q \colon \boldsymbol{R} \leftarrow \mathbb{F}_q^{k_R}\}$ is uniform over $\mathbb{F}_q^Q$.

Example: Reed-Solomon Codes. Let $k, n \in \mathbb{N}$. Let $\mathbb{F}_q$ be a field, let $s_1, \ldots, s_n$ be distinct points in $\mathbb{F}_q$, and let $S := \{s_1, \ldots, s_n\}$. The *Reed-Solomon* code $\mathcal{RS}[n, k, S]$ is defined as

$$\mathcal{RS}[n, k, S] := \{(p(s_1), \ldots, p(s_n)) \mid p \in \mathbb{F}_q[X], \deg(p) \leq k - 1\} \ ,$$

with associated encoding function

$$\mathsf{Enc}\colon (M_0, \ldots, M_{k-1}) \mapsto (p_{\boldsymbol{M}}(s_1), \ldots, p_{\boldsymbol{M}}(s_n)) \ ,$$

where $p_{\boldsymbol{M}} := \sum_{i=0}^{k-1} M_i X^i$. It is well known that $\mathcal{RS}[n, k, S]$ is a $[n, k, n - k + 1]$ code and that furthermore, when considering a randomised encoding function for some $k_R < k$, the Reed-Solomon code is k_R-query uniform.

We state results on the decodability and list-decodability of Reed-Solomon codes.

Lemma 1 (Berlekamp-Welch algorithm [KR07]). *The code $\mathcal{RS}[n, k, S]$ is $\lfloor \frac{n-k}{2} \rfloor$-error correcting, and Dec is computable in time $O(n^2)$.*

Lemma 2 (Guruswami-Sudan algorithm [McE03,Gur06]). *The code $\mathcal{RS}[n, k, S]$ is (e, ℓ)-list decodable with $e = n - 1 - \sqrt{(k-1)n}$ and $\ell = O(n^2)$. Moreover, the list L can be computed in polynomial time.*

Note that some references such as [McE03] allow LDec to produce codewords with distance greater than e as part of the list L as they can be easily discarded. We assume that this has already been done and L contains only codewords within Hamming distance e of the input vector.

Finally, note that while both run in polynomial time, the asymptotic complexity of LDec in Lemma 2 is higher than that of Dec in Lemma 1.

2.4 Implicit-Only PAKE

A PAKE is a cryptographic protocol π that allows two parties to agree on a shared key over an unauthenticated channel assuming that both parties know the same shared password (i.e., a potentially low entropy string), and guarantees that

each party gets a freshly sampled random key if the input passwords are different. The absence of an authenticated channel and the low-entropy assumption on the shared password mean that an iPAKE has to take man-in-the-middle and offline dictionary attacks into account.

The main building block of our construction is an *implicit-only PAKE* (or iPAKE for short) put forward by Dupont et al. [DHP+18]. An iPAKE is a specific type of PAKE that only achieves *implicit* authentication with respect to the honest parties as well as the adversary. This means that at the end of the protocol execution, the two interacting parties do not know if they derived the same key or not. Moreover, an adversary launching an active attack on the protocol by trying to guess the password does not get any feedback on the correctness of their guesses. Dupont et al. [DHP+18] formally defined an iPAKE as an ideal functionality $\mathcal{F}_{\mathsf{iPAKE}}$ which we recall in Fig. 12 of the Supplementary Material.

Correctness. Roy and Xu [RX23] showed that ideal UC-functionalities for PAKE like $\mathcal{F}_{\mathsf{PAKE}}$ by [CHK+05] do not offer correctness. This shortcoming of $\mathcal{F}_{\mathsf{PAKE}}$ also applies to the $\mathcal{F}_{\mathsf{iPAKE}}$ functionality. In a nutshell, all these functionalities offer an adversarial interface to *prevent* successful key exchange that is not subject to any restrictions. Since protocols that emulate a functionality can allow for the same attacks than the functionality, even a PAKE where parties always output random keys could securely realize a PAKE functionality. Further, Roy and Xu [RX23, Thm. 2] show that it is impossible to incorporate correctness into $\mathcal{F}_{\mathsf{PAKE}}$ in the two-party setting. However, they show how to overcome this limitation by defining PAKE as a three-party protocol, where the third party is the man-in-the-middle adversary. This formulation is equivalent to demanding correctness from the PAKE protocol as a separate property [RX23, Thm. 3]. We therefore also demand that an iPAKE protocol has correctness. Let $(K, K') \leftarrow \mathrm{out}_{\mathcal{A},\pi}\langle \mathcal{P}_0(\mathsf{pw}), \mathcal{P}_1(\mathsf{pw})\rangle$ denote that $\mathcal{P}_0$ outputs K and $\mathcal{P}_1$ outputs K' when executing π on input $\mathsf{pw} \in \mathcal{D}$, respectively and in the presence of adversary $\mathcal{A}$. Here $\mathcal{D}$ denotes a finite alphabet (or dictionary). We adapt their correctness definition to the setting of implicit-only PAKE here.

Definition 1. *We say an iPAKE protocol π is* ε-correct *if for all $\mathsf{pw} \in \mathcal{D}$, for some finite alphabet $\mathcal{D}$, and for all passive (i.e., honest-but-curious) adversaries $\mathcal{A}$, the probability that in an execution of π with honest parties $\mathcal{P}_0, \mathcal{P}_1$ the first party $\mathcal{P}_0$ on input pw outputs a different key than the second party $\mathcal{P}_1$ on input pw in the presence of $\mathcal{A}$ is negligible, i.e.,*

$$\Pr[K \neq K' \mid (K, K') \leftarrow \mathrm{out}_{\mathcal{A},\pi}\langle \mathcal{P}_0(\mathsf{pw}), \mathcal{P}_1(\mathsf{pw})\rangle] \leq \varepsilon(\lambda),$$

where the probability is taken over the random coins of $\mathcal{P}_0$,$\mathcal{P}_1$, and $\mathcal{A}$. We call ε the correctness error*. We just say π is correct if it has negligible correctness error.*

2.5 Split Authentication

Another building block in our fPAKE construction is *Split Authentication*, put forward by Barak et al. [BCL+05]. It is essentially a protocol that implements something very close to an authenticated channel in an unauthenticated setting. The adversary's only additional ability is that they can run two separate executions of an authentication protocol (one with each party) without the two communicating parties realizing it. In particular, if both parties are honest, split authentication forces the adversary to decide at the beginning of the communication whether to launch a man-in-the-middle attack and run a separate protocol with each party, or to just observe the messages being exchanged and potentially delay their delivery.

Barak et al. [BCL+05] formalized Split Authentication as an ideal functionality $\mathcal{F}_{\mathsf{SA}}$ which we recall in Fig. 13 of the Supplementary Material.

3 Fuzzy Password-Authenticated Key Exchange

Fuzzy Password-Authenticated Key Exchange (fPAKE for short) is a cryptographic protocol allowing two parties to agree on a shared key in the following setting: the two parties are connected by an unauthenticated channel, but each of them holds a *noisy* version of a shared low-entropy password.

3.1 The $\mathcal{F}_{\mathsf{fPAKE}}$ ideal functionality

Dupont et al. [DHP+18] formalized fPAKE as a UC ideal functionality $\mathcal{F}_{\mathsf{fPAKE}}$ which we recall in Fig. 3. Here we describe $\mathcal{F}_{\mathsf{fPAKE}}$ at a high level and refer the reader to the work of Dupont et al. for a more detailed discussion.

Parties $\mathcal{P}_0, \mathcal{P}_1$ initiate a password-authenticated key exchange by sending a NewSession message to the ideal functionality including their version of the password. To allow for initiator-responder-style protocols, the ideal functionality allows parties to specify their role, i.e., whether they are the Sender or the Receiver. Upon receiving a NewSession message, the functionality records the received password pw and informs the adversary about the NewSession request.

The adversary can then try to guess the recorded password through a TestPwd query. This interface allows an adversary to mount an active attack on the protocol. The recorded pw is marked as `compromised` if adversary's guess is "close enough" and `interrupted` otherwise. The adversary is informed about the result of their guess. To capture the closeness formally, $\mathcal{F}_{\mathsf{fPAKE}}$ is parametrized by *error tolerance* δ and a metric d, and two passwords $\mathsf{pw}, \mathsf{pw}'$ are considered close enough for key exchange to succeed if $d(\mathsf{pw}, \mathsf{pw}') \leq \delta$.

The functionality outputs a secret key to a party $\mathcal{P}_i$ after receiving a NewKey instruction from the adversary. This captures the ability of network attackers to arbitrarily delay the termination of the protocol. $\mathcal{F}_{\mathsf{fPAKE}}$ decides on the secret key being sent in three different ways (for a formal definition of each case, see Fig. 3.):

(i) The secret key is specified by the adversary if the adversary successfully guessed $\mathcal{P}_i$'s recorded password, which can happen either through a TestPwd query, or by corrupting $\mathcal{P}_{1-i}$ and submitting a password on their behalf. This case captures the fact that a protocol participant with a close enough password can bias the final key.
(ii) If both parties are honest, their recorded passwords are δ-close and $\mathcal{F}_{\mathsf{fPAKE}}$ already sent a secret key to party $\mathcal{P}_{1-i}$, then $\mathcal{P}_i$ gets the same key as $\mathcal{P}_{1-i}$. This requirement ensures the core functionality of an fPAKE; namely, that honest parties with close enough passwords output the same key.
(iii) In any other case, $\mathcal{F}_{\mathsf{fPAKE}}$ sends a freshly sampled random key to ensures pseudorandomness of keys overall, and uniformity of keys output by parties with non-matching passwords.

Dupont et al. [DHP+18] additionally defined a modified TestPwd interface that gives more freedom in defining the leakage to the adversary after a TestPwd query. The functionality is now additionally parametrized by a *leakage threshold* $\gamma \geq \delta$ and three leakage functions L_c, L_m, L_f defining the leakage to the adversary depending on the closeness of their guess to the tested password. See Fig. 5 for a graphical explanation of the extended TestPwd interface and Fig. 4 for a formal definition of the extension.

We are particularly interested in a TestPwd interface that leaks positions at which passwords match if the passwords are sufficiently close (following [DHP+18], we call this leakage the *mask*). Our proof of security will rely on this leaked information. Formally, let $\mathcal{F}^M_{\mathsf{fPAKE}}$ be the ideal functionality from Fig. 3 except that the TestPwd interface is defined according to Fig. 4 with the following leakage functions (here $\mathsf{pw} = (\mathsf{pw}_1, \ldots, \mathsf{pw}_n)$ and $\mathsf{pw}' = (\mathsf{pw}'_1, \ldots, \mathsf{pw}'_n)$):

$$L^M_c(\mathsf{pw}, \mathsf{pw}') \coloneqq (\{j \text{ s.t. } \mathsf{pw}_j = \mathsf{pw}'_j\}, \text{“correct guess”}),$$
$$L^M_m(\mathsf{pw}, \mathsf{pw}') \coloneqq (\{j \text{ s.t. } \mathsf{pw}_j = \mathsf{pw}'_j\}, \text{“wrong guess”}),$$
$$L^M_f(\mathsf{pw}, \mathsf{pw}') \coloneqq \text{“wrong guess”}.$$

Correctness of fPAKE. For the same reason as for an iPAKE protocol (see discussion in Section 2.4), we also additionally require that a fuzzy PAKE protocol has correctness. Let $(K, K') \leftarrow \mathrm{out}_{\mathcal{A},\pi}\langle \mathcal{P}_0(\mathsf{pw}), \mathcal{P}_1(\mathsf{pw}')\rangle$ denote that $\mathcal{P}_0$ outputs K and $\mathcal{P}_1$ outputs K' when executing π on input pw and pw', respectively and in the presence of adversary $\mathcal{A}$.

Definition 2. *We say a fuzzy PAKE protocol π is* (δ, ε)-correct *if for all $\mathsf{pw}, \mathsf{pw}' \in \mathcal{D}$ with $d(\mathsf{pw}, \mathsf{pw}') \leq \delta$, where $\mathcal{D}$ is a finite alphabet, and for all passive (i.e., honest-but-curious) adversaries $\mathcal{A}$ the probability that in an execution of π with honest parties $\mathcal{P}_0, \mathcal{P}_1$ the first party $\mathcal{P}_0$ on input pw outputs a different key than the second party $\mathcal{P}_1$ on input pw' in the presence of $\mathcal{A}$ is bounded above:*

$$\Pr[K \neq K' \mid (K, K') \leftarrow \mathrm{out}_{\mathcal{A},\pi}\langle \mathcal{P}_0(\mathsf{pw}), \mathcal{P}_1(\mathsf{pw}')\rangle] \leq \varepsilon(\lambda),$$

where the probability is taken over the random coins of $\mathcal{P}_0$,$\mathcal{P}_1$, and $\mathcal{A}$. We call ε the correctness error*. We just say π is δ-correct if it has negligible correctness error.*

Functionality $\mathcal{F}_{\mathsf{fPAKE}}$

The functionality $\mathcal{F}_{\mathsf{fPAKE}}$ is parameterized by a security parameter λ and error tolerance δ and leakage threshold $\gamma \geq \delta$. It interacts with an adversary $\mathcal{A}$ and two parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

On input (NewSession, sid, $\mathsf{pw}^{\mathcal{P}_i}$, role) from party $\mathcal{P}_i$, where $\mathsf{pw}^{\mathcal{P}_i}$ is a password and $\mathsf{role} = \mathsf{sender}$ implies that $\mathcal{P}_i$ wishes to initiate a key exchange, while $\mathsf{role} = \mathsf{receiver}$ implies that $\mathcal{P}_i$ wishes to respond:

- Send (NewSession, sid, $\mathcal{P}_i$, role) to $\mathcal{A}$;
- If one of the following is true, record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$ and mark this record `fresh`:
  - This is the first NewSession query;
  - This is the second NewSession query and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}^{\mathcal{P}_{1-i}})$.

On (TestPwd, sid, $\mathcal{P}_i$, pw*) from the adversary $\mathcal{A}$:
If there is a `fresh` record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$, then set $d \leftarrow d(\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^*)$ and do:

- If $d \leq \delta$, mark the record `compromised` and reply to $\mathcal{A}$ with "correct guess";
- If $d > \delta$, mark the record `interrupted` and reply to $\mathcal{A}$ with "wrong guess".

On (NewKey, sid, $\mathcal{P}_i$, sk) from the adversary $\mathcal{A}$:
If there is no record of the form $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$, or if this is not the first NewKey query for $\mathcal{P}_i$, then ignore this query. Otherwise:

- If at least one of the following is true, then output $(\mathsf{sid}, \mathsf{sk})$ to player $\mathcal{P}_i$:
  - The record is `compromised`;
  - The record is `fresh`, $\mathcal{P}_{1-i}$ is corrupted, and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}^{\mathcal{P}_{1-i}})$ with $d(\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}) \leq \delta$.
- If this record is `fresh`, both parties are honest, there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}^{\mathcal{P}_{1-i}})$ with $d(\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}) \leq \delta$, a key sk' was sent to $\mathcal{P}_{1-i}$, and $(\mathcal{P}_{1-i}, \mathsf{pw}^{\mathcal{P}_{1-i}})$ was `fresh` at the time, then output $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$.
- In any other case, pick a new random key sk' of length λ and send $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$.
- Mark the record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$ as `completed`.

Fig. 3. Ideal functionality $\mathcal{F}_{\mathsf{fPAKE}}$ [DHP+18]

On (TestPwd, sid, $\mathcal{P}_i$, pw*) from the adversary $\mathcal{A}$:
If there is a `fresh` record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$, then set $d \leftarrow d(\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^*)$ and do:

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

Fig. 4. A modified TestPwd interface to allow for different leakage [DHP+18]

3.2 On the insecurity of previous fuzzy PAKE constructions

Dupont et al. [DHP+18] proposed a concrete fPAKE protocol (which they call $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$) that uses an implicit PAKE (iPAKE) and an error-correcting code (ECC)[7].

[7] As already discussed in the introduction, Dupont et al. [DHP+18] use the terminology of Robust Secret Sharing instead of ECC.

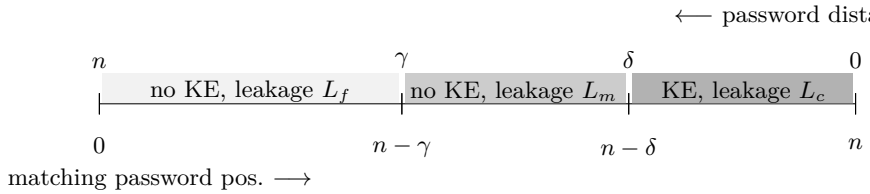


Fig. 5. A graphical explanation of the relaxed TestPwd interface of Figure 4. Here, "KE" stands for successful key exchange (e.g., when 0 to δ errors occur), and "no KE" stands for failure of key exchange, i.e., both parties outputting uniformly random strings.

In this section we explain why, in fact, the $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ protocol does not realize the $\mathcal{F}_{\mathsf{fPAKE}}$ functionality. In order to do so, we first need to discuss the main ideas of $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$. We refer the reader to the original work [DHP+18] for more details.

Description of $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$. Let $\mathsf{pw} = (\mathsf{pw}_1, \cdots, \mathsf{pw}_n) \in \mathcal{D} = \mathcal{L}^n$, for some finite alphabet $\mathcal{L}$, be the input password of the Sender and $\mathsf{pw} = (\mathsf{pw}'_1, \cdots, \mathsf{pw}'_n) \in \mathcal{D} = \mathcal{L}^n$ the input password of the Receiver. We typically talk about passwords being vectors of *password characters*. In the first stage of the protocol, the two parties engage in n iPAKE protocols. In the t-th execution, the Sender's input is the t-th password character pw_t and the Receiver's input is pw'_t. Parties receive outputs $K_t, K'_t \in \mathbb{F}_q$ respectively. The iPAKE with the correctness property guarantees that if $\mathsf{pw}_t = \mathsf{pw}'_t$ and there is no attack, then $K_t = K'_t$. Otherwise the keys K_t and K'_t are independent. At the end of this stage, the parties hold *character keys* $\boldsymbol{K} := (K_t)_{t\in[n]}, \boldsymbol{K'} := (K'_t)_{t\in[n]} \in \mathbb{F}_q^n$ respectively.

In the second stage, the Sender samples a random field element $U \leftarrow \mathbb{F}_q$ and encodes it using an error correcting code with a hiding property. Let $\boldsymbol{C} \in \mathbb{F}_q^n$ denote the resulting codeword. The Sender now performs a one-time-pad and sends $\boldsymbol{E} \coloneqq \boldsymbol{C} + \boldsymbol{K}$ to the Receiver. The Receiver computes $\boldsymbol{C'} \coloneqq \boldsymbol{E} - \boldsymbol{K'}$ and decodes the obtained codeword to U'. The idea is that if $\boldsymbol{K}$ and $\boldsymbol{K'}$ only differ at a few positions, then both codewords $\boldsymbol{C}$ and $\boldsymbol{C'}$ decode to the same value thanks to the error correcting property of the code, and hence both parties output the same key.

In order to realize the $\mathcal{F}_{\mathsf{fPAKE}}$ functionality, one needs to bind the n iPAKE executions and the final message together, as otherwise, active attacks on individual iPAKE instances allow to derive information about the corresponding password characters as observed already in [DHP+18]. To this end, the protocol $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ actually makes use of a *labelled* iPAKE (or ℓ-iPAKE for short). In a nutshell, ℓ-iPAKE allows each party to additionally input a public label that serves as a public authentication string (i.e., any tampering with the label can be detected efficiently). The Sender then samples a signing key pair $(\mathsf{sk}, \mathsf{vk})$ at the beginning of the $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ protocol and uses vk as their label in all ℓ-iPAKE executions. The Receiver checks that all the output labels of the ℓ-iPAKE executions are the same and aborts if this is not the case. In the last message, the Sender

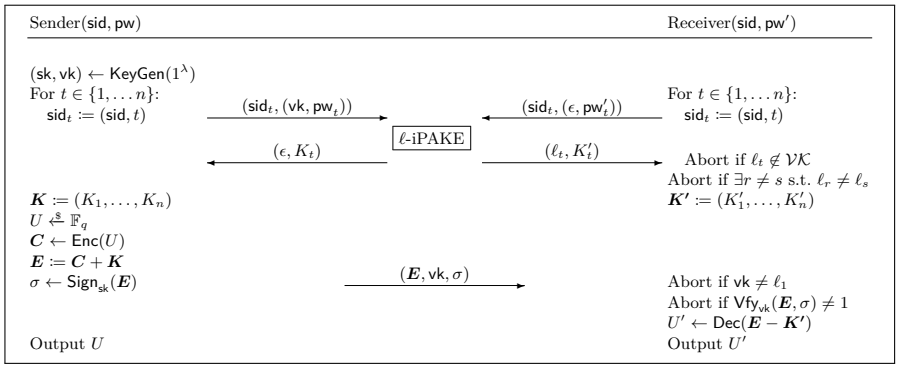


Fig. 6. Description of the $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ protocol from [DHP+18] that uses n instances of an ℓ-iPAKE and a signature scheme $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vfy})$ whose verification key space is $\mathcal{VK}$ and an error-correcting code $(\mathsf{Enc}, \mathsf{Dec})$.

sends their signature $\sigma = \mathsf{Sign}_{\mathsf{sk}}(\boldsymbol{E})$ and vk to the Receiver in addition to $\boldsymbol{E}$. The Receiver aborts if vk is not equal to the output labels of the ℓ-iPAKE instances or with the signature does not verify. See Fig. 6 for a schematic description of the protocol.

Attack on $\pi^{\mathsf{RSS}}_{\mathit{fPAKE}}$. It turns out that the way ℓ-iPAKE instances and the last message were tightened together in the $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ protocol is not sufficient to realize the $\mathcal{F}_{\mathsf{fPAKE}}$ functionality. Let us assume that both parties enter passwords $\mathsf{pw}, \mathsf{pw}'$ whose distance is exactly at the threshold δ, i.e., $d(\mathsf{pw}, \mathsf{pw}') = \delta$. This means that the fPAKE would yield matching keys for both parties if there was no active attack on the protocol. The adversary can now make one of the ℓ-iPAKE key exchanges fail by making a wrong guess (without loss of generality, assume the first one). This perfectly emulates a situation where the first password values pw_1 and pw'_1 mismatch. Now, imagine a simulator that ensures indistinguishability of the ideal execution of $\mathcal{F}_{\mathsf{fPAKE}}$ from the real protocol execution $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ where parties make calls to the ℓ-iPAKE instances. The simulator's task is to figure out whether the described attack actually constitutes a successful DoS attack against the whole fPAKE or not, i.e., whether the passwords entered by both parties were already at the error tolerance δ and if $\mathsf{pw}_1 = \mathsf{pw}'_1$. If so, the result of the attack in the real world is that both parties compute random keys, otherwise they compute the same key. The only leakage about the passwords accessible by the simulator is through the TestPwd interface of $\mathcal{F}_{\mathsf{fPAKE}}$. Calling this interface, however, requires the simulator to provide a password γ-close to pw or pw', as otherwise parties will already receive randomized keys in the ideal world because of the session getting `interrupted`. The probability that the simulator guesses the password depends on the dictionary size but is not negligibly close to 1. This results in a significant distinguishing advantage for the environment. We highlight that this attack has practical implications on the protocol. For the sake

of simplicity, assume that the passwords must match exactly (i.e., $\delta = 0$), and that the password is encoded as a bit string. An attacker can completely retrieve a user's n-bit password from only n executions of the protocol with that user by guessing one bit per execution. Whenever key exchange is successful the bit was guessed correctly. For $\delta > 0$, the attack needs slightly more guesses as the attacker must ensure that the distance from his guess to the user's password is exactly at the δ boundary before the bit-wise guessing will work. We now describe the attack more formally.

Formalizing the attack on $\pi^{\mathsf{RSS}}_{\mathit{fPAKE}}$ To formalize the attack, we need to show that for every simulator $\mathcal{S}$ there exists an environment $\mathcal{Z}$ that can distinguish the interaction with the hybrid world adversary and the protocol from the interaction with the simulator and the ideal functionality with non-negligible probability. In fact, in Fig. 7, we describe a distinguishing environment $\mathcal{Z}$ that can distinguish the hybrid and ideal worlds with non-negligible probability *for all* PPT simulators.

Let us first look at the pseudocode in Fig. 7 and consider that $\mathcal{Z}$ interacts with the hybrid world (i.e., the protocol $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ making calls to $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$, and the real world dummy adversary $\mathcal{A}$ that does what $\mathcal{Z}$ tells them to do). One can easily observe that if the environment's choice bit is set to $b = 0$, then the attack results in DoS (i.e., output keys sk and sk' are chosen uniformly at random) and if $b = 1$, the attack does not disturb the key exchange (i.e. $\mathsf{sk} = \mathsf{sk}'$). Hence, if $\mathcal{Z}$ interacts with the hybrid world, they will correctly output HYBRID except with negligible probability (note that $\Pr[\mathsf{sk} = \mathsf{sk}' \mid \mathsf{sk} \xleftarrow{\$} \mathbb{F}_q, \mathsf{sk}' \xleftarrow{\$} \mathbb{F}_q] = \frac{1}{q}$, where $q \approx 2^\lambda$).

Let us now focus on the ideal world, where $\mathcal{Z}$ interacts with the ideal functionality $\mathcal{F}_{\mathsf{fPAKE}}$ and a simulator $\mathcal{S}$. The task of $\mathcal{S}$ is to make $\mathcal{F}_{\mathsf{fPAKE}}$ produce an output indistinguishable from the hybrid world described above. Let us first summarize all the information that $\mathcal{S}$ gets. From the environment, $\mathcal{S}$ receives $\mathsf{pw}^*_1 \in \mathbb{F}_p$ and we can assume that $\mathcal{S}$ knows that this character is different from pw_1. According to the definition of $\mathcal{F}_{\mathsf{fPAKE}}$, $\mathcal{S}$ gets no information about the passwords $\mathbf{pw}, \mathbf{pw}'$ from $\mathcal{F}_{\mathsf{fPAKE}}$ by default. The only way $\mathcal{S}$ could get information via $\mathcal{F}_{\mathsf{fPAKE}}$ is through the TestPwd interface, where $\mathcal{S}$ has to guess $\mathbf{pw}$ or $\mathbf{pw}'$. If $\mathcal{S}$ makes a wrong guess of one of the passwords, this password gets marked as `interrupted` which means that the simulator loses all power over the keys being output to both parties (i.e., each party gets a randomly sampled key). In order to get useful information about the passwords via the TestPwd interface without causing random keys, the simulator would have to make a close enough guess of one of the passwords. But since $\mathcal{S}$ knows only that $\mathsf{pw}_1 \neq \mathsf{pw}^*_1$ and the passwords were chosen uniformly at random, the probability that $\mathcal{S}$ succeeds in guessing is not negligibly close to 1 (the exact probability depends, of course, on the size of the dictionary and γ). .

Hence, if $\mathcal{S}$ would be a simulator that could ensure indistinguishability of the ideal and hybrid world, this would imply that $\mathcal{S}$ is able to guess b correctly with non-negligible probability. Since b was chosen uniformly at random from $\{0, 1\}$ by $\mathcal{Z}$, this is information theoretically impossible.

1. Sample a bit $b \in \{0,1\}$ uniformly at random.
2. Sample $\mathbf{pw} = (\mathsf{pw}_1, \ldots, \mathsf{pw}_n) \in \mathbb{F}_p^n$ uniformly at random.
3. Choose a random $\mathbf{pw'} = (\mathsf{pw}'_1, \ldots, \mathsf{pw}'_n) \in \mathbb{F}_p^n$ s.t. (1) $d(\mathbf{pw}, \mathbf{pw'}) = \delta$ and (2) $\mathsf{pw}'_1 = \mathsf{pw}_1$ if $b = 0$ and $\mathsf{pw}'_1 \neq \mathsf{pw}_1$ if $b = 1$.
4. Choose $\mathsf{pw}^*_1 \in \mathbb{F}_p \setminus \{\mathsf{pw}_1\}$.
5. Send $(\textsc{NewSession}, \mathsf{sid}, \mathbf{pw}, \text{Sender})$ to $\mathcal{P}_0$ and $(\textsc{NewSession}, \mathsf{sid}, \mathbf{pw'}, \text{Receiver})$ to party $\mathcal{P}_1$, where sid chosen arbitrarily.
6. Upon receiving $(\textsc{NewSession}, (\mathsf{sid}, 1), \mathcal{P}_0, \text{Sender})$ and $(\textsc{NewSession}, (\mathsf{sid}, 1), \mathcal{P}_1, \text{Receiver})$, instruct the adversary to send $(\textsc{TestPwd}, (\mathsf{sid}, 1), \mathcal{P}_0, \mathsf{pw}^*_1)$.
7. Instruct the adversary to send $(\textsc{NewKey}, (\mathsf{sid}, 1), \mathcal{P}_0, \mathsf{sk}_1)$ and then $(\textsc{NewKey}, (\mathsf{sid}, 1), \mathcal{P}_1, \mathsf{sk}_1)$ for an arbitrary key sk_1.
8. For every $t = 2, \ldots, n$:
   (a) Upon receiving $(\textsc{NewSession}, (\mathsf{sid}, t), \mathcal{P}_0, \text{Sender})$ and $(\textsc{NewSession}, (\mathsf{sid}, t), \mathcal{P}_1, \text{Receiver})$, instruct the adversary to send $(\textsc{NewKey}, (\mathsf{sid}, t), \mathcal{P}_0, \mathsf{sk}_t)$ and then $(\textsc{NewKey}, (\mathsf{sid}, t), \mathcal{P}_1, \mathsf{sk}_t)$ for an arbitrary key sk_t.
9. Upon receiving $(\mathsf{sid}, \mathsf{sk})$ from $\mathcal{P}_0$ and $(\mathsf{sid}, \mathsf{sk})$ from $\mathcal{P}_1$ distinguish the following two cases:
   - $b = 0$: If $\mathsf{sk} \neq \mathsf{sk}'$, output HYBRID. Otherwise, output IDEAL.
   - $b = 1$: If $\mathsf{sk} = \mathsf{sk}$, output HYBRID. Otherwise, output IDEAL.

Fig. 7. Pseudocode of an environment $\mathcal{Z}$ distinguishing the interaction with the $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ protocol in the $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$ hybrid world from the interaction with the $\mathcal{F}_{\mathsf{fPAKE}}$ ideal functionality.

3.3 Towards repairing the previous construction

In their work, Dupont et al. mention an alternative way of binding the iPAKE instances and the last message together (see footnote 6 on page 26 of the eprint version of their paper [DHP+17]). Instead of using labels and a one-time signature scheme, they suggest letting the two parties sign every message using the split transformation put forward by Barak et al. [BCL+05]. A natural first step when trying to fix the $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ protocol is hence to follow this alternative approach. Unfortunately, it turns out that applying the split transformation is not straightforward.

At a very high level, the split transformation is a generic way to transform a protocol realizing an ideal functionality $\mathcal{F}$ *assuming authenticated channels* into a protocol that achieves the same without authenticated channels.[8] Instead, the transformed protocol works in the $\mathcal{F}_{\mathsf{SA}}$-hybrid world, where $\mathcal{F}_{\mathsf{SA}}$ is the ideal functionality for split authentication recalled in Section 2.5.

Intuitively, this seems to be exactly what is needed to bind the iPAKE instances and the message of $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ together. Let $\pi^{\mathsf{plain}}_{\mathsf{fPAKE}}$ denote the protocol which is defined exactly as $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ except that it uses $\mathcal{F}_{\mathsf{iPAKE}}$ instead of $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$ instances and the sender does not sign the last message. Assume that the sender and the

[8] This is an oversimplified statement as the transformed protocol does not realize $\mathcal{F}$ but its *split variant*, denoted $s\mathcal{F}$. See the work of Barak et al. [BCL+05] for more details.

receiver have an authenticated channel available which they can use for all the communication happening during all the n iPAKE executions and parties can use the same channel for the last message. Then the $\pi^{\mathsf{plain}}_{\mathsf{fPAKE}}$ protocol would be a secure fuzzy PAKE. Unfortunately, this intuition is misleading as it does not reflect the UC formalization of the $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$ protocol correctly.

The core of the problem is that the result of Barak et al. does not apply to hybrid protocols like $\pi^{\mathsf{plain}}_{\mathsf{fPAKE}}$. In a bit more detail, recall that the protocol $\pi^{\mathsf{plain}}_{\mathsf{fPAKE}}$ is defined in the $\mathcal{F}_{\mathsf{iPAKE}}$-hybrid world. This, in particular, means that the only message sent between the sender and the receiver in $\pi^{\mathsf{plain}}_{\mathsf{fPAKE}}$ is the last message. The rest of the protocol consists of the communication between a party and $\mathcal{F}_{\mathsf{iPAKE}}$ ideal functionality. Sending (only) the last message over an authenticated channel would not help us to bind the n iPAKE executions together, so such a protocol could not realize $\mathcal{F}_{\mathsf{fPAKE}}$. Hence, applying the split transformation does not have the desired effect of protecting against active attacks on individual password characters, like the one described in Fig. 7, in $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$.

Our approach. To address this issue, we define a protocol that does not work in the $\mathcal{F}_{\mathsf{iPAKE}}$-hybrid world. Instead, it lets the Sender and the Receiver run the code of a protocol π_{iPAKE} realizing $\mathcal{F}_{\mathsf{iPAKE}}$ directly. In a bit more detail, let us assume that the Sender received $\mathsf{pw} = (\mathsf{pw}_1, \ldots, \mathsf{pw}_n)$ as input. For every $t \in [n]$, our protocol instructs the Sender to run the code of the π_{iPAKE}-sender on input pw_t. Whenever the π_{iPAKE}-sender would have sent a message to the π_{iPAKE}-receiver instance, the Sender does so via the channel available for the two parties in our protocol. The Receiver in our protocol is defined analogously. This way, all communication happening during the iPAKE executions and the last message are sent via the same channel between the Sender and the Receiver. Hence, the split transformation will have the desired effect. We formally prove this intuitive statement in the next section, where we define our final protocol precisely.

Potential alternative approach. Another way to resolve the insufficient binding of the iPAKE instances and the last message together could be to define a new building block, "batched iPAKE". This new iPAKE primitive would take as input a vector $(\mathsf{pw}_t)_{t\in[n]}$ and output a vector $(K_t)_{t\in[n]}$. An adversary would be allowed to make a TestPwd query, but would always have to test all password characters $(\mathsf{pw}^*_t)_{t\in[n]}$. Hence, the adversary would be forced to launch a man-in-the-middle attack either against all iPAKE instances, or none of them, which would guarantee the binding of the iPAKEs. In order to bind iPAKEs to the last message, we would require the "batch iPAKE" to be labelled. As in $\pi^{\mathsf{RSS}}_{\mathsf{fPAKE}}$, the sender would use their verification key as a label and then use the corresponding signing key to sign the last message.

4 Our Construction

General idea. The idea behind our construction is very simple: we modify the fuzzy PAKE of Dupont et al. [DHP+18] by letting the receiver apply list decoding

instead of unique decoding. Since now the receiver ends up with a list of potential key candidates (or codewords, more precisely), we additionally let the sender compute a "hint" on which codeword is the correct one. More formally, the sender adds the hash of the original codeword $h := H(\boldsymbol{C})$ to the last message. The receiver applies list decoding which results in a list of candidates $\boldsymbol{C}^1, \ldots, \boldsymbol{C}^\ell$. They then find the one that satisfies $h = H(\boldsymbol{C}^j)$ and compute $U \leftarrow \mathsf{Dec}(\boldsymbol{C}^j)$. The value h is a hint on which of the candidates is the correct one. The benefit of our construction is that it can correct more errors in the password: for a Reed-Solomon code of rank k and block length n, unique decoding can correct up to $\lfloor \frac{n-k}{2} \rfloor$ errors (Lemma 1), while list decoding outputs a list containing the original codeword in the presence of $n - 1 - \sqrt{(k-1)n}$ errors. Fig. 2 provides several numerical examples of these terms, demonstrating that for growing password sizes, the list decoding approach yields a significantly better error correction capability than the unique decoding approach in [DHP+18].

On the other hand, it is *not* intuitively clear that the extra information passed from the sender to the receiver does not void any of the security guarantees of the protocol. To give an example, consider a different kind of hint $\bar{h} := H(\mathsf{sk})$, where sk is the output key of the sender. Such a hint helps the receiver to choose its output key just as well as the hint $h := H(\boldsymbol{C})$. However, a fuzzy PAKE protocol sending $\bar{h}$ cannot realize $\mathcal{F}_{\mathsf{fPAKE}}$. The reason is that $\bar{h}$ depends deterministically on secret information known to the sender (namely its output key sk), and hence it leaves no wiggle room for a simulator, who simply has to guess sk in order to simulate correctly. While the hint $H(\boldsymbol{C})$ does include some randomness contributed by the sender to the encoding algorithm Enc, it is not straightforward to see that the resulting fuzzy PAKE is simulatable, meaning that it does not involuntarily reveal any information about the sender's password through the hint.

In the remainder of this section, we hence carefully analyze the security of our fuzzy PAKE protocol given in Fig. 8.

Formal protocol description. The protocol is depicted in Fig. 8. To be able to fix previous issues of imperfect binding of messages, and as detailed in Section 3.2, we let parties send messages to each other over functionality $\mathcal{F}_{\mathsf{SA}}$ (the functionality was briefly discussed in Section 2.5 and is formally defined in Fig. 13). The protocol parties take a password from a dictionary $\mathcal{D} = \mathcal{L}^n$ as input, where $\mathcal{L}$ is some finite alphabet. We write that the sender chooses $\boldsymbol{s} \xleftarrow{\$} \mathbb{F}_q^k$ and encodes it using $\mathsf{Enc}(\boldsymbol{s})$. One can equivalently say that the sender chooses a message $s_0 \xleftarrow{\$} \mathbb{F}_q$ and randomness $\boldsymbol{r} \xleftarrow{\$} \mathbb{F}_q^{k-1}$ and uses the randomized encoding $\overline{\mathsf{Enc}}(s_0, \boldsymbol{r})$. To make our protocol description modular, let π_j denote an instance of a protocol that UC-realizes $\mathcal{F}_{\mathsf{iPAKE}}$. We write $X \leftarrow \pi_j(\mathcal{P}_i, \mathsf{sid}, \mathsf{pw}^{\mathcal{P}_i})$ to denote that $\mathcal{P}_i$ outputs X when running π_j on sid and $\mathsf{pw}^{\mathcal{P}_i}$ and $K \leftarrow \pi_j(\mathcal{P}_i, \mathsf{sid}, X)$ to denote that $\mathcal{P}_i$ outputs K as response to input X when running π_j on sid. For the sake of simplicity, we assume that the π_j protocols are one-round protocols. For multi-round protocols, one can send every message again over $\mathcal{F}_{\mathsf{SA}}$ to the respective other party, like in the first round. A more formal description of the protocol, using the interfaces of $\mathcal{F}_{\mathsf{fPAKE}}$ can be found in Figure 14.

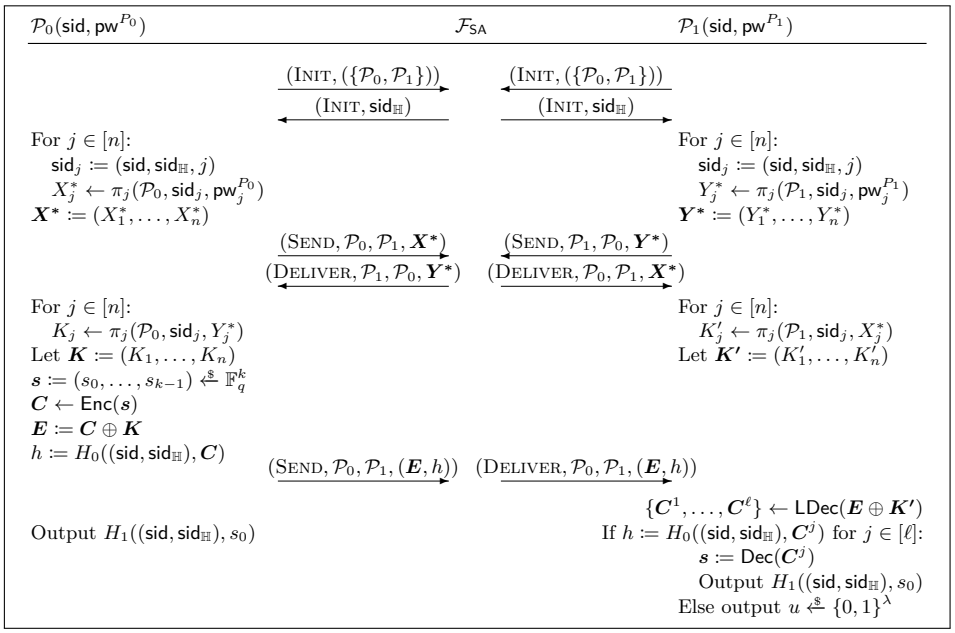


Fig. 8. The protocol in the $\mathcal{F}_{\mathsf{SA}}$-hybrid model. We drop sid from all messages for brevity. We recall $\mathcal{F}_{\mathsf{SA}}$ in Figure 13 in the appendix, but for the sake of understanding our protocol it is enough to know that $\mathcal{F}_{\mathsf{SA}}$ is used to transmit the protocol messages. With π_j we denote an instances of a protocol that UC-realizes $\mathcal{F}_{\mathsf{iPAKE}}$ (cf. Figure 12). We write $X \leftarrow \pi_j(\mathcal{P}_i, \mathsf{sid}, \mathsf{pw}_j^{\mathcal{P}_i})$ to denote that $\mathcal{P}_i$ outputs X when running π_j on sid with password $\mathsf{pw}_j^{\mathcal{P}_i}$ and $K \leftarrow \pi_j(\mathcal{P}_i, \mathsf{sid}, X)$ to denote that $\mathcal{P}_i$ outputs K as response to input X when running π_j on sid.

We start by proving the correctness of our protocol.

Theorem 1. *If $\mathcal{C}$ is a randomized linear code which is efficiently δ-list decodable, H_1 is a ε_2-collision-resistant hash function and π is ε_1-correct and UC-realizes $\mathcal{F}_{\mathsf{iPAKE}}$ then the protocol in Figure 8 is (δ, ε_0)-correct for*

$$\varepsilon_0 = n\varepsilon_1 + \varepsilon_2.$$

Remark 1 (Implications for previous results). Note that our construction implies (a fixed version of) the original fuzzy PAKE of Dupont et al. [DHP+18] when falling back to unique decoding algorithms. Indeed, one can simply set the list size of the list decoding to one and thus obtain unique decoding instead of list decoding as in [DHP+18], preserving our security analysis. There are two notable differences in this "fallback" version of our fuzzy PAKE and the one of [DHP+18]. First, our version applies a proper binding mechanism by using the split functionality $\mathcal{F}_{\mathsf{SA}}$ for sending messages [BCL+05], at the cost of one signature per message (see Appendix D for the concrete instantiation). Second, our security proof does not rely on the smoothness property of the underlying code. This enables the usage of a variety of codes for which such properties are not well researched, or do not hold at all.

Proof. Assume that two honest parties $\mathcal{P}_0, \mathcal{P}_1$ execute the protocol from figure Fig. 8 on input pw^{P_0} and pw^{P_1} with $d(\mathsf{pw}^{P_0}, \mathsf{pw}^{P_1}) \leq \delta$, respectively. Further, assume that the adversary $\mathcal{A}$ attacking this protocol is passive. Each instance π_j of π is ε_1-correct by assumption (see Definition 1). We construct a sequence of hybrids $\mathcal{H}_0, \ldots, \mathcal{H}_n$ where in the j-th hybrid we abort the execution of the protocol if the two parties of π_j output different keys even though the respective inputs $\mathsf{pw}_j^{P_0}$ and $\mathsf{pw}_j^{P_1}$ are the same. By the ε_1-correctness of π, $\mathcal{H}_j$ differs from $\mathcal{H}_{j-1}$ at most by ε_1 for all $j \in \{1, \ldots, n\}$. Now in $\mathcal{H}_n$, as we have $d(\mathsf{pw}^{P_0}, \mathsf{pw}^{P_1}) \leq \delta$, there are at least $n - \delta$ instances of π where $\mathcal{P}_0$ and $\mathcal{P}_1$ run on matching inputs $\mathsf{pw}_j^{P_0}$ and $\mathsf{pw}_j^{P_1}$. Thus, we have $K_j = K'_j$ for at least $n - \delta$ keys. So the codeword $\boldsymbol{C'} = \boldsymbol{E} \oplus \boldsymbol{K'}$ that the receiver computes differs from the sender's $\boldsymbol{C}$ on at most δ entries. As $\mathcal{C}$ is δ-list decodable, the list that is output by LDec contains the same $\boldsymbol{C}$ that the sender encoded. Now assume that there are two possible words $\boldsymbol{C}, \boldsymbol{C}^*$ with $\boldsymbol{s} = \mathsf{Dec}(\boldsymbol{C})$ and $\boldsymbol{s}^* = \mathsf{Dec}(\boldsymbol{C}^*)$ such that $H_1((\mathsf{sid}, \mathsf{sid}_H), s_0) = H_1((\mathsf{sid}, \mathsf{sid}_H), s_0^*)$ with probability ε_2. One can construct an adversary against the collision-resistance of H_1 that internally runs $\mathcal{P}_0(\mathsf{pw}^{P_0})$ and $\mathcal{P}_1(\mathsf{pw}^{P_1})$ and outputs s_0, s_0^* as above. We get

$$\Pr[K \neq K' \mid (K, K') \leftarrow \mathrm{out}_{\mathcal{A},\pi}\langle \mathcal{P}_0(\mathsf{pw}^{P_0}), \mathcal{P}_1(\mathsf{pw}^{P_1})\rangle] \leq n\varepsilon_1 + \varepsilon_2.$$

Theorem 2. *If $\mathcal{C}$ is a $\mathcal{RS}[n, k, S]$ code over $\mathbb{F}_q$ for $q = 2^{\Omega(\lambda)}$, π UC-realizes $\mathcal{F}_{\mathsf{iPAKE}}$ and is correct, and H_0 and H_1 are modeled as random oracles then the protocol in Figure 8 UC-realizes $\mathcal{F}_{\mathsf{fPAKE}}^{M}$ in the $\mathcal{F}_{\mathsf{SA}}$-hybrid model with error tolerance $\delta = n - 1 - \sqrt{(k-1)n}$ and leakage threshold $\gamma = n - k$.*

We discuss the proof of Theorem 2 in Section 5.

4.1 Comparison with properties and proof techniques in [DHP+18]

As previously discussed, prior work [DHP+18] constructs fuzzy PAKE protocols from linear error-correcting codes, using the fact that the errors in noisy codewords can be uniquely corrected up to the *unique decoding radius*. However, in this work, we wish to offer improved PAKE functionality by using list-decoding algorithms to correct errors beyond the unique decoding radius. The proof strategy employed in [DHP+18] does not readily generalise to this setting, and the presence of the hint h becomes crucial in the proof of Theorem 2, as we will explain.

The PAKE construction of [DHP+18] builds fuzzy PAKE protocols from robust secret-sharing (RSS) schemes with various special properties, which are in turn constructed from linear error-correcting codes. First, we summarize the RSS properties used in the PAKE construction of [DHP+18] and rephrase them as properties of error-correcting codes. Then, we describe the techniques used in the proof of the *reconstruction* and *smoothness* properties, and how this proof strategy leads to challenges when we generalize to list-decoding. Finally, we explain how the inclusion of the hint h allows us to circumvent these issues.

RSS properties in [DHP+18]. An RSS scheme consists of a sharing algorithm, which takes a message as input and produces n output shares, and a reconstruction algorithm, which, on input a list of (possibly corrupted) shares, returns a secret.

In this work, in line with our study of PAKE protocols from a decoding perspective, we phrase our algorithms in terms of randomised error-correcting codes as in Section 2.2, instead of RSS schemes. With this view, the randomised encoding algorithm $\overline{\mathsf{Enc}}$ simply corresponds to a share algorithm with message space generalised from $\mathbb{F}$ to $\mathbb{F}^{k_M}$, and explicit random input $\boldsymbol{R} \in \mathbb{F}^{k_R}$.

The PAKE construction of [DHP+18] uses the following properties of randomised encoding algorithms (with their corresponding RSS terminology from [DHP+18] included for clarity):

- B-query uniformity (Section 2.3), which corresponds to strong B-privacy for RSS;
- the existence of an efficient algorithm for e-error correction (Section 2.2), which corresponds to e-robustness for RSS;
- for all $E \subseteq [n]$ with $|E| \leq e$, all $\boldsymbol{M} \in \mathbb{F}^{k_M}$, the distribution of

$$\left\{ \mathsf{Dec}(\boldsymbol{V}) \colon \boldsymbol{R} \leftarrow \mathbb{F}^{k_R}, \boldsymbol{V}|_E := \overline{\mathsf{Enc}}(\boldsymbol{M}, \boldsymbol{R})|_E, \boldsymbol{V}|_{\bar{E}} \leftarrow \mathbb{F}^{\bar{E}} \right\} \tag{1}$$

  is uniform over $\mathbb{F}^{k_M}$. This corresponds to e-smoothness for RSS;
- e-smoothness on random secrets. This is similar to e-smoothness except that $\boldsymbol{M}$ is also sampled uniformly at random from $\mathbb{F}^{k_M}$.

The authors of [DHP+18] give a construction of a randomised linear code with each of the above properties from a standard linear error correcting code. The construction was originally given as an RSS.

Lemma 3 ([DHP+18, Lemma 5], restated). *Given an* $[n+1, k, n-k+2]$*-linear code* $\mathcal{C}_0$*, there is a* $[n, k, d = n-k+1]$ *randomised linear code* $\mathcal{C}$ *with* $k_M = 1$*,* $(k-1)$*-smoothness,* $(k-1)$*-query uniformity,* $\lfloor \frac{n+k}{2} \rfloor$*-error correction and* $\lfloor \frac{n+k}{2} \rfloor - 1$ *smoothness on random secrets.*

Arguing smoothness and decoding properties for Lemma 3. The proof of Lemma 3 in [DHP+18] starts by defining $\mathcal{C}$ to be the code obtained by puncturing $\mathcal{C}_0$ in its last coordinate. Then, $(k-1)$-smoothness is proved with respect to the following decoding algorithm. On input a codeword in $\mathcal{C}'$, the decoding algorithm selects the subvector of k entries of the codeword and then inverts the submatrix of the generator matrix of $\mathcal{C}'$ corresponding to these entries on the subvector to find the original message. Finally, this message vector is combined with the column of the generator matrix of $\mathcal{C}$ corresponding to its last coordinate.

Intuitively, in the case of Reed-Solomon codes, this is like choosing k codeword entries of a punctured codeword with the last entry missing, and then interpolating to find out what the last entry should be.

With this decoding algorithm, it is relatively easy to prove smoothness, because $\boldsymbol{V}$ in Equation (1) has at least $n-k+1$ random entries, so at least one entry out of the k entries is random. Based on the MDS properties of $\mathcal{C}$, one can show that the "last coordinate" computed from the k selected entries must also be random.

In order for this decoding algorithm to be robust, it must be able to select k error-free positions. Otherwise, the submatrix inversion step will incorporate an error and produce an incorrect message. In the case of Reed-Solomon codes, this could be achieved by using another decoding algorithm (such as the Berlekamp-Welch algorithm) to identify these positions.

Therefore, proving the smoothness property relies heavily on the MDS properties of $\mathcal{C}$ and the existence of efficient decoding algorithms to allow errors in the codeword to be located.

Moving beyond the unique decoding radius. In this work, we intend to construct a PAKE scheme which works for errors beyond the unique decoding radius. In this high-error setting, the natural replacement for a decoding algorithm is a list-decoding algorithm, which produces all possible codewords within a particular radius of the transmitted word. The first challenge for our PAKE algorithm to overcome was to allow the sender and receiver to agree on the same password, even though decoding may no longer produce a unique codeword which can be used to determine the password. We solve this problem by way of the hint $h = H(\boldsymbol{C})$.

Another challenge is the fact that smoothness, an important property in the security proofs of [DHP+18], relies heavily on the special decoding strategy introduced, which must apply a linear map to error-free positions in a codeword in order to ensure robustness. In this high-error setting, it may be impossible to identify a unique set of error-free positions in a codeword. Thus, the decoding strategy above, which involves inverting the generator matrix on an error-free set of codeword entries, may not be possible either. Therefore, it seems challenging to

construct a list-decoding algorithm with a generalised smoothness property via a linear decoding algorithm that still returns the correct codeword somewhere in its list, as the linear decoding step may always be applied to strings with errors instead of error-free positions.

However, amazingly, the hint that we introduce lets us circumvent this issue. In order to prove our PAKE protocol secure, we require only that the list decoding algorithm will not produce the same codeword provided by the hint after many random errors are added to it. This follows almost trivially as adding a large number of random errors produces a string which is so far away from the original codeword that the list decoding algorithm cannot produce the original codeword. This relates to $\mathbf{G}_8$ and Lemma 4 in the security proof.

5 Proof of Security

To prove that the protocol in Figure 8 UC-realizes $\mathcal{F}_{\mathsf{fPAKE}}^{M}$, we have to show that every PPT environment $\mathcal{Z}$ has at most negligible advantage in distinguishing a real execution of the protocol from an ideal execution, with $\mathcal{F}_{\mathsf{fPAKE}}^{M}$ and the simulator that we show in Figs. 9 to 10. We show this by a sequence of hybrid games. In the following, we give intuition for each of the game hops and prove the necessary lemmas. We defer the full proof of Theorem 2 to Section A of the Supplementary Material.

On using other iPAKE protocols than EKE: The current simulator in Figures 9 to 10 is written with an iPAKE protocol in mind in which each party only sends one message. However, the simulator can be easily adapted to work with any iPAKE protocol as follows: Instead of forwarding only one set of messages $(\{Z_j^*\}_{j=1}^n$ between $\mathcal{Z}$ and the iPAKE simulators, $\mathcal{S}$ would forward all iPAKE messages between $\mathcal{Z}$ and the corresponding iPAKE simulator.

Proof (Sketch). We depict a UC execution of Fig. 8 in Fig. 14 in Section D of the Supplementary Material, which makes explicit all the interfaces of $\mathcal{F}_{\mathsf{fPAKE}}$. The general proof strategy of showing indistinguishability of this protocol from $\mathcal{F}_{\mathsf{fPAKE}}$ is to start with the real-world execution of the protocol from Fig. 14 and replace step-by-step parts of the protocol by simulated parts. The goal is to reach the ideal-world execution where the simulator can work without using the secret passwords of the participating parties but with help from the ideal functionality $\mathcal{F}_{\mathsf{fPAKE}}^{M}$. In the following we give intuition for every game-hop we make:

- $\mathbf{G}_0$: The proof starts with the real execution of the protocol.
- $\mathbf{G}_1$: We move the whole execution into one machine which we will call simulator. We also add dummy parties and a functionality that merely forwards the input of the parties (i.e. NewSession calls) to the simulator. Furthermore, the functionality forwards the output of the simulator to the parties as NewKey messages. This allows the simulator to execute the protocol on the real inputs $\mathsf{pw}^{\mathcal{P}_i}$ and $\mathsf{pw}^{\mathcal{P}_{1-i}}$. The goal of the remaining steps is to add

On $(\textsc{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathsf{role})$ from $\mathcal{F}_{\mathsf{fPAKE}}$ or $(\textsc{Init}, (\{\mathcal{P}_0, \mathcal{P}_1\}, \mathsf{sid}))$ from a corrupted $\mathcal{P}_i$ to $\mathcal{F}_{\mathsf{SA}}$:
- Create record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \bot \rangle$ and mark it `fresh`.
- Send $(\textsc{Init}, (\{\mathcal{P}_0, \mathcal{P}_1\}, \mathsf{sid}))$ to $\mathcal{Z}$ in the name of $\mathcal{F}_{\mathsf{SA}}$.

On $(\textsc{Init}, \mathsf{sid}, \mathcal{P}_i, \mathbb{H}, \mathsf{sid}_{\mathbb{H}})$ from $\mathcal{Z}$ in the name of $\mathcal{A}$ to $\mathcal{F}_{\mathsf{SA}}$:
- Perform the same checks as $\mathcal{F}_{\mathsf{SA}}$ on $\mathsf{sid}_{\mathbb{H}}$ and $\mathbb{H}$ and ignore the message if one of them fails.
- Record $\langle \mathbb{H}, \mathsf{sid}_{\mathbb{H}} \rangle$.
- If $\mathcal{P}_i$ is corrupted then provide output $(\textsc{Init}, \mathsf{sid}, \mathsf{sid}_{\mathbb{H}})$ to $\mathcal{P}_i$. Regardless:
  - If $\mathbb{H} = \{\mathcal{P}_0, \mathcal{P}_1\}$ then set $\mathsf{MitM} := 0$.
  - Else if $\mathbb{H} = \{\mathcal{P}_i\}$ set $\mathsf{MitM} := 1$.
- Retrieve $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \bot \rangle$ and replace $\bot$ by MitM.
- If there is not already an instance of $\mathcal{S}_{\mathsf{iPAKE}}$ for session-id $(\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j)$ for $j \in [n]$ create one. // I.e., create $\{\mathcal{S}^j_{\mathsf{iPAKE}}\}_{j=1}^n$ if $\mathsf{MitM} = 1$ or if $\mathsf{MitM} = 0$ and $\{\mathcal{S}^j_{\mathsf{iPAKE}}\}_{j=1}^n$ are not yet created for $\mathcal{P}_{1-i}$.
- For all $j \in [n]$ send $(\textsc{NewSession}, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j), P_i, \mathsf{role})$ to $\mathcal{S}^j_{\mathsf{iPAKE}}$.
- Upon receiving output Z_j^* from all n instances of $\mathcal{S}_{\mathsf{iPAKE}}$, send $(\mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \mathrm{msg} := (Z_1^*, \ldots, Z_n^*))$ formatted as coming from $\mathcal{F}_{\mathsf{SA}}$ to $\mathcal{Z}$ and store $\langle \mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \mathrm{msg} \rangle$.

On instruction from $\mathcal{Z}$ to send $(\textsc{Deliver}, \mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \mathrm{msg} := \{Z_j^*\}_{j=1}^n)$ to $\mathcal{F}_{\mathsf{SA}}$:
- If $\mathsf{MitM} = 0$:
  - If message msg is the one that $\mathcal{S}$ sent, i.e. $\mathcal{S}$ has a record $\langle \mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \mathrm{msg} \rangle$:
    - Give instruction to $\mathcal{S}^j_{\mathsf{iPAKE}}$ to send message Z_j^* to $\mathcal{P}_{1-i}$ in the name of $\mathcal{P}_i$.
    - Remove one appearance of the record $\langle \mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \mathrm{msg} \rangle$.
  - Else: do nothing. // message manipulated
- Else // if $\mathsf{MitM} = 1$
  - Give instruction to $\mathcal{S}^j_{\mathsf{iPAKE}}$ to send message Z_j^* to $\mathcal{P}_{1-i}$ in the name of $\mathcal{P}_i$.

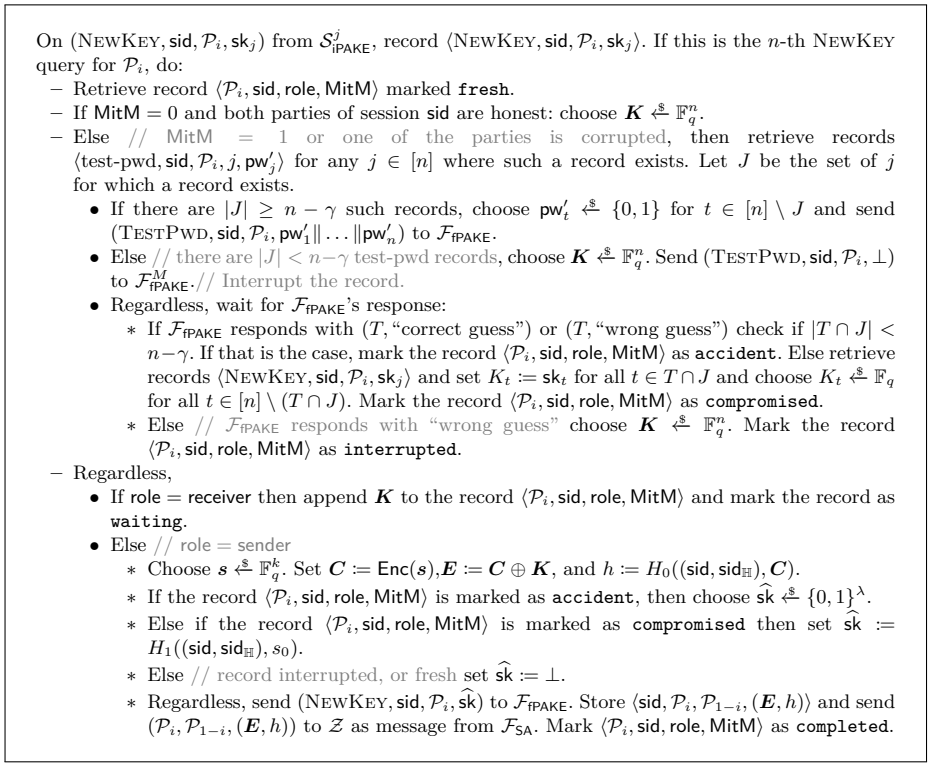
On $(\textsc{TestPwd}, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j), \mathcal{P}_i, \mathsf{pw}'_j)$ from $\mathcal{S}^j_{\mathsf{iPAKE}}$ to $\mathcal{F}^j_{\mathsf{iPAKE}}$
- store $\langle \text{test-pwd}, \mathsf{sid}, \mathcal{P}_i, j, \mathsf{pw}'_j \rangle$. // Remember the query. No need to answer

On instruction from $\mathcal{Z}$ to send $(\textsc{Deliver}, \mathsf{sid}, \mathcal{P}_{1-i}, \mathcal{P}_i, \mathrm{msg} := (\boldsymbol{E}', h'))$ to $\mathcal{F}_{\mathsf{SA}}$:
- Retrieve record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM}, \boldsymbol{K} \rangle$ marked `waiting`.
- If $\mathsf{MitM} = 0$ and both parties of session sid are honest:
  - Remove one appearance of the record $\langle \mathsf{sid}, \mathcal{P}_{1-i}, \mathcal{P}_i, (\boldsymbol{E}', h') \rangle$. // Record exists because no MitM attack.
  - If $\mathsf{role} = \mathsf{receiver}$ then send $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \bot)$ to $\mathcal{F}_{\mathsf{fPAKE}}$.
  - Else, ignore this message. // $\mathsf{role} = \mathsf{sender}$ does not expect this message
- Else // $\mathsf{MitM} = 1$ or one of the parties is corrupted
  - If $\mathcal{P}_i$ is honest and $\mathsf{role} = \mathsf{receiver}$ then
    - Compute $\{\boldsymbol{C}^1, \ldots, \boldsymbol{C}^\ell\} \leftarrow \mathsf{LDec}(\boldsymbol{E}' \oplus \boldsymbol{K})$.
    - If $h' = H_0((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C}^j)$ for $j \in [\ell]$ then set $\boldsymbol{s} := \mathsf{Dec}(\boldsymbol{C}^j)$.
    - If the record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ is marked as `accident`, then choose $\widehat{\mathsf{sk}} \xleftarrow{\$} \{0,1\}^\lambda$.
    - Else if the record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ is marked as `compromised` then set $\widehat{\mathsf{sk}} := H_1((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0)$.
    - Else // record interrupted set $\widehat{\mathsf{sk}} := \bot$.
    - Regardless, send $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \widehat{\mathsf{sk}})$ to $\mathcal{F}_{\mathsf{fPAKE}}$.
  - Else if $\mathcal{P}_i$ is corrupt then output $(\textsc{Received}, \mathsf{sid}, \mathcal{P}_{1-i}, \mathcal{P}_i, (\boldsymbol{E}', h'))$ to $\mathcal{P}_i$.
  - Else, ignore this message. // $\mathsf{role} = \mathsf{sender}$ does not expect this message

On a query $((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0)$ to H_1:
- If there is a record $\langle H_1, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0, k \rangle$ return k. Else draw $k \xleftarrow{\$} \{0,1\}^\lambda$, record $\langle H_1, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0, k \rangle$ and return k.

On a query $((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C})$ to H_0:
- If there is a record $\langle H_0, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C}, \rho \rangle$ return ρ. Else draw $\rho \xleftarrow{\$} \{0,1\}^\lambda$. If there is another record $\langle H_0, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C}', \rho' \rangle$ with $\boldsymbol{C} \neq \boldsymbol{C}'$ but $\rho = \rho'$ abort the simulation. Else record $\langle H_0, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C}, \rho \rangle$ and return ρ.

Fig. 9. The first part of the simulator $\mathcal{S}$ for $\mathcal{F}_{\mathsf{fPAKE}}$ and our Fuzzy PAKE with list-decoding protocol. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{SA}}$ and the random oracles H_0 and H_1. If we write "retrieve record" and the record does not exist, $\mathcal{S}$ ignores the message.



On $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk}_j)$ from $\mathcal{S}^j_{\mathsf{iPAKE}}$, record $\langle \textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk}_j \rangle$. If this is the n-th NewKey query for $\mathcal{P}_i$, do:

- Retrieve record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ marked `fresh`.
- If $\mathsf{MitM} = 0$ and both parties of session sid are honest: choose $\boldsymbol{K} \xleftarrow{\$} \mathbb{F}_q^n$.
- Else // MitM = 1 or one of the parties is corrupted, then retrieve records $\langle \text{test-pwd}, \mathsf{sid}, \mathcal{P}_i, j, \mathsf{pw}'_j \rangle$ for any $j \in [n]$ where such a record exists. Let J be the set of j for which a record exists.
  - If there are $|J| \geq n - \gamma$ such records, choose $\mathsf{pw}'_t \xleftarrow{\$} \{0,1\}$ for $t \in [n] \setminus J$ and send $(\textsc{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}'_1 \| \ldots \| \mathsf{pw}'_n)$ to $\mathcal{F}_{\mathsf{fPAKE}}$.
  - Else // there are $|J| < n-\gamma$ test-pwd records, choose $\boldsymbol{K} \xleftarrow{\$} \mathbb{F}_q^n$. Send $(\textsc{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \perp)$ to $\mathcal{F}^M_{\mathsf{fPAKE}}$.// Interrupt the record.
  - Regardless, wait for $\mathcal{F}_{\mathsf{fPAKE}}$'s response:
    - If $\mathcal{F}_{\mathsf{fPAKE}}$ responds with $(T, \text{“correct guess”})$ or $(T, \text{“wrong guess”})$ check if $|T \cap J| < n-\gamma$. If that is the case, mark the record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ as `accident`. Else retrieve records $\langle \textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk}_j \rangle$ and set $K_t \coloneqq \mathsf{sk}_t$ for all $t \in T \cap J$ and choose $K_t \xleftarrow{\$} \mathbb{F}_q$ for all $t \in [n] \setminus (T \cap J)$. Mark the record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ as `compromised`.
    - Else // $\mathcal{F}_{\mathsf{fPAKE}}$ responds with “wrong guess” choose $\boldsymbol{K} \xleftarrow{\$} \mathbb{F}_q^n$. Mark the record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ as `interrupted`.
- Regardless,
  - If $\mathsf{role} = \mathsf{receiver}$ then append $\boldsymbol{K}$ to the record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ and mark the record as `waiting`.
  - Else // role = sender
    - Choose $\boldsymbol{s} \xleftarrow{\$} \mathbb{F}_q^k$. Set $\boldsymbol{C} \coloneqq \mathsf{Enc}(\boldsymbol{s}), \boldsymbol{E} \coloneqq \boldsymbol{C} \oplus \boldsymbol{K}$, and $h \coloneqq H_0((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C})$.
    - If the record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ is marked as `accident`, then choose $\widehat{\mathsf{sk}} \xleftarrow{\$} \{0,1\}^\lambda$.
    - Else if the record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ is marked as `compromised` then set $\widehat{\mathsf{sk}} \coloneqq H_1((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0)$.
    - Else // record interrupted, or fresh set $\widehat{\mathsf{sk}} \coloneqq \perp$.
    - Regardless, send $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \widehat{\mathsf{sk}})$ to $\mathcal{F}_{\mathsf{fPAKE}}$. Store $\langle \mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, (\boldsymbol{E}, h) \rangle$ and send $(\mathcal{P}_i, \mathcal{P}_{1-i}, (\boldsymbol{E}, h))$ to $\mathcal{Z}$ as message from $\mathcal{F}_{\mathsf{SA}}$. Mark $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} \rangle$ as `completed`.

Fig. 10. The second part of the simulator $\mathcal{S}$ for $\mathcal{F}_{\mathsf{fPAKE}}$ and our Fuzzy PAKE with list-decoding protocol. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{SA}}$ and the random oracles H_0 and H_1. If we write “retrieve record” and the record does not exist, $\mathcal{S}$ ignores the message.

and adapt interfaces of the ideal functionality until we reach $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ and to make the simulator independent of the input passwords $\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}$ until we reach the simulator as described in Figs. 9 to 10.

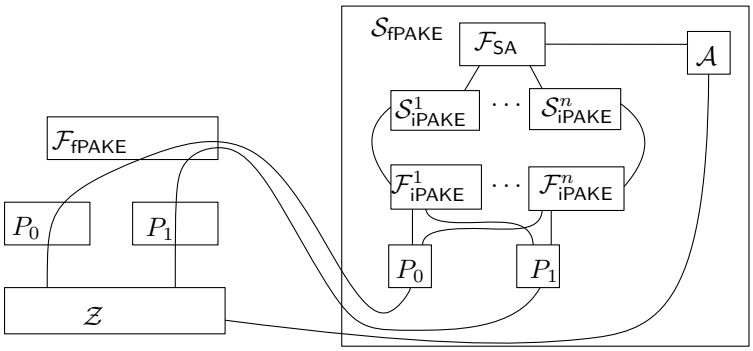


Fig. 11. This figure shows the layout of our simulator $\mathcal{S}_{\mathsf{fPAKE}}$ in $\mathbf{G}_3$ in a setting without MitM attack.

- $\mathbf{G}_2$: In this game the simulator aborts on collisions of H_0.
- $\mathbf{G}_3$: In this game we use the hypothesis that the iPAKE protocol π_j UC-realizes $\mathcal{F}_{\mathsf{iPAKE}}$. There is a simulator $\mathcal{S}_{\mathsf{iPAKE}}$ that interacts with $\mathcal{F}_{\mathsf{iPAKE}}$ and simulates a protocol execution of π. Thus, we can replace each instance π_j of π by an instance $\mathcal{S}^j_{\mathsf{iPAKE}}$ of $\mathcal{S}_{\mathsf{iPAKE}}$. The simulator $\mathcal{S}$ must also simulate $\mathcal{F}_{\mathsf{iPAKE}}$ towards $\mathcal{S}_{\mathsf{iPAKE}}$ to make sure that the simulation works. Note that $\mathcal{S}$ still needs the input passwords $\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}$ to internally run $\mathcal{F}_{\mathsf{iPAKE}}$.
- $\mathbf{G}_4$: In this game we equip $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ with the TestPwd interface. Also the NewKey and TestPwd interfaces mark records. Also the NewKey interface reacts to the marked records. However, $\mathcal{S}$ does not use the TestPwd interfaces yet.
- $\mathbf{G}_5$: $\mathcal{S}_{\mathsf{iPAKE}}$ might send TestPwd queries to $\mathcal{F}_{\mathsf{iPAKE}}$. $\mathcal{S}$ still uses the input passwords to answer these. The games $\mathbf{G}_5$ to $\mathbf{G}_8$ will changes this. In $\mathbf{G}_5$ we argue that $\mathcal{S}_{\mathsf{iPAKE}}$ will never send a TestPwd query to $\mathcal{F}_{\mathsf{iPAKE}}$ if both parties are honest and no man-in-the-middle attack is mounted. So in that case $\mathcal{S}$ can ignore TestPwd queries of $\mathcal{S}_{\mathsf{iPAKE}}$.
- $\mathbf{G}_6$: In this game we start dealing with attacks on an honest sender. Such an attack will lead to TestPwd queries from the $\mathcal{S}_{\mathsf{iPAKE}}$. $\mathcal{S}$ can also send a TestPwd query to $\mathcal{F}^{M}_{\mathsf{fPAKE}}$. We leverage the fact that communication runs over the split authentication functionality. Because of $\mathcal{F}_{\mathsf{SA}}$ the $\mathcal{F}_{\mathsf{iPAKE}}$ instances of the sender run in a different authentication set than the iPAKE instances of the (malicious) receiver. Thus, $\mathcal{F}_{\mathsf{iPAKE}}$ will never have a successful key exchange. Either the key is adversarially chosen or it is random. This is what mitigates the attack from Section 3.2.
  The general idea in this game is that $\mathcal{S}$ collects the TestPwd queries of the iPAKE simulators, concatenates the passwords and uses this to ask a

TestPwd query to $\mathcal{F}_{\mathsf{fPAKE}}$. If the password was close enough (i.e. within γ distance), then $\mathcal{S}$ receives the position of the correct password characters as leakage from $\mathcal{F}^M_{\mathsf{fPAKE}}$. Our simulator then uses this knowledge to simulate the $\mathcal{F}_{\mathsf{iPAKE}}$ to $\mathcal{Z}$. If $\mathcal{S}$'s password guess was too far away, it will not receive any leakage information from $\mathcal{F}^M_{\mathsf{fPAKE}}$. In that case $\mathcal{S}$ still uses the input passwords $\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}$ to produce output keys K_j for the iPAKE sessions.

The only difference to $\mathbf{G}_5$ is that in the case of an attack on an honest sender, in $\mathbf{G}_6$ the sender gets a random key as output from the functionality and not from the simulator anymore. However, by the $k-1$ query uniformity of the code we can argue that the output was already independent and uniformly random in the previous game.

- $\mathbf{G}_7$: In this game we change how $\mathcal{S}$ simulates iPAKE keys of an honest sender when there is a MitM attack or the receiver is corrupt. In the case that the adversary attacked less than $n-\gamma$ iPAKE sessions or guessed less than $n-\gamma$ characters of the password correctly, $\mathcal{S}$ just uses uniformly random iPAKE keys K_j for all $j \in [n]$. So even for the few successfully attacked iPAKE sessions the output key will be random. Again, we can use $(k-1)$-query uniformity to argue that the few correct entries of $\boldsymbol{C}$ that an adversary obtains already looked uniformly random before.
- $\mathbf{G}_8$: In this game, we deal with attacks on honest receivers. Like in the previous game $\mathcal{S}$ can use split authentication and the TestPwd interface of $\mathcal{F}^M_{\mathsf{fPAKE}}$ with leakage to deal with $n-\gamma$ or more attacked iPAKE sessions. In the case where less than $n-\gamma$ iPAKE sessions were attacked, $\mathcal{S}$ does not try to decode but instead interrupts the record by sending a TestPwd query to $\mathcal{F}^M_{\mathsf{fPAKE}}$ with $\mathsf{pw} = \perp$. Then it sends a NewKey query with $\mathsf{sk} = \perp$ to $\mathcal{F}^M_{\mathsf{fPAKE}}$, which causes the output of the receiver to be a independent, uniformly random value. One can use Lemma 4 to see that in the case of less than $n-\gamma$ password characters the LDec algorithm will not output anything that matches the hint h and, thus, already in the previous game the receiver's output was an independent uniform random value.
- $\mathbf{G}_9$: This game syntactically changes $\mathcal{S}$ to remove usage of the provided input passwords $\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}$ in dealing with NewSession and TestPwd queries. We made sure in the previous games that $\mathcal{S}$ does not need $\mathsf{pw}^{\mathcal{P}_i}$ and $\mathsf{pw}^{\mathcal{P}_{1-i}}$ anymore for those queries.
- $\mathbf{G}_{10}$: In this game $\mathcal{S}$ is modified such that honest and undisturbed parties produce output without $\mathcal{S}$ using the provided inputs for the parties. For this, $\mathcal{S}$ can use the NewKey interface of $\mathcal{F}^M_{\mathsf{fPAKE}}$. The functionality $\mathcal{F}^M_{\mathsf{fPAKE}}$ will produce output according to the input passwords. If $d(\mathsf{pw}, \mathsf{pw}') \leq \delta$ then $\mathcal{F}^M_{\mathsf{fPAKE}}$ produces matching passwords. Because of Lemma 6 the real-world protocol does the same. If $d(\mathsf{pw}, \mathsf{pw}') > \delta$ then $\mathcal{F}^M_{\mathsf{fPAKE}}$ produces independent keys. By Lemma 5 the real-world protocol does the same. We also replace the message $(\boldsymbol{E}, h)$ by a uniformly random message, which is indistinguishable for $\mathcal{Z}$ as the iPAKE keys of honest parties are hidden from $\mathcal{Z}$.
- $\mathbf{G}_{11}$: This is the ideal-world experiment with $\mathcal{F}^M_{\mathsf{fPAKE}}$ and the simulator from Figs. 9 to 10.

The following lemma essentially says that given a target vector $\boldsymbol{C}$, a vector $\tilde{\boldsymbol{C}}$ with at least $\gamma + 1 = n - k + 1$ random entries will have $d(\boldsymbol{C}, \tilde{\boldsymbol{C}}) \geq \gamma + 1$. This is necessary in our protocol to make sure that a malicious sender with a far away password cannot make the key exchange succeed. In the security proof of our protocol we use this lemma in $\mathbf{G}_8$.

Lemma 4. *For* $q = 2^{\Omega(\lambda)}, \forall n, k \in \mathbb{N}$ *with* $n \geq k$: $\forall A \subseteq [n]$ *with* $|A| \leq k - 1$: $\forall \boldsymbol{C}, \tilde{\boldsymbol{C}} \in \mathbb{F}_q^n$ *with* $\tilde{\boldsymbol{C}}|_{\bar{A}} \xleftarrow{\$} \mathbb{F}_q^{n-|A|}$:

$$\Pr[d(\boldsymbol{C}, \tilde{\boldsymbol{C}}) \geq n - k + 1] \geq 1 - \mathsf{negl}(\lambda).$$

The probability is taken over the randomness used to sample $\tilde{\boldsymbol{C}}|_{\bar{A}}$.

Proof. Remember that $\gamma = n - k$. Observe that $d(\boldsymbol{C}, \tilde{\boldsymbol{C}}) = d(\boldsymbol{C}|_A, \tilde{\boldsymbol{C}}|_A) + d(\boldsymbol{C}|_{\bar{A}}, \tilde{\boldsymbol{C}}|_{\bar{A}})$. The statement must even hold for $d(\boldsymbol{C}|_A, \tilde{\boldsymbol{C}}|_A) = 0$ and A maximally large, i.e., $|A| = k - 1$ and, thus, $|\bar{A}| = n - k + 1 = \gamma + 1$. What remains to be shown is that $d(\boldsymbol{C}|_{\bar{A}}, \tilde{\boldsymbol{C}}|_{\bar{A}}) \geq n - k + 1 = \gamma + 1$ with overwhelming probability. For $d(\boldsymbol{C}|_{\bar{A}}, \tilde{\boldsymbol{C}}|_{\bar{A}}) \geq \gamma + 1$ to hold, $\boldsymbol{C}|_{\bar{A}}$ and $\tilde{\boldsymbol{C}}|_{\bar{A}}$ must be distinct in all $\gamma + 1$ coordinates. The probability that $\boldsymbol{C}|_{\bar{A}}$ and $\tilde{\boldsymbol{C}}|_{\bar{A}}$ are distinct in a specific coordinate is $1 - \frac{1}{q}$. Thus, the probability that $\boldsymbol{C}|_{\bar{A}}$ and $\tilde{\boldsymbol{C}}|_{\bar{A}}$ are distinct in all $\gamma + 1$ coordinates is $(1 - \frac{1}{q})^{\gamma+1}$. For constant γ and $q = 2^\lambda$ this probability is overwhelming in λ.

The next lemma is similar to Lemma 4. One important difference is that in Lemma 4 $\boldsymbol{C}$ was an arbitrary vector in $\mathbb{F}_q^n$, whereas in Lemma 5 $\boldsymbol{C}$ is a valid codeword. We use this lemma to guarantee that in our protocol in a session with an honest sender and an honest receiver, whose passwords missmatch in more than δ positions, both parties get independent random keys. In particular this lemma guarantees that the receiver will not be able to decode the codeword, which will make them output a random key. This is important in the transition from $\mathbf{G}_9$ to $\mathbf{G}_{10}$.

Lemma 5. *For* $s \xleftarrow{\$} \mathbb{F}_q$, $\boldsymbol{C} \leftarrow \mathsf{Enc}(s)$, $\tilde{\boldsymbol{C}} \in \mathbb{F}_q^n$ *and* $A \subseteq [n]$ *such that* $|A| \leq k-1$, $\boldsymbol{C}|_A = \tilde{\boldsymbol{C}}|_A$, *and* $\tilde{\boldsymbol{C}}|_{\bar{A}} \xleftarrow{\$} \mathbb{F}_q^{n-|A|}$, *we have that* $\boldsymbol{C} \in \mathsf{LDec}(\tilde{\boldsymbol{C}})$ *with negligible probability.*

Proof. From Lemma 4 we get that $d(\boldsymbol{C}, \tilde{\boldsymbol{C}}) \geq n - k + 1$ with overwhelming probability. Because $n - k + 1 > n - 1 - \sqrt{(k-1)n} = \delta$ and $\mathsf{LDec}(\tilde{\boldsymbol{C}})$ does not output candidates $\boldsymbol{C}$ with $d(\boldsymbol{C}, \tilde{\boldsymbol{C}}) > \delta$ by Lemma 2, we get that $\boldsymbol{C} \in \mathsf{LDec}(\tilde{\boldsymbol{C}})$ with negligible probability.

Lemma 6 can be seen as the counterpart of Lemma 5, as we use it to guarantee that in our protocol two honest parties that have close enough passwords (i.e. $\leq \delta$) get the same session key. In detail, this lemma guarantees that the honest receiver is able to correctly decode $\boldsymbol{C}' = \boldsymbol{E} \oplus \boldsymbol{K}'$, which then leads to successful key exchange. We use this lemma in the transition from $\mathbf{G}_9$ to $\mathbf{G}_{10}$.

Lemma 6. *For* $s \xleftarrow{\$} \mathbb{F}_q$, $\boldsymbol{C} \leftarrow \mathsf{Enc}(s)$, $\tilde{\boldsymbol{C}} \in \mathbb{F}_q^n$ *and* $A \subseteq [n]$ *such that* $|A| \geq n-\delta$, $\boldsymbol{C}|_A = \tilde{\boldsymbol{C}}|_A$, *and* $\tilde{\boldsymbol{C}}|_{\bar{A}} \xleftarrow{\$} \mathbb{F}_q^{n-|A|}$, *we have that* $\boldsymbol{C} \in \mathsf{LDec}(\tilde{\boldsymbol{C}})$ *with overwhelming probability.*

Proof. This follows directly from Lemma 2 with $\delta = e = n - 1 - \lceil\sqrt{(k-1)n}\rceil$.

References

ABM+21. Shashank Agrawal, Saikrishna Badrinarayanan, Payman Mohassel, Pratyay Mukherjee, and Sikhar Patranabis. BETA: Biometric-enabled threshold authentication. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 290–318. Springer, Heidelberg, May 2021.

ABMR20. Shashank Agrawal, Saikrishna Badrinarayanan, Pratyay Mukherjee, and Peter Rindal. Game-set-MATCH: Using mobile devices for seamless external-facing biometric matching. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1351–1370. ACM Press, November 2020.

All22. WiFi Alliance. Wpa3 specification version 3.1. https://www.wi-fi.org/download.php?file=/sites/default/files/private/WPA32022.

BBCW21. Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. Provable security analysis of FIDO2. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 125–156, Virtual Event, August 2021. Springer, Heidelberg.

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

BCL+05. Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 361–377. Springer, Heidelberg, August 2005.

BCL22. Jonathan Bootle, Alessandro Chiesa, and Siqi Liu. Zero-knowledge iops with linear-time prover and polylogarithmic-time verifier. In *Proceedings of the 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '22, pages 275–304, 2022.

BDFK12. Jens Bender, Özgür Dagdelen, Marc Fischlin, and Dennis Kügler. The pace|aa protocol for machine readable travel documents, and its security. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*, volume 7397 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 2012.

BDK+05. Xavier Boyen, Yevgeniy Dodis, Jonathan Katz, Rafail Ostrovsky, and Adam Smith. Secure remote authentication using biometric data. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 147–163. Springer, Heidelberg, May 2005.

BFK09. Jens Bender, Marc Fischlin, and Dennis Kügler. Security analysis of the PACE key-agreement protocol. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio A. Ardagna, editors, *Information Security, 12th International Conference, ISC 2009, Pisa, Italy, September 7-9, 2009. Proceedings*, volume 5735 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2009.

BFK13. Jens Bender, Marc Fischlin, and Dennis Kügler. The pace|ca protocol for machine readable travel documents. In Roderick Bloem and Peter Lipp, editors, *Trusted Systems - 5th International Conference, INTRUST 2013, Graz, Austria, December 4-5, 2013, Proceedings*, volume 8292 of *Lecture Notes in Computer Science*, pages 17–35. Springer, 2013.

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.

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.

CCG+07. Hao Chen, Ronald Cramer, Shafi Goldwasser, Robbert de Haan, and Vinod Vaikuntanathan. Secure computation from random error correcting codes. In *Proceedings of the 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT' 07, pages 291–310, 2007.

CDBN15. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

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.

CFP+16. Ran Canetti, Benjamin Fuller, Omer Paneth, Leonid Reyzin, and Adam D. Smith. Reusable fuzzy extractors for low-entropy distributions. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 117–146. Springer, Heidelberg, May 2016.

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.

CNPR22. Cas Cremers, Moni Naor, Shahar Paz, and Eyal Ronen. CHIP and CRISP: Protecting all parties against compromise through identity-binding PAKEs. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 668–698. Springer, Heidelberg, August 2022.

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+17. Pierre-Alain Dupont, Julia Hesse, David Pointcheval, Leonid Reyzin, and Sophia Yakoubov. Fuzzy password-authenticated key exchange. Cryptology ePrint Archive, Paper 2017/1111, 2017. `https://eprint.iacr.org/2017/1111`.

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.

DORS08. Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam D. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM J. Comput.*, 38(1):97–139, 2008.

EHOR20. Andreas Erwig, Julia Hesse, Maximilian Orlt, and Siavash Riahi. Fuzzy asymmetric password-authenticated key exchange. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 761–784. Springer, Heidelberg, December 2020.

Gur06. Venkatesan Guruswami. Algorithmic results in list decoding. *Found. Trends Theor. Comput. Sci.*, 2(2), 2006.

HL19. Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR TCHES*, 2019(2):1–48, 2019. `https://tches.iacr.org/index.php/TCHES/article/view/7384`.

HS14. Feng Hao and Siamak Fayyaz Shahandashti. The SPEKE protocol revisited. *IACR Cryptol. ePrint Arch.*, page 585, 2014.

Hv22. Feng Hao and Paul C. van Oorschot. SoK: Password-authenticated key exchange - theory, practice, standardization and real-world lessons. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22*, pages 697–711. ACM Press, May / June 2022.

Jab96. David P. Jablon. Strong password-only authenticated key exchange. *Comput. Commun. Rev.*, 26(5):5–26, 1996.

JLHG22. Mingming Jiang, Shengli Liu, Shuai Han, and Dawu Gu. Fuzzy authenticated key exchange with tight security. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *ESORICS 2022, Part II*, volume 13555 of *LNCS*, pages 337–360. Springer, Heidelberg, September 2022.

KR07. Michel Kulhandjian and Atri Rudra. Lecture 27: Berlekamp–welch algorithm, 2007.

Mac01. Philip MacKenzie. On the security of the SPEKE password-authenticated key exchange protocol. *IACR Cryptol. ePrint Arch.*, page 57, 2001.

McE03. R J McEliece. The guruswami—sudan decoding algorithm for reed–solomon codes. IPN Progress Report 42-153, 2003.

PC20. Thitikorn Pongmorrakot and Rahul Chatterjee. tpake: Typo-tolerant password-authenticated key exchange. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering - 10th International Conference, SPACE 2020, Kolkata, India, December 17-21, 2020, Proceedings*, volume 12586 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2020.

RW04. Renato Renner and Stefan Wolf. The exact price for unconditionally secure asymmetric cryptography. In Christian Cachin and Jan Camenisch, edi-

tors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 109–125. Springer, Heidelberg, May 2004.

RX23. Lawrence Roy and Jiayu Xu. A universally composable PAKE with zero communication cost - (and why it shouldn't be considered uc-secure). In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *Public-Key Cryptography - PKC 2023 - 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, May 7-10, 2023, Proceedings, Part I*, volume 13940 of *Lecture Notes in Computer Science*, pages 714–743. Springer, 2023.

Wei16. Mor Weiss. Secure computation and probabilistic checking, 2016.

WHC+21. Mei Wang, Kun He, Jing Chen, Zengpeng Li, Wei Zhao, and Ruiying Du. Biometrics-authenticated key exchange for secure messaging. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2618–2631. ACM Press, November 2021.

Supplementary Material

A Full proof

In the following, we discuss in detail why the game hops presented in Section 5 show that no PPT environment $\mathcal{Z}$ has noticeable advantage in distinguishing a real execution of the protocol from Figure 8 from an ideal execution, with $\mathcal{F}_{\mathsf{fPAKE}}^{M}$ and the simulator from Figs. 9 to 10.

Proof. The first game, denoted as $\mathbf{G}_0$ is the real execution. We gradually change the experiment until we reach the ideal execution, denoted as $\mathbf{G}_{11}$. In the following, we introduce the games in between and argue directly why each game is indistinguishable from its predecessor. We write $\Pr[\mathbf{G}_i]$ as a shorthand for the probability that the environment outputs 1 in game $\mathbf{G}_i$, where the probability is taken over the random coins of the environment and all parties.

Game $\mathbf{G}_0$: The real execution.

Game $\mathbf{G}_1$: Create functionality and simulator. In this step, we introduce a simulator to the experiment. We move the whole execution of the real protocol to the simulator. Concretely, this means that the simulator runs all parties of the protocol with their real code. To do this the simulator needs the inputs of the parties (which the simulator won't get any more in later games). We therefore introduce dummy parties and an ideal functionality that merely forwards all messages, i.e., a dummy party forwards its input $(\textsc{NewSession}, \mathsf{sid}, \mathsf{pw}^{\mathcal{P}_i}, \mathsf{role})$ to the functionality and the functionality forwards $(\textsc{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i}, \mathsf{role})$ to the simulator (without changing or suppressing $\mathsf{pw}^{\mathcal{P}_i}$). Finally, we equip the functionality with interfaces $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ that allow the simulator to make any dummy parties output the key sk that the simulator wants.
Note that the internal parties inside of the simulator now execute the protocol exactly as in the real world and also have the same output. We just

changed where the parties execute their code. This is merely a syntactical change. Thus, both hybrids are identically distributed, i.e.,

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_0].$$

Game $\mathbf{G}_2$: Abort on collisions of H_0. In this game, we change the simulator such that it aborts the simulation whenever there are two inputs $\boldsymbol{C}, \boldsymbol{C}'$ with $\boldsymbol{C} \neq \boldsymbol{C}'$ such that $H_0((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C}) = H((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C}')$. As the output for each new hash query is chosen uniformly at random, two values collide with probability $1/2^{\lambda}$. When there are η queries to H_0 we get

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq \binom{\eta}{2} 2^{-\lambda}.$$

Game $\mathbf{G}_3$: Exchanging real iPAKE messages by simulated ones. In this game, we only change the simulator. We introduce a variable MitM, which is set to 1, if there is a man-in-the-middle attack and to 0 otherwise. More precisely, on a message $(\textsc{Init}, \mathsf{sid}, \mathcal{P}_i, \mathbb{H}, \mathsf{sid}_{\mathbb{H}})$ from $\mathcal{Z}$ to the simulated $\mathcal{F}_{\mathsf{SA}}$, $\mathcal{S}$ sets $\mathsf{MitM} = 0$, if $\mathbb{H} = \{\mathcal{P}_0, \mathcal{P}_1\}$ and $\mathsf{MitM} = 1$, otherwise. From now on, the simulator runs one internal instance of $\mathcal{F}_{\mathsf{iPAKE}}$, presented in Figure 12, for each distinct session of π. Whenever a party $\mathcal{P}_i$ in our protocol starts a new session of π on a new sid $(\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j)$ with $\mathsf{pw}_j^{\mathcal{P}_i}$, $\mathcal{S}$ creates an instance of $\mathcal{F}_{\mathsf{iPAKE}}$ with the same sid and gives it the input $(\textsc{NewSession}, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j), \mathsf{pw}_j, \mathsf{role})$ in the name of $\mathcal{P}_i$. $\mathcal{S}$ also invokes one iPAKE simulator $\mathcal{S}_{\mathsf{iPAKE}}$ for each instance of $\mathcal{F}_{\mathsf{iPAKE}}$, which is guaranteed to exist, because π UC-realizes $\mathcal{F}_{\mathsf{iPAKE}}$. The situation is depicted in Fig. 11. If the environment is not doing a MitM attack on the communication, i.e., $\mathsf{sid}_{\mathbb{H}}$ is the same for both parties, then there will be n instances of $\mathcal{F}_{\mathsf{iPAKE}}$. If the environment is doing a MitM attack, i.e., $\mathsf{sid}_{\mathbb{H}}$ is different for $\mathcal{P}_0$ and $\mathcal{P}_1$, then there will be $2n$ instances of $\mathcal{F}_{\mathsf{iPAKE}}$, because then each party has their own n $\mathcal{F}_{\mathsf{iPAKE}}$ instances that run on different session-ids than the other party's instances. $\mathcal{S}$ can simply execute the code of $\mathcal{F}_{\mathsf{iPAKE}}$, as in this game $\mathcal{S}$ still knows the parties' passwords and, thus, all inputs to the $\mathcal{F}_{\mathsf{iPAKE}}$ instances. When an instance of $\mathcal{F}_{\mathsf{iPAKE}}$ outputs $((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j), \mathsf{sk})$ to $\mathcal{P}_i$ as response to a $\textsc{NewKey}$ query of the corresponding $\mathcal{S}_{\mathsf{iPAKE}}$, $\mathcal{S}$ gives this to $\mathcal{P}_i$ as if coming from π_j. Whenever all n $\mathcal{S}_{\mathsf{iPAKE}}$ belonging to $\mathcal{P}_i$ produce a message Z_j^* for $\mathcal{Z}$, simulating that $\mathcal{P}_i$ sent this message, $\mathcal{S}$ gives $(Z_1^*, \ldots, Z_n^*)$ to $\mathcal{F}_{\mathsf{SA}}$ as message from $\mathcal{P}_i$. This is the same as $\mathcal{P}_i$ would have done when all n π_j had produced a message Z_j^*. Finally, whenever $\mathcal{F}_{\mathsf{SA}}$ delivers a message $(Z_1^*, \ldots, Z_n^*)$ to $\mathcal{P}_i$ (upon instruction from $\mathcal{Z}$), $\mathcal{S}$ gives the instruction to deliver Z_j^* to $\mathcal{S}_{\mathsf{iPAKE}}^j$ in the name of the environment.

To show that $\mathbf{G}_3 \overset{c}{\approx} \mathbf{G}_2$ we use a hybrid argument. In the j-th step of the hybrid argument, we exchange the j-th instance π_j of π by one instance of the ideal execution $\mathcal{F}_{\mathsf{iPAKE}}^j$ and $\mathcal{S}_{\mathsf{iPAKE}}^j$. After at most $2n$ steps (or n steps if there is no man-in-the-middle attack) we have exchanged all instances of π and reached $\mathbf{G}_3$.

To see that an individual hybrid step is valid, observe that $\mathcal{S}$ perfectly emulates the internal $\mathcal{F}_{\mathsf{iPAKE}}$ instances. Furthermore, because π UC-realizes $\mathcal{F}_{\mathsf{iPAKE}}$, the outputs of the simulators $\mathcal{S}_{\mathsf{iPAKE}}$ are indistinguishable from the messages of π.
In a bit more detail, if there is an index l, such that an environment $\mathcal{Z}_l$ can distinguish between hybrids l and $l+1$, then we can use $\mathcal{Z}_l$ to build an environment $\mathcal{Z}^*$ that can distinguish between an execution of π and an execution of the ideal protocol with $\mathcal{F}_{\mathsf{iPAKE}}$. Let $\mathcal{S}_l$ be the simulator in $\mathbf{G}_3$ at hybrid step l. The environment $\mathcal{Z}^*$ runs $\mathcal{Z}_l$ and $\mathcal{S}_l$ and whenever a party $\mathcal{P}_i$ starts a session of π_{l+1}, $\mathcal{Z}^*$ gives the input to their test-session, which is either the real version of π, or the ideal version with $\mathcal{F}_{\mathsf{iPAKE}}$. If $\mathcal{Z}_l$ guesses that it is in hybrid l, $\mathcal{Z}^*$ outputs that it is in the real world, and if $\mathcal{Z}_l$ guesses that it is in hybrid $l+1$ then $\mathcal{Z}^*$ outputs that it is in the ideal world. $\mathcal{Z}^*$ has the same advantage as $\mathcal{Z}_l$.
Thus, we have that

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq 2n\mathbf{Dist}^{\pi,\{\mathcal{F}_{\mathsf{iPAKE}},\mathcal{S}_{\mathsf{iPAKE}}\}}_{\mathcal{Z}^*}(\lambda),$$

where $\mathbf{Dist}^{\pi,\{\mathcal{F}_{\mathsf{iPAKE}},\mathcal{S}_{\mathsf{iPAKE}}\}}_{\mathcal{Z}^*}(\lambda)$ denotes the distinguishing advantage that a PPT environment $\mathcal{Z}^*$ has in distinguishing a real-world execution of π from an ideal-world execution with $\mathcal{F}_{\mathsf{iPAKE}}$ and $\mathcal{S}_{\mathsf{iPAKE}}$.

Game $\mathbf{G}_4$: Functionality makes records of running sessions, marks them, and reacts accordingly. In this game, we only change the ideal functionality. Now, whenever the functionality receives input $(\textsc{NewSession}, \mathsf{sid}, \mathsf{pw}^{\mathcal{P}_i}, \mathsf{role})$ from the dummy party $\mathcal{P}_i$ it records $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{pw}^{\mathcal{P}_i}\rangle$ if this was the first NewSession query or if it was the second NewSession query and there already is a record $\langle \mathcal{P}_{1-i}, \mathsf{sid}, \mathsf{pw}^{\mathcal{P}_{1-i}}\rangle$. It marks the record as `fresh`. This means that the ideal functionality now has the full NewSession interface of $\mathcal{F}_{\mathsf{fPAKE}}$ with the addition that it still forwards the passwords to the simulator. Next, we add the TestPwd interface as in the full $\mathcal{F}^M_{\mathsf{fPAKE}}$ functionality, thus, records can also be marked `interrupted` or `compromised`. Furthermore, we change the functionality's behavior on $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ queries such that it now checks if a record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{pw}^{\mathcal{P}_i}\rangle$ exists. If so, it outputs a uniformly random key when the corresponding record is marked as `interrupted`. Also, it outputs the adversarially provided key sk when the corresponding record is marked as `compromised` or the other party is corrupted as in $\mathcal{F}_{\mathsf{fPAKE}}$. Note, however, that the functionality in case of `fresh` records still uses the key provided by the simulator. This will change in a later game. Regardless of how the record was marked before, the functionality now marks the record as `completed`.
Note that the records and markings are only internal and their only effect to the outside is through the NewKey interface. In the NewKey interface the functionality still outputs the key chosen by the simulator, except if the corresponding record is `interrupted`. However, so far our simulator never calls $\mathcal{F}_{\mathsf{fPAKE}}$'s TestPwd interface and, therefore, currently a record can never be `interrupted`. Thus, the changes to the functionality are only

syntactical and we have

$$\Pr[\mathbf{G}_4] = \Pr[\mathbf{G}_3].$$

Game $\mathbf{G}_5$: Simulator handles TestPwd queries of $\mathcal{F}^M_{\mathsf{fPAKE}}$ for honest and undisturbed session. In this game, we only modify the simulator. In the case of an honest session, i.e. when there is no MitM attack and no party is corrupted, $\mathcal{S}$ does not use the passwords of the parties to simulate TestPwd of $\mathcal{S}_{\mathsf{iPAKE}}$ queries anymore. Whenever $\mathcal{S}$ receives a TestPwd query from $\mathcal{S}_{\mathsf{iPAKE}}$ for a session with two honest parties and where $\mathsf{MitM} = 0$, $\mathcal{S}$ just ignores the query.

This means, for hones sessions, $\mathcal{S}$ can simulate the NewSession and the TestPwd interfaces of $\mathcal{F}_{\mathsf{iPAKE}}$ to $\mathcal{S}_{\mathsf{iPAKE}}$ without using the password $\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}$ of the parties. However, $\mathcal{S}$ still uses $\mathsf{pw}^{\mathcal{P}_i}$ and $\mathsf{pw}^{\mathcal{P}_{1-i}}$ to simulate $\mathcal{F}_{\mathsf{iPAKE}}$'s reaction on NewKey queries from $\mathcal{S}_{\mathsf{iPAKE}}$.

Below we argue why this does not change the distribution of the game. The only way that ignoring TestPwd queries from the $\mathcal{S}_{\mathsf{iPAKE}}$ can change the distribution of the game is when a $\mathcal{S}_{\mathsf{iPAKE}}$ asks a TestPwd query. However, we only changed the simulator for the case of an honest session and in that case, both parties receive the same $\mathsf{sid}_{\mathbb{H}'} = \mathsf{sid}_{\mathbb{H}}$ from $\mathcal{F}_{\mathsf{SA}}$ TBD: There is something wrong.]and $\mathcal{F}_{\mathsf{SA}}$ behave like the normal authenticated channel functionality. Thus, the adversary can either let the session proceed without inserting or changing messages, or it can completely prevent the iPAKE sessions from finishing. Intuitively, in an honest and undisturbed iPAKE session, $\mathcal{S}_{\mathsf{iPAKE}}$ cannot ask a TestPwd query, because $\mathcal{S}_{\mathsf{iPAKE}}$ does not know the password and, thus, this would most likely interrupt the session. Then both parties would get distinct keys, even when they used the same passwords. This should not happen in an honest session.

More formally we argue as follows. As we assumed that the iPAKE protocol π is *correct* (see Definition 1) it follows that the $\mathcal{S}_{\mathsf{iPAKE}}$ simulators will not send a TestPwd query to $\mathcal{S}$ as both parties are honest and the adversary only eavesdrops on the session. (In the terminology of [RX23], $\mathcal{S}_{\mathsf{iPAKE}}$ is a *perfectly reasonable simulator*. This follows from the fact that π is correct and UC-realizes $\mathcal{F}_{\mathsf{iPAKE}}$, see [RX23, Thm. 1].) Thus, we get

$$\Pr[\mathbf{G}_5] = \Pr[\mathbf{G}_4].$$

Game $\mathbf{G}_6$: Simulator handles attacks on honest senders. Previously, we argued that $\mathcal{S}$ does not need to send a TestPwd query to $\mathcal{F}^M_{\mathsf{fPAKE}}$ if there is no man-in-the-middle attack and both parties are honest to deal with TestPwd of $\mathcal{S}_{\mathsf{iPAKE}}$s. In this game, we change the simulator's behavior when there is a man-in-the-middle attack against an honest sender or when the receiver is corrupted.

The high-level idea in this game is that $\mathcal{S}$ collects the TestPwd queries from the iPAKE simulators, concatenates them, and then asks one TestPwd query to $\mathcal{F}^M_{\mathsf{fPAKE}}$. If the guess was close enough, $\mathcal{S}$ learns the position of the correct password characters. Then $\mathcal{S}$ marks the record of the (internal) $\mathcal{F}_{\mathsf{iPAKE}}$ instance as `compromised` if the corresponding character was correct,

or as `interrupted` if that character was wrong. If the guess of $\mathcal{S}$ was too far away, $\mathcal{S}$ still uses the password of the sender to check which of the characters were correct and then sets the $\mathcal{F}_{\mathsf{iPAKE}}$ entry accordingly.

Next, we describe the changes to $\mathcal{S}$ in more detail. When $\mathcal{S}$ receives a $(\textsc{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i}, \mathsf{role})$ message from $\mathcal{F}_{\mathsf{fPAKE}}$ then $\mathcal{S}$ records $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role} \rangle$. In the following, we only change $\mathcal{S}$'s behavior if $\mathcal{S}$ recorded $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{sender} \rangle$, i.e., if $\mathcal{P}_i$ is a sender. On getting a $(\textsc{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}'_j)$ query from $\mathcal{S}^j_{\mathsf{iPAKE}}$, $\mathcal{S}$ stores the password guess and does not do anything else, because $\mathcal{F}^j_{\mathsf{iPAKE}}$ would also not answer to the query. (Remember that $\mathcal{F}_{\mathsf{iPAKE}}$ models implicit-only PAKE.) When getting a $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ query from $\mathcal{S}^j_{\mathsf{iPAKE}}$, $\mathcal{S}$ stores sk. As soon as $\mathcal{S}$ receives a NewKey query from all n $\mathcal{S}_{\mathsf{iPAKE}}$s, $\mathcal{S}$ does the following: Let m be the number of iPAKE TestPwd queries for $\mathcal{P}_i$ that $\mathcal{S}$ received, so $m \leq n$. We have $m < n$ when there are some simulators $\mathcal{S}^j_{\mathsf{iPAKE}}$ that give input NewKey to $\mathcal{S}$ without first giving input TestPwd to $\mathcal{S}$. Let δ and γ be as in Figure 4.

- If $m = n$, then $\mathcal{S}$ concatenates all the iPAKE password guesses and asks a TestPwd query to $\mathcal{F}^M_{\mathsf{fPAKE}}$. Concretely, after receiving $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk}_j)$ from $\mathcal{S}^j_{\mathsf{iPAKE}}$, for all $j \in [n]$, $\mathcal{S}$ retreives the stored pw'_j and sends $(\textsc{TestPwd}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}'_1 \| \ldots \| \mathsf{pw}'_n)$ to $\mathcal{F}^M_{\mathsf{fPAKE}}$. If the guess was correct, i.e., at least $n - \delta$ correct password characters, then $\mathcal{S}$ continues the simulation of $\mathcal{P}_i$ with the sk_j from the NewKey queries to $\mathcal{F}^j_{\mathsf{iPAKE}}$, where the corresponding password character pw_j was correct. For the incorrect pw_j, $\mathcal{S}$ uses random iPAKE keys, which is the same as $\mathcal{F}_{\mathsf{iPAKE}}$ would do. If the guess was wrong, but there were at least $n - \gamma$ correct password characters, $\mathcal{S}$ learns the position of the correct characters from $\mathcal{F}^M_{\mathsf{fPAKE}}$ through the leakage function. $\mathcal{S}$ continues the simulation as above, i.e., with setting the iPAKE keys to sk_j if the j-th password character was correct and to a random value if it was incorrect. If the guess was wrong and $\mathcal{S}$ does not get the position of the correct characters, i.e., less than $n - \gamma$ correct characters, then $\mathcal{S}$ uses the input password of the sender (which in this game it still gets from $\mathcal{F}_{\mathsf{fPAKE}}$) to check for which $j \in [n]$ the TestPwd queries of $\mathcal{S}^j_{\mathsf{iPAKE}}$ to $\mathcal{F}_{\mathsf{iPAKE}}$ was correct. For the wrong positions $\mathcal{S}$ chooses a uniformly random key and for the correct positions j the simulator uses the sk_j provided by $\mathcal{S}_{\mathsf{iPAKE}}$.
- If $m < n - \gamma$, then $\mathcal{S}$ uses the input password $\mathsf{pw}^{\mathcal{P}_i}$ of the honest sender as above to check which TestPwd queries to $\mathcal{F}_{\mathsf{iPAKE}}$ were correct. For the wrong positions $\mathcal{S}$ chooses a uniformly random key and for the correct positions j the simulator uses the sk_j provided by $\mathcal{S}_{\mathsf{iPAKE}}$. Additionally $\mathcal{S}$ sends a TestPwd query for $\mathcal{P}_i$ with $\mathsf{pw} = \bot$ to $\mathcal{F}^M_{\mathsf{fPAKE}}$ in order to make sure that the record is `interrupted`.
- If $n - \gamma \leq m < n$, $\mathcal{S}$ arbitrarily chooses the missing password characters and asks a TestPwd query to $\mathcal{F}^M_{\mathsf{fPAKE}}$. If the guess was wrong and $\mathcal{S}$ does not get leakage from $\mathcal{F}^M_{\mathsf{fPAKE}}$, then $\mathcal{S}$ acts as above and uses the input password $\mathsf{pw}^{\mathcal{P}_i}$ of the honest sender to check which TestPwd queries from $\mathcal{S}_{\mathsf{iPAKE}}$ to $\mathcal{F}_{\mathsf{iPAKE}}$ were correct. For the wrong positions $\mathcal{S}$ chooses a

uniformly random key and for the correct positions j the simulator uses the sk_j provided by $\mathcal{S}_{\mathsf{iPAKE}}$.
Let's now consider the case that $\mathcal{S}$ gets leakage from $\mathcal{F}^M_{\mathsf{fPAKE}}$ (either because the guess was correct, or at most γ characters were wrong). Then $\mathcal{S}$ needs to check if this was only because of the missing password characters that $\mathcal{S}$ arbitrarily guessed, or because $\mathcal{Z}$ did guess enough correct characters. Because $\mathcal{S}$ learned from $\mathcal{F}^M_{\mathsf{fPAKE}}$ the position of the correct characters, $\mathcal{S}$ can check if $\mathcal{Z}$ did indeed guess enough correct characters. If that is the case, then $\mathcal{S}$ continues the simulations with the sk_j from the NewKey queries to $\mathcal{F}_{\mathsf{iPAKE}}$ for the correct positions and with random keys for the incorrect positions. Otherwise (if $\mathcal{Z}$ did not guess enough correct characters), $\mathcal{S}$ uses the sender's password to determine the position of the correct characters and then sets the iPAKE keys as in a case as above. The only issue is in the case that $\mathcal{Z}$ guessed less than $n - \gamma$ correct characters, but $\mathcal{S}$'s guesses of the remaining characters were responsible for exceeding $n - \gamma$, then $\mathcal{F}^M_{\mathsf{fPAKE}}$'s record is compromised, although it should be interrupted. $\mathcal{S}$ recognizes this case, through learning the positions of the correct password characters from the leakage function. Thus, $\mathcal{S}$ can remedy the issue by setting a random key in the NewKey query to $\mathcal{F}^M_{\mathsf{fPAKE}}$, thereby imitating a `interrupted` record. Note that $\mathcal{S}$ can do this as the outputs that the internal $\mathcal{F}^j_{\mathsf{iPAKE}}$ functionalities give to parties are not reported to $\mathcal{Z}$ directly but only to $\mathcal{S}$.

In all of the above cases, if the record is `interrupted` ($\mathcal{S}$ recognizes this by getting "wrong guess" and no leakage as an answer to the TestPwd query), $\mathcal{S}$ sets $\mathsf{sk} = \perp$ in the NewKey query to $\mathcal{F}^M_{\mathsf{fPAKE}}$, as $\mathcal{F}^M_{\mathsf{fPAKE}}$ would ignore sk anyways. If the record is `compromised`, $\mathcal{S}$ still sets sk to the key that the simulated $\mathcal{P}_i$ would output.

We argue in the following why the difference between $\mathbf{G}_6$ and $\mathbf{G}_5$ is at most negligible. When there is a man-in-the-middle attack happening or the receiver is corrupted $\mathcal{S}$ has n iPAKE simulators for the $\mathcal{P}_i$ session (and another n iPAKE simulators for the $\mathcal{P}_{1-i}$ session if $\mathcal{P}_{1-i}$ is not corrupted). This is because the adversary plays a man-in-the-middle attack ($\mathcal{S}$ knows this) and therefore $\mathcal{F}_{\mathsf{SA}}$ (played by $\mathcal{S}$) provides different session identifier $\mathsf{sid}_{\mathbb{H}'} \neq \mathsf{sid}_{\mathbb{H}}$ to $\mathcal{P}_i$ and $\mathcal{P}_{i-1}$. Therefore $\mathcal{P}_i$ runs its own instance of $\mathcal{F}_{\mathsf{iPAKE}}$ with $\mathsf{sid}_{\mathbb{H}}$ (actually n of them). This instance is in particular independent of $\mathcal{P}_{1-i}$'s instance $\mathcal{F}'_{\mathsf{iPAKE}}$ that runs with $\mathsf{sid}_{\mathbb{H}'} \neq \mathsf{sid}_{\mathbb{H}}$. Therefore, $\mathcal{F}_{\mathsf{iPAKE}}$ is not in the case where two honest parties execute the protocol. This means, either the adversary compromises the session and makes $\mathcal{P}_i$ output an adversarially chosen sk' or the adversary makes $\mathcal{P}_i$ output a random key either by interrupting the session with a wrong password guess in a TestPwd to $\mathcal{F}_{\mathsf{iPAKE}}$ query or by sending NewKey without a TestPwd query to $\mathcal{F}_{\mathsf{iPAKE}}$. $\mathcal{S}$ would ignore NewKey queries for $\mathcal{P}_{1-i}$, because $\mathcal{F}_{\mathsf{iPAKE}}$ would also ignore them, because $\mathcal{P}_{1-i}$ never sent a NewSession query (for that specific session-id). $\mathcal{S}$ would also ignore TestPwd queries from $\mathcal{S}_{\mathsf{iPAKE}}$ for $\mathcal{P}_{1-i}$ for the same reason. Note that this is what prevents the attack from Section 3.2.

We distinguish two cases depending on how many characters of the tested password were correct. Let us first consider the case where $\mathcal{Z}$ guessed less than $n-\gamma = t$ password characters correctly (this includes the case that $\mathcal{S}$'s guess of the additional characters made the guess appear correct to $\mathcal{F}^{M}_{\mathsf{fPAKE}}$). In this case, $\mathcal{S}$ still uses the honest sender's password and sets the output of the $\mathcal{F}_{\mathsf{iPAKE}}$ instances the same way as in $\mathbf{G}_5$. The only difference to $\mathbf{G}_5$ is that the record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$ of $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ is now interrupted and, thus, the output of the sender is a uniformly random value instead of $H_1((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0)$. Let us explain why this change is not detectable by $\mathcal{Z}$. Because H_1 is a random oracle, $H_1((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0)$ is uniformly random for anybody that did not query H_1 on $((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0)$. The chance of guessing s_0 is negligible. The only other values that could contain information about s_0 are the hint $h = H_0((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C})$ and $\boldsymbol{E} = \boldsymbol{C} \oplus \boldsymbol{K}$. Because $\mathcal{Z}$ guessed at most $n-\gamma-1 = k-1$ password characters correctly, $\mathcal{Z}$ can recover at most $k-1$ coordinates of $\boldsymbol{C}$. By the $k-1$ query uniformity of the code, these (at most) $k-1$ undisturbed coordinates appear uniformly random and, thus, do not reveal any information about s_0. The hint h is also uniformly random unless one has queried $((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C})$ to H_0. Let us now bound the probability that $\mathcal{Z}$ queries $\boldsymbol{C}$ to H_0. Since $\boldsymbol{E} = \boldsymbol{C} \oplus \boldsymbol{K}$ and $\boldsymbol{K}$ is uniformly random (and unknown to $\mathcal{Z}$) in at least $\gamma + 1$ positions, $\mathcal{Z}$ would need to correctly guess the remaining coordinates in $\mathbb{F}_q^{\gamma+1}$. The probability that $\mathcal{Z}$ can do so in polynomial time is negligible.
Let us now consider the case that $\mathcal{Z}$ guessed at least $n-\gamma$ password characters correctly. In this case, $\mathcal{S}$ learned the position of the correct characters via the leakage function and, thus, simulates the $\mathcal{F}_{\mathsf{iPAKE}}$ identically to $\mathbf{G}_5$. Therefore, the session key that $\mathcal{S}$ sends to $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ via the NewKey interface is identically distributed to $\mathbf{G}_5$. The only difference is that in $\mathbf{G}_6$ the corresponding record of $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ is `compromised` instead of `fresh`. However, until now, $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ outputs $\mathcal{S}$'s key in both cases. Therefore, also in the case that $\mathcal{Z}$ guessed at least $n-\gamma$ password characters correctly, $\mathbf{G}_5$ and $\mathbf{G}_6$ are identically distributed, whereby we get

$$\Pr[\mathbf{G}_6] = \Pr[\mathbf{G}_5].$$

Game $\mathbf{G}_7$: Use random iPAKE keys in case of attack on honest sender.
Again we only change the simulator in the case of a MitM attack, or if the receiver is corrupted. If the password guess of $\mathcal{Z}$ is correct in at most $k-1 = n-\gamma-1$ characters, then $\mathcal{S}$ uses random iPAKE keys for all $\mathcal{F}_{\mathsf{iPAKE}}$ instances of the sender. This means that the view of $\mathcal{Z}$ on $\boldsymbol{C}' \coloneqq \boldsymbol{E} \oplus \boldsymbol{K}'$ is different in $\mathbf{G}_6$ and $\mathbf{G}_7$. Let $t \leq k-1$ be the number of correct password characters. In $\mathbf{G}_6$ the receiver still obtained the original code word on t entries, whereas in this game all entries are uniformly random. However, as $t \leq k-1$, the $k-1$-query uniformity of the code guarantees that $\boldsymbol{C}'$ was already uniformly random in $\mathbf{G}_6$. Therefore, $\mathbf{G}_7$ is identically distributed as$\mathbf{G}_6$.
Note that in the case of a MitM attack, or if the receiver is corrupted, the simulator does not use the password of the honest sender anymore and

instead only uses the information that it received from $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ as an answer to a TestPwd query. We get

$$\Pr[\mathbf{G}_7] = \Pr[\mathbf{G}_6].$$

Game $\mathbf{G}_8$: Simulator handles attacks on honest receivers. In this game, we change the simulator's behavior when there is a man-in-the-middle attack against an honest receiver happening or when the sender is corrupted. More precisely, we only change $\mathcal{S}$'s behavior if $\mathcal{S}$ recorded $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{sender}, \mathsf{MitM} = 1\rangle$ or if $\mathcal{S}$ recorded $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{sender}, \mathsf{MitM} = 0\rangle$ but $\mathcal{P}_{1-i}$ is corrupted.
Like in $\mathbf{G}_6$, $\mathcal{S}$ now records TestPwd queries from $\mathcal{S}^{j}_{\mathsf{iPAKE}}$ for $j \in [n]$. When it received all n NewKey queries from $\mathcal{S}^{j}_{\mathsf{iPAKE}}$ for $\mathcal{P}_i$, it distinguishes the following cases:

- If $\mathcal{S}$ recorded $m \geq n - \gamma$ TestPwd queries with a guess pw_j of $\mathcal{S}^{j}_{\mathsf{iPAKE}}$ then $\mathcal{S}$ randomly chooses the missing $n - m$ password-entries and submits a TestPwd query to $\mathcal{F}_{\mathsf{fPAKE}}$. If $\mathcal{S}$ receives leakage from $\mathcal{F}_{\mathsf{fPAKE}}$ as response $\mathcal{S}$ behaves exactly as in $\mathbf{G}_6$. This means, $\mathcal{S}$ sets K_j to the value sk_j that was chosen by $\mathcal{S}^{j}_{\mathsf{iPAKE}}$ for the j where the corresponding guess of pw_j was correct and sets K_j to random values for those $j \in [n]$ where the guess of pw_j was wrong or there was no TestPwd query of $\mathcal{S}^{j}_{\mathsf{iPAKE}}$ for that j. In particular, if $\mathcal{F}_{\mathsf{fPAKE}}$ answers that the guess was correct, then $\mathcal{S}$ must check if this was due to its own added password entries or because of the guesses from the $\mathcal{S}_{\mathsf{iPAKE}}$s and adjust its NewKey query to $\mathcal{F}_{\mathsf{fPAKE}}$ accordingly. However, in the case where $\mathcal{S}$ does only get the answer "wrong guess" and no leakage from $\mathcal{F}_{\mathsf{fPAKE}}$, $\mathcal{S}$ does not use LDec. Instead, $\mathcal{S}$ simply sends a NewKey query to $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ with $\mathsf{sk} = \perp$, as the record is `interrupted` and $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ would ignore sk anyways.
- If there were less than $n-\gamma$ TestPwd queries of $\mathcal{S}^{j}_{\mathsf{iPAKE}}$ recorded, then $\mathcal{S}$ asks a TestPwd query in order to interrupt the record of $\mathcal{F}^{M}_{\mathsf{fPAKE}}$. Then $\mathcal{S}$ acts as above when it receives no leakage. This means, it does not decode $\boldsymbol{E}$ but instead sends a NewKey query to $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ with $\mathsf{sk} = \perp$.

Like in $\mathbf{G}_6$, when $\mathcal{S}$ receives leakage from $\mathcal{F}_{\mathsf{fPAKE}}$ the distribution of the experiment does not change as $\mathcal{S}$ uses the same session keys sk_j from $\mathcal{S}^{j}_{\mathsf{iPAKE}}$ as in $\mathbf{G}_6$ for the correct $j \in [n]$ and random keys K_j for the other positions. We stress again here that we leverage the fact that communication runs over $\mathcal{F}_{\mathsf{SA}}$ for that argument. As $\mathsf{MitM} = 1$ (or the sender is corrupted) the $\mathcal{F}_{\mathsf{iPAKE}}$ instances of the receiver will never successfully exchange keys, as they run in a different authentication set, i.e., with a different session-id, than the iPAKE of the sender. This is what prevents the attack from Section 3.2.
However, in the case where $\mathcal{S}$ does not receive leakage or less than $n - \gamma$ of the $\mathcal{S}^{j}_{\mathsf{iPAKE}}$ sent a TestPwd query, the behavior of $\mathcal{S}$ changed. In $\mathbf{G}_7$, as we either have $\mathsf{MitM} = 1$ or a corrupt sender, $\mathcal{Z}$ can instruct the adversary to send a modified message $(\boldsymbol{E}', h')$ to the honest receiver. But in $\mathbf{G}_7$ the receiver would at least have $\gamma + 1$ indices $j \in [n]$ where the corresponding iPAKE key K_j is uniformly random. That is because either $\mathcal{S}^{j}_{\mathsf{iPAKE}}$ did

send NewKey without sending TestPwd or because the provided password guess pw_j from the TestPwd query of $\mathcal{S}^j_{\mathsf{iPAKE}}$ was wrong. But then, the receiver computes $\mathsf{LDec}(\boldsymbol{E}' \oplus \boldsymbol{K}')$, where $\boldsymbol{E}' \oplus \boldsymbol{K}'$ is uniformly random in at least $\gamma + 1$ positions. Now we need to argue that also in $\mathbf{G}_7$ the honest receiver would have output an independent, uniformly random key. First consider the case that $\mathcal{Z}$ has not queried the random oracle H_0 on a vector $v \in \mathbb{F}_q^n$ such that $H_0((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), v) = h'$. Then the probability that a candidate codeword $\boldsymbol{C}_j$ output by LDec matches the hint h' is negligible and the honest receiver will output $u \xleftarrow{\$} \{0,1\}^\lambda$. Now consider the case that $\mathcal{Z}$ did query $v \in \mathbb{F}_q^n$ such that $H_0((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), v) = h'$. Then, v is uniquely determined by h' because since $\mathbf{G}_2$ the simulator aborts on collisions of H_0. Here we can apply Lemma 4 by setting $\boldsymbol{C} = v, \tilde{\boldsymbol{C}} = \boldsymbol{C}$ and A to be the indices of the correct password guesses of the $\mathcal{S}^j_{\mathsf{iPAKE}}$ and get that $d(v, \boldsymbol{C}') \geq \gamma + 1$, which also means that $d(v, \boldsymbol{C}) \geq \delta + 1$. Since LDec does not output candidates with a distance greater than δ to $\boldsymbol{C}$, LDec will not output v and, thus, none of the candidates will match the hint. Therefore, the honest sender will output $u \xleftarrow{\$} \{0,1\}^\lambda$.
Thus, in $\mathbf{G}_7$ in both cases, the honest sender outputs a fresh uniformly random key with overwhelming probability which is the same as in $\mathbf{G}_8$.

$$\Pr[\mathbf{G}_8] \approx \Pr[\mathbf{G}_7].$$

Game $\mathbf{G}_9$: Simulator does not give passwords to $\mathcal{F}_{\mathsf{iPAKE}}$ instances. In this game, we change the simulator. Whenever the simulator receives a message $(\text{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i}, \mathsf{role})$ from $\mathcal{F}_{\mathsf{fPAKE}}$ for an honest party $\mathcal{P}_i$ it does not use $\mathsf{pw}^{\mathcal{P}_i}$ as input for $\mathcal{F}_{\mathsf{iPAKE}}$. Concretely, instead of internally running $\mathcal{F}^j_{\mathsf{iPAKE}}$ on input $(\text{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i}_j, \mathsf{role})$ the simulator forwards $(\text{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathsf{role})$ (without the j-th bit of $\mathsf{pw}^{\mathcal{P}_i}$) to $\mathcal{S}^j_{\mathsf{iPAKE}}$. Further, $\mathcal{S}$ checks if this is the first NewSession query or this is the second NewSession query and if there is a record $\langle \mathcal{F}_{\mathsf{iPAKE}}, \mathcal{P}_{1-i}, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j)\rangle$. If so, $\mathcal{S}$ creates a record $\langle \mathcal{F}_{\mathsf{iPAKE}}, \mathcal{P}_i, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j)\rangle$ and marks it `fresh`. On a NewKey query, $\mathcal{S}$ still behaves exactly as $\mathcal{F}_{\mathsf{iPAKE}}$ by using $\mathsf{pw}^{\mathcal{P}_i}$ to check if a matching output has to be given to the parties.
For every iPAKE instance, the view on the execution did not change, as $\mathcal{S}$ behaves as $\mathcal{F}_{\mathsf{iPAKE}}$ would.

$$\Pr[\mathbf{G}_9] = \Pr[\mathbf{G}_8].$$

Game $\mathbf{G}_{10}$: Simulator makes honest and undisturbed parties output a key. In this game, we change the simulator and the functionality. First, we change $\mathcal{F}^M_{\mathsf{fPAKE}}$'s behavior on receiving a query $(\text{NewKey}, \mathsf{sid}, \mathcal{P}_i, \mathsf{sk})$ from $\mathcal{S}$. Now, $\mathcal{F}^M_{\mathsf{fPAKE}}$ treats these queries exactly like in Fig. 3. That means concretely, if the record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$ is `fresh` and both parties are honest and run on input passwords with $d(\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}) \leq \delta$ then both parties receive the same random key.
We also change the simulator. When the simulator receives a message $(\text{NewSession}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i}, \mathsf{role})$ from $\mathcal{F}_{\mathsf{fPAKE}}$ for an honest party $\mathcal{P}_i$ where there is no man-in-the-middle attack $\mathcal{S}$ discards $\mathsf{pw}^{\mathcal{P}_i}$ and stores $\perp$ instead of $\mathsf{pw}^{\mathcal{P}_i}$ in all records of the form

$\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{pw}^{\mathcal{P}_i} \rangle$. We also change how $\mathcal{S}$ reacts on receiving all n NewKey queries from $\mathcal{S}^j_{\mathsf{iPAKE}}$ for $j \in [n]$.

- If the role of $\mathcal{P}_i$ is $\mathsf{role} = \mathsf{receiver}$ then $\mathcal{S}$ marks its record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} = 0 \rangle$ as `waiting` but produces no output for $\mathcal{P}_i$. Instead, on a message $(\textsc{Deliver}, \mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, (\boldsymbol{E}, h))$ the simulator sends a message $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \perp)$ to $\mathcal{F}_{\mathsf{fPAKE}}$ if $(\boldsymbol{E}, h)$ is not tampered with.
- If the role of $\mathcal{P}_i$ is $\mathsf{role} = \mathsf{sender}$ then $\mathcal{S}$ marks its record $\langle \mathcal{P}_i, \mathsf{sid}, \mathsf{role}, \mathsf{MitM} = 0 \rangle$ as `completed` and chooses $K_1, \ldots, K_n$ uniformly at random. The simulator uses these keys to produce $(\boldsymbol{E}, h)$ like an honest party would do. $\mathcal{S}$ sends $(\textsc{Send}, \mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, (\boldsymbol{E}, h))$ to $\mathcal{Z}$ formatted as being sent from $\mathcal{P}_i$ to $\mathcal{P}_{1-i}$ via $\mathcal{F}_{\mathsf{SA}}$. Then, $\mathcal{S}$ sends $(\textsc{NewKey}, \mathsf{sid}, \mathcal{P}_i, \perp)$ to $\mathcal{F}_{\mathsf{fPAKE}}$.

Note that the changes to $\mathcal{F}^M_{\mathsf{fPAKE}}$ only affect the case where both parties are honest and no man-in-the-middle attack is happening. Because if a man-in-the-middle attack is mounted then $\mathcal{S}$ will send a TestPwd query to $\mathcal{F}^M_{\mathsf{fPAKE}}$ (see $\mathbf{G}_6$ and $\mathbf{G}_8$) either marking $\mathcal{F}^M_{\mathsf{fPAKE}}$'s record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$ as `interrupted` or as `compromised`, so the record will not be `fresh`. The rest of the NewKey interface of $\mathcal{F}^M_{\mathsf{fPAKE}}$ was already like in Fig. 3 since $\mathbf{G}_4$.

In the first case $\mathsf{role} = \mathsf{receiver}$ only the output of the party is changed. This is because the keys $K_1, \ldots, K_n$ (output by $\mathcal{F}^j_{\mathsf{iPAKE}}$) were only used internally by $\mathcal{P}_i$ in $\mathbf{G}_9$. In this game, $\mathcal{P}_i$ produces output as the ideal functionality $\mathcal{F}_{\mathsf{fPAKE}}$ provides the dummy party with output on receiving NewKey from $\mathcal{S}$. In $\mathbf{G}_9$ the output was $H_1((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0)$. In the second case $\mathsf{role} = \mathsf{sender}$, $\mathcal{F}_{\mathsf{fPAKE}}$ provides the dummy party also with output instead of $H_1((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0)$. Note that we assumed no man-in-the-middle attack and thus, $\mathcal{F}_{\mathsf{fPAKE}}$ will produce output for $\mathcal{P}_i$ and $\mathcal{P}_{1-i}$ depending on the input passwords $\mathsf{pw}^{\mathcal{P}_i}$ and $\mathsf{pw}^{\mathcal{P}_{1-i}}$. If $d(\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}) \leq \delta$ then $\mathcal{F}_{\mathsf{fPAKE}}$ provides matching outputs $\mathsf{sk} = \mathsf{sk}'$ to $\mathcal{P}_i$ and $\mathcal{P}_{1-i}$ and if $d(\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}) > \delta$ then the output sk of $\mathcal{P}_i$ is drawn independently of the output sk' of $\mathcal{P}_{1-i}$. Clearly, by Lemma 6 both parties output the same random string $\mathsf{sk} = \mathsf{sk}'$ in $\mathbf{G}_9$ if $d(\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}) \leq \delta$. Further, in $\mathbf{G}_9$ if $d(\mathsf{pw}^{\mathcal{P}_i}, \mathsf{pw}^{\mathcal{P}_{1-i}}) > \delta$, we can use Lemma 5 to see that both parties output different keys. That is because the passwords mismatch on more than δ characters and so the codeword $\boldsymbol{C}' = \boldsymbol{E} \oplus \boldsymbol{K}'$ computed by the receiver will with overwhelming probability not decode to $\boldsymbol{C}$.

It remains to argue that $\mathcal{Z}$'s view on the protocol transcript does not change. Note that in $\mathbf{G}_9$ $(\boldsymbol{E}, h)$ was computed as $\boldsymbol{E} \coloneqq \boldsymbol{C} \oplus \boldsymbol{K}$, where $\boldsymbol{K} = (K_1, \ldots, K_n)$ were the outputs of $\mathcal{F}^j_{\mathsf{iPAKE}}$ for $j \in [n]$. In $\mathbf{G}_{10}$ $\boldsymbol{K}$ is uniformly random. The output K_j of $\mathcal{F}^j_{\mathsf{iPAKE}}$ is also a uniformly random string when both parties are honest and no man-in-the-middle attack happened. The only difference is that both parties, $\mathcal{P}_i$ and $\mathcal{P}_{1-i}$ receive the same K_j as output for that session. But as we argued above, the receiver only uses K_j internally, so $\mathcal{Z}$ does not see whether both parties output the same K_j for a iPAKE session. We get

$$\Pr[\mathbf{G}_{10}] \approx \Pr[\mathbf{G}_9].$$

Game $\mathbf{G}_{11}$: Simulator does not get inputs of the parties anymore. In this game, we change the functionality. Instead of sending

(NewSession, $\mathsf{sid}, \mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i}, \mathsf{role}$) to $\mathcal{S}$ the functionality now keeps the input $\mathsf{pw}^{\mathcal{P}_i}$ secret and sends only (NewSession, $\mathsf{sid}, \mathcal{P}_i, \mathsf{role}$) to $\mathcal{S}$.
Since $\mathbf{G}_{10}$ $\mathcal{S}$ does not need the input $\mathsf{pw}^{\mathcal{P}_i}$ anymore to simulate the n instances of $\mathcal{F}_{\mathsf{iPAKE}}$ for two honest parties with no man-in-the-middle attack. Further, we argued in $\mathbf{G}_6$ and $\mathbf{G}_8$ that in a session with a corrupted party or where a man-in-the-middle attack is mounted $\mathcal{S}$ also does not need $\mathsf{pw}^{\mathcal{P}_i}$ as it can either use random keys $K_1, \ldots, K_n$ or it learns through a TestPwd query to $\mathcal{F}_{\mathsf{fPAKE}}$ which iPAKE sessions were successful. Overall, the simulator does not use the provided input $\mathsf{pw}^{\mathcal{P}_i}$ anymore. Hence, we get

$$\Pr[\mathbf{G}_{11}] = \Pr[\mathbf{G}_{10}].$$

Note that $\mathbf{G}_{11}$ is the same as the ideal execution of $\mathcal{F}_{\mathsf{fPAKE}}$ with the simulator from Figs. 9 to 10. This concludes the proof.

B Additional preliminaries

We recap the implicit PAKE functionality of Dupont et al. [DHP+18] in Fig. 12 as well as the split authentication functionality of Barak et al. [BCL+05] in Fig. 13.

C On the similarities to AMD codes

In Section 4, we explained how our protocol can narrow down the list of "candidate decodings" that are output of the list-decoding to a single codeword by using the hint h. Cramer et al. [CDD+15] had a similar idea. To construct a robust secret sharing scheme, they use a list-decodable code. But before encoding the message with this code, they apply an *Algebraic Manipulation Detection* (AMD) code to the message. Intuitively, these codes allow to detect manipulation of an encoded message and abort decoding if the codeword was manipulated. Now, one could suspect that the hint $h = H_0(\boldsymbol{C})$ serves as AMD code in our construction. But we argue in the following that hashing the codeword and appending the hash is not an AMD code. Roughly speaking, if we would use our construction without split authentication it would be trivial for an adversary to tamper with the codeword and the hint, such that a receiver decodes the wrong message. We elaborate on this more formally:

An AMD code is defined as follow:

Definition 3 (Algebraic Manipulation Detection Code). *Let q be a prime-power, $n > k$ be integers and $\rho > 0$. A (k, n, ρ)-AMD $\mathcal{C}$ consists of a probabilistic encoding algorithm $\mathsf{Enc} : \mathbb{F}_q^k \to \mathbb{F}_q^n$ and a deterministic decoding algorithm $\mathsf{Dec} : \mathbb{F}_q^n \to \mathbb{F}_q^k \cup \{\perp\}$, such that the following holds for every $\boldsymbol{x} \in \mathbb{F}_q^k$:*

- Correctness: $\mathsf{Dec}(\mathsf{Enc}(\boldsymbol{x})) = \boldsymbol{x}$ *with probability 1.*
- Manipulation detection: *for every $\Delta \in \mathbb{F}_q^n$ and for $\boldsymbol{C} \leftarrow \mathsf{Enc}(\boldsymbol{x})$ it holds that $\mathsf{Dec}(\boldsymbol{x} \oplus \Delta) \in \{\perp, \boldsymbol{x}\}$ except with probability at most ρ.*

The functionality $\mathcal{F}_{\mathsf{iPAKE}}$ is parameterized by a security parameter λ. It interacts with an adversary $\mathcal{A}$ and the (dummy) parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Upon receiving a query (NewSession, sid, $\mathsf{pw}^{\mathcal{P}_i}$, role) from party $\mathcal{P}_i$:**
  - Send (NewSession, sid, $\mathcal{P}_i$, role) to $\mathcal{A}$;
  - If one of the following is true, record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$ and mark this record `fresh`:
    - This is the first NewSession query
    - This is the second NewSession query and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}^{\mathcal{P}_{1-i}})$
- **Upon receiving a query (TestPwd, sid, $\mathcal{P}_i$, pw^*) from $\mathcal{A}$:**
  If there is a `fresh` record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$, then:
  - If $\mathsf{pw}^{\mathcal{P}_i} = \mathsf{pw}^*$, mark the record `compromised`;
  - If $\mathsf{pw}^{\mathcal{P}_i} \neq \mathsf{pw}^*$, mark the record `interrupted`.
- **Upon receiving a query (NewKey, sid, $\mathcal{P}_i$, sk) from $\mathcal{A}$, where $|\mathsf{sk}| = \lambda$:**
  If there is no record of the form $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$, or if this is not the first NewKey query for $\mathcal{P}_i$, then ignore this query. Otherwise:
  - If at least one of the following is true, then output (sid, sk) to player $\mathcal{P}_i$:
    - The record is `compromised`
    - $\mathcal{P}_i$ is corrupted
    - The record is `fresh`, $\mathcal{P}_{1-i}$ is corrupted, and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}^{\mathcal{P}_{1-i}})$ with $\mathsf{pw}^{\mathcal{P}_i} = \mathsf{pw}^{\mathcal{P}_{1-i}}$
  - If this record is `fresh`, both parties are honest, there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}^{\mathcal{P}_{1-i}})$ with $\mathsf{pw}^{\mathcal{P}_i} = \mathsf{pw}^{\mathcal{P}_{1-i}}$, a key sk' was sent to $\mathcal{P}_{1-i}$, and $(\mathcal{P}_{1-i}, \mathsf{pw}^{\mathcal{P}_{1-i}})$ was `fresh` at the time, then output $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$;
  - In any other case, pick a new random key sk' of length λ and send $(\mathsf{sid}, \mathsf{sk}')$ to $\mathcal{P}_i$.
  - Mark the record $(\mathcal{P}_i, \mathsf{pw}^{\mathcal{P}_i})$ as `completed`.

Fig. 12. Functionality $\mathcal{F}_{\mathsf{iPAKE}}$ [DHP+18].

Functionality $\mathcal{F}_{\mathsf{SA}}$

Initialization:

Upon activation with input $(\textsc{Init}, \mathsf{sid})$, where $\mathsf{sid} = (\mathbb{P}, \mathsf{sid}')$ and $\mathbb{P}$ is a set of PIDs, record $\mathbb{P}$ and forward it to the adversary.

On $(\textsc{Init}, \mathsf{sid}, \mathcal{P}, \mathbb{H}, \mathsf{sid}_{\mathbb{H}})$ **from** the adversary, verify that:
- $\mathcal{P} \in \mathbb{P}$ and $\mathcal{P} \in \mathbb{H}$, and
- for all recorded sets $\mathbb{H}'$, either
  - $\mathbb{H} \cap \mathbb{H}'$ contains only corrupt parties and $\mathsf{sid}_{\mathbb{H}} \neq \mathsf{sid}_{\mathbb{H}'}$, or
  - $\mathbb{H}' = \mathbb{H}$ and $\mathsf{sid}_{\mathbb{H}} = \mathsf{sid}_{\mathbb{H}'}$.

If so, then output $(\textsc{Init}, \mathsf{sid}, \mathsf{sid}_{\mathbb{H}})$ to $\mathcal{P}$ and record $(\mathbb{H}, \mathsf{sid}_{\mathbb{H}})$ if not yet recorded. Else, do nothing.

Message authentication:

On input $(\textsc{Send}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \text{msg})$ **from** $\mathcal{P}$, where $\mathcal{P}' \in \mathbb{P}$, send $(\mathcal{P}, \mathcal{P}', \text{msg})$ to $\mathcal{A}$ and add $(\mathcal{P}, \mathcal{P}', \text{msg})$ to an (initially empty) list $\mathbb{W}$ of waiting messages. Note that the same triple can appear multiple times in the list. Else do nothing.

On $(\textsc{Deliver}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \text{msg})$ from $\mathcal{A}$, proceed as follows:
- If $\mathcal{P}'$ did not previously receive an $(\textsc{Init}, \mathsf{sid}, \cdot)$ output then do nothing.
- Else, if $\mathcal{P}$ is in the authentication set $\mathbb{H}$ of $\mathcal{P}'$, and $\mathcal{P}$ is honest, then do:
  - If there is a triple $(\mathcal{P}, \mathcal{P}', \text{msg}) \in \mathbb{W}$, then remove one appearance of the triple from $\mathbb{W}$ and output $(\textsc{Received}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \text{msg})$ to $\mathcal{P}'$.
  - Otherwise do nothing.
- Else (i.e., $\mathcal{P}'$ received $(\textsc{Init}, \mathsf{sid}, \cdot)$, and either $\mathcal{P}$ is corrupted or $\mathcal{P} \notin \mathbb{H}$), output $(\textsc{Received}, \mathsf{sid}, \mathcal{P}, \mathcal{P}', \text{msg})$ to $\mathcal{P}'$, regardless of $\mathbb{W}$.

Fig. 13. Ideal functionality $\mathcal{F}_{\mathsf{SA}}$ for split authentication [BCL+05].

We can try to formalize the hint mechanism as above by setting $\mathsf{Enc}(\boldsymbol{x}) := \boldsymbol{x} \| H(\boldsymbol{x})$. The decoding parses a codeword as $\boldsymbol{C} := \boldsymbol{x} \| h$ and outputs $\boldsymbol{x}$ if $h = H(\boldsymbol{x})$ and else an error symbol $\perp$. But it turns out that this is *not* an AMD code. The reason is that manipulation detection must hold for all $\boldsymbol{\Delta} \in \mathbb{F}_q^n$. Let $\boldsymbol{x}' \in \mathbb{F}_q^k$ with $\boldsymbol{x}' \neq \boldsymbol{x}$. Then manipulation detection must hold in particular for the vector

$$\boldsymbol{\Delta} = (\boldsymbol{x} \oplus \boldsymbol{x}', H(\boldsymbol{x}) \oplus H(\boldsymbol{x}')).$$

But the decoding algorithm would now output $\boldsymbol{x}'$ and not $\perp$ or $\boldsymbol{x}$ on input $\boldsymbol{x} \oplus \boldsymbol{\Delta}$.

Now, why does this weakness not allow for an attack on our fuzzy PAKE protocol? The reason is that we send messages over a split-authenticated channel. If both parties are authenticated, i.e., in the same authentication set $\mathbb{H}$ then an adversary can not tamper with the message by adding $\boldsymbol{\Delta}$ as above to the codeword. Else if there is a MitM attack happening then we distinguish two cases. (1) If the adversary (playing as corrupted sender) guesses at least $n - \delta$ password characters of the receiver correctly then the receiver gets iPAKE keys of the adversaries choice. That means, the adversary can anyways make the

client output a shared key of its choice by changing the iPAKE keys accordingly. So tampering with h does not give the adversary additional power. (2) If the adversary guessed less than $n - \delta$ password characters correctly then there is already so much randomness added to the codeword by the iPAKE keys that the list decoding algorithm will output some specific codeword only with negligible probability.

D UC Execution and Instantiation of our fuzzy PAKE

We give a UC execution of our fuzzy PAKE protocol of Figure 8 using the interfaces of $\mathcal{F}_{\mathsf{fPAKE}}$ in Fig. 14. We can instantiate the building blocks of our

The parties $\mathcal{P}_0$ and $\mathcal{P}$ communicate with the hybrid functionalities $\mathcal{F}_{\mathsf{SA}}$ and $\mathcal{F}_{\mathsf{RO}}$.

On (NewSession, $\mathsf{sid}, \mathcal{P}_i, \mathsf{role}$) from $\mathcal{F}_{\mathsf{fPAKE}}$ or (Init, $(\{\mathcal{P}_i, \mathcal{P}_{1-i}\}, \mathsf{sid})$) from $\mathcal{Z}$ in the name of $\mathcal{P}_i$ to $\mathcal{F}_{\mathsf{SA}}$
- Record $\langle \mathsf{sid}, \mathsf{role} \rangle$
- Send (Init, $(\{\mathcal{P}_i, \mathcal{P}_{1-i}\}, \mathsf{sid})$) to $\mathcal{F}_{\mathsf{SA}}$.

On (Init, $\mathsf{sid}, \mathsf{sid}_H$) from $\mathcal{F}_{\mathsf{SA}}$
- For $j \in [n]$:
  - Run $X_j^* \leftarrow \pi_j(\mathcal{P}_i, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j), \mathsf{pw}_j)$
- Send (Send, $\mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, (X_1^*, \ldots, X_n^*)$) to $\mathcal{F}_{\mathsf{SA}}$

On (Deliver, $\mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, (Y_1^*, \ldots, Y_n^*)$) from $\mathcal{F}_{\mathsf{SA}}$ and if $\mathcal{P}_i$ recorded $\langle \mathsf{sid}, \mathsf{sender} \rangle$
- For $j \in [n]$:
  Run $K_j \leftarrow \pi_j(\mathcal{P}_i, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j), Y_j^*)$
- Let $\boldsymbol{K} \coloneqq (K_1, \ldots, K_n)$
- $\boldsymbol{s} = (s_0, \ldots, s_{k-1}) \xleftarrow{\$} \mathbb{F}_q^k$
- $\boldsymbol{C} \leftarrow \mathsf{Enc}(\boldsymbol{s})$
- $\boldsymbol{E} \coloneqq \boldsymbol{C} \oplus \boldsymbol{K}$
- $h \coloneqq H_0((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C})$
- Send (Send, $\mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, (\boldsymbol{E}, h)$) to $\mathcal{F}_{\mathsf{SA}}$
- Send $((\mathsf{sid}, \mathsf{sid}_H), s_0)$ to $\mathcal{F}_{\mathsf{RO}}$. On response ρ from $\mathcal{F}_{\mathsf{RO}}$ output ρ.

On (Deliver, $\mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, (Y_1^*, \ldots, Y_n^*)$) from $\mathcal{F}_{\mathsf{SA}}$ and if $\mathcal{P}_i$ recorded $\langle \mathsf{sid}, \mathsf{receiver} \rangle$
- For $j \in [n]$:
  Run $K_j \leftarrow \pi_j(\mathcal{P}_i, (\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}, j), Y_j^*)$
- Let $\boldsymbol{K} \coloneqq (K_1, \ldots, K_n)$
- Append $\boldsymbol{K}$ to $\langle \mathsf{sid}, \mathsf{receiver} \rangle$

On (NewSession, $\mathsf{sid}, \mathcal{P}_i, \mathsf{role}$) from $\mathcal{F}_{\mathsf{fPAKE}}$ or (Init, $(\{\mathcal{P}_i, \mathcal{P}_{1-i}\}, \mathsf{sid})$) from $\mathcal{Z}$ in the name of $\mathcal{P}_i$ to $\mathcal{F}_{\mathsf{SA}}$
- Send (Init, $(\{\mathcal{P}_i, \mathcal{P}_{1-i}\}, \mathsf{sid})$) to $\mathcal{F}_{\mathsf{SA}}$.

On (Deliver, $\mathsf{sid}, \mathcal{P}_i, \mathcal{P}_{1-i}, \mathrm{msg} := (\boldsymbol{E}', h')$) from $\mathcal{F}_{\mathsf{SA}}$ and if $\mathcal{P}_i$ recorded $\langle \mathsf{sid}, \mathsf{receiver} \rangle$:
- Run $\{\boldsymbol{C}^1, \ldots, \boldsymbol{C}^\ell\} \leftarrow \mathsf{LDec}(\boldsymbol{E} \oplus \boldsymbol{K}')$
- If $h = H_0((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), \boldsymbol{C}^j)$ for a $j \in [\ell]$
  - $\boldsymbol{s}' \leftarrow \mathsf{Dec}(\boldsymbol{C}^j)$
  - Send $((\mathsf{sid}, \mathsf{sid}_{\mathbb{H}}), s_0)$ to $\mathcal{F}_{\mathsf{RO}}$. On response ρ from $\mathcal{F}_{\mathsf{RO}}$ output ρ.
- else output $u \xleftarrow{\$} \{0,1\}^\lambda$

Fig. 14. A UC Execution of our Protocol.

protocol with EKE2 [BPR00] as iPAKE and a signature-based transformation to instantiate $\mathcal{F}_{\mathsf{SA}}$ by Barak et al. [BCL+05]. The fully instantiated protocol is depicted in Fig. 15.

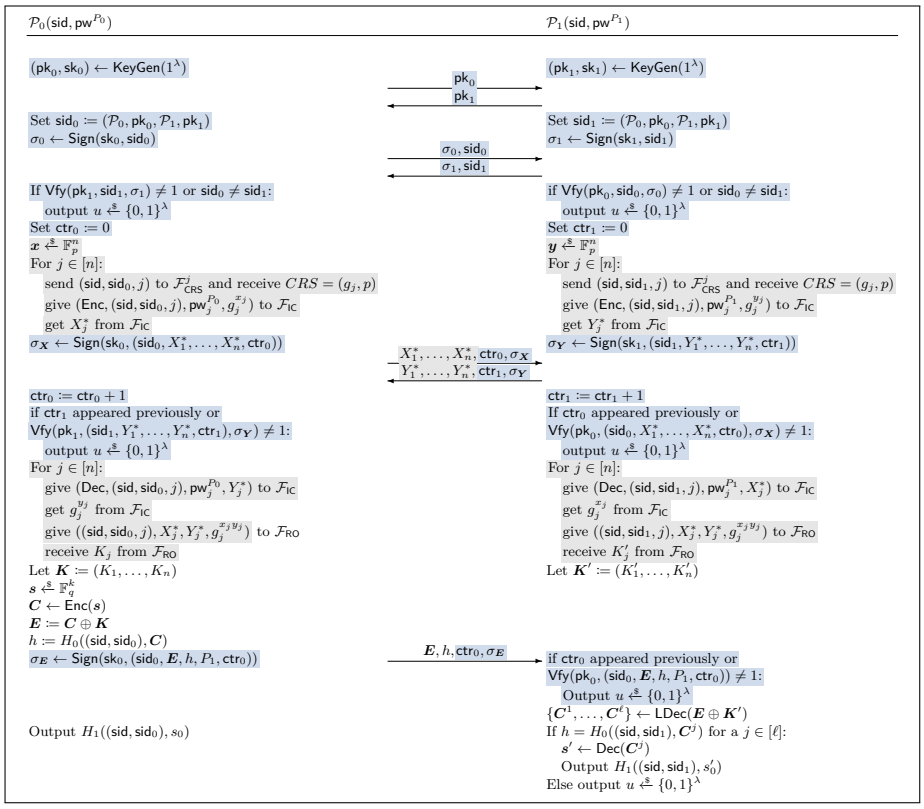


Fig. 15. The protocol in the $\mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{IC}}$-hybrid model without any black-box PAKE functionality but with explicit EKE and with signing like in the split transform. Assume that a party aborts the execution if any check fails. Parts in grey are for the n EKE2 key exchanges and parts in blue are for the split-transform-like authentication.