SweetPAKE: Key exchange with decoy passwords

Afonso Arriaga[1] Peter Y.A. Ryan[1] Marjan Škrobot[1]

[1]University of Luxembourg,
{afonso.delerue, peter.ryan, marjan.skrobot}@uni.lu

Abstract

Decoy accounts are often used as an indicator of the compromise of sensitive data, such as password files. An attacker targeting only specific known-to-be-real accounts might, however, remain undetected. A more effective method proposed by Juels and Rivest at CCS'13 is to maintain additional fake passwords associated with each account. An attacker who gains access to the password file is unable to tell apart real passwords from fake passwords, and the attempted usage of a false password immediately sets off an alarm indicating a password file compromise. Password-Authenticated Key Exchange (PAKE) has long been recognised for its strong security guarantees when it comes to low-entropy password authentication and secure channel establishment, without having to rely on the setup of a PKI. In this paper, we introduce SweetPAKE, a new cryptographic primitive that offers the same security guarantees as PAKE for key exchange, while allowing clients with a single password to authenticate against servers with n candidate passwords for that account and establish a secure channel. Additional security properties are identified and formalized to ensure that (a) high-entropy session keys are indistinguishable from random, even if later on the long-term secret password becomes corrupted (forward secrecy); (b) upon password file leakage, an adversary cannot tell apart real from fake passwords; and (c) a malicious client cannot trigger a false alarm. We capture these properties by extending well-established game-based definitions of PAKE. Furthermore, we propose a new UC formulation that comprehensively unifies both SweetPAKE (session key indistinguishability and sugarword indistinguishability) and a related notion known as Oblivious-PAKE. Finally, we propose efficient SweetPAKE and Oblivious-PAKE protocols constructed from Password-Authenticated Public-Key Encryption (PAPKE) that satisfy all the proposed notions.

Keywords: SweetPAKE, Honeywords, PAKE, Password-Authenticated Public-Key Encryption, Oblivious PAKE.

1 Introduction

Data breaches and leakage of user credentials are unfortunately becoming more common. It comes as a result of the rise of the Internet and the interconnectedness of devices, along with inadequate security measures to protect sensitive data. Some resources such as the '*Have I Been Pwned*' website[1], or even modern password managers, regularly monitor websites, forums, and marketplaces with anonymized access, where criminals usually sell or publish stolen credentials. However, organizations and service providers must act promptly, but face significant challenges in detecting such data breaches. According to a 2022 data security report by IBM [21], businesses took an average of approximately 9 months to identify and report a data breach. It is therefore of the utmost importance that system architects take a *security-by-design* approach and put in place adequate mechanisms to detect a compromise of user credentials.

[1]As of 2022, HIBP collected 11M+ credentials from more than 600 compromised websites. The project's website is available at `https://haveibeenpwned.com`.

One simple tactic to detect leakage of user credentials is to create decoy accounts. This has the clear benefit that no upgrade to the way users are authenticated within the system is necessary. Suspicious activity in those accounts could indicate a password file compromise, as they are not associated with real users. However, an adversary might be able to tell apart real usernames from those that are fake, or target only specific accounts known to belong to legitimate users, and thus avoid triggering the alert mechanism in place.

Another approach, proposed by Juels and Rivest [24], is to maintain decoy passwords instead. Each user account is associated with one real password and multiple fake passwords. Using the authors' terminology, fake passwords are called *honeywords*, the real password is called *the sugarword*, and together they form *the sweetwords* associated with each account. This so-called *Honeywords* method incorporates an auxiliary computer device called *the Honeychecker*, with a simple API and an operating system possibly different from that of the authenticated server, whose sole purpose is to raise an alarm whenever a user used a honeyword as their password. The authentication server stores the password file, containing n sweetwords per user. The Honeychecker stores the index of each user's sugarword (and nothing else). The Honeychecker is connected to the authentication server via a secure channel, and every time a user logs in, the index of the sweetword used (together with the username) is transmitted to the Honeychecker. Obviously, if honeywords are not generated carefully, an adversary might also be able to spot the sugarword among sweetwords. A generation algorithm is said to be (perfectly) *flat* if the winning probability of guessing the sugarword given n sweetword is $1/n$. The authors propose multiple generation algorithms that are perfectly flat (or ϵ-flat, meaning that a negligible advantage ϵ is tolerated). We refer the reader to [24] for more details on these honeyword generation algorithms.

The authors of [24] assume there is a server-authenticated TLS channel between the user and the authentication server, over which the password is transmitted, every time the client authenticates. In practice, this is usually supported by a PKI, where a trusted Certificate Authority (CA) asserts the identity of the server and its public key via an X.509 certificate. Needless to say that this is a considerable trust assumption since modern browsers and operating systems include more than a hundred certificates in their truststore, all of which could potentially issue such a certificate. Phishing attacks are recurring, whether the attacker was able to masquerade as the authentication server because it obtained a valid certificate or because it was able to trick the victim. A better approach to overcome the limitations of client-authentication via preshared password over server-authenticated TLS channel that has seen wide-scale deployment in recent years [31, 18, 5] relies on Password-Authenticated Key Exchange (PAKE) protocols. This primitive allows two parties that share a low-entropy password to establish a high-entropy session key, with the assurance that offline dictionary attacks are computationally hard, and online password guesses are limited to one per interaction. Phishing attacks are effectively mitigated, which stands as a significant advantage of PAKE protocols. This raises the following research questions:

- ***How to build a PAKE-style protocol that admits decoy passwords?***
  In contrast to conventional PAKE protocols, servers store n sweetwords per client. In addition to the session key, a server running the protocol also outputs the index of the client's password, to be transmitted to the Honeychecker.
- ***What security guarantees such protocol provides?***

To answer these questions, we introduce a new cryptographic primitive that we call *SweetPAKE*, which is a generalization of PAKE[2], and allows for direct integration with the *Honeywords* method from [24].

[2]Looking ahead, the SweetPAKE game-based security model for the case ($n = 1$) matches the widely-used *Real-or-Random* model for PAKE by [3].

We emphasize that our research centers on the development of a mutual authentication protocol between a client and a server, via a shared low-entropy password. On the server side, the client's password is placed among decoys. An initial client registration and decoy passwords generation is required. This process, similarly to the setup required by conventional PAKE protocols, can be executed through various methods, ranging from in-person registration to online registration facilitated by a server-authenticated TLS channel. All the benefits of PAKE persist even when the secure channel for the registration phase is established via a PKI, as the registration procedure for a new client is a one-time setup requirement.

1.1 Our contribution

We consider a system that, upon enrollment, allows a client to share a password (i.e. the sugarword) with an authentication server. This sugarword is on the authentication server side hidden amongst $(n-1)$ honeywords.

Thus, each authentication server in its password file, for each client, stores a vector containing n sweetwords (the sugarword plus the $(n-1)$ honeywords). The *index* corresponding to the position of the sugarword in the vector of sweetwords is together with the client's identity stored by the Honeychecker. The client enrollment process (registration, password selection, and honeywords generation) could be carried out with the help of the registration authority or by the authentication server itself, assuming that the index is an ephemeral value that is discarded and never stored after being transmitted to the Honeychecker.

Later on, a SweetPAKE protocol allows the client and the authentication server to agree on a secure session key as long as the client's password is among the n sweetwords. The authentication server silently transmits the index of the matching sweetword to the Honeychecker. If it does not match the index of the sugarword, the Honeychecker raises an alarm to the system administrator, signalling a password file compromise.
In this paper, our contribution is multi-fold:

- **Architecture.** We abstract the general architecture considered in [24] to a setting where we don't assume a server-authenticated channel between client and server (possibly supported by a PKI).

- **Security model**. We identify and formalize three security properties for SweetPAKE to ensure that (a) high-entropy session keys are indistinguishable from random, even if later on the long-term secret password becomes corrupted (commonly referred to as *forward secrecy*); (b) upon password file leakage, an active adversary cannot tell apart real passwords from those that are fake and never used by legitimate clients; and (c) a malicious client cannot trigger a false alarm. All these properties are first formalized based on the well-known Real-or-Random definition for PAKE [3]. We then model the first two properties in the Universal Composability framework, with a single ideal functionality. UC definitions are notorious for capturing arbitrary correlations between passwords, which is particularly relevant in the context of SweetPAKE as we are then able to assure security with respect to a larger set of honeyword generation algorithms.

- **Naive approach.** We explain why the naive approach of running up to n times a PAKE protocol is insufficient to meet the above-mentioned requirements, in addition to being inefficient due to too many rounds of communication.

- **BeePAKE.** We build a SweetPAKE protocol from Password-Authenticated Public-Key Encryption (PAPKE) [12], message authentication codes, key derivation functions and random

permutations. We call our protocol BeePAKE and formally prove that it satisfies all three security properties.

- **Relations with other notions.** We explore the relations between our newly proposed primitive with other notions in the literature, namely Oblivious PAKE [25]. To the best of our knowledge, no UC definition was previously proposed for O-PAKE. The one we propose here unifies both SweetPAKE and O-PAKE. We then show how BeePAKE protocol can be modified to achieve UC-secure O-PAKE.

1.2 Related Work

In this section, we briefly discuss the state-of-the-art PAKE protocols and security definitions, introduce PAPKE [12] and three important notions that are relevant to our problem: Oblivious PAKE [25], HoneyPAKE [7], and HPAKE [27]. Finally, we mention a few complementary approaches.

PAKE. Password-Authenticated Key Exchange allows secure session key establishment over insecure networks between two or more parties who only share a low-entropy password. The main advantage of PAKE over a typical authentication approach is that it prevents offline dictionary attacks and by design avoids man-in-the-middle attacks in the context of phishing. The most widely accepted security definitions for PAKE can be classified as (i) *game-based*; and (ii) *simulation-based* definitions. Security model 'Find-then-Guess' (FtG) [9] and its extension 'Real-or-Random' (RoR) [3] are prominent within the game-based category. The simulation-based definition within Canetti's Universal Composability framework [14] is the gold standard (with the known caveat that the definition is impossible to satisfy in the plain model). Starting with EKE [10], many PAKE protocols have been proposed and rigorously studied over the last three decades (see a recent survey [19]). In 2019, as a result of the IETF standardization competition, CPace [17, 4] and OPAQUE [23] were selected as the winning protocols. Today, PAKE protocols can be found in many places, most notably in electronic passports (PACE), wireless network security protocols such as WPA3-Personal (SAE) and Eduroam (EAP-PWD), Apple Cloud Key Vault (SRP), and IoT network protocol Thread (J-PAKE).

PAPKE. Password-Authenticated Public-Key Encryption (PAPKE) [12] enables secure end-to-end encryption between two entities without relying on a trusted third party or other mechanisms for authentication, such as a PKI. In a nutshell, this primitive is similar to regular Public-Key Encryption (PKE), except that the key generation algorithm takes a password as extra input and produces a secret key and an *authenticated* public key. The encryption algorithm also takes as extra input a password. The essence of this primitive is that ciphertexts are only decryptable if the password used for encryption matches the one used to generate the key pair. Two constructions for PAPKE have been proposed based on the ideal cipher model or random oracle model.
The authors show that a 2-round PAKE protocol can trivially be constructed from PAPKE as follows: (1) the client generates a key pair with their password and sends the authenticated public key to the server; (2) the server samples a random session key and encrypts it with the password previously shared with the client, and sends the ciphertext back to the client; (3) if the client is able to successfully decrypt the ciphertext, both parties agree on a common session key. In our protocol BeePAKE, we leverage this idea and let the server encrypt the session key under the authenticated public key and each sweetword. The number of communication rounds remains unchanged, regardless of the number of decoy passwords. Other building blocks are however necessary to ensure the protocol satisfies the three security properties we aim for.

HoneyPAKE. Becerra et al. [7] were the first to observe the benefits of combining PAKE with the Honeywords methods, for simultaneous authentication, secure-channel establishment, and password file leakage detection. However, in their setting, each client has two passwords (similar to 2FA-type authentication mechanisms). On the server side, only the second password is mixed with $(n-1)$ honeywords. This change significantly simplifies the problem. Namely, a trivial solution to the problem is as follows: a client and a server establish a secure channel using the first password, and only then the client sends the second password (or some function of it) through already established secure channel. This second password is the one that might trigger an alarm if it matches a honeyword. Essentially, the solution is simply replacing the PKI-authenticated channel with a password-authenticated channel established using PAKE protocol. The authors propose a new protocol named HoneyPAKE that follows this strategy with some optimizations. The problem we address in our work is how to construct a PAKE-style protocol that allows for decoy passwords on the server side. In the design of such protocol, we restrict our attention to the setting where the client is only allowed to store a *single* low-entropy password as a long-term secret. We also formalize the security guarantees our construction satisfies, which includes a novel property of false-alarm protection against malicious clients.

Oblivious PAKE. The work of Kiefer and Manulis on Oblivious Password-Authenticated Key Exchange (O-PAKE) [25] is close to the problem at hand. In their work, the client shares *one* password with each server, but the client is allowed to use multiple passwords within an O-PAKE session. The protocol succeeds if and only if one of those input passwords matches the one stored on the server side. Such protocol is useful in scenarios where the client no longer remembers which of their passwords has been registered at a particular server. At first glance, this seems to be exactly the problem we are addressing with SweetPAKE, with the roles of client and server reversed. However, there are a few, but important, differences in the security definitions between O-PAKE and SweetPAKE: (1) O-PAKE password setup is somewhat incompatible with the one from SweetPAKE (see Section 4); (2) SweetPAKE demands two additional properties (i.e. 'sugarword indistinguishability' and 'false alarm protection') that are specific to our use case. Interestingly, in [25], the proposed compiler relies on Index-Hiding Message Encoding (IHME) [28] and a secure PAKE to achieve O-PAKE, which is a completely different approach from the one taken in this paper.

We also propose a UC definition for this primitive, which overcomes the setup incompatible previously mentioned, and therefore unifies both Oblivious PAKE and SweetPAKE (session key and sugarword indistinguishability) definitions.

HPAKE. Li, Wang, and Liang [27] address a similar problem to the one that we independently explore in this work, making it crucial to draw comparisons between their findings and ours. In their paper, the authors introduce a game-based security model to address session key indistinguishability, as well as what we refer to as *sugarword indistinguishability*. However, their approach does not account for false alarm protection against malicious clients, nor does it offer a clear definition within the UC framework. Their proposed construction follows a white-box approach, building upon OPAQUE, which has the advantage of being an *augmented* PAKE. In contrast, our contribution lies in a black-box construction that inherently offers enhanced modularity. Additionally, we establish a parallel with Oblivious PAKE.

Alternative approaches. An interesting line of work, complementary to ours, involves preventing the offline guessing attempts on user's password from the authenticating server by elevating it into a strong secret using an auxiliary service or device [6, 22]. Moreover, it is noteworthy that significant

efforts for alternatives to passwords are underway with passwordless standards such as WebAuthn[3] gaining prominence, particularly with the widespread adoption of the term *passkeys* by various industry players. WebAuthn relies on high-entropy public-keys, which are public in nature. Despite these ongoing developments, given their human-memorable nature and widespread use, passwords are far from being deprecated and are likely to remain relevant in many contexts [11].

2 SweetPAKE Security Model

Security definitions for PAKE have been extensively studied in the literature and fall into either game-based or simulation-based categories. The most widely accepted game-based definition is the Real-or-Random model by [3], which is an extension of the Find-then-Guess (FtG) model by [9]. The popular choice for simulation-based definitions is that of [14], outlined in the Universal Composability (UC) framework of [13].

2.1 Model considerations

In this paper, we propose three game-based security properties for SweetPAKE building upon the Real-or-Random (RoR) PAKE model from [3] and two other related works [25, 7]. We select the RoR PAKE model variant over the original FtG as it has been shown to provide tighter compositional properties with higher-level protocols that make use of the session key [30]. In our view, game-based definitions convey security guarantees in simple terms and are generally easier to manage from a provable security standpoint. In case of SweetPAKE, game-based definitions and corresponding proofs provide modularity by allowing the reader to easily pinpoint which building blocks are required to achieve each of the three proposed security notions. However, the main drawback often pointed out is that game-based definitions fail to capture arbitrary correlations between passwords – in game-based models, clients' passwords are sampled independently and uniformly at random[4] from a password dictionary by the challenger. In contrast, PAKE definitions within the UC framework result in a stronger notion and successfully captures the scenario where clients register related passwords with different servers. Furthermore, it also ensures security under arbitrary protocol composition.

UC definition is the gold standard for PAKE and arguably better suited for capturing arbitrary password correlations, which often occur in practice and moreover in the context of decoy passwords (depending on the adopted method [24]). Therefore, we extend our security analysis to the Universal Composability framework to leave no stone unturned.

In line with the above discussion, and starting with our game-based models, we will assume that the underlying SweetPAKE initialization procedure provides the flatness of sweetwords within each initialized client's vector and the independence between sugarwords in the system. Whilst the first assumption is addressed by the seminal work of Juels and Rivest [24], the second assumption is standard for corruption models captured by game-based definitions [9, 3], where passwords are uniformly sampled or chosen from a distribution with min-entropy κ.

How to generate honeywords that look like they could be real passwords is an interesting problem addressed by [24]. Employing the terminology of the authors, this property of the honeyword generation procedure is called *flatness*.

[3] `https://www.w3.org/TR/webauthn-2/`.

[4] Although various authors adopting game-based PAKE modelling use the uniform password distribution to simplify the security model and analysis, other distributions could be considered as in [2], as long as passwords are still sampled independently and carry enough entropy for meaningful security.

Definition 1 (Flatness.). *Let* Flat *be a honeywords generation procedure that takes as input a secret sugarword* pw *and a public parameter* n *representing the number of sweetwords in the game and outputs a randomly permuted array* $\mathbf{P}[n]$ *containing the sugarword and* $(n-1)$ *honeywords. To win the game, an adversary* $\mathcal{A}$ *with access to* $\mathbf{P}[n]$ *must guess the index of the sugarword* pw *in it. The honeywords generation procedure* Flat *is perfectly flat, if for any adversary against the game above, the inequality*

$$\mathsf{Adv}^{flatness}_{\mathcal{A},\mathsf{Flat}}(n) = \frac{1}{n}$$

holds, where the probability is taken over the choice of sugarword pw*, honeywords generation procedure* Flat*, and any random coins used by the adversary to produce its guess.*

As pointed out in [24], one good way of generating a sugarword and honeywords is to first generate a list of n sweetwords, and then randomly pick one element of this list to be the sugarword. This becomes the real password, and the remaining elements the honeywords. In more general terms, our game-based model captures any password setup procedure in which sugarwords are picked independently from each other and the honeywords generation procedure is flat, while our UC formulation handles arbitrary correlations of sweetwords/passwords, providing strong security guarantees.

2.2 SweetPAKE architecture

Although similar to PAKE in many aspects, SweetPAKE has several differences that must be incorporated into the security model. To pinpoint exactly what these differences are and what one might expect of SweetPAKE protocol in terms of functionality and security guarantees, let's first abstract the architecture of a system where SweetPAKE is run to establish session keys for secure communication between clients and servers.

A client $C \in \mathbb{C}$ has a password $pw_{C,S}$ for a server $S \in \mathbb{S}$. The server S has a list of n sweetwords for client C. One element on this list matches password $pw_{C,S}$. The index i, where the password $pw_{C,S}$ matches the i-th element of S's list of sweetwords for C, is stored with the hard-to-corrupt Honeychecker. We assume that the initial setup is carried out with the help of a registration service that generates $(n-1)$ honeywords, picks a random index $i \in \{1..n\}$, creates a list with the generated honeywords and the client's password placed in position i and sends the list to the server and the index to the Honeychecker.

We also assume that secure channels exist everywhere except between the client and the server. Here, we analyse the strongest variant of those proposed in [7], in which the client has a single password per server, there's no setup of pre-shared keys, seeds or nonces, and communication between the server and the Honeychecker is unidirectional from the server to the Honeychecker. Thus, the goal of the SweetPAKE protocol is to establish session keys between the client and server with mutual authentication, and for the server to be able to identify the index of the client's password so that it can be forwarded to the Honeychecker. The overall architecture is depicted in Figure 1.

2.3 Game-based security

Informally, there are three security properties we would like to capture for SweetPAKE protocols: (1) indistinguishability of session keys, similar to regular PAKE protocols; (2) difficulty of spotting the sugarword (real password) among sweetwords; (3) difficulty of raising a false alarm to the Honeychecker. To model all three properties, we use a standard PAKE model from [3], and we highlight that Test query is relevant only to the indistinguishability of the session keys property. Furthermore, instead of a single password per client, servers store a list of sweetwords per client –

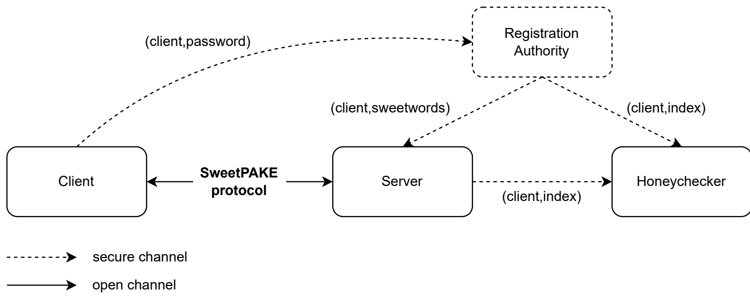

Figure 1: SweetPAKE architecture.

one of which is the sugarword. In our model, we consider a setting where multiple clients and servers run many protocol executions of SweetPAKE, possibly in parallel.

Augmented PAKE (aPAKE) protocols allow servers to hold a one-way hash of the password instead of the password in clear. This serves to impede potential adversaries' attempts at swiftly recovering the password (and impersonating the client) if the server's security is ever compromised. In the context of *strong* augmented PAKE (saPAKE) [23], safeguarding against pre-computation attacks is also required. In our model, we exclusively focus on extending the balanced PAKE scenario (also known as symmetric PAKE) to the setting where the server possesses multiple passwords per client, stored in clear. We do not deal with the kind of asymmetry between client and server known as augmented PAKE[5].

There are established generic compilers from *balanced* PAKE to *augmented* PAKE [15, 20], and further from *augmented* PAKE to *strong augmented* PAKE [23]. However, this work does not explore whether these compilers can be applied to SweetPAKE. Detecting password file leakage is an orthogonal problem to slowing down password recovery once leakage occurs. It could even be argued that measures taken to slow down password recovery might inadvertently prolong the detection of such leaks.

Protocol participants. A party in a SweetPAKE protocol is either a client $C \in \mathbb{C}$ or a server $S \in \mathbb{S}$. The set of all parties $\mathbb{U}$ is the union $\mathbb{C} \cup \mathbb{S}$. Each party might run multiple protocol execution instances. We denote by U^i the i-th execution of party U. We consider the Honeychecker to be a passive protocol participant as it only receives a client identity and an index from the server, via a secure channel, while its further actions are separate from the SweetPAKE protocol. The Honeychecker cannot be corrupted within our model.

Setup. Each pair of parties (C, S) share a secret password $pw_{C,S}$. On the server side, this secret is mixed with $(n-1)$ honeywords. The value of n is a fixed security parameter of the protocol. Games are initialized by a challenger that for each pair $(C, S) \in \mathbb{C} \times \mathbb{S}$ first samples the sugarword $pw_{C,S}$ independently and uniformly at random from dictionary $\mathbb{D}$ and then applies a function Flat on input $(pw_{C,S}, n)$. The function Flat outputs a randomly permuted array $\mathbf{P}_{C,S}[n]$ containing the sugarword and $(n-1)$ honeywords. An index i that contains the sugarword position in $\mathbf{P}_{C,S}[n]$ is then forwarded to the Honeychecker. We assume that Flat generates honeywords that are flat with respect to the sugarword.

[5]Note that in the case of symmetric PAKEs and SweetPAKE, it is indeed technically feasible for the server to store password hashes, given that the client also hashes its password before initiating the protocol. However, the resulting protocol wouldn't offer the guarantees provided by an augmented PAKE, such as requiring the client to know the actual password (not just the password hash), or safeguarding against pre-computation attacks.

Once the challenger completes the sampling, the server S is associated with a bi-dimensional array (of n sweetwords per client) and the client C with a list of passwords (one password per server). For an algorithmic description of this initialization procedure, see Figure 2. Notice that upon initialization, S is unaware of which sweetwords are sugarwords, but after a legitimate interaction between C and S, the server S will learn the sugarword of the client C because the index is an output of the protocol on the server's side.

```
Initialize:

for  S ∈ 𝕊  //  for each server
     for  C ∈ ℂ  //  for each client
          pw_{C,S} ← 𝔻  //  pick a sugarword
          P_{C,S}[n] ← Flat(pw_{C,S}, n)  //  create sweetwords
          lookup  i  such that  P_{C,S}[i] = pw_{C,S}
          forward sugarword position  i  to Honeychecker
```

Figure 2: Initialization procedure common to all games.

Party instance state. Each party instance U^i holds a state that the party keeps updating during the protocol execution. It takes the form of ($U^i.pid$, $U^i.trace$, $U^i.internal$, $U^i.key$, $U^i.accept$) where

- $U^i.pid$ is the partner identifier of U^i, which is initially $\perp$ and remains so until U^i starts running the protocol;
- $U^i.trace$ is the communication transcript of U^i, and when U^i terminates, $U^i.trace$ containing the full or a partial communication transcript is used as the session identifier, made explicit by the protocol;
- $U^i.internal$ is the internal state of U^i and might hold secrets unique to U^i (such as secret exponents);
- $U^i.key$ is the session key of U^i, which remains $\perp$ until the party instance U^i accepts;
- $U^i.index$ is the password index U^i, which remains $\perp$ until the party instance U^i accepts. This is only relevant if $U \in \mathbb{S}$, otherwise it stays $\perp$ forever;
- $U^i.accept$ is a boolean flag that keeps track of the acceptance state of U^i. For simplicity, we assume that instance U^i terminates once $U^i.accept = true$.

Correctness. An untampered protocol execution between a client instance C^i (holding password $pw_{C,S}$) and a server instance S^j (holding a list $\mathbf{P}_{C,S}$ of sweetwords for C) results in C^i and S^j accepting, becoming partners and outputting the same key $C^i.key = S^j.key$ if $pw_{C,S} \in \mathbf{P}_{C,S}$. Furthermore, S^j also outputs an index i such that $\mathbf{P}_{C,S}[i] = pw_{C,S}$. As with the Honeywords method, [24], when a legitimate client authenticates, the server learns the correct index.

Partner instances. A client instance C^i and a server instance S^j are partnered if both instances have terminated in accepting state with (a) identical session identifiers $C^i.trace = S^j.trace$, and (b) matching partner identifiers $C^i.pid = S$ and $S^j.pid = C$. The correctness implies key equality. We emphasize that client instances are never partnered with each other, nor are server instances.

Adversary model. We model the adversary $\mathcal{A}$ as a PPT algorithm. After the password initialization procedure, the challenger proceeds by generating global parameters, and Common Reference String (CRS) if applicable, as described by the SweetPAKE protocol specifications - these values are passed on to the adversary $\mathcal{A}$. From thereon, the interaction between the challenger and the adversary occurs via oracles, provided by the challenger and queried at will by the adversary, essentially modelling the adversary's capability in real world systems. Common to all three SweetPAKE security games we discuss below are the following oracles:

- $\mathsf{Execute}(C^i, S^j)$: This oracle models an honest execution of the protocol between a client instance C^i and a server instance S^j. The transcript of the protocol containing all messages exchanged between the parties is output to the adversary.
- $\mathsf{Send}(U^i, m)$: This oracle models an active attack, in which the adversary is able to send a message m of its choice to party instance U^i. Recall that in [3] model, the adversary is capable of dropping, injecting, and modifying protocol messages at will. The message that party instance U^i would output upon receiving message m is passed on to the adversary.
- $\mathsf{Reveal}(U^i)$: This oracle models the capability of corrupting a session key. When queried on U^i, this oracle outputs the session key of U^i. For a meaningful definition, if the game has a Test oracle, $\mathsf{Reveal}(U^i)$ outputs $\perp$ in case U^i or its partner (if there is one) was previously queried to Test.
- $\mathsf{Corrupt}(C, S)$: This oracle models the corruption of a client and outputs the password that client C registered with server S. This query is essential to capture perfect forward secrecy (PFS) since session keys should not be compromised even if long-term passwords used in the protocol are compromised.

I. Real-or-Random indistinguishability of the session key. In this game, the adversary is challenged to distinguish between real and random session keys. To do so, it has at its disposal a Test oracle that when queried, outputs a key. Here, we adopt the extension from [3] that allows the adversary to place multiple Test queries. For a meaningful definition – and following the terminology commonly employed in PAKE – we restrict this oracle to only output a key when queried on a *fresh* instance. Let's define what this entails.

Freshness: Party instance U^i is fresh if (a) it terminated in accepting state, (b) it was not queried to Reveal or Test, and (c) in case of having exactly

- 0 partner instances: $\mathsf{Corrupt}(C, S)$, where $C = U$ or $S = U$, was not called prior to U^i accepting;
- 1 partner instance: neither Reveal or Test was queried to the partner instance;
- 2 or more partner instances: no further conditions are imposed.

Oracle specific to RoR SweetPAKE security game:

- $\mathsf{Test}(U^i)$: When the game is initialized, a bit b is sampled. If $b = 0$, this oracle outputs the real session key for party instance U^i, otherwise it outputs a random key of the length of the real key. This oracle provides a challenge to the adversary and can only be queried on *fresh*

party instances. If party instance U^i is not fresh, this oracle outputs $\perp$. All Test queries are answered according to the same challenge bit b. Once the adversary places a query on U^i to this oracle, it is restricted to call Reveal on U^i or its partner.

Eventually the adversary terminates and outputs a guess bit b'.

Definition 2. *A SweetPAKE protocol* Π *has session keys indistinguishable from random, if for any PPT adversary against the game above, the inequality*

$$Adv^{RoR}_{\mathcal{A},\Pi}(\lambda, n) \stackrel{\text{def}}{=} |\Pr[b = b'] - \frac{1}{2}| \leq \frac{n \cdot q_s}{|\mathbb{D}|} + \epsilon(\lambda)$$

holds for negligible term $\epsilon(\lambda)$ *and* n *sweetwords per pair* (C, S)*, where* $|\mathbb{D}|$ *is cardinality of the password dictionary and* q_s *is an upper-bound on the number of queries to* Send *oracle.*

In essence, we require the best course of action of adversary to be guessing one of the n sweetwords associated with some pair (C, S).

II. Sugarword indistinguishability. Unfortunately, from time to time servers get compromised and some passwords are leaked. This game aims to capture the difficulty of guessing the sugarword among sweetwords, even for adversaries that have some control over the network.

Oracle specific to sugarword indistinguishability security game:

- $\mathsf{Leak}(S, C)$: This oracle models a partial compromise of a server's passwords authentication file. When queried, it outputs $\mathbf{P}_{C,S}$ to the adversary, which contains all sweetwords associated with the client C at the server S. Because now the adversary has all candidate passwords for C at S, it is restricted from querying $\mathsf{Send}(C, \cdot)$. Otherwise, it is trivial to test out all sweetwords and figuring out which one is the sugarword. In the real world, clients are usually the ones initiating the communication and rarely are the systems designed otherwise. Therefore, we argue that this minimal and inevitable restriction we impose in our model brings out a meaningful security definition.

Eventually the adversary terminates and outputs a pair (C, S) and a guess index i. The adversary is admissible if $pw_{C,S}$ was not revealed via $\mathsf{Corrupt}$ query.

Definition 3. *A SweetPAKE protocol* Π *is sugarword indistinguishable, if for any admissible PPT adversary against the game above, the inequality*

$$\begin{aligned} Adv^{SIND}_{\mathcal{A},\Pi}(\lambda, n) &\stackrel{\text{def}}{=} \Pr[\mathbf{P}_{C,S}[i] = pw_{C,S}] \\ &\leq \frac{1}{n} + \frac{n \cdot q_s^*}{|\mathbb{D}|} + \epsilon(\lambda) \end{aligned}$$

holds for negligible term $\epsilon(\lambda)$ *and* n *sweetwords per pair* (C, S)*, where* $|\mathbb{D}|$ *is cardinality of the password dictionary and* q_s^* *is an upper-bound on the number of inquiries to* $\mathsf{Send}(C, \cdot)$ *oracle before the leak.*

This means that no adversary can do better than testing n passwords per interaction with C before the leak. Thus $(n \cdot q_s^*)/|\mathbb{D}|)$ term, and randomly guessing the index of the sugarword, which yields a successful guess with probability $1/n$.

III. False alarm protection. From the correctness of SweetPAKE, a server instance accepts not only with the real password of the client, but also with the associated honeywords. In such a case, Honeychecker would trigger the alarm. The third property we want to capture is the difficulty of an adversary triggering the alarm without the password file being leaked (or partially leaked) – this is in fact a *false* alarm. In this game, the adversary has access to the usual oracles and its goal is to originate such an event that we (unimaginatively) name *Event E*. Note that oracles Test from game RoR and Leak from game SIND are not needed here and deliberately not included.

Event E: An adversary queries $\mathsf{Send}(S^j, \cdot)$ and as a result party instance S^j accepts outputting a pair (C, i) to Honeychecker, which corresponds to a honeyword, i.e.,

$$\mathbf{P}_{C,S}[i] \neq pw_{C,S}.$$

Definition 4. *A SweetPAKE protocol* Π *offers false alarms protection if for any PPT adversary* $\mathcal{A}$ *against game FAP the inequality*

$$Adv^{FAP}_{\mathcal{A},\Pi}(\lambda, n) \stackrel{\text{def}}{=} \Pr[E] \leq \frac{q_s^{\diamond} \cdot (n-1)}{|\mathbb{H}|} + \epsilon(\lambda) \tag{1}$$

holds for negligible term $\epsilon(\lambda)$ *and* n *sweetwords per pair* $(C, S) \in \mathbb{C} \times \mathbb{S}$*, where* $|\mathbb{H}|$ *is cardinality of the honeyword dictionary induced by* $\mathsf{Flat}(pw_{C,S}, n)$ *over the sugarword and parameter* n*, and* $q_s^{\diamond}$ *is an upper-bound on the number of inquiries to* $\mathsf{Send}(S, \cdot)$ *oracle for any* $S \in \mathbb{S}$*.*

This means that no adversary can do better than testing $(n-1)$ honeywords per interaction with the server in the absence of a password file leak.

2.4 UC security

We propose a new UC definition for SweetPAKE, which reconciles the divergences in password sampling between SweetPAKE and O-PAKE. This is because, in the UC framework, passwords are arbitrarily chosen by the environment $\mathcal{Z}$. The functionality described in Figure 3 primarily reflects the security aspects of SweetPAKE's session key indistinguishability (but also extends to sugarword indistinguishability as we will discuss in a moment) and the security guarantees provided by O-PAKE protocols. This definition draws inspiration from the PAKE definition presented in [14] while incorporating the minimal necessary extensions to account for the protocol asymmetry in that one party holds 1 password whereas the other party holds n passwords. In more detail, the adversary might place a $\mathsf{TestPwC}$ to test n passwords against 1, or a $\mathsf{TestPwS}$ query to test 1 password against n. As with the ideal functionality for PAKE, if the adversary succeeds, the record is marked as $\mathsf{compromised}$ and the adversary is given the freedom to set the session key via a NewKey query.

In the UC framework, the selection of passwords is carried out by an environment $\mathcal{Z}$, which maintains a permanent communication channel with the adversary $\mathcal{A}$. This arrangement inherently models scenarios where honest parties' passwords may become compromised at any point during the protocol execution. It is important to note that the environment $\mathcal{Z}$ can expose various levels of information to the adversary $\mathcal{A}$, such as a client password, the complete set of passwords associated with a server, or even just a subset of those.

Perhaps the most notable benefit of security definitions within the UC framework is the ability to capture arbitrary password correlations, in contrast to the limitations imposed by game-based definitions, in which the challenger samples passwords according to some fixed distribution (usually the uniform distribution over the dictionary). This models practical use-cases where clients register the same or similar passwords with multiple servers, or typing errors when asked to input their

1. **New Session Client**. On input $(\mathsf{NewSessionC}, sid, \mathcal{P}_i, \mathcal{P}_j, pw)$ from party $\mathcal{P}_i$:
   - Ignore this query if record $(sid, \mathcal{P}_i, \cdot, \cdot, \cdot, \cdot, \cdot)$ already exists.
   - Otherwise, record $(sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{fresh}, pw, \perp, client)$ and send $(\mathsf{NewSessionC}, sid, \mathcal{P}_i, \mathcal{P}_j)$ to $\mathcal{A}$.

---

2. **New Session Server**. On input $(\mathsf{NewSessionS}, sid, \mathcal{P}_i, \mathcal{P}_j, \mathbf{P}_{C,S})$ from party $\mathcal{P}_i$ where $\mathbf{P}_{C,S} \in \mathbb{D}^n$:
   - Ignore this query if record $(sid, \mathcal{P}_i, \cdot, \cdot, \cdot, \cdot, \cdot)$ already exists.
   - Otherwise, record $(sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{fresh}, \mathbf{P}_{C,S}, \perp, server)$ and send $(\mathsf{NewSessionS}, sid, \mathcal{P}_i, \mathcal{P}_j)$ to $\mathcal{A}$.

---

3. **Password Guess on Client**. On input $(\mathsf{TestPwC}, sid, \mathcal{P}_i, \mathbf{P}^*_{C,S})$ from adversary $\mathcal{A}$:
   - Retrieve the record of the $(sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{fresh}, pw, \perp, client)$ (abort if no such record exists).
   - If $pw = \mathbf{P}^*_{C,S}[\cdot]$, update the record to $(sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{compromised}, pw, \perp, client)$, and send $(\mathsf{TestPwC}, sid, correct)$ to $\mathcal{A}$.
   - Else, update record to $(sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{interrupted}, pw, \perp, client)$ and send $(\mathsf{TestPwC}, sid, wrong)$ to $\mathcal{A}$.

---

4. **Password Guess on Server**. On input $(\mathsf{TestPwS}, sid, \mathcal{P}_i, pw^*)$ from adversary $\mathcal{A}$:
   - Retrieve the record of the $(sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{fresh}, \mathbf{P}_{C,S}, \perp, server)$ (abort if no such record exists).
   - If $pw^* \in \mathbf{P}_{C,S}$
     - Initialize vector $\mathbf{M} = [0] * n$
     - $\forall m \in \{1..n\}$ s.t. $\mathbf{P}^*_{C,S}[m] = pw$, mark $\mathbf{M}[m] = 1$
     - Update the record to $(sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{compromised}, \mathbf{P}_{C,S}, \perp, server)$
     - Send $(\mathsf{TestPwC}, sid, correct\ , \mathbf{M}\ )$ to $\mathcal{A}$.
   - Else, update record to $(sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{interrupted}, \mathbf{P}_{C,S}, \perp, server)$ and send $(\mathsf{TestPwS}, sid, wrong)$ to $\mathcal{A}$.

---

5. **Session Key**. On input $(\mathsf{NewKey}, sid, \mathcal{P}_i, k^*)$ from adversary $\mathcal{A}$ where $|k^*| = \lambda$:
   - Retrieve record $(sid, \mathcal{P}_i, \mathcal{P}_j, status, p, k, r)$ for $status \in \mathsf{fresh}, \mathsf{interrupted}, \mathsf{compromised}$, abort if no such record exist.
   - If $status = \mathsf{compromised}$ or either $\mathcal{P}_i$ or $\mathcal{P}_j$ is corrupted, set $k \leftarrow k^*$.
   - If $status = \mathsf{fresh}$ and there exists a record $(sid, \mathcal{P}_j, \mathcal{P}_i, \mathsf{completed}, \cdot, k', r')$ such that $r \neq r'$ and $status$ of $\mathcal{P}_j$ switched from fresh to $\mathsf{completed}$ before it received $(\mathsf{NewKey}, sid, k')$, set $k \leftarrow k'$.
   - Else set $k \leftarrow\!\!\$\ 0, 1^\lambda$.
   - Update the record to $(sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{completed}, p, k, r)$, and output $(\mathsf{NewKey}, sid, k)$ to $\mathcal{P}_i$.

Figure 3: SweetPAKE/O-PAKE ideal functionality $\mathcal{F}_{\mathsf{OPAKE}}$. To capture sugarword indistinguishability, remove the gray box in TestPwS.

password. We also note that since we are in the symmetric PAKE setting, passwords are stored in clear on the server side. This means that our ideal functionality does not need an interface to steal the password file as the password file contains exactly the passwords input by the environment $\mathcal{Z}$ upon new session calls to the ideal functionality[6].

An important observation is that trivial protocols such as one that simply uses PAPKE to encrypt individually each password, without further modifications to the protocol, reveals to the decryptor the index of the correct password. For this vanilla protocol to be provably secure, the functionality must also leak the indices of matching passwords. To see why, consider an environment $\mathcal{Z}$ that samples n *high-entropy* passwords, initializes party P_i with these, randomly selects one of the passwords and gives it to its real-world adversary $\mathcal{A}$ to interact with P_i. Real-world adversary $\mathcal{A}$ is the decryptor and therefore figures out the index of the given password, and sends it to $\mathcal{Z}$. Finally $\mathcal{Z}$ outputs 1 if the index corresponds to the password randomly picked. For such environment $\mathcal{Z}$, there is no simulator capable of simulating the real-world without receiving the index from the functionality. In fact, the functionality must leak *all* indices of matching passwords, has the environment might place repeated passwords when initializing P_i. Note, however, that this is only relevant when testing 1 password against n. In the reverse direction, the simulator can always compute itself which indices match by querying one password at the time and placing place holders $\perp$ in the other $n-1$ password slots.

To capture sugarword indistinguishability as well, we simply lift from $\mathsf{TestPwS}$ the leakage of indices matching the queried password (see Figure 3). The vanilla protocol (plus its sugarword indistinguishability extension) is depicted in Figure 5. To be able to capture sugarword indistinguishability, the encryptor must remove duplicate passwords (and replace them with place holders $\perp$), and shuffle the vector of ciphertexts. Intuitively, deduplication ensures there is at most 1 match. Shuffling renders the matching index irrelevant.

It is a deliberate choice not to try to capture *false alarm protection* within our definition as this would require changing the syntax of the protocol to also output an index (in the SweetPAKE setting, this is the index the server sends to the Honeychecker), which is not relevant in the Oblivious PAKE setting. To the best of our knowledge, this is the first UC formulation of O-PAKE primitive.

3 BeePAKE: A game-based secure SweetPAKE protocol

In this section, we first review why simple composition of secure Password Authenticated Key Exchange (PAKE) protocol executions is not adequate for building SweetPAKE. This exercise will give an insight into design choices we made in Section 3.2. Then, we present a practical construction that securely realizes SweetPAKE primitive as defined in Section 2.

3.1 Naive proposal

Let's consider a simple (but less efficient) way to build SweetPAKE. Suppose we select a forward-secure PAKE protocol with key confirmation[7] such as a variant of SPAKE2 from [1].

---

[6] In the *augmented* setting, when the environment Z registers a password, in the real world this password is not stored in clear on the server side. Corrupting the server still grants access to the passwords, but compromising the password file with the hashes has meaningful practical implications too. Such compromise subsequently allows the attacker to (1) impersonate the server to the client, and (2) find the password via an offline dictionary attack. There is a practical difference between corrupting the server and seeing the passwords in clear, and compromising the password file with the hashes.

[7] Note that the explicit authentication, using for instance a secure MAC, allows an authentication server to identify which PAKE session is the correct one and appropriately submit the index to the Honeychecker.

In case there are n sweetwords stored per user on the authentication server S, a client C could initiate n independent PAKE executions with S. Perhaps more sensibly, the Client should bundle n instances into 1, to minimize the number of message flows. For each PAKE execution, C must generate a distinct Diffie-Hellman key share using the *same* password but *different* ephemeral exponents to be in line with the standard PAKE security definitions (both UC and game-based). To each message, the server responds by executing the SPAKE2 protocol with a different password from the list of sweetwords for the client C, and a distinct ephemeral exponent – n key confirmations are also produced. The client C then checks which key confirmation is valid and sends their own key confirmation. Finally, The server S identifies the sugarword with the client's key confirmation.

Remark 1. The main drawback of our naive protocol is the computational burden imposed on the client side. Since all key shares coming from the client use the same password, one may try to optimize client computation and reuse the same key share in each of the n instances. However, note that both RoR-secure and UC-secure PAKE are insufficient to guarantee security in case of message repetitions (e.g. $X = g^x M^{pw}$ in SPAKE2) over multiple PAKE sessions.

Remark 2. Sugarword indistinguishability is not satisfied by our naive protocol without taking additional care. More specifically, with the intention to improve efficiency, it would be natural to stop the execution of this naively built SweetPAKE as soon as the correct PAKE session is identified of the client side. However, in a sequential execution of multiple instances, this leaks the index of the sugarword. Even if the n sessions are bundled together, timing attacks are still of concern. Active adversaries have another attack vector: intercept and repeat messages (e.g. repeating n times the same messages from S to reveal if it contains the sugarword). Looking ahead, the shuffling technique introduced in our BeePAKE protocol could be adapted to the naive construction as well.

3.2 Secure SweetPAKE construction

Here we introduce an efficient and conceptually simple construction that satisfies all three SweetPAKE gamed-based security properties defined in Section 2. This generic construction that we call BeePAKE can be seen as a compiler that takes four primitives (see Appendix B for the details) and combines them into a secure SweetPAKE: (a) a UC-secure password authenticated public-key encryption algorithm from [12], (b) secure PRF, (c) random permutation RP, and (d) secure MAC algorithm. The resulting construction offers standard model security if underlying primitives are standard model secure[8].

BeePAKE compiler. On a high level our compiler (see Figure 4) works as follows. A client C and a server S first run PAPKE with multiple-ciphertexts[9], each corresponding to one of n sweetwords stored by the server S for the client C. For each ciphertext, the server encapsulates a fresh secret key that is derived from an initial secret seed using a fixed-length output pseudorandom function PRF. Before being sent out, the vector containing ciphertexts is shuffled to hide the order of passwords within the password file. After successful decryption, the client expands the obtained fresh secret using PRF to achieve client-to-server explicit authentication and establish a secure channel with the server. In addition, the client provides an ephemeral index carrying information about successful ciphertext decryption, thus reducing the amount of work on the server side. As a result of a successful BeePAKE execution both the client and the server jointly share a high-entropy session key.

[8]Unfortunately, the two existing PAPKE instantiations from [12] rely on idealized assumptions.
[9]The security model of PAPKE admits multiple ciphertexts per authenticated public key from the same encryptor.

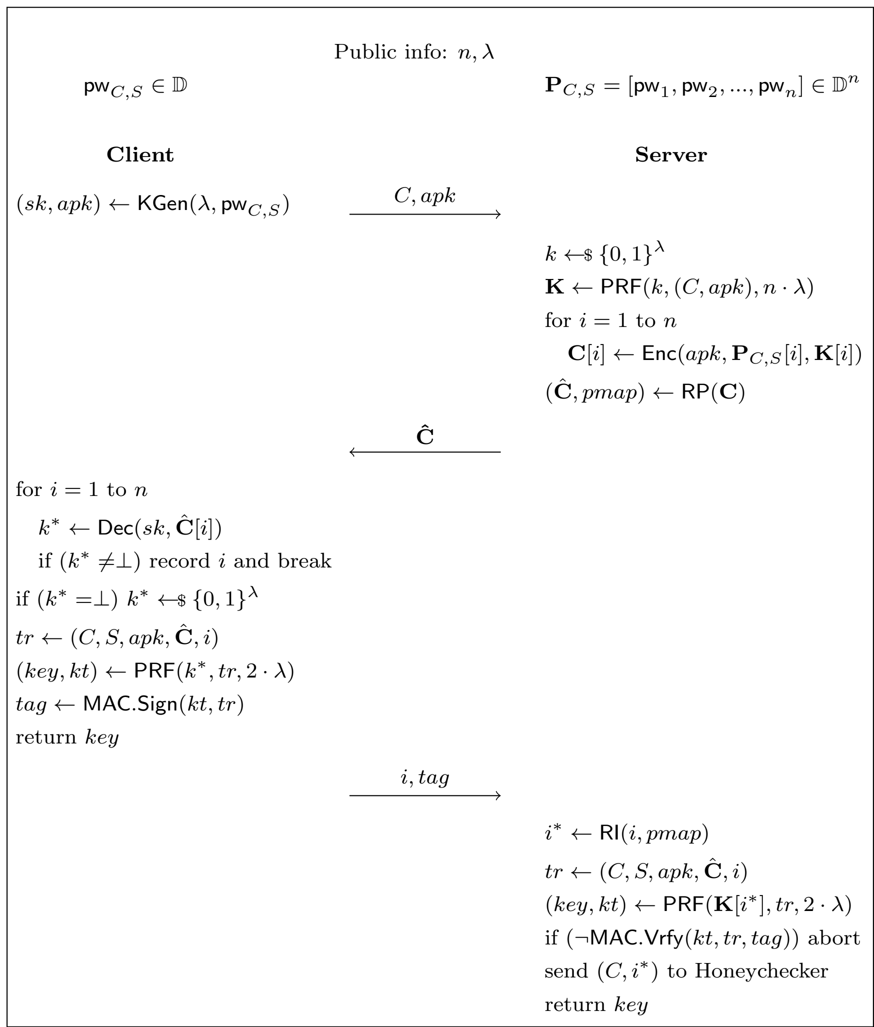

Figure 4: BeePAKE protocol. Random permutation RP can be inverted by RI with auxiliary mapping information *pmap*.

Protocol description. First, a client C wishing to authenticate computes a public key *apk* (authenticated with the password) obtained as the output of PAPKE's key generation algorithm $\mathsf{KGen}(\lambda, pw_{C,S})$ and sends it (together with its identity C) to a server S.

Second, the server S generates a random seed k, inputs it into $\mathsf{PRF}(k, (C, apk), n \cdot \lambda)$ and as a result obtains a vector of keys $\mathbf{K}$ of size $n \cdot \lambda$ bits. Next, the server encrypts the keys stored in $\mathbf{K}$ one at a time[10] by feeding the client's authenticated public key *apk* together with a password candidate pw_i from the password file $\mathbf{P}_{C,S}$ into PAPKE's encryption algorithm $\mathsf{Enc}(apk, \mathbf{P}_{C,S}[i], \mathbf{K}[i])$. The server, then, shuffles the vector of ciphertexts $\mathbf{C}$, sends a resulting vector $\hat{\mathbf{C}}$ to the client, and stores the mapping between $\mathbf{C}$ and $\hat{\mathbf{C}}$ denoted as *pmap*.

Third, upon receiving the server's message, the client checks the format and size of the vector

[10] The PAPKE encryption procedure can be executed concurrently.

$\hat{\mathbf{C}}$ and, if valid, goes through it by trying to decrypt each ciphertext using PAPKE's decryption algorithm $\mathsf{Dec}(sk, \hat{\mathbf{C}}[i])$. If no ciphertext decryption from $\hat{\mathbf{C}}$ is successful, the client aborts. Otherwise, upon successful decryption, the client retrieves the plaintext containing the secret value k^*, and stores an index i of the well-formed ciphertext within $\hat{\mathbf{C}}$. Once a valid ciphertext is found, the client aborts the decryption procedure[11]. Then C uses $\mathsf{PRF}(k^*, tr, 2 \cdot \lambda)$ to expand k^* into two keys (key, kt). The key kt is used to compute the tag tag over the transcript of the first two message rounds and index i. Finally, C sends tag together with index i to the server.

Fourth, the server inverts the random permutation RP using RI with auxiliary mapping information $pmap$, thus obtaining the true index of the sugarword i^*. This helps the server to retrieve the corresponding secret $\mathbf{K}[i^*]$ and to derive the same key pair (key, kt) as the client. Then, the server checks tag. If tag verification is successful, the server forwards (C, i^*) to the Honeychecker and sets key to be its session key with C. Otherwise, the server aborts. In the background, Honeychecker checks whether the pair (C, i^*) is valid and acts accordingly.

Design choices. The main idea behind this construction is to rely on the stronger PAPKE primitive instead of PAKE. Thereby, a SweetPAKE server may use the same authenticated public key coming from a client for multiple encryptions using different passwords as input to the encryption algorithm from PAPKE. This approach significantly reduces the size of the client's first message compared to the naive approach and minimizes computational effort on the client side. Also, it allows us to prove optimal security bounds for all three security properties. Note that we use PAPKE in PAKE mode, meaning that we only encrypt secret keys – this keeps the size of the ciphertext vector $\hat{\mathbf{C}}$ small. We reduce the computation on the server side by sending a temporary index in the last client's message without jeopardizing sugarword indistinguishability.

The random permutation RP thwarts timing attacks against sugarword indistinguishability. Thus, a client can stop the decryption of vector $\hat{\mathbf{C}}$ as soon as it finds a valid ciphertext.

To achieve false alarm protection against an adversary that knows the sugarword (either an insider or due to sugarword compromise), we encrypt a different secret key for each sweetword (on the server side). This approach, together with the use of secure PRF and MAC, prevents the adversary (on the client side) from creating a valid tag for a wrong index. To generate a vector $\mathbf{K}$, we only sample a seed k that we subsequently expand using PRF, instead of sampling n session keys separately. This choice reduces the number of random bits needed for the protocol execution.

A session key in SweetPAKE will be accepted and the session successfully terminated from the server side in case an adversary or (an honest user accidentally) submits any sweetword. In case it's a honeyword, it's up to the Honeychecker to either raise a silent alarm to the system administrator or directly take action, dropping the connection and preventing further communication.

On a final note, we would like to refer the reader to Appendix A for a performance comparison of our BeePAKE protocol against the bundled multi-session SPAKE2 described above. For this analysis, we focus on the computational cost of both client and server, and the total bandwidth required to execute the protocols. For an estimation of the computational cost, we counted the most costly operations, which are likely to impact the overall performance: group exponentiation, group multiplication, hash into group, ideal cipher on groups (which largely depends on the groups and IC instantiation [29]). For concrete run time and bandwidth requirements of both protocols, we instantiated BeePAKE and SPAKE2 over Curve25519. Furthermore, BeePAKE is instantitated with the PAPKE-IC-DHIES [12] variant, IC with Rinjdael-256, and AE with AES-128-GCM, for an overall security parameter of $\lambda = 128$. As expected, our BeePAKE shines in terms of the client's computational effort, and compares favorably in terms bandwidth, without impacting the server's

[11]Note that this optimization does not enable timing attacks against sugarword indistinguishability property since $\hat{\mathbf{C}}$ is previously shuffled using a random permutation.

computational cost.

3.3 Security analysis of BeePAKE protocol

In this section, we show that BeePAKE protocol described in Figure 4 satisfies three security notions defined for SweetPAKE. To prove this, we will assume that underlying PAPKE primitive UC-realizes PAPKE functionality $\mathcal{F}_{\mathsf{PAPKE}}$ from [12]. Note that our SweetPAKE definition admits correlations between sweetwords coming out of Flat algorithm. As a consequence, game-based PAPKE definitions from [12] (i.e. IND-CCKA and AUTH-CTXT) are insufficient for our purpose. Therefore we need to rely on the security provided with UC PAPKE. We resolve a syntax issue between UC and game-based definitions[12] using dummy session identifiers as done in [14] and [12].

3.3.1 Session key indistinguishability of BeePAKE

In Theorem 1 we establish indistinguishability of the session keys (including forward secrecy) for the BeePAKE protocol presented in Figure 4. The proof of the theorem is in Appendix D.

Theorem 1. *Let* PAPKE *be a secure password-authenticated public-key encryption scheme that realizes the UC-PAPKE functionality* $\mathcal{F}_{\mathsf{PAPKE}}$ *[12], and* PRF *be secure fixed-length output pseudo-random function. Then, the BeePAKE protocol* Π *described in Figure 4 is a RoR-secure SweetPAKE. More precisely, the advantage of any PPT adversary is bounded by*

$$\mathsf{Adv}^{\mathsf{RoR}}_{\hat{\mathcal{A}},\Pi}(\lambda, n) \leq \frac{n \cdot q_s}{|\mathbb{D}|} + \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_3,\mathsf{PRF}}(\lambda) + \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}'_3,\mathsf{PRF}}(\lambda) + \epsilon(\lambda), \tag{2}$$

where n is the number of sweetwords, q_s denotes the number of Send *queries asked by $\hat{\mathcal{A}}$, $|\mathbb{D}|$ is the size of the password dictionary, and $\epsilon(\lambda)$ is negligible.*

3.3.2 Sugarword indistinguishability of BeePAKE

In Theorem 2 we establish sugarword indistinguishability for BeePAKE. The proof of the theorem is in Appendix E.

Theorem 2. *Let* PAPKE *be a secure password-authenticated public-key encryption scheme that realizes the UC-PAPKE functionality* $\mathcal{F}_{\mathsf{PAPKE}}$ *[12],* PRF *be a secure fixed-length output pseudo-random function, and* RP *be a random permutation. Then, the BeePAKE protocol* Π *described in Figure 4 satisfies the sugarword indistinguishablity. More precisely, the advantage of any PPT adversary is bounded by*

$$\mathsf{Adv}^{\mathsf{SIND}}_{\hat{\mathcal{A}},\Pi}(\lambda, n) \leq \frac{1}{n} + \frac{n \cdot q^*_s}{|\mathbb{D}|} + \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_1,\mathsf{PRF}}(\lambda) + \epsilon(\lambda), \tag{3}$$

where n is the number of sweetwords, q^*_s denotes the number of Send *queries to* client *instances before* Leak *query has been asked, $|\mathbb{D}|$ is the size of the password dictionary, and $\epsilon(\lambda)$ is negligible.*

3.3.3 False alarm protection of BeePAKE

In Theorem 3 we establish that BeePAKE satisfies false alarm protection. The proof of the theorem is in Appendix F.

[12] All three interfaces provided with $\mathcal{F}_{\mathsf{PAPKE}}$ take as input a globally unique session identifier *sid*, which is in contrast with our SweetPAKE notions.

Theorem 3. *Let* PAPKE *be a secure password-authenticated public-key encryption scheme that realizes the UC-PAPKE functionality* $\mathcal{F}_{\mathsf{PAPKE}}$ *[12],* PRF *be a secure fixed-length output pseudo-random function, and* MAC *be a one-time unforgeable message authentication algorithm. Then, the BeePAKE protocol* Π *described in Figure 4 provides false alarm protection. More precisely, the advantage of any PPT adversary is bounded by*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{FAP}}_{\hat{\mathcal{A}},\Pi}(\lambda, n) \leq \frac{(n-1)\cdot q_s^{\diamond}}{|\mathbb{H}|} \ + \ \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_1,\mathsf{PRF}}(\lambda) \ + \ \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_5,\mathsf{PRF}}(\lambda) \\ + \, q_s^{*}\cdot \mathsf{Adv}^{\mathsf{1uf}}_{\mathcal{B}_6,\mathsf{MAC}}(\lambda) \ + \ \epsilon(\lambda)\end{aligned} \tag{4}$$

where n *is the number of sweetwords,* $q_s^{\diamond}$ *denotes the number of* Send *queries to* server *instances asked by* $\hat{\mathcal{A}}$*,* q_s^{*} *denotes the number of* Send *queries to* client *instances,* $|\mathbb{H}|$ *is the size of the honeyword dictionary induced by the underlying honeyword generation algorithm, and* $\epsilon(\lambda)$ *is negligible.*

4 PAPKE-2-OPAKE: A UC-secure Oblivious PAKE protocol

In this section, we propose a simple and intuitive Oblivious PAKE protocol based on PAPKE from [12], a secure PRF, and a random permutation. In our construction, the random permutation is optional and only needed to achieve $\mathcal{F}_{\mathsf{OPAKE}}$ variant that implies sugarword indistinguishability, i.e. without leaking the indices of matching passwords (refer to Figure 3, removing the gray box). We adopt a similar approach as in Theorem 1, taking advantage of the fact that PAPKE has already been established as secure in the UC model (as shown in [12]). We demonstrate in Section 4.4 that a simplified version of BeePAKE, called PAPKE-2-OPAKE (Figure 5), is sufficient to achieve UC-secure SweetPAKE/O-PAKE. Nevertheless, it's crucial to highlight that in this particular version, neither party learns the index of the correct password. As a result, this protocol is only applicable to the O-PAKE setting, rendering the protection against false alarms irrelevant.

4.1 Oblivious PAKE

An O-PAKE protocol allows a forgetful client to test multiple passwords in one shot without leaking its password guesses to a legitimate server or a malicious party. More precisely, in O-PAKE, a client shares a distinct password with each server in the system and is allowed to submit multiple passwords within a single O-PAKE session with a server. The client succeeds if and only if one of those input passwords matches the one stored on the server side – as a result of a successful O-PAKE run, the client and the server establish a common session key.

4.2 Oblivious PAKE vs SweetPAKE

Real-or-Random (RoR) O-PAKE from [25] and RoR SweetPAKE defined in Section 2 are both particular cases of RoR PAKE [3] where we restrict our attention to the setting of a single password ($n = 1$). Besides differences in the password initialization procedures, these two definitions address the same problem (with respect to session key indistinguishability), only with the roles of client and server reversed. But these differences in how the challenger selects and assigns passwords yields non-equivalent security definitions nevertheless.

SweetPAKE provides security assurances beyond those of session key indistinguishability, as it allows the detection of a password file compromise on the server side (captured by sugarword indistinguishability and false alarm protection). Also, in O-PAKE, the set of passwords the client inputs is a subset of all passwords the client has registered with every server within the model – this captures the uncertainty the client has regarding which of their passwords was registered. In SweetPAKE, candidate passwords (i.e. the sweetwords) are stored on the server side and generated according to a honeyword generation procedure that assures flatness with respect to the sugarword. Thus, in the SweetPAKE setting, it makes little sense to model honeywords as real passwords the client registers with other servers, and that an adversary may selectively corrupt.

4.3 Constructing Oblivious PAKE from PAPKE

In our PAPKE-2-OPAKE protocol depicted in Figure 5, Alice first identifies herself to Bob. Then, Bob sends his authenticated public key, which Alice uses to encapsulate a single secret key k under multiple (n) passwords[13], and in that process forms a ciphertext vector that is then sent to Bob. Alice at this point has all information to derive the session key. Upon receiving the ciphertext vector, Bob tries to decrypt each ciphertext one by one. As soon as a single ciphertext decryption is successful, Bob no longer needs to decrypt more ciphertexts. It then computes the session key and terminates the protocol. If no ciphertext is valid, Bob samples a random key and terminates.

Note that since our SweetPAKE/O-PAKE ideal functionality (see Fig. 3) is implicitly rejecting, the server in PAPKE-2-OPAKE protocol must randomly sample the session key in case of a wrong password guess. To have the party abort in such circumstances, we would need to modify our functionality to accommodate one-sided or mutual explicit authentication as in [16, 8]. Also, to achieve sugarword indistinguishability, Alice must additionally substitute all duplicate passwords in the password vector $\mathbf{P}_{A,B}$ with $\perp$ and then shuffle $\mathbf{P}_{A,B}$ before encrypting the session key.

4.4 Security analysis of our Oblivious PAKE construction

In this section, we present a straightforward method for compiling PAPKE to Oblivious PAKE. Our method involves a stripped-down version of the BeePAKE protocol, and we prove its UC-security properties in Theorem 4 with reference to the ideal functionality shown in Figure 3. It is worth noting that our construction differs from the approach described in [25], which relies on different building blocks and leverages on the concept of *Index-Hiding Message Encoding* (IHME).

Theorem 4. *Let* PAPKE *be a secure password-authenticated public-key encryption scheme that UC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{PAPKE}}$ *(Appendix C). Then, the* PAPKE-2-OPAKE *protocol* Π *described in Figure 5 UC-realizes O-PAKE ideal functionality (Figure 3) in the presence of static-corruption adversaries.*

The proof of Theorem 4 can be found in Appendix G. For a full-fledged SweetPAKE with protection against false alarms, we refer the reader to our BeePAKE construction (Figure 4), which requires an extra round of communication and a MAC.

[13]Remember that the security model of PAPKE admits multiple ciphertexts per authenticated public key from the same encryptor. Also, note that the passwords that Alice submits to the PAPKE encryption algorithm may come from various places (e.g. the password manager's list or can be typed by a human).

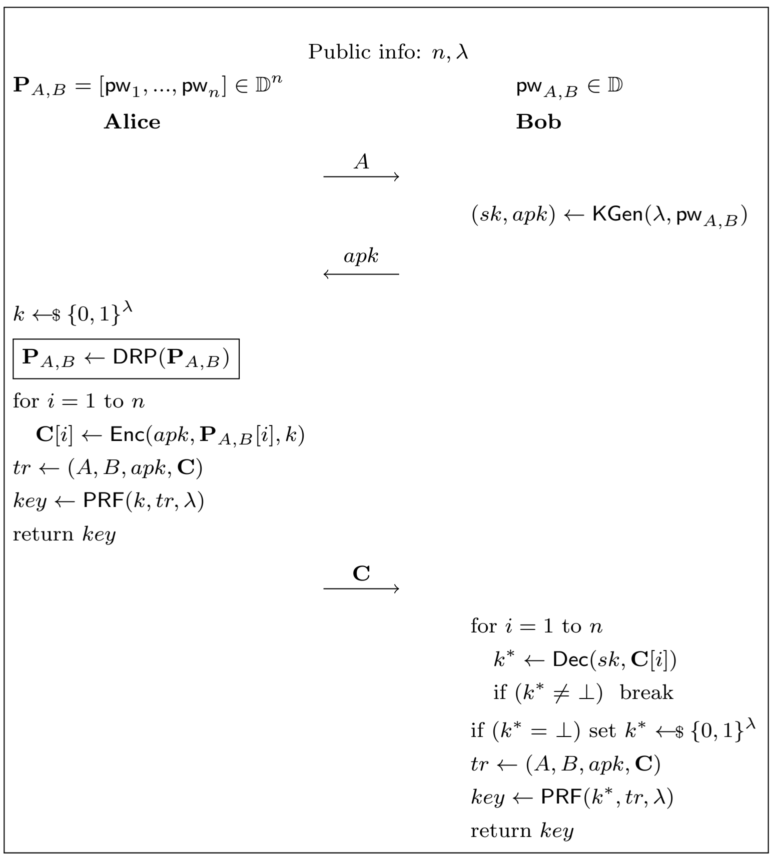

Figure 5: An O-PAKE protocol called PAPKE-2-OPAKE that realizes $\mathcal{F}_{\mathsf{OPAKE}}$ from UC-secure PAPKE. Add the protocol step in text box if sugarword indistinguishability is required. Deduplication and random permutation algorithm DRP first substitutes password duplicates with $\perp$ and then shuffles the vector using RP.

5 Conclusion and future directions

A PAKE-style protocol that admits decoy passwords is useful to put in place as a password file leakage detection mechanism, similar to the Honeywords method by Juels and Rivest [24], while preserving the strong security guarantees provided by PAKE when it comes to low-entropy password-authentication and secure channel establishment. We called this new primitive SweetPAKE.

In this paper, we extended the well-established PAKE model by Abdalla et al. [3] and modelled 3 security properties of SweetPAKE: (1) indistinguishability of session keys (with forward secrecy); (2) indistinguishability of sugarwords; and (3) false alarm protection. We also extend the UC model for PAKE [14] to account for the protocol asymmetry in that one party holds a single password whereas the other party holds n passwords, and to handle arbitrary correlations between passwords / sweetwords.

We leverage the PAPKE primitive by Bradley et al. [12] to encrypt the session key under multiple candidate passwords and combine it with a key derivation function, a secure message authentication code and a random permutation to build BeePAKE, a SweetPAKE protocol that satisfies simulta-

neously all three properties.

The SweetPAKE notion is closely related to O-PAKE by Kiefer and Manulis [25], with the roles of client and server reversed. With regard to game-based definitions, it is easy to see that O-PAKE implies indistinguishability of session keys of SweetPAKE, sufficing to initialize n times more servers within the model to avoid password overlap.

The status of whether the reverse is true, or if O-PAKE implies sugarword indistinguishability, remains unresolved questions, but it does not seems to be the case. More importantly, the UC definition we introduce overcomes these differences and unifies SweetPAKE (session key and sugarword indistinguishability) and O-PAKE.

In Section 4.3 we show an alternative construction to [25] on how to build an O-PAKE protocol, inspired by our BeePAKE protocol. However, the protocol is considerably simpler, as it builds on top of only PAPKE and PRF. A compelling future direction is to show if the O-PAKE protocol proposed in [25] and based on PAKE and IHME can be transformed into a SweetPAKE. Interestingly, their protocol is instantiated with SPAKE2, which has a tight proof under the gap-CDH assumption [1]. However, for sugarword indistinguishability, we most likely need to assume a decisional variant of the underlying hard problem, so that an adversary cannot link the candidate values extracted from the IHME structure.

Finally, our game-based definitions of SweetPAKE admit that sweetwords associated with a client may be correlated as long as they are "flat", i.e. any of them is equally likely (or ϵ-close) to be the sugarword. However, like game-based definitions of PAKE, our game-based models do not capture arbitrary correlations between sugarwords (i.e. between "real" passwords). Reason being that corrupting a party might reveal information that allows the attacker to trivially infer the password of another party. As with PAKE, to provide this security guarantee one must refer to the UC model.

Acknowledgements

We thank the anonymous reviewers of AsiaCCS 2024 for their comments and suggestions. Afonso Arriaga and Marjan Škrobot received support from the Luxembourg National Research Fund (FNR) under the CORE Junior project (C21/IS/16236053/FuturePass). Peter Y.A. Ryan received support from the Luxembourg National Research Fund (FNR) under the CORE project (C21/IS/16221219/ImPAKT). Additionally, we extend our thanks to Steve Meireles for his contribution to the implementation of the proposed protocols.

References

[1] Michel Abdalla, Manuel Barbosa, Tatiana Bradley, Stanisław Jarecki, Jonathan Katz, and Jiayu Xu. Universally composable relaxed password authenticated key exchange. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 278–307, Cham, 2020. Springer.

[2] Michel Abdalla, Olivier Chevassut, and David Pointcheval. One-time verifier-based encrypted key exchange. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005*, pages 47–64, Berlin, Heidelberg, 2005. Springer.

[3] Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. Password-based authenticated key exchange in the three-party setting. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005*, pages 65–84, Berlin, Heidelberg, 2005. Springer.

[4] Michel Abdalla, Björn Haase, and Julia Hesse. Security analysis of cpace. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 711–741, Cham, 2021. Springer.

[5] Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, pages 191–208, Berlin, Heidelberg, 2005. Springer.

[6] Tolga Acar, Mira Belenkiy, and Alptekin Küpçü. Single password authentication. *Computer Networks*, 57(13):2597–2614, 2013.

[7] José Becerra, Peter B. Rønne, Peter Y. A. Ryan, and Petra Sala. Honeypakes. In Vashek Matyáš, Petr Švenda, Frank Stajano, Bruce Christianson, and Jonathan Anderson, editors, *Security Protocols XXVI*, pages 63–77, Cham, 2018. Springer.

[8] Hugo Beguinet, Céline Chevalier, David Pointcheval, Thomas Ricosset, and Mélissa Rossi. Get a cake: Generic transformations from key encaspulation mechanisms to password authenticated key exchanges. In Mehdi Tibouchi and XiaoFeng Wang, editors, *Applied Cryptography and Network Security*, pages 516–538, Cham, 2023. Springer.

[9] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, pages 139–155, Berlin, Heidelberg, 2000. Springer.

[10] S.M. Bellovin and M. Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, Los Alamitos, CA, USA, 1992. IEEE Computer Society.

[11] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[12] Tatiana Bradley, Jan Camenisch, Stanislaw Jarecki, Anja Lehmann, Gregory Neven, and Jiayu Xu. Password-authenticated public-key encryption. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 442–462, Cham, 2019. Springer.

[13] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[14] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Phil MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, pages 404–421, Berlin, Heidelberg, 2005. Springer.

[15] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006*, pages 142–159, Berlin, Heidelberg, 2006. Springer.

[16] Adam Groce and Jonathan Katz. A new framework for efficient password-based authenticated key exchange. In *Proceedings of the 17th ACM Conference on Computer and Communications*

Security, CCS '10, pages 516–525, New York, NY, USA, 2010. Association for Computing Machinery.

[17] Björn Haase and Benoît Labrique. Aucpace: Efficient verifier-based pake protocol tailored for the iiot. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):1–48, Feb. 2019.

[18] Feng Hao and Peter Ryan. J-pake: Authenticated key exchange without pki. In Marina L. Gavrilova, C. J. Kenneth Tan, and Edward David Moreno, editors, *Transactions on Computational Science XI: Special Issue on Security in Computing, Part II*, pages 192–206, Berlin, Heidelberg, 2010. Springer.

[19] Feng Hao and Paul C. van Oorschot. Sok: Password-authenticated key exchange – theory, practice, standardization and real-world lessons. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, pages 697–711, New York, NY, USA, 2022. Association for Computing Machinery.

[20] Jung Yeon Hwang, Stanislaw Jarecki, Taekyoung Kwon, Joohee Lee, Ji Sun Shin, and Jiayu Xu. Round-reduced modular construction of asymmetric password-authenticated key exchange. In Dario Catalano and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 485–504, Cham, 2018. Springer.

[21] Ponemon Institute. The cost of a data breach report. IBM Security, 2022.

[22] Stanislaw Jarecki, Hugo Krawczyk, Maliheh Shirvanian, and Nitesh Saxena. Device-enhanced password protocols with optimal online-offline protection. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 177–188, New York, NY, USA, 2016. Association for Computing Machinery.

[23] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. Opaque: An asymmetric pake protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 456–486, Cham, 2018. Springer.

[24] Ari Juels and Ronald L. Rivest. Honeywords: Making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 145–160, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Franziskus Kiefer and Mark Manulis. Oblivious pake: Efficient handling of password trials. In Javier Lopez and Chris J. Mitchell, editors, *Information Security*, pages 191–208, Cham, 2015. Springer.

[26] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, volume 2. Addison-Wesley, USA, 3 edition, 1997.

[27] Wenting Li, Ping Wang, and Kaitai Liang. Hpake: Honey password-authenticated key exchange for fast and safer online authentication. *IEEE Transactions on Information Forensics and Security*, 18:1596–1609, 2023.

[28] Mark Manulis, Benny Pinkas, and Bertram Poettering. Privacy-preserving group discovery with linear complexity. In Jianying Zhou and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 420–437, Berlin, Heidelberg, 2010. Springer.

[29] Bruno Freitas Dos Santos, Yanqi Gu, and Stanislaw Jarecki. Randomized half-ideal cipher on groups with applications to uc (a)pake. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 128–156, Cham, 2023. Springer.

[30] Marjan Skrobot and Jean Lancrenon. On composability of game-based password authenticated key exchange. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 443–457, Los Alamitos, CA, USA, 2018. IEEE Computer Society.

[31] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin. Using the secure remote password (srp) protocol for tls authentication. Technical Report 5054, RFC Editor, 2007.

A Comparison between N*SPAKE2 and BeePAKE-PAPKE-IC-DHIES

In Table 1 we compare the computational cost and bandwidth of BeePAKE-PAPKE-IC-DHIES against the benchmark N*SPAKE2, which consists of running n instances of SPAKE2 [1] in parallel. For a fair comparison, we have optimized N*SPAKE in the following ways: (1) messages are bundled to minimize the number of flows; (2) the client only computes M^{pw} and N^{pw} once. To obtain concrete values, we have set the number of sweetwords per client to $n = 20$, and targeted an overall security parameter $\lambda = 128$. We instantiated both SPAKE2 and PAPKE-IC-DHIES with Curve25519. The IC in PAPKE-IC-DHIES is instantiated with Rijndael-256 using the techniques from [29], and AE with AES-128-GCM. Note that the estimate column corresponds to the worst-case scenario where every single ciphertext needs to be decrypted by the client until they find the one encrypted with the sugarword. In practice, this happens on average after decrypting half of the ciphertexts, which is reflected in the measurement experiment. Code for measurements was implemented in Python 3, and runtime was assessed on a Mac laptop with an Intel Core i5 chip and 16GB of RAM. Measurements were taken 50 times, averaging the results after removing the top and bottom 10%.

B Building blocks

In this section we provide a brief description of primitives used as building blocks in our BeePAKE (Fig. 4) and Oblivious PAKE (Fig. 5) constructions.

Definition 5 (PAPKE [12]). *Let $\mathbb{D}$ be a dictionary of possible passwords, and $\mathcal{M}$ be a message space. A password-authenticated public-key encryption scheme is a tuple of algorithms* $\mathsf{PAPKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *behaving as follows:*

- $\mathsf{KGen}(\lambda, pw)$*: on input a security parameter λ and a password $pw \in \mathbb{D}$,* KGen *algorithm outputs an authenticated public key apk and a secret key sk.*
- $\mathsf{Enc}(apk, pw, m)$*: on input an authenticated public key apk, a password pw, and a message $m \in \mathcal{M}$,* Enc *algorithm outputs a ciphertext c.*
- $\mathsf{Dec}(sk, c)$*: on input a secret key sk and ciphertext c,* Dec *algorithm outputs a message $m \in \mathcal{M} \cup \{\perp\}$, where $\perp$ denotes invalid ciphertext.*

Correctness. *Let password $pw \in \mathbb{D}$, key pair (apk, sk) be an output of $\mathsf{KGen}(\lambda, pw)$, and ciphertext c be an output of $\mathsf{Enc}(apk, pw, m)$.* PAPKE *is said to be* correct *if we have that $m \leftarrow \mathsf{Dec}(sk, c)$.*

N*SPAKE2			
Client		Server	
Estimation	Measured ($n = 20$)	Estimation	Measured ($n = 20$)
$(2n+2)$ exp + $2n$ mult	99.14 ms	$4n$ exp + $2n$ mult	215.01 ms
Bandwidth			
General cost		Cost for $n = 20$ ($\lambda = 128$, Curve22519)	
$2n$ group elements + $(n+1) * \lambda$ bits		1616 bytes	
Message flows: 3			

BeePAKE-PAPKE-IC-DHIES			
Client		Server	
Estimation	Measured ($n = 20$)	Estimation	Measured ($n = 20$)
$(n+1)$ exp + 1 ic	48.55 ms	$2n$ exp + n ic	181.45 ms
Bandwidth			
General cost		Cost for $n = 20$ ($\lambda = 128$, Curve22519, Rijndael-256, AES-128-GCM)	
$(n+1)$ group elements + $(n+1) * \lambda$ bits + n overhead of AE		1568 bytes	
Message flows: 3			

Table 1: Comparison between N*SPAKE2 and BeePAKE-PAPKE-IC. The terms 'exp', 'mult' and 'ic' refer to the number of group exponentiations, the number of group multiplications, and the number of encryptions or decryption of group elements, respectively. The overhead of 1 AES-128-GCM ciphertext is 12 bytes for the IV and 16 bytes for the tag.

In our paper, we will assume that underlying PAPKE primitive UC-realizes PAPKE functionality $\mathcal{F}_{\mathsf{PAPKE}}$ from [12] (see Figure 1).

Definition 6 (Pseudorandom Function)**.** *Let* PRF *be a fixed-length output pseudorandom function family parameterized by its output length. Given a key* k *in the key space* $\mathcal{K}$*, a bitstring* $m \in \mathcal{M}$*, and an output length* l *in bits,* PRF *outputs a value in* $\{0,1\}^l$*. We define the security of a fixed-length output pseudorandom function family against any PPT adversary* $\mathcal{A}$ *making at most* q *queries in the following distinguishing game:*

1. *For* $i = 1, . . . , q' \leq q$*:* $\mathcal{A}$ *chooses arbitrary values* m_i *and receives the value* $\mathsf{PRF}(k, m_i, l)$ *(queries by* $\mathcal{A}$ *are adaptive, i.e., each query may depend on the responses to previous ones).*
2. $\mathcal{A}$ *chooses values* m *such that* $m \notin \{m_1, ..., m_q\}$*.*
3. *A bit* $b \in \{0,1\}$ *is chosen at random. If* $b = 0$*, attacker* $\mathcal{A}$ *is provided with the output of* $\mathsf{PRF}(k, m, l)$*, else* $\mathcal{A}$ *is given a random string of* l *bits.*
4. *Step 1 is repeated for up to* $q - q'$ *queries with the restriction on challenged* m*.*
5. $\mathcal{A}$ *outputs a bit* $b' \in \{0,1\}$*. It wins if* $b' = b$*.*

We define the advantage of an algorithm $\mathcal{A}$ *in winning* prf *security game as* $\mathsf{Adv}^{\mathsf{prf}}_{\mathcal{A},\mathsf{PRF}}(\lambda) = |\ 2 \cdot \Pr[b' = b] - 1\|$*. We say that* PRF *is* prf*-secure if for all PPT algorithms* $\mathcal{A}$*,* $\mathsf{Adv}^{\mathsf{prf}}_{\mathcal{A},\mathsf{PRF}}(\lambda)$ *is negligible in the security parameter* λ*.*

Definition 7 (One-time MAC security)**.** *A* MAC *scheme with key space* $\mathcal{K}$ *and domain* $\mathcal{D}$ *is one-time unforgeable if, for any PPT adversary* $\mathcal{A}$*, the following advantage term is negligible in the security parameter* λ*.*

$$\mathsf{Adv}^{\mathsf{1uf}}_{\mathcal{A},\mathsf{MAC}} \stackrel{\text{def}}{=} \Pr\Big[t \leftarrow \mathsf{MAC}(k, m) : (m, t) \leftarrow \mathcal{A}^{\mathsf{MAC}(k,\cdot)}(\lambda); k \leftarrow \mathcal{K}\Big]$$

In *one-time* MAC, the adversary can only call MAC oracle once.

Definition 8 (Random Permutation)**.** *A uniform random permutation* RP *is one in which each of the* $n!$ *possible permutations are equally likely.*

Fisher–Yates shuffle algorithm [26] implements a random permutation in $\mathcal{O}(n)$ time complexity, assuming function $rand()$ that generates a random numbers in $\mathcal{O}(1)$ time.

C Ideal Functionality for PAPKE

For completeness, we include in Figure 6 the ideal functionality for PAPKE from [12].

D Security Proof of Theorem 1

Proof. We will use the game-hopping technique in this proof, starting with the game G_0 where Test bit is fixed to 0 (real keys) and ending with the game G_7 where Test bit is fixed to 1 (random keys).

Let $\hat{\mathcal{A}}$ be an adversary against the Real-or-Random (RoR) indistinguishability of the session keys property and $\mathcal{C}^{ror}$ be a challenger administrating this security game. Let G_x be the event that $\hat{\mathcal{A}}$ outputs 1 in Game x.

1. **Key Generation**. On input ($\mathsf{KEYGEN}, sid, pwd$) from party $\mathcal{P}$:
   - If $sid \neq (\mathcal{P}, sid')$ or a record ($\mathsf{keyrec}, sid, \cdot, \cdot$) exists, ignore.
   - Send (KEYGEN, sid) to $\mathcal{A}$ and wait for ($\mathsf{KEYCONF}, sid, apk, \mathcal{M}$) from $\mathcal{A}$.
   - If a record ($\mathsf{badkeys}, sid, apk_j, \cdot$) with $apk_j = apk$ exists, abort.
   - Create a record ($\mathsf{keyrec}, sid, apk, pwd$) and output ($\mathsf{KEYCONF}, sid, apk$) to $\mathcal{P}$.

---

2. **Encryption**. On input ($\mathsf{ENCRYPT}, sid, apk', pwd', m$) from party $\mathcal{Q}$ where $m \in \mathcal{M}$:
   - If a record ($\mathsf{keyrec}, sid, apk, pwd$) with $apk = apk'$ exists and $\mathcal{P}$ (from $sid = (\mathcal{P}, sid')$) is honest, then do the following:
     - Send ($\mathsf{ENC\text{-}L}, sid, |m|$) to $\mathcal{A}$ and wait for ($\mathsf{CIPHERTEXT}, sid, c$) from $\mathcal{A}$.
     - Abort if a record ($\mathsf{enrec}, sid, \cdot, c$) for c already exists.
     - Create a record ($\mathsf{enrec}, sid, m', c$) with $m' \leftarrow m$ if $pwd' = pwd$, and $m \leftarrow \perp$, else.
   - Otherwise, do the following:
     - If a record ($\mathsf{badkeys}, sid, apk_j, pwd_j$) with $apk_j = apk'$ exists, set $pwd^* \leftarrow pwd_j$.
     - Else, send ($\mathsf{GUESS}, sid, apk'$) to $\mathcal{A}$, wait for ($\mathsf{GUESS}, sid, pwd^*$) from $\mathcal{A}$ and create record ($\mathsf{badkeys}, sid, apk', pwd^*$).
     - If $pwd' = pwd^*$, send ($\mathsf{ENC\text{-}M}, sid, m$) to $\mathcal{A}$, and wait for ($\mathsf{CIPHERTEXT}, sid, c$) from $\mathcal{A}$.
     - If $pwd' \neq pwd^*$, send ($\mathsf{ENC\text{-}L}, sid, |m|$) to $\mathcal{A}$, and wait for ($\mathsf{CIPHERTEXT}, sid, c$) from $\mathcal{A}$.
   - Output ($\mathsf{CIPHERTEXT}, sid, c$) to $\mathcal{Q}$.

---

3. **Decryption**. On input ($\mathsf{DECRYPT}, sid, c$) from party $\mathcal{P}$:
   - If $sid \neq (\mathcal{P}, sid')$ or no record ($\mathsf{keyrec}, sid, apk, pwd$) exists, ignore.
   - If a record ($\mathsf{enrec}, sid, m, c$) for c exists, where $m \in \mathcal{M} \cup \{\perp\}$:
     - Output ($\mathsf{PLAINTEXT}, sid, m$) to $\mathcal{P}$.
   - Else:
     - Send ($\mathsf{DECRYPT}, sid, c$) to $\mathcal{A}$ and wait for ($\mathsf{PLAINTEXT}, sid, m, pwd^*$) from $\mathcal{A}$.
     - If $pwd^* = pwd$, set $m' \leftarrow m$ and $m' \leftarrow \perp$ otherwise.
     - Create a record ($\mathsf{enrec}, sid, m', c$).
     - Output ($\mathsf{PLAINTEXT}, sid, m'$) to $\mathcal{P}$.

Figure 6: Ideal functionality $\mathcal{F}_{\mathsf{PAPKE}}$ [12].

Game G_0. (Fixing a bit - real keys). This is a RoR game with a fixed challenge bit $b = 0$. Real session keys are provided to $\hat{\mathcal{A}}$ when it queries Test oracle.

Game G_1. (Building an environment). The challenger of this game $\mathcal{C}^{ror}$ doubles as an environment $\mathcal{Z}$ interacting with a dummy adversary $\mathcal{A}$ in the real world.

Simulation. The challenger for $\mathcal{C}^{ror}$ runs the simulation for $\hat{\mathcal{A}}$ as follows. Acting as an environment, $\mathcal{C}^{ror}$ executes the initialization procedure, defining all sweetwords and sugarwords associated with clients and servers according to Figure 2. After this, $\hat{\mathcal{A}}$ may start making queries. $\mathcal{C}^{ror}$ answers to $\hat{\mathcal{A}}$'s queries as presented in Figures 7 and 8. On high level, $\mathcal{C}^{ror}$ replaces the calls to algorithms $\mathsf{PAPKE.KGen}$, $\mathsf{PAPKE.Enc}$, and $\mathsf{PAPKE.Dec}$ in this game with calls to the parties in the real world. The challenger resolves syntax issues between our game-based and UC PAPKE security notions with use of dummy sids and by tracking and recording progress of instances with lists $\mathbf{L}_{clients}$ and $\mathbf{L}_{servers}$.

Note that $\mathcal{Z}$ does not flip any coin and always forwards the real session key to the adversary $\hat{\mathcal{A}}$ when it asks a Test query. In the end, $\mathcal{Z}$ outputs whatever $\hat{\mathcal{A}}$ outputs.

$$\Pr[\mathsf{G}_0] = \Pr[\mathsf{G}_1] \tag{5}$$

Game G_2. (Transition to SIM). The challenger of this game now doubles as an environment $\mathcal{Z}$ interacting with a simulator SIM and ideal functionality $\mathcal{F}_{\mathsf{PAPKE}}$. Since we assume that underlying PAPKE used in BeePAKE protocol UC-realizes $\mathcal{F}_{\mathsf{PAPKE}}$, there exists a SIM that simulates adversary $\mathcal{A}$ such that this transition is unnoticeable to $\mathcal{Z}$. The environment $\mathcal{Z}$ outputs whatever $\hat{\mathcal{A}}$ outputs, and therefore this has to be unnoticeable to $\hat{\mathcal{A}}$, as well.

$$|\Pr[\mathsf{G}_1] - \Pr[\mathsf{G}_2]| = |Real(\mathcal{Z}, \mathcal{A}, \mathsf{PAPKE}) - Ideal(\mathcal{Z}, \mathsf{SIM}, \mathcal{F}_{\mathsf{PAPKE}})| \leq \epsilon(\lambda) \tag{6}$$

Game G_3. (Randomize session keys for $\mathsf{Execute}$ queries). The challenger modifies Test query such that independent random session keys are provided when $\hat{\mathcal{A}}$ calls Test oracle targeting an instance that has accepted as a result of a call to an $\mathsf{Execute}$ query (see Figure 9).

When $\mathcal{Z}$ inputs $(\mathsf{ENCRYPT}, sid, apk', \mathbf{P}_{C,S}[i], \mathbf{K}[i])$ to party S^j as a result of an $\mathsf{Execute}$ query, a record $(\mathsf{keyrec}, sid, apk, pw_{C,S})$ always exist such that $apk = apk'$ because environment $\mathcal{Z}$ uses the apk' that is provided by SIM in the key confirmation message $(\mathsf{KEYCONF}, sid, apk, \mathcal{M})$. Note that apk' cannot be or become a "badkey" and SIM is never given the possibility to place a password guess. Recall that we restrict our attention to dummy adversaries that simply follow the instruction of the environment, which as shown in [13] is equivalent to the notion of accepting arbitrary PPT adversarial strategies. Hence, SIM simulates the ciphertexts only from the length of the message, which in our case is the length of the key parameterized by λ.

When $\mathcal{Z}$ inputs $(\mathsf{DECRYPT}, sid, c)$ as a result of a call to $\mathsf{Execute}$ oracle, there is always a record $(\mathsf{enrec}, sid, m, c)$ of the plaintext m corresponding to c. Therefore, SIM cannot select m of its choosing (as specified in else branch of ideal $\mathcal{F}_{\mathsf{PAPKE}}$ decryption interface).

Note that key is computed as a result of two calls to PRF, while key' is sampled uniformly at random. Both key and key' are independent of the $trace \leftarrow (C, S, apk, \hat{\mathbf{C}}, i, tag)$ since $\hat{\mathbf{C}}$ was computed from the message length and PRF is a secure PRF. Therefore,

$$|\Pr[\mathsf{G}_2] - \Pr[\mathsf{G}_3]| = \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_3,\mathsf{PRF}} + \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}'_3,\mathsf{PRF}} \leq \epsilon(\lambda) \tag{7}$$

Game G_4. Let bad_1 be the event that when $\hat{\mathcal{A}}$ queries $\mathsf{Send}(S^j, M = (C, apk))$, no prior $\mathsf{Corrupt}(C, S)$ query occurred, simulator SIM guesses any password from the list $\mathbf{P}_{C,S}$. Let bad_2 be the event that

when $\hat{\mathcal{A}}$ queries $\mathsf{Send}(C^i, M = (S, \hat{\mathbf{C}}))$, the SIM provides the correct password with the plaintext. When bad_1 or bad_2 happen, the challenger aborts and $\hat{\mathcal{A}}$ wins the game. [Technically, the challenger is unable to determine when it has to abort.] Since passwords are sampled by the environment $\mathcal{Z}$ uniformly at random, the probability of challenger aborting is bounded by $n \cdot q_s/|\mathbb{D}|$. Note that q_s is the total number of asked Send queries, which comprises both $\mathsf{Send}(S^j, M = (C, apk))$ and $\mathsf{Send}(C^i, M = (S, \hat{\mathbf{C}}))$. On an additional note, when $\hat{\mathcal{A}}$ impersonates the client, SIM is given 1 password guess targeting n (possibly) distinct passwords. When $\hat{\mathcal{A}}$ impersonates the server, SIM is given n password guesses, each targeting one password.

$$\Pr[\mathsf{G}_3] = \Pr[\mathsf{G}_4] + \frac{n \cdot q_s}{|\mathbb{D}|} \tag{8}$$

Game G_5. The challenger samples random keys for instances that terminate in the accepting state. Recall that a legitimate adversary can only call Test or Reveal on fresh instances. In case the instance was corrupted before accepting, a legitimate adversary cannot make a Test or Reveal query on that instance. Bad events set in the previous game assure that ciphertexts produced by SIM (and the transcripts that go along) are independent of session keys.

$$\Pr[\mathsf{G}_4] = \Pr[\mathsf{G}_5] \tag{9}$$

Game G_6. The challenger of this game doubles as an environment $\mathcal{Z}$ interacting with a dummy adversary $\mathcal{A}$ in the real world, again, but still computes random session keys for $\hat{\mathcal{A}}$.

$$|\Pr[\mathsf{G}_6] - \Pr[\mathsf{G}_5]| = |Real(\mathcal{Z}, \mathcal{A}, \mathsf{PAPKE}) - Ideal(\mathcal{Z}, \mathsf{SIM}, \mathcal{F}_{\mathsf{PAPKE}})| \leq \epsilon(\lambda) \tag{10}$$

Game G_7. The challenger no longer doubles as environment $\mathcal{Z}$ and computes everything itself. Notice that Game G_7 is RoR with fixed $b = 1$. Random session keys are provided to $\hat{\mathcal{A}}$ when it queries Test oracle.

$$\Pr[\mathsf{G}_6] = \Pr[\mathsf{G}_7] \tag{11}$$

Putting all together,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{RoR}}_{\hat{\mathcal{A}},\Pi}(\lambda, n) &= 2 \cdot \Pr[Succ] - 1 \\ &= \Pr[\mathsf{G}_7] - \Pr[\mathsf{G}_0] \\ &\leq \frac{n \cdot q_s}{|\mathbb{D}|} + \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_3,\mathsf{PRF}} + \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}'_3,\mathsf{PRF}} + \epsilon(\lambda)\end{aligned}$$

□

E Security Proof of Theorem 2

Proof. Let $\hat{\mathcal{A}}$ be an adversary against the sugarword indistinguishability property and $\mathcal{C}^{sind}$ be a challenger administrating this security game.

Game G_0. **(Original game)**. The original game for the sugarword indistinguishability property. Adversary $\hat{\mathcal{A}}$ terminates outputting (C, S, i).

Game G_1. **(Pseudo-random function)**. Remove calls to PRF and sample keys of vector $\mathbf{K}$ independently at random. This game decouples k^* from other values in the vector $\mathbf{K}$. Even if tag and PRF leak k^*, this does not help $\hat{\mathcal{A}}$ to determine vector $\mathbf{K}$ and k^*'s position in it.

$$|\Pr[\mathsf{G}_0] - \Pr[\mathsf{G}_1]| = \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_1,\mathsf{PRF}} \leq \epsilon(\lambda) \tag{12}$$

Game G_2. (Building an environment). The challenger of this game doubles as an environment $\mathcal{Z}$ interacting with a dummy adversary $\mathcal{A}$ in the real world.

Simulation. The challenger for $\mathcal{C}^{sind}$ runs the simulation for $\hat{\mathcal{A}}$ in the same way as in the proof of Theorem 1 with the following three changes due to game G_1: (1) For each $\mathsf{Send}(S^j, M = (C, apk))$ query, instead of sampling seed k to be used as PRF input, $\mathcal{C}^{sind}$ samples uniformly at random n independent keys and places them in vector $\mathbf{K}$; (2) The same change is made for $\mathsf{Execute}$ queries; (3) Test queries are not available to $\hat{\mathcal{A}}$. See Figure 10 for details.

$$\Pr[\mathsf{G}_1] = \Pr[\mathsf{G}_2] \tag{13}$$

Game G_3. (Transition to SIM). The challenger of this game now doubles as an environment $\mathcal{Z}$ interacting with a simulator SIM and ideal functionality $\mathcal{F}_{\mathsf{PAPKE}}$. Since we assume that underlying PAPKE used in BeePAKE protocol UC-realizes $\mathcal{F}_{\mathsf{PAPKE}}$, there exists a SIM that simulates adversary $\mathcal{A}$ such that this transition is unnoticeable to $\mathcal{Z}$. If $\hat{\mathcal{A}}$ correctly guesses an index, $\mathcal{Z}$ guesses "real world". Otherwise, $\mathcal{Z}$ guesses "ideal world".

$$|\Pr[\mathsf{G}_2] - \Pr[\mathsf{G}_3]| = |Real(\mathcal{Z}, \mathcal{A}, \mathsf{PAPKE}) - Ideal(\mathcal{Z}, \mathsf{SIM}, \mathcal{F}_{\mathsf{PAPKE}})| \leq \epsilon(\lambda) \tag{14}$$

Game G_4. (Wrong password guesses). Ideal functionality $\mathcal{F}_{\mathsf{PAPKE}}$ is replaced with $\mathcal{F}'_{\mathsf{PAPKE}}$ which considers all password guesses of SIM as being wrong (but is unchanged otherwise). No more correct password guesses.

Notice that q_s^* corresponds to the number of Send queries to client instances, excluding those queries asked after the Leak query has been made. Since servers do not know the index of the sugarword, the adversary obtains no information about the index when interacting with servers.

$$\Pr[\mathsf{G}_4] \leq \Pr[\mathsf{G}_3] + \frac{n \cdot q_s^*}{|\mathbb{D}|} \tag{15}$$

Game G_5. (Random Permutation). At this point, only honestly generated ciphertexts (on an individual level) might decrypt to something other than $\perp$. Which ciphertext decrypts might still reveal information regarding the correct index. However, this information is destroyed by the random permutation RP that shuffles ciphertexts in each execution. Therefore, the probability of correctly guessing an index in this game is $1/n$.

$$\Pr[\mathsf{G}_5] = \frac{1}{n} \tag{16}$$

Putting all together,

$$\mathsf{Adv}^{\mathsf{SIND}}_{\hat{\mathcal{A}},\Pi}(\lambda, n) \leq \frac{1}{n} + \frac{n \cdot q_s^*}{|\mathbb{D}|} + \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_1,\mathsf{PRF}} + \epsilon(\lambda) \tag{17}$$

□

F Security Proof of Theorem 3

Proof. Let $\hat{\mathcal{A}}$ be an adversary against the false alarm protection property and $\mathcal{C}^{fap}$ be a challenger administrating this security game.

Game G_0. (Original game). The original game for the false alarm protection property. Adversary $\hat{\mathcal{A}}$ runs in t-time and places at most q queries.

Game G_1. (Pseudo-random function I). Remove calls to PRF and sample keys of vector $\mathbf{K}$ independently at random. As in proof of Theorem 2, this game hop decouples k^* from other values in the vector $\mathbf{K}$.

$$|\Pr[\mathsf{G}_0] - \Pr[\mathsf{G}_1]| = \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_1,\mathsf{PRF}} \leq \epsilon(\lambda) \tag{18}$$

Game G_2. (Building an environment). The challenger of this game $\mathcal{C}^{fap}$ doubles as an environment $\mathcal{Z}$ interacting with a dummy adversary $\mathcal{A}$ in the real world.

Simulation. Due to similarity with the order of game hops, the challenger for $\mathcal{C}^{fap}$ runs the simulation for $\hat{\mathcal{A}}$ in the same manner as the challenger for $\mathcal{C}^{sind}$ in the proof of Theorem 2 (see Figure 10).

$$\Pr[\mathsf{G}_1] = \Pr[\mathsf{G}_2] \tag{19}$$

Game G_3. (Transition to SIM). The challenger of this game now doubles as an environment $\mathcal{Z}$ interacting with a simulator SIM and ideal functionality $\mathcal{F}_{\mathsf{PAPKE}}$. Since we assume that underlying PAPKE used in BeePAKE protocol UC-realizes $\mathcal{F}_{\mathsf{PAPKE}}$, there exists a SIM that simulates adversary $\mathcal{A}$ such that this transition is unnoticeable to $\mathcal{Z}$. If event E occurs, the environment $\mathcal{Z}$ guesses "real world". If $\hat{\mathcal{A}}$ depletes its resources (number of queries available) without triggering event E, $\hat{\mathcal{A}}$ lost and environment $\mathcal{Z}$ guesses "ideal world".

$$|\Pr[\mathsf{G}_2] - \Pr[\mathsf{G}_3]| = |Real(\mathcal{Z}, \mathcal{A}, \mathsf{PAPKE}) - Ideal(\mathcal{Z}, \mathsf{SIM}, \mathcal{F}_{\mathsf{PAPKE}})| \leq \epsilon(\lambda) \tag{20}$$

Game G_4. (Encrypting zeros). The challenger encrypts 0^λ for all honeywords. What passwords are sugarwords is known to the challenger, and ciphertexts encrypted with mismatching pairs (apk, pw) decrypt to $\perp$ according to the ideal functionality $\mathcal{F}_{\mathsf{PAPKE}}$. Refer to Figure 11.

Simulation. On the server side (encryption), if apk is a "goodkey", SIM receives $(\mathsf{ENC\text{-}L}, sid, \lambda)$. Since $\lambda = |\mathbf{K}[i]|$ for all i, nothing changes. If apk is a "badkey", SIM is allowed to place a password GUESS. Notice that for every such Send query, $\mathcal{Z}$ asks $(n-1)$ encryptions with replaced keys (0^λ).

On the client side (decryption), if a record $(\mathsf{ENCREC}, sid, m, c)$ exists, then the correct record will be returned, which for replaced keys is the $\perp$ symbol. If no record exists, SIM has one password guess via $(\mathsf{PLAINTEXT}, sid, m, pw^*)$ per $(\mathsf{DECRYPT}, sid, c)$. In practice, we are accounting for the probability of adversary $\hat{\mathcal{A}}$ encrypting c under the correct password associated with apk, which will then allow the adversary to remove that password from possible candidates for sweetwords. In the reduction, we count all $\mathsf{PLAINTEXT}$ answers from SIM, which is upper bounded by $1/|\mathbb{H}|$ per ciphertext decryption $(\mathsf{DECRYPT}, sid, c)$ because of flatness (by definition, all honeywords are all equally likely). The last flow on the server side remains unchanged, meaning that fixed keys 0^λ are not used. Instead, PRF expands over k^* as per protocol description. Unless SIM successfully guesses one of the honeywords in either GUESS or $\mathsf{PLAINTEXT}$ answers, the ciphertexts and decryption outputs are produced from the exact same view that SIM had in Game G_3. Given that honeywords within each vector $\mathbf{P}_{C,S}$ are all equally likely, the probability of each honeyword guess is $1/|\mathbb{H}|$. SIM places at most $n-1$ honeyword guesses for every $\mathsf{Send}(S, \cdot)$ query $\hat{\mathcal{A}}$ asks to a server instance. Hence,

$$\Pr[\mathsf{G}_4] \leq \Pr[\mathsf{G}_3] + \frac{(n-1) \cdot q_s^\diamond}{|\mathbb{H}|} \tag{21}$$

Game G_5. (Pseudo-random function II). At this point, none of the keys that would allow $\hat{\mathcal{A}}$

to forge a *tag* that triggers a false alarm are part of the *trace*. We remove calls to PRF and sample (key, kt) independently at random. The challenger keeps track of values on both client and server sides. Note that this change decouples *key* from *kt*, so even if *key* somehow leaks *kt* is protected.

$$|\Pr[\mathsf{G}_4] - \Pr[\mathsf{G}_5]| = \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_5,\mathsf{PRF}} \leq \epsilon(\lambda) \tag{22}$$

Game G_6. (Secure MAC). The challenger selects one Send query to embed the challenge from MAC (essentially not sampling *kt* and marking it as a challenge key). All subsequent queries that require computation of a MAC under the challenge key *kt* (either *tag* creation or *tag* verification) are answered with the help of MAC oracle. Finally, if adversary $\hat{\mathcal{A}}$ is able to trigger a false alarm and the challenge was embedded in the correct place, which happens with probability $1/q_s^*$, it is essentially forging a *tag*. Therefore,

$$\Pr[\mathsf{G}_6] \leq \Pr[\mathsf{G}_5] + q_s^* \cdot \mathsf{Adv}^{\mathsf{1uf}}_{\mathcal{B}_6,\mathsf{MAC}} \leq \epsilon(\lambda) \tag{23}$$

To conclude,

$$\mathsf{Adv}^{\mathsf{FAP}}_{\hat{\mathcal{A}},\Pi}(\lambda, n) \leq \frac{(n-1)\cdot q_s^{\diamond}}{|\mathbb{H}|} + \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_1,\mathsf{PRF}}(\lambda + \mathsf{Adv}^{\mathsf{prf}}_{\mathcal{B}_5,\mathsf{PRF}}(\lambda) + q_s^* \cdot \mathsf{Adv}^{\mathsf{1uf}}_{\mathcal{B}_6,\mathsf{MAC}}(\lambda) + \epsilon(\lambda) \tag{24}$$

□

G Security Proof of Theorem 4

Here we provide the proof of Theorem 4, showing that our generic PAPKE-2-OPAKE construction and its extension, both presented in Figure 5, securely realize the UC OPAKE functionality $\mathcal{F}_{\mathsf{OPAKE}}$[14] shown in Figure 3. Throughout the proof, we add text in box in all the places where proofs for two constructions differ.

Proof overview. To prove the Theorem 4 we must show that the following two interactions are indistinguishable by the environment: 1) "real world" interaction in which the environment $\mathcal{Z}$, adversary $\mathcal{A}$ and parties $\mathcal{P}_i$ and $\mathcal{P}_j$ execute the protocol from Fig. 5 based on the ideal functionality $\mathcal{F}_{\mathsf{PAPKE}}$ (Fig. 6); 2) "ideal world" interaction in which the simulator SIM interacts with $\mathcal{F}_{\mathsf{OPAKE}}$ and simulates the interaction between the parties $\mathcal{P}_i$ and $\mathcal{P}_j$ and $\mathcal{F}_{\mathsf{PAPKE}}$ towards $\mathcal{A}$ in the presence of the environment $\mathcal{Z}$. Our proof follows the proof strategy from [12], where authors prove that their PAPKE-2-PAKE compiler realizes standard UC PAKE functionality.

The simulator. Here we provide the description of the simulation algorithm SIM. We use "$\mathcal{F}_{\mathsf{PAPKE}}$" to refer to the sub-procedure that simulator runs internally and is described in the proof.

First message round. After receiving $(\mathsf{NewSessionS}, sid', \mathcal{P}_i, \mathcal{P}_j)$ from $\mathcal{F}_{\mathsf{OPAKE}}$, SIM sends a message (sid') from "$\mathcal{P}_i$" to "$\mathcal{P}_j$" via $\mathcal{A}$.

Second message round. After receiving $(\mathsf{NewSessionC}, sid', \mathcal{P}_j, \mathcal{P}_i)$ from $\mathcal{F}_{\mathsf{OPAKE}}$, SIM waits for a message (sid') sent to "$\mathcal{P}_j$" from $\mathcal{A}$. Then SIM sets $sid \leftarrow (\mathcal{P}_j, sid')$ and runs "$\mathcal{F}_{\mathsf{PAPKE}}$" on input "$(\mathsf{KEYGEN}, sid, \perp)$ from $\mathcal{P}_j$" to generate *apk*. Then, SIM sends a message (sid', apk) from "$\mathcal{P}_j$" to

[14] The extended protocol securely realize $\mathcal{F}_{\mathsf{OPAKE}}$ with sugarword indistinguishability.

"$\mathcal{P}_i$" via $\mathcal{A}$.

Third message round. After receiving message (sid', apk') from $\mathcal{A}$ sent to "$\mathcal{P}_i$" (apk' possibly being modified by $\mathcal{A}$), SIM first samples $k \leftarrow\!\!\$ \{0,1\}^\lambda$, sets $sid \leftarrow (\mathcal{P}_j, sid')$ and runs "$\mathcal{F}_{\mathsf{PAPKE}}$" on input "$(\mathsf{ENCRYPT}, sid, apk', \bot, k)$ from $\mathcal{P}_i$" n-times to generate $\mathbf{C}$. SIM shuffles $\mathbf{C}$. Then, in case of a correct password guess (or corrupted instance), SIM computes *key* as a result of PRF over a key k and transcript $(P_i, P_j, apk, \mathbf{C})$, or otherwise samples *key* at random, and sends $(\mathsf{NewKey}, sid', \mathcal{P}_i, key)$ to $\mathcal{F}_{\mathsf{OPAKE}}$ followed by a message $(sid', \mathbf{C})$ from "$\mathcal{P}_i$" to "$\mathcal{P}_j$" via $\mathcal{A}$.

Final round. After receiving message $(sid', \mathbf{C}')$ sent to "$\mathcal{P}_j$" from $\mathcal{A}$, SIM runs "$\mathcal{F}_{\mathsf{PAPKE}}$" n-times on input "$(\mathsf{DECRYPT}, sid, \mathbf{C}'[ind])$ from $\mathcal{P}_j$" to obtain a plaintext m. In case of a correct password guess (or corrupted instance), SIM sets $k^* \leftarrow m$ computes *key* as a result of PRF over a key k^* and transcript $(P_i, P_j, apk, \mathbf{C})$ or otherwise samples *key* at random, and sends to $\mathcal{F}_{\mathsf{OPAKE}}$ $(\mathsf{NewKey}, sid', \mathcal{P}_j, key)$.

Proof. Here we argue that the environment $\mathcal{Z}$ cannot distinguish between the two worlds.

First message round. In both worlds $(\mathsf{NewSessionS}, sid', \mathcal{P}_i, \mathcal{P}_j, \mathbf{P}_{A,B})$ sent from $\mathcal{Z}$ is treated in the same way, and hence views of $\mathcal{Z}/\mathcal{A}$ are identical for the first message round.

Second message round. This round starts when $\mathcal{Z}$ calls $(\mathsf{NewSessionC}, sid', \mathcal{P}_j, \mathcal{P}_i, pw_{A,B})$, and $\mathcal{A}$ sends (sid') to $\mathcal{P}_j$. Then, $\mathcal{A}$ receives (KEYGEN, sid) from $\mathcal{F}_{\mathsf{PAPKE}}$ (or "$\mathcal{F}_{\mathsf{PAPKE}}$") and replies with $(\mathsf{KEYCONF}, sid, apk)$. Finally, $\mathcal{A}$ receives (sid', apk) from $\mathcal{P}_j$ (or "$\mathcal{P}_j$"). Views of $\mathcal{Z}/\mathcal{A}$ are identical for the second message round.

Third message round. After receiving (sid', apk') from $\mathcal{A}$ sent to $\mathcal{P}_i$:

1) If $apk = apk'$ (where apk was provided by $\mathcal{A}$ for that session), then $\mathcal{A}$ receives n-times $(\mathsf{ENC\text{-}L}, sid, |m|)$ from $\mathcal{F}_{\mathsf{PAPKE}}$ (or "$\mathcal{F}_{\mathsf{PAPKE}}$") and replies with n different $(\mathsf{CIPHERTEXT}, sid, c_{ind})$. Then $\mathcal{P}_i$ sends to $\mathcal{P}_j$ a message $(sid', \mathbf{C})$ where $\mathbf{C} = \{c_1, ..., c_n\}$ and is shuffled with RP (save *pmap*) and outputs $(\mathsf{NewKey}, sid', \mathcal{P}_j, key)$. Note that *key* is computed by $\mathcal{P}_i$ in the real world, while in the ideal world it is chosen by $\mathcal{F}_{\mathsf{OPAKE}}$.

2) If $apk \neq apk'$, we have two cases to consider:

2a) if a record $(\mathsf{badkeys}, sid, apk_q, pwd_q)$ with $apk_q = apk'$ exists, then if $pwd_q = \mathbf{P}_{A,B}[i]$ then $(\mathsf{ENC\text{-}M}, sid, k)$ is sent to $\mathcal{A}$ from $\mathcal{F}_{\mathsf{PAPKE}}$ (or "$\mathcal{F}_{\mathsf{PAPKE}}$") or $(\mathsf{ENC\text{-}L}, sid, |m|)$ if $pwd_q \neq \mathbf{P}_{A,B}[i]$. In the ideal world, SIM sends $(\mathsf{TestPwS}, sid', \mathcal{P}_i, pwd_q)$ to $\mathcal{F}_{\mathsf{OPAKE}}$. If password guess is correct $\mathcal{F}_{\mathsf{OPAKE}}$ answers $(\mathsf{correct}, \mathbf{M})$ where the vector $\mathbf{M}$ contains position(s) of the correct password in $\mathbf{P}_{A,B}$. Based on $\mathbf{M}$, "$\mathcal{F}_{\mathsf{PAPKE}}$" sends either $(\mathsf{ENC\text{-}M}, sid, k)$ or $(\mathsf{ENC\text{-}L}, sid, |m|)$ to $\mathcal{A}$ while respecting the order of calls. Otherwise, SIM receives wrong, and "$\mathcal{F}_{\mathsf{PAPKE}}$" sends n $(\mathsf{ENC\text{-}L}, sid, |m|)$ to $\mathcal{A}$; In case of $\mathcal{F}_{\mathsf{OPAKE}}$ that captures sugarword indistinguishability, vector $\mathbf{M}$ is not provided, but since $\mathsf{DRP}(\mathbf{P}_{A,B})$ is added to the extended protocol, SIM then sends a single $(\mathsf{ENC\text{-}M}, sid, k)$ and $n-1$ $(\mathsf{ENC\text{-}L}, sid, |m|)$ in random order to $\mathcal{A}$. 2b) Otherwise, if no such record exists, $\mathcal{A}$ receives $(\mathsf{GUESS},\ sid, apk')$ from $\mathcal{F}_{\mathsf{PAPKE}}$ (or "$\mathcal{F}_{\mathsf{PAPKE}}$"), and answers with $(\mathsf{GUESS},\ sid, pwd^*)$. If password guess is correct, then $(\mathsf{ENC\text{-}L}, sid, k)$ is sent to $\mathcal{A}$ from $\mathcal{F}_{\mathsf{PAPKE}}$ or $(\mathsf{ENC\text{-}L}, sid, |m|)$ if otherwise. In the ideal world, SIM sends $(\mathsf{TestPwS}, sid', \mathcal{P}_i, pwd^*)$ and answers in the same way as in case 2a) based on the reply from $\mathcal{F}_{\mathsf{OPAKE}}$. Notice that in both cases 2a) and 2b), $\mathcal{A}$ after receiving n queries $(\mathsf{ENC\text{-}}\cdot, sid, \cdot)$ replies with n different $(\mathsf{CIPHERTEXT}, sid, c_{ind})$. Then $\mathcal{P}_i$ (or "$\mathcal{P}_i$") sends to $\mathcal{P}_j$ a message $(sid', \mathbf{C})$ and outputs $(\mathsf{NewKey}, sid', \mathcal{P}_i, key)$. Note that *key* is computed by $\mathcal{P}_i$ in the real world, while in the ideal world

it is chosen by $\mathcal{F}_{\mathsf{OPAKE}}$.

Final output round. After receiving $(sid', \mathbf{C}')$ sent to "$\mathcal{P}_j$" from $\mathcal{A}$:

1) If a record $(\mathsf{enrec}, sid, k'/\perp, \mathbf{C}'[ind])$ exists for all ciphertexts within the vector $\mathbf{C}'$, then $\mathcal{Z}$ receives $(\mathsf{NewKey}, sid', \mathcal{P}_j, key)$ from $\mathcal{P}_j$ (or "$\mathcal{P}_j$"). There are three cases to consider depending on how session key key is defined: 1a) honest execution between $\mathcal{P}_i$ and $\mathcal{P}_j$ with overlapping passwords ($pw \in \mathbf{P}^*_{A,B}$): in the real world, $\mathcal{P}_i$ will ask $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}^*_{A,B}[ind], k')$ where $\mathbf{P}^*_{A,B}[ind] = pw_{A,B}$ (among other queries) and k' is a random secret sampled by P_i that will be used with transcript as input to PRF that outputs the session key shared between $\mathcal{P}_i$ and $\mathcal{P}_j$. In the ideal world, SIM sends $(\mathsf{NewKey}, sid', \mathcal{P}_i, \perp)$ to $\mathcal{F}_{\mathsf{OPAKE}}$ - since password from $\mathcal{P}_j$ and password file $\mathcal{P}_i$ are overlapping, $\mathcal{F}_{\mathsf{OPAKE}}$ will assign them the same session key. 1b) honest execution between $\mathcal{P}_i$ and $\mathcal{P}_j$ without overlapping passwords ($pw_{A,B} \notin \mathbf{P}^*_{A,B}$): in the real world, $\mathcal{P}_i$ queries $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}^*_{A,B}[ind], k')$, where $pw_{A,B} \notin \mathbf{P}^*_{A,B}$ and k' is a random secret sampled by P_i that will create a record $(\mathsf{enrec}, sid, \perp, c')$ in $\mathcal{F}_{\mathsf{PAPKE}}$. Then, after it receives $(\mathsf{DECRYPT}, sid, \mathbf{C}[ind]')$ from $\mathcal{P}_j$, $\mathcal{F}_{\mathsf{PAPKE}}$ returns $(\mathsf{PLAINTEXT}, sid, \perp)$, and $\mathcal{P}_j$ sets $k \leftarrow\!\!\$ \{0,1\}^\lambda$ and computes the key. In the ideal world, we proceed as in case 1a) (by calling $(\mathsf{NewKey}, sid', \mathcal{P}_i, \perp)$, but since password from $\mathcal{P}_j$ and password file from $\mathcal{P}_i$ are not overlapping, key will be a random value of size λ. 1c) corrupted parties: the environment $\mathcal{Z}$ creates a corrupted $\mathcal{P}^*$, and queries $(\mathsf{ENCRYPT}, sid, apk, pwd^*_{ind}, k'_{ind})$ for any k'_{ind} in $\{0,1\}^\lambda$. In the real world, if $pwd^*_{ind} = pw_{A,B}$, k^* is set to k'_{ind} and key is computed using k^* and transcript. In case none of password guesses were correct, sample $k^* \leftarrow\!\!\$ \{0,1\}^\lambda$ and compute key. In the ideal world, for each pwd^*_{ind} that SIM receives through $(\mathsf{ENCRYPT}, sid, apk, pwd^*_{ind}, k^*_{ind})$ it submits $(\mathsf{TestPwC}, sid', \mathcal{P}_j, \mathbf{P}_{ind}, \perp)$ to $\mathcal{F}_{\mathsf{OPAKE}}$ such that $\mathbf{P}$ is built in the following way: password guess pwd^*_{ind} is inserted on all n positions in $\mathbf{P}_{ind}$. If as an answer to a TestPwC call $\mathcal{F}_{\mathsf{OPAKE}}$ returns correct, session $\mathcal{P}_j$ is compromised, k^*_{ind} that corresponds to the first correct password occurrence in $\mathbf{C}'$ is then used to compute $key = \mathsf{PRF}(k^*_{ind}, P_i, P_j, apk, \mathbf{C}')$, and SIM sends $(\mathsf{NEWKEY}, sid', key)$ to $\mathcal{F}_{\mathsf{OPAKE}}$ - the session key of $\mathcal{P}_j$ is set to key. Note that SIM using this strategy to query $\mathcal{F}_{\mathsf{OPAKE}}$ can obtain position(s) of correct password guess(es) in $\mathbf{C}^*$. This means SIM can select the correct k^*_{ind} and thus compute the appropriate key.

2) If a record $(\mathsf{enrec}, sid, k', \mathbf{C}'[ind])$ does not exist for any ciphertext in the vector $\mathbf{C}'$, then $\mathcal{A}$ receives $(\mathsf{DECRYPT}, sid, \mathbf{C}'[ind])$ from $\mathcal{F}_{\mathsf{PAPKE}}$ (or "$\mathcal{F}_{\mathsf{PAPKE}}$") and replies with its answer as follows: $(\mathsf{PLAINTEXT}, sid, k^*_{ind}, pwd^*_{ind})$. After this, $\mathcal{Z}$ will receive from $\mathcal{P}_j$ (or "$\mathcal{P}_j$") $(\mathsf{NewKey}, sid', \mathcal{P}_j, key)$, where $key = \mathsf{PRF}(k^*_{ind}, P_i, P_j, apk, \mathbf{C}')$ for the first password guess (if any out of n) such that $pwd^*_{ind} = pw_{A,B}$. In case all password guesses were wrong, sample $k^* \leftarrow\!\!\$ \{0,1\}^\lambda$ and compute key. In the ideal world, SIM sends n $(\mathsf{TestPwC}, sid', \mathcal{P}_j, \mathbf{P}_{ind})$ calls to $\mathcal{F}_{\mathsf{OPAKE}}$ in the same way as in 1c. If password guess is correct ($\mathbf{P}_{ind}[\cdot] = pw_{A,B}$), $\mathcal{F}_{\mathsf{OPAKE}}$ answers with correct, and position ind of correct guess and underlying secret k^*_{ind} are obtained, SIM computes $key = \mathsf{PRF}(k^*_{ind}, P_i, P_j, apk, \mathbf{C}')$, sends $(\mathsf{NEWKEY}, sid', key)$, and P_j's session key is set to key. Otherwise, SIM receives wrong, and P_j's session key key is picked at random and $(\mathsf{NEWKEY}, sid', key)$ is sent.

3) Modified ciphertext vector. Here we treat the case when a record $(\mathsf{enrec}, sid, k', \mathbf{C}'[ind])$ does not exist for all ciphertexts but only for some within the vector $\mathbf{C}'$. If the received vector $\mathbf{C}'$ contains any ciphertext for which records $(\mathsf{enrec}, sid, k', \mathbf{C}'[i])$ does not exists (but also contains at least one with existing records), then $\mathcal{A}$ will receive the following $(\mathsf{DECRYPT}, sid, \mathbf{C}'[i])$ from $\mathcal{F}_{\mathsf{PAPKE}}$ (or "$\mathcal{F}_{\mathsf{PAPKE}}$") and reply with $(\mathsf{PLAINTEXT}, sid, k^*_{ind}, pwd^*_{ind})$ for all ciphertexts with non-existing record.

3a) Honest password records and wrong password guesses - real world. If none of $pwd^*_{ind} = pw_{A,B}$, and it does not exist $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}^*_{A,B}[ind], k')$ where $\mathbf{P}^*_{A,B}[ind] = pw_{A,B}$ (corresponding to $\mathbf{C}'[ind]$'s with records), $\mathcal{P}_j$ sets $k^* \leftarrow\!\!\$ \{0,1\}^\lambda$ and computes the key. If none of $pwd^*_{ind} = pw_{A,B}$, but it does exist $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}^*_{A,B}[ind], k')$ where $\mathbf{P}^*_{A,B}[ind] = pw_{A,B}$ (corresponding to $\mathbf{C}'[ind]$'s with records), $\mathcal{P}_j$ sets $k^* = k'$ and computes the $key = \mathsf{PRF}^*(k^*, P_i, P_j, apk, \mathbf{C}')$. Notice that since transcripts differ between $\mathcal{P}_i$ and $\mathcal{P}_j$, the session key shared will be different due to prf-security of PRF.

3b) Honest password records and correct password guesses – real world. If there is a record $(\mathsf{PLAINTEXT}, sid, k^*_{ind}, pwd^*_{ind})$ received as an answer to $(\mathsf{DECRYPT}, sid, \mathbf{C}'[ind])$ such that pwd^*_{ind} is equal to $pw_{A,B}$, we branch in two cases:

3b1) Correct password guess in front of honest records. If $\mathbf{C}'[ind]$ is placed in vector $\mathbf{C}'$ in front of any ciphertext for which records $(\mathsf{enrec}, sid, k', \mathbf{C}'[i])$ exists, then $\mathcal{Z}$ receives $(\mathsf{NewKey}, sid', \mathcal{P}_j, key)$ from $\mathcal{P}_j$ (or "$\mathcal{P}_j$"), where $key = \mathsf{PRF}(k^*_{ind}, P_i, P_j, apk, \mathbf{C}')$;

3b2) Correct password guess after even a single honest record with overlapping password. If there exists a record as follows $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}^*_{A,B}[ind], k')$ where $\mathbf{P}^*_{A,B}[ind] = pw_{A,B}$ (corresponding to $\mathbf{C}'[ind]$'s with records) and there exists adversarial $pwd^*_q = pw_{A,B}$ such that $ind < q$, $\mathcal{P}_j$ sets $k^* = k'$ and computes the $key = \mathsf{PRF}^*(k^*, P_i, P_j, apk, \mathbf{C}')$. Notice that since transcripts differ between $\mathcal{P}_i$ and $\mathcal{P}_j$, the session key shared will be different due to prf-security of PRF. If such (ENCRYPT record does not exist, we are again in 3b1).

In the ideal world, SIM knows all adversarial password guesses and corresponding secrets (i.e. $(\mathsf{DECRYPT}, sid, \mathbf{C}'[ind])$ and corresponding $(\mathsf{PLAINTEXT}, sid, k^*_{ind}, pwd^*_{ind}))$ and all asked queries $(\mathsf{ENCRYPT}, sid, apk, \bot, k')$ with corresponding adversarial ciphertexts. Thus, SIM sends at most $n-1$ $(\mathsf{TestPwC}, sid', \mathcal{P}_j, \mathbf{P})$ where $\mathbf{P}$ is set as pwd^*_{ind} in all n positions (as in 1c and 2) so that index of a correct password guess (if exists) is known. If all password guesses to $\mathcal{F}_{\mathsf{OPAKE}}$ are wrong, SIM receives wrong, and P_j's session key key is picked at random and $(\mathsf{NEWKEY}, sid', key)$ is sent (even if there is a honest ciphertext with the correct password within the received ciphertext vector, key will be random due to PRF security.). If password guess is correct $(\mathbf{P}_{ind}[\cdot] = pw_{A,B})$, $\mathcal{F}_{\mathsf{OPAKE}}$ answers with correct, and correct password pwd^*_{ind}, its position ind in the ciphertext vector and underlying secret k^*_{ind} are obtained. Now SIM needs to test whether there exists any honest ciphertext with a correct password pwd^*_{ind} within the received ciphertext vector and does so with calling $(\mathsf{TestPwS}, sid', \mathcal{P}_i, pwd^*_{ind})$ to $\mathcal{F}_{\mathsf{OPAKE}}$ and gets the vector of indices $\mathbf{M}$ which will show him in which position in the original honest ciphertext vector was the correct password (if any). If honest ciphertext is in front of the ciphertext with the correct password guess pwd^*_{ind} from the adversary, SIM picks key randomly (secure due to PRF security), or otherwise SIM computes $key = \mathsf{PRF}(k^*_{ind}, P_i, P_j, apk, \mathbf{C}')$ and sends $(\mathsf{NEWKEY}, sid', key)$, and P_j's session key is set to key. For extended protocol SIM does not obtain the vector $\mathbf{M}$ but it can use correct or wrong indication from $\mathcal{F}_{\mathsf{OPAKE}}$ to understand if it is in password overlapping or password non-overlapping case. In case of former SIM counts the number of honest ciphertexts before the correct password guess in the ciphertext vector (this number we mark as p), and submits in $(\mathsf{NEWKEY}, sid', key)$ a random key key with probability p/n and adversarial key $key = \mathsf{PRF}(k^*_{ind}, P_i, P_j, apk, \mathbf{C}')$ in all other cases.

□

$\mathsf{Send}(C^i, M = start_client_instance)$ query:

- sample sid
- input $(\mathsf{KEYGEN}, sid, pw_{C,S})$ to C^i and receive $(\mathsf{KEYCONF}, sid, apk, \mathcal{M})$ via $\mathcal{A}$
- record (C^i, apk, sid) in list $\mathbf{L}_{clients}$
- return (C, apk)

$\mathsf{Send}(S^j, M = (C, apk))$ query:

- if $\nexists$ $(C^{i\prime}, apk', sid')$ in $\mathbf{L}_{clients}$ such that $C = C'$ and $apk = apk'$: sample sid // client impersonation
- else: $sid \leftarrow sid'$
- sample key $k \leftarrow\!\!\$ \{0,1\}^\lambda$ and generate $\mathbf{K} \leftarrow \mathsf{PRF}(k, (C, apk), n \cdot \lambda)$
- for $i = 1$ to n
  - input $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}_{C,S}[i], \mathbf{K}[i])$ to party S^j
  - receive $(\mathsf{CIPHERTEXT}, sid, \mathbf{C}[i])$ via $\mathcal{A}$
- $(\hat{\mathbf{C}}, pmap) \leftarrow \mathsf{RP}(\mathbf{C})$
- record $(S^j, C, apk, \mathbf{K}, \hat{\mathbf{C}}, pmap)$ in list $\mathbf{L}_{servers}$
- return $(S, \hat{\mathbf{C}})$

$\mathsf{Send}(C^i, M = (S, \hat{\mathbf{C}}))$ query:

- if $\nexists$ $(C^{i\prime}, *, sid')$ in $\mathbf{L}_{clients}$ s.t. $C^i = C^{i\prime}$: return $(invalid)$ // instance C^i does not exist yet
- else: $sid \leftarrow sid'$
- for $i = 1$ to n
  - input $(\mathsf{DECRYPT}, sid, \hat{\mathbf{C}}[i])$ to party C^i
  - receive $(\mathsf{PLAINTEXT}, sid, k^*)$ via $\mathcal{A}$, if $(k^* \neq \perp)$ record i and break
- if $(k^* = \perp)$ return $(abort)$
- else
  - $tr \leftarrow (C, S, apk, \hat{\mathbf{C}}, i)$
  - $(key, kt) \leftarrow \mathsf{PRF}(k^*, tr, 2 \cdot \lambda)$
  - $tag \leftarrow \mathsf{MAC.Sign}(kt, tr)$
  - $C^i.accept \leftarrow true,\quad C^i.key \leftarrow key$
  - return $(i, tag, (accept))$

$\mathsf{Send}(S^j, M = (i, tag))$ query:

- if $\nexists$ $(S^{j\prime}, C', apk', \mathbf{K}', \hat{\mathbf{C}}', pmap')$ in $\mathbf{L}_{servers}$ s.t. $S^j = S^{j\prime}$: return $(invalid)$ // invalid query, instance S^j does not exist yet
- else: $C \leftarrow C'$, $apk \leftarrow apk'$, $\mathbf{K} \leftarrow \mathbf{K}'$, $\hat{\mathbf{C}} \leftarrow \hat{\mathbf{C}}'$, $pmap \leftarrow pmap'$
- $i^* \leftarrow \mathsf{RI}(i, pmap)$
- $tr \leftarrow (C, S, apk, \hat{\mathbf{C}}, i)$
- $(key, kt) \leftarrow \mathsf{PRF}(\mathbf{K}[i^*], tr, 2 \cdot \lambda)$
- if $(\neg\mathsf{MAC.Vrfy}(kt, tr, tag)$ return $(abort)$
- else
  - $S^j.accept \leftarrow true,\quad S^j.key \leftarrow key\quad S^j.index \leftarrow i^*$
  - return $(accept)$

Figure 7: In the proofs of Theorem 1, the challenger answers $\hat{\mathcal{A}}$ queries by replacing the calls to algorithms PAPKE.KGen, PAPKE.Enc, and PAPKE.Dec in with calls to the parties in the real world via environment $\mathcal{Z}$. The pseudocode provided here specifies how Send queries are answered.

$\mathsf{Execute}(C^i, S^j)$ query:

- sample sid
- input $(\mathsf{KEYGEN}, sid, pw_{C,S})$ to C^i and receive $(\mathsf{KEYCONF}, sid, apk, \mathcal{M})$ via $\mathcal{A}$
- sample key $k \leftarrow\!\!\$ \{0,1\}^\lambda$ and generate $\mathbf{K} \leftarrow \mathsf{PRF}(k, (C, apk), n \cdot \lambda)$
- for $i = 1$ to n
  - input $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}_{C,S}[i], \mathbf{K}[i])$ to party S^j
  - receive $(\mathsf{CIPHERTEXT}, sid, \mathbf{C}[i])$ via $\mathcal{A}$
- $(\hat{\mathbf{C}}, pmap) \leftarrow \mathsf{RP}(\mathbf{C})$
- for $i = 1$ to n
  - input $(\mathsf{DECRYPT}, sid, \hat{\mathbf{C}}[i])$ to party C^i
  - receive $(\mathsf{PLAINTEXT}, sid, k^*)$ via $\mathcal{A}$
    * if $(k^* \neq \perp)$ record i and break
- $tr \leftarrow (C, S, apk, \hat{\mathbf{C}}, i)$
- $(key, kt) \leftarrow \mathsf{PRF}(k^*, tr, n \cdot \lambda)$
- $tag \leftarrow \mathsf{MAC.Sign}(kt, tr)$
- $C^i.accept \leftarrow true$, $C^i.key \leftarrow key$, $S^j.accept \leftarrow true$, $S^j.key \leftarrow key$
- $trace \leftarrow (C, S, apk, \hat{\mathbf{C}}, i, tag)$
- return $trace$

$\mathsf{Reveal}(U^i)$:

- if $U^i.accept = true$
  - return $U^i.key$

$\mathsf{Corrupt}(C, S)$ query:

- return $pw_{C,S}$

$\mathsf{Test}(U^i)$ query:

- return $U^i.key$

Figure 8: In the proof of Theorem 1, the challenger answers $\hat{\mathcal{A}}$ queries by replacing the calls to algorithms PAPKE.KGen, PAPKE.Enc, and PAPKE.Dec in with calls to the parties in the real world via environment $\mathcal{Z}$. The pseudocode provided here specifies how Execute, Reveal, Corrupt, Test queries are answered.

When $\hat{\mathcal{A}}$ queries $\mathsf{Test}(U^a)$ to a fresh instance U^a created as the result of $\mathsf{Execute}(C^i, S^j)$ where $C^i = U^a$ or $S^j = U^a$:

- if $U^a.replaced = \perp$
  - sample $key' \leftarrow\!\!\$ \{0,1\}^\lambda$
  - $C^i.replaced \leftarrow key'$
  - $S^j.replaced \leftarrow key'$
  - return $U^a.replaced$ to $\hat{\mathcal{A}}$.

Figure 9: In the proof of Theorem 1, the challenger in game G_3 randomizes session keys for Execute queries.

$\mathsf{Send}(S^j, M = (C, apk))$ query:

- if $\nexists\ (C^{i'}, apk', sid')$ in $\mathbf{L}_{clients}$ such that $C = C'$ and $apk = apk'$
  - sample sid // client impersonation attack
- else
  - $sid \leftarrow sid'$
- for $i = 1$ to n
  - $\boxed{\text{sample } \mathbf{K}[i] \leftarrow \{0,1\}^\lambda}$
- for $i = 1$ to n
  - input $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}_{C,S}[i], \mathbf{K}[i])$ to party S^j
  - receive $(\mathsf{CIPHERTEXT}, sid, \mathbf{C}[i])$ via $\mathcal{A}$
- $(\hat{\mathbf{C}}, pmap) \leftarrow \mathsf{RP}(\mathbf{C})$
- record $(S^j, C, apk, \mathbf{K}, \hat{\mathbf{C}}, pmap)$ in list $\mathbf{L}_{servers}$
- return $(S, \hat{\mathbf{C}})$

$\mathsf{Execute}(C^i, S^j)$ query:

- sample sid
- input $(\mathsf{KEYGEN}, sid, pw_{C,S})$ to C^i and receive $(\mathsf{KEYCONF}, sid, apk, \mathcal{M})$ via $\mathcal{A}$
- for $i = 1$ to n
  - $\boxed{\text{sample } \mathbf{K}[i] \leftarrow \{0,1\}^\lambda}$
- for $i = 1$ to n
  - input $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}_{C,S}[i], \mathbf{K}[i])$ to party S^j
  - receive $(\mathsf{CIPHERTEXT}, sid, \mathbf{C}[i])$ via $\mathcal{A}$
- $(\hat{\mathbf{C}}, pmap) \leftarrow \mathsf{RP}(\mathbf{C})$
- for $i = 1$ to n
  - input $(\mathsf{DECRYPT}, sid, \hat{\mathbf{C}}[i])$ to party C^i
  - receive $(\mathsf{PLAINTEXT}, sid, k^*)$ via $\mathcal{A}$
    - if $(k^* \neq \perp)$ record i and break
- $tr \leftarrow (C, S, apk, \hat{\mathbf{C}}, i)$
- $(key, kt) \leftarrow \mathsf{PRF}(k^*, tr, 2 \cdot \lambda)$
- $tag \leftarrow \mathsf{MAC.Sign}(kt, tr)$
- $C^i.accept \leftarrow true$, $C^i.key \leftarrow key$, $S^j.accept \leftarrow true$, $S^j.key \leftarrow key$
- $trace \leftarrow (C, S, apk, \hat{\mathbf{C}}, i, tag)$
- return $trace$

Figure 10: In the proofs of Theorem 2 and Theorem 3, the challenger answers $\hat{\mathcal{A}}$ queries by replacing the calls to algorithms PAPKE.KGen, PAPKE.Enc, and PAPKE.Dec in with calls to the parties in the real world via environment $\mathcal{Z}$. The pseudocode provided here specifies how $\mathsf{Send}(S^j, M = (C, apk))$ and Execute queries are answered. For Send queries to the client or key confirmation flow to the server, refer to Figure 7.

$\mathsf{Send}(S^j, M = (C, apk))$ query:

- if $\nexists\ ({C^i}', apk', sid')$ in $\mathbf{L}_{clients}$ such that $C = C'$ and $apk = apk'$
  - sample sid // client impersonation attack
- else
  - $sid \leftarrow sid'$
- for $i = 1$ to n, sample $\mathbf{K}[i] \leftarrow \{0,1\}^\lambda$
- for $i = 1$ to n
  - if $\mathbf{P}_{C,S}[i] \neq pw_{C,S}$
    - $\boxed{\text{input } (\mathsf{ENCRYPT}, sid, apk, \mathbf{P}_{C,S}[i], 0^\lambda) \text{ to party } S^j}$
  - else input $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}_{C,S}[i], \mathbf{K}[i])$ to party S^j
  - receive $(\mathsf{CIPHERTEXT}, sid, \mathbf{C}[i])$ via $\mathcal{A}$
- $(\hat{\mathbf{C}}, pmap) \leftarrow \mathsf{RP}(\mathbf{C})$
- record $(S^j, C, apk, \mathbf{K}, \hat{\mathbf{C}}, pmap)$ in list $\mathbf{L}_{servers}$
- return $(S, \hat{\mathbf{C}})$

$\mathsf{Execute}(C^i, S^j)$ query:

- sample sid
- input $(\mathsf{KEYGEN}, sid, pw_{C,S})$ to C^i and receive $(\mathsf{KEYCONF}, sid, apk, \mathcal{M})$ via $\mathcal{A}$
- for $i = 1$ to n, sample $\mathbf{K}[i] \leftarrow \{0,1\}^\lambda$
- for $i = 1$ to n
  - if $\mathbf{P}_{C,S}[i] \neq pw_{C,S}$
    - $\boxed{\text{input } (\mathsf{ENCRYPT}, sid, apk, \mathbf{P}_{C,S}[i], 0^\lambda) \text{ to party } S^j}$
  - else input $(\mathsf{ENCRYPT}, sid, apk, \mathbf{P}_{C,S}[i], \mathbf{K}[i])$ to party S^j
  - receive $(\mathsf{CIPHERTEXT}, sid, \mathbf{C}[i])$ via $\mathcal{A}$
- $(\hat{\mathbf{C}}, pmap) \leftarrow \mathsf{RP}(\mathbf{C})$
- for $i = 1$ to n
  - input $(\mathsf{DECRYPT}, sid, \hat{\mathbf{C}}[i])$ to party C^i
  - receive $(\mathsf{PLAINTEXT}, sid, k^*)$ via $\mathcal{A}$
    - if $(k^* \neq \perp)$ record i and break
- $tr \leftarrow (C, S, apk, \hat{\mathbf{C}}, i)$
- $(key, kt) \leftarrow \mathsf{PRF}(k^*, tr, 2 \cdot \lambda)$
- $tag \leftarrow \mathsf{MAC.Sign}(kt, tr)$
- $acc_{C^i} \leftarrow true, \quad key_{C^i} \leftarrow key, \quad acc_{S^j} \leftarrow true, \quad key_{S^j} \leftarrow key$
- $trace \leftarrow (C, S, apk, \hat{\mathbf{C}}, i, tag)$
- return $trace$

Figure 11: In the proof of Theorem 3, the challenger encrypts 0^λ for all honeywords. What passwords are sugarwords is known to the challenger, and ciphertexts encrypted with mismatching pairs (apk, pw) decrypt to $\perp$ according to the ideal functionality $\mathcal{F}_{\mathsf{PAPKE}}$.