CRISP: Compromise Resilient Identity-based Symmetric PAKE

Moni Naor*, Shahar Paz**, and Eyal Ronen(✉)***

Abstract. Password Authenticated Key Exchange (PAKE) protocols allow parties to establish a shared key based only on the knowledge of a password, without leaking any information about it. In this work, we propose a novel notion called "Identity-based PAKE" (iPAKE) that is resilient to the compromise of one or more parties. iPAKE protocols protect all parties in a symmetric setting, whereas in Asymmetric PAKE (aPAKE) only one party (a server) is protected. Binding each party to its identity prevents impersonation between devices with different roles and allows the revocation of compromised parties.

We further strengthen the notion by introducing "Strong iPAKE" (siPAKE), similar to "Strong aPAKE" (saPAKE), which is additionally immune to pre-computation. To mount an (inevitable) offline dictionary attack, an adversary must first compromise a device and only then start an exhaustive search over the entire password dictionary. Rather than storing its password in the clear, each party derives a password file using its identity and a secret random salt ("salted hash"). Although the random salts are independently selected, any pair of parties is able to establish a cryptographically secure shared key from these files.

We formalize iPAKE and siPAKE notions in the Universally Composable (UC) framework and propose a compiler from PAKE to iPAKE using Identity-Based Key-Agreement. We then present CRISP: a construction of siPAKE from any PAKE using bilinear groups with "Hash2Curve". We prove CRISP's UC-security in the Generic Group Model (GGM) and show that each offline password guess requires at least one pairing operation.

Keywords: Password authentication, Identity based key exchange, PAKE.

* Department of Computer Science and Applied Mathematics, Weizmann Institute of Science. Supported in part by grant 950/16 from the Israel Science Foundation. Incumbent of the Judith Kleeman Professorial Chair. Email: moni.naor@weizmann.ac.il

** School of Computer Science, Tel Aviv University. Email: shaharps@tau.ac.il

*** School of Computer Science, Tel Aviv University. Member of the Check Point Institute for Information Security. Email: er@eyalro.net

1 Introduction

Establishing secure communication over insecure channels has been the goal of cryptography for over two millennia. Following the birth of modern cryptography in Diffie and Hellman's work, the possibility of securely communicating without prearranged keys arose, but doing it in an authenticated manner remained unresolved.

Password Authenticated Key Exchange (PAKE) protocols allow parties to negotiate a strong secret key based only on the knowledge of a shared (and possibly low-entropy) password. To prevent offline attacks, PAKE protocols do not leak any information about the password to passive adversaries.

However, even in the ideal implementation, this is not entirely satisfactory since it does not deal with the issue of password storage, leaving it open to compromise. Asymmetric PAKE (aPAKE) protocols stipulated or introduced asymmetry in the setting, calling some participants "servers" and other "clients", assuming only servers store some function (hash) of the password. Only servers are afforded protection from compromising. This asymmetry does not fit all settings, as in practice it is quite common for passwords to be stored insecurely on devices/clients.[1]

Moreover, some use cases are explicitly required to be symmetric (e.g., by the Wi-Fi Alliance consortium for the WPA3 protocol [Wi-18]). In those symmetric multi-party settings, we would like to allow any pair of parties to create a secure channel, while still providing the following security guarantees:

1. *Compromise resilience.* Protect password stored by all parties.
2. *Impersonation prevention.* Different parties can have different roles or permissions that depend on their identity.
3. *Revocation.* Protect the network from compromised parties without changing the password.[2]

Our "Identity Based" solution: Combine the password with identities and "salt" when creating the password file for a party from the actual password. As part of the protocol session, we verify both the password and the declared identity of the peer (the other party).

Note that we also aim to prevent *pre-computation attacks*, where it is possible to perform most of the computation before a device is compromised. After compromising a device, the pre-computation is used to extract the password rather quickly. In saPAKE protocols such as OPAQUE [JKX18], such protection was provided only to servers, leaving clients vulnerable. See Section 1 for summary of the notions.

This work provides four main contributions: We introduce and formalize two novel notions: iPAKE ("weak" Identity-based PAKE functionality, i.e., one involving the identity in addition to the password) and siPAKE (Strong Identity-based PAKE functionality) that additionally protects from pre-computation attacks against *all* parties. We suggest an instantiation of an iPAKE protocol, and also a siPAKE protocol called CRISP (Compromise Resilient Identity-based Symmetric PAKE) and prove its UC-security in the Generic Group Model (GGM).

[1] Many users allow their browsers to save their password. Those passwords are then accessible from any program running under the same user [Wri16], which makes them an easy target for any malware. Critical flaws in password managers expose user passwords even in a locked state, and there have been multiple breaches and security issues in many popular password managers [Bed19].

[2] The revocation is effective for a limited time. Even with compromise resilience, the time required by an attacker to recover the password after a compromise is dependent on the entropy of the password and the computational cost of each guess.

Protected Parties		none	server	all
Pre-computation	vulnerable	PAKE [CHK+05]	aPAKE [GMR06]	iPAKE [Section 3]
	resilient	–	saPAKE [JKX18]	siPAKE [Section 3]

Table 1: PAKE notions under party compromise attack.

CRISP is actually a compiler that transforms any PAKE protocol into a compromise resilient, identity-based, and symmetric PAKE protocol, realizing the ideal functionality of siPAKE. It is based on a bilinear group with pairing and "Hash-to-Group". A key property of this protocol is that each offline password guess requires a computational cost equivalent to one pairing operation and that pre-computation is of no avail (in an ideal model).

1.1 Passwords and PAKE

Passwords are arguably the most widely used authentication method today. They are used in a wide range of applications from authentication on the internet (e.g., email and bank servers), wireless network encryption (e.g., Wi-Fi), and enterprise network authentication (e.g., Kerberos [NYHR05] and EAP-pwd [HZ10]). Today, most of the widely adopted protocols are vulnerable to a wide range of attacks and implementation errors that might leak the password. For example, transmitting the plain password under some secure channel (e.g., TLS or SSH connection), which usually relies upon some setup assumptions, such as Public Key Infrastructure (PKI), and may result in plaintext passwords being logged by the server [Kre19].

In recent years, a significant effort has been made to try and standardize a more secure approach for password authentication based on the use of Password Authenticated Key Exchange (PAKE) protocols (e.g. Kerberos with SPAKE-2$^+$ [MSHH18], Wi-Fi with Dragonfly [Wi-18], and combining TLS with PAKEs [BF18]).

1.2 Formalizations of PAKE

Canetti et al. [CHK+05] were the first to formalize PAKE in the Universal Composability framework (UC) [Can01]. Their ideal functionality $\mathcal{F}_{\text{PAKE}}$ (denoted $\mathcal{F}_{\text{pwKE}}$) changes each party's password with a randomly chosen key (for the session), only allowing the adversary an online attack where a single guess may be made to some party's password.

Asymmetric PAKE (aPAKE) protocols, also called Augmented PAKE, address the problem of password compromise from long term storage by introducing *asymmetry*, separating parties into "clients" and "servers". While clients supply their passwords on every session, servers use a "password file" generated in a setup phase. The password should not be extractable from such a file, *ideally not permitting servers to impersonate clients*. Since the password domain is generally considered to be rather small, this requirement is impossible: one can try validating every possible password against the password file until one is accepted. This is known as *"offline dictionary attack"*.

The formulation of Strong Asymmetric PAKE $\mathcal{F}_{\text{saPAKE}}$ [JKX18] addresses an issue with the original $\mathcal{F}_{\text{aPAKE}}$ of [GMR06], that allowed a pre-computation attack: password guesses could have been submitted before a server compromise. Most of the computational work could have been done prior to the actual compromise of the password file, allowing "instantaneous" password recovery upon compromise. For example, the attacker can pre-compute the hash value for all passwords in a given dictionary in advance, then once a server is compromised, simply find the pre-image for the

compromised hash value to find the password immediately. However, in all previous solutions, the clients (and all parties in the symmetric case) are not protected in case of compromise.

1.3 Our Contributions

In this paper, we propose to protect all PAKE participants against compromise attacks by embedding $\mathcal{F}_{\text{saPAKE}}$'s notion of security into the symmetric setting. We formalize an *ideal symmetric identity-based compromise-resilient* PAKE functionality $\mathcal{F}_{\text{siPAKE}}$. We also formalize the weaker functionality $\mathcal{F}_{\text{iPAKE}}$ that similarly to $\mathcal{F}_{\text{aPAKE}}$ allows pre-computation attacks. The notations of $\mathcal{F}_{\text{siPAKE}}$ and $\mathcal{F}_{\text{iPAKE}}$ are strictly stronger than $\mathcal{F}_{\text{saPAKE}}$ and $\mathcal{F}_{\text{aPAKE}}$, respectively.

When applying compromise resilience to both parties, a problem arises. $\mathcal{F}_{\text{saPAKE}}$ prevents impersonation of clients by a compromised server, by using the clients' knowledge of the password itself. However, in the symmetric setting, all parties use password files. To prevent impersonation of a non-compromised party we need to distinguish between password files of the different parties. Our solution is to combine identities into the password files: in the setup phase, when creating a password file for a party from the actual password, that party's identity is integrated into the file. When engaging in a protocol, the peer can verify the declared identity with respect to the established key: if both parties output the same key, then both use the same passwords and declare their true identity.

By binding the password file to a certain identity, one can revoke a compromised password file by refusing interaction with the corresponding identity. Since we prevent identity impersonation, the revoked password file is useless, except for the inevitable post-compromise offline dictionary attack.

In addition to defining symmetric compromise resilience formally, we provide an instantiation of an iPAKE protocol. Moreover, we present CRISP: a concrete realization of $\mathcal{F}_{\text{siPAKE}}$ functionality based on any symmetric PAKE protocol and a bilinear group with pairing and "Hash-to-Group". CRISP is the first PAKE protocol resilient to pre-computation attack protecting *all* parties (as opposed to protecting against pre-computation attacks of servers only). We model pre-computation resilience by binding bilinear pairing operations in the real world with offline-test queries in the ideal world, therefore testing N passwords against a compromised CRISP password file costs at least N bilinear pairing computations.

We define GGM as UC ideal functionality $\mathcal{F}_{\text{GG}}$, GGM with pairing and hash-to-group as $\mathcal{F}_{\text{GGP}}$. We prove CRISP's UC-security in GGM in the ($\mathcal{F}_{\text{PAKE}}$,$\mathcal{F}_{\text{GGP}}$)-hybrid world.

1.4 Structure of the Paper

We discuss various methods for compromise resilience in section 2. The ideal functionalities for the extensions of PAKE (including the UC Modelling of Generic Groups) are described in section 3. An iPAKE protocol appears in section 4 and the CRISP protocol is described in section 5. In section 6 we described the computational cost of running the CRISP protocol and of the brute-force attack. We also propose several optimization to the protocol. Conclusions and open problems are presented in section 7.

2 Compromise Resilience

The compromise resilience of PAKE protocols is determined by the information stored on the device in the offline phase (i.e., in the password file), and information sent during the online phase of the protocol. These determine the computational costs required from an adversary to recover the original password from the password file and the possibility of performing a trade-off between the

offline pre-computation cost (performed before the compromise of the password file) and the online computation cost (performed after the compromise). For simplicity, we assume that the adversary has a password dictionary that contains the real password, and the brute-force computational cost is proportional to the size of the dictionary. Our adversary might target multiple passwords used by different users.

We survey known methods for achieving various levels of compromise resilience and also give examples for systems using them:

1. **Plaintext password:** The password is stored as-is in the password file. No computation is required for password recovery. This is the case for the WPA3 protocol in Wi-Fi [Wi-18], and the client-side for augmented PAKEs.
2. **Hash of the password:** A one-way function of the password is computed and stored in the password file. The adversary can compute the resulting hash value of each password in the dictionary and save it in a data structure that allows for password recovery with O(1) computation cost. This can be done once, amortizing the cost of the pre-computation over multiple passwords recoveries. This option is only beneficial when using a high entropy password chosen from a password dictionary that is too large to pre-compute.
3. **Hash of the password and public identifiers:** A one-way function of the password and some public identifiers of the connection is computed and stored in the password file. For example, the public identifiers can be derived from the SSID (network name) in Wi-Fi or a combination of the server and user's name. In this case, pre-computation is still possible, but it is not possible to amortize the cost of recovering multiple passwords with different public identifiers.
4. **Hash of the password and public "salt":** A one-way function of the password and a randomly generated value ("salt") is computed and stored in the password file. The "salt" is sent as part of the PAKE protocol, and so the adversary can learn it before compromising the device. Again pre-computation is possible. This is the case for the server side in aPAKE protocols.
5. **Hash of the password and random *private* "salt":** In this case, the random "salt" is never revealed, and no pre-computation is possible. This level of protection is offered by saPAKE and our novel siPAKE protocols. A brute-force attack is inevitably possible post-compromise, as shown below.

2.1 Black Box Brute-force attack

Post-compromise brute-force dictionary attacks are inevitable for any PAKE protocol. In our attack, we assume that the correct password is in the dictionary and that the PAKE protocol fails if two different passwords are used by the two parties. The attack is straightforward:

1. Retrieve password file FILE_c from compromised device.
2. For every password π_i in the dictionary
   (a) Derive password file FILE_i.
   (b) Use FILE_c and FILE_i to simulate both parties of the PAKE protocol.
   (c) If the same key was negotiated by the simulated parties, π_i is the correct password.

The cost of each password guess in the black-box attack is the cost of deriving the password file from a password, and running the protocol twice. Note that the password file derivation can be done in pre-computation.

Functionality $\mathcal{F}_{\text{PAKE}}$, with security parameter κ, interacting with parties $\{\mathcal{P}_i\}_{i=1}^n$ and an adversary $\mathcal{S}$.

Upon (NEWSESSION, $sid, \mathcal{P}_j, \pi_i$) **from** $\mathcal{P}_i$**:**
- Send (NEWSESSION, $sid, \mathcal{P}_i, \mathcal{P}_j$) to $\mathcal{S}$
- If there is no record $\langle$SESSION, $\mathcal{P}_i, \mathcal{P}_j, \cdot, \cdot\rangle$:
  - ▷ record $\langle$SESSION, $\mathcal{P}_i, \mathcal{P}_j, \pi_i\rangle$ and mark it FRESH

Upon (TESTPWD, $sid, \mathcal{P}_i, \pi'$) **from** $\mathcal{S}$**:**
- Retrieve $\langle$SESSION, $\mathcal{P}_i, \mathcal{P}_j, \pi_i\rangle$ marked FRESH
- If $\pi_i = \pi'$: mark the session COMPROMISED and return "correct guess" to $\mathcal{S}$
- otherwise: mark the session INTERRUPTED and return "wrong guess" to $\mathcal{S}$

Upon (NEWKEY, $sid, \mathcal{P}_i, K'$) **from** $\mathcal{S}$**:**
- Retrieve $\langle$SESSION, $\mathcal{P}_i, \mathcal{P}_j, \pi_i\rangle$ not marked COMPLETED
- If it is marked COMPROMISED, or either $\mathcal{P}_i$ or $\mathcal{P}_j$ is corrupted: $K_i \leftarrow K'$
- else if it is marked FRESH, $\pi_i = \pi_j$ and there is a record $\langle$KEY, $\mathcal{P}_j, \pi_j, K_j\rangle$: $K_i \leftarrow K_j$
- otherwise: pick $K_i \xleftarrow{\text{R}} \{0,1\}^\kappa$
- If the session is marked FRESH: record $\langle$KEY, $\mathcal{P}_i, \pi_i, K_i\rangle$
- Mark the session COMPLETED and send $\langle ssid, K_i\rangle$ to $\mathcal{P}_i$

Fig. 1: Symmetric PAKE functionality $\mathcal{F}_{\text{PAKE}}$

3 Ideal Functionalities

3.1 Notation

Throughout this paper we use the following notations:
- κ is a security parameter;
- π denotes a password;
- id denotes some party's identifier;
- q is a large prime number $q \geq 2^\kappa$;
- $\mathbb{Z}_q$ denotes the field of integers modulo q, $\mathbb{Z}_q^\star = \mathbb{Z}_q \backslash \{0\}$;
- x denotes an element of $\mathbb{Z}_q$;
- F denotes a polynomial in $\mathbb{Z}_q[X]$;
- $\mathtt{X}$ denotes a formal variable in a polynomial (indeterminate);
- $\mathbb{G}$ denotes a cyclic group of order q;
- $[x]_{\mathbb{G}}$ denotes a member of group $\mathbb{G}$, identified by the exponent x of some public generator $g \in \mathbb{G}$: $[x]_{\mathbb{G}} = g^x$;
- $\{0,1\}^n$ denotes the set of binary strings of length n;
- $\{0,1\}^\star$ denotes the set of binary strings of any length;
- $\mathcal{P}$ denotes a party interacting in either real or ideal world;
- $x \xleftarrow{\text{R}} S$ denotes sampling x from uniform distribution over set S.

3.2 Symmetric PAKE Functionality

The (symmetric) PAKE functionality $\mathcal{F}_{\text{PAKE}}$, originated by [CHK+05] (denoted $\mathcal{F}_{\text{pwKE}}$ there), is depicted in Fig. 1. For clarity, we rephrased it to explicitly record keys handed to parties using KEY records.

3.3 (Strong) Identity-based PAKE Functionality

In Fig. 2 we present the Identity-based PAKE functionality $\mathcal{F}_{\text{iPAKE}}$ and Strong Identity-based PAKE functionality $\mathcal{F}_{\text{siPAKE}}$. Essentially, they preserve the symmetry of $\mathcal{F}_{\text{PAKE}}$ while adopting the

notion of password files and party compromise from the strong asymmetric PAKE functionality $\mathcal{F}_{\text{saPAKE}}$ of [JKX18] (found in Appendix A). Note that the symmetric functionality $\mathcal{F}_{\text{siPAKE}}$ is strictly stronger than the asymmetric $\mathcal{F}_{\text{saPAKE}}$: given a $\mathcal{F}_{\text{siPAKE}}$ functionality, it is trivial to realize the $\mathcal{F}_{\text{saPAKE}}$ functionality. The user party U simply computes its password file on each session, when receiving UsrSession query from the environment. However, we are not aware of any direct extension of $\mathcal{F}_{\text{saPAKE}}$ to $\mathcal{F}_{\text{siPAKE}}$.

The main addition relative to $\mathcal{F}_{\text{saPAKE}}$ is the notion of identities (id_i) assigned by the environment to parties. Without them, a single party compromise would allow the adversary to compromise any sub-session by impersonating that party. Having $\mathcal{F}_{\text{siPAKE}}$ inform a party of its peer identity prevents the attack. It is possible to revoke known compromised identities, or only permit communication with certain identities (e.g., a user will only allow connecting to servers and vice versa).

For symmetry, we restored the notation of parties as $\{\mathcal{P}_i\}_{i=1}^n$: All parties invoke StorePwdFile before starting a session and all use the password file instead of providing a password when starting a session; UsrSession query was eliminated, and SvrSession was renamed NewSession as in $\mathcal{F}_{\text{PAKE}}$. We also parametrized queries on $\mathcal{P}_i$ and $\mathcal{P}_j$ where $\mathcal{F}_{\text{saPAKE}}$ omitted them, since in the symmetric setting those queries may be applied to several parties (e.g., StealPwdFile applying to any party). On the other hand, we omit $\mathcal{P}_j$ from StorePwdFile; in our setting a password file is derived for each party independently, and is not bound to specific peers.

$\mathcal{F}_{\text{siPAKE}}$ introduces a new query OfflineComparePwd, allowing the adversary to test whether two stolen password files correspond to the same password. In the real world, such attack is always possible by an adversary simulating the protocol for those parties, and comparing the resulting keys. We argue that in most real-world settings, all parties of the same session use the same password (e.g., devices connecting to the same WiFi network), thus such a query is both inevitable and non-beneficial for the adversary.

Notice the four types of records used by $\mathcal{F}_{\text{siPAKE}}$:

1. $\langle \textbf{FILE}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$ records represent password files created for each party $\mathcal{P}_i$, and are derived from the actual password π_i of that party together with its identity id_i. Similar type of records exist in $\mathcal{F}_{\text{saPAKE}}$ (without identities) only for the server.
2. $\langle \textbf{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{id}_i, \pi_i \rangle$ records represent party $\mathcal{P}_i$'s view of a sub-session with identifier $ssid$ between $\mathcal{P}_i$ and $\mathcal{P}_j$. Similar type of records exist in $\mathcal{F}_{\text{PAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$, without identities.
3. $\langle \textbf{KEY}, ssid, \mathcal{P}_i, \pi, K_i \rangle$ records represent sub-session keys K_i created for party $\mathcal{P}_i$ participating in sub-session $ssid$ with password π_i, and whose session was not compromised or interrupted. These records were also implicitly created in $\mathcal{F}_{\text{PAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$, but appear here explicitly for the sake of clarity.
4. $\langle \textbf{IMP}, ssid, \mathcal{P}_i, \mathsf{id}' \rangle$ records represent permissions for the adversary to set the peer identity observed by party $\mathcal{P}_i$ in sub-session $ssid$ to id'. When $\mathsf{id}'=\star$ this record acts as a "wild card", permitting the adversary to select any identity.

Additionally, $\mathcal{F}_{\text{iPAKE}}$ uses the following record type:

5. $\langle \textbf{OFFLINE}, \mathcal{P}_i, \pi' \rangle$ records represent an offline-guess π' for party $\mathcal{P}_i$'s password, submitted by $\mathcal{S}$ before compromising $\mathcal{P}_i$. If $\mathcal{P}_i$ is later compromised, $\mathcal{S}$ will instantly learn if the guess was successful, i.e., $\pi'=\pi_i$.

Identities verification is implicit. When no attack is carried out by the adversary, both parties report each other's real identities. However, when the adversary succeeds in an online attack,

Functionality $\mathcal{F}_{\text{siPAKE}}$, with security parameter κ, interacting with parties $\{\mathcal{P}_i\}_{i=1}^n$ and adversary $\mathcal{S}$.

Upon (StorePwdFile, $sid, \mathsf{id}_i, \pi_i$) **from** $\mathcal{P}_i$**:**
- If there is no record $\langle \text{FILE}, \mathcal{P}_i, \cdot, \cdot \rangle$:
  - ▷ record $\langle \text{FILE}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$ and mark it UNCOMPROMISED

Upon (StealPwdFile, $sid, \mathcal{P}_i$) **from** $\mathcal{S}$**:**
- If there is a record $\langle \text{FILE}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$:
  - ▷ $\pi \leftarrow \begin{cases} \pi_i & \text{if there isa record } \langle \text{OFFLINE}, \mathcal{P}_i, \pi' \rangle \text{ with } \pi'{=}\pi_i \\ \bot & \text{otherwise} \end{cases}$
  - ▷ mark the file COMPROMISED and return $\langle$"password file stolen", $\mathsf{id}_i, \pi\rangle$ to $\mathcal{S}$
- otherwise: return "no password file" to $\mathcal{S}$

Upon (OfflineTestPwd, $sid, \mathcal{P}_i, \pi'$) **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{FILE}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$
- If it is marked COMPROMISED:
  - ▷ return "correct guess" to $\mathcal{S}$ if $\pi_i = \pi'$, and "wrong guess" otherwise
- otherwise: Record $\langle \text{OFFLINE}, \mathcal{P}_i, \pi' \rangle$

Upon (OfflineComparePwd, $sid, \mathcal{P}_i, \mathcal{P}_j$) **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{FILE}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$ and $\langle \text{FILE}, \mathcal{P}_j, \mathsf{id}_j, \pi_j \rangle$ both marked COMPROMISED
- Return "passwords match" to $\mathcal{S}$ if $\pi_i = \pi_j$, and "passwords differ" otherwise

Upon (NewSession, $sid, ssid, \mathcal{P}_j$) **from** $\mathcal{P}_i$**:**
- Retrieve $\langle \text{FILE}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$ and Send (NewSession, $ssid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{id}_i$) to $\mathcal{S}$
- If there is no record $\langle \text{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, \cdot \rangle$:
  - ▷ record $\langle \text{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ and mark it FRESH

Upon (OnlineTestPwd, $sid, ssid, \mathcal{P}_i, \pi'$) **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ marked FRESH or COMPROMISED
- Record $\langle \text{IMP}, ssid, \mathcal{P}_i, \star \rangle$
- If $\pi_i = \pi'$: mark the session COMPROMISED and return "correct guess" to $\mathcal{S}$
- otherwise: mark the session INTERRUPTED and return "wrong guess" to $\mathcal{S}$

Upon (Impersonate, $sid, ssid, \mathcal{P}_i, \mathcal{P}_k$) **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ marked FRESH or COMPROMISED
- Retrieve $\langle \text{FILE}, \mathcal{P}_k, \mathsf{id}_k, \pi_k \rangle$ marked COMPROMISED
- Record $\langle \text{IMP}, ssid, \mathcal{P}_i, \mathsf{id}_k \rangle$
- If $\pi_i = \pi_k$: mark the session COMPROMISED and return "correct guess" to $\mathcal{S}$
- otherwise: mark the session INTERRUPTED and return "wrong guess" to $\mathcal{S}$

Upon (NewKey, $sid, ssid, \mathcal{P}_i, \mathsf{id}', K'$) **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ not marked COMPLETED and $\langle \text{FILE}, \mathcal{P}_j, \mathsf{id}_j, \pi_j \rangle$
- If $\mathcal{P}_i$ is honest: ignore the query if either the session is marked FRESH and $\mathsf{id}' {\neq} \mathsf{id}_j$, or it is COMPROMISED and $\langle \text{IMP}, ssid, \mathcal{P}_i, \mathsf{id} \rangle$ is not recorded for both $\mathsf{id} {\in} \{\mathsf{id}', \star\}$
- If the session is marked COMPROMISED, or either $\mathcal{P}_i$ or $\mathcal{P}_j$ is corrupted: $K_i {\leftarrow} K'$
- else if it is marked FRESH, $\pi_i {=} \pi_j$ and there is a record $\langle \text{KEY}, ssid, \mathcal{P}_j, K_j \rangle$: $K_i {\leftarrow} K_j$
- otherwise: pick $K_i \xleftarrow{\text{R}} \{0,1\}^\kappa$
- If the session is marked FRESH: record $\langle \text{KEY}, ssid, \mathcal{P}_i, K_i \rangle$
- Mark the session COMPLETED and send $\langle ssid, \mathsf{id}', K_i \rangle$ to $\mathcal{P}_i$

Fig. 2: Functionalities $\mathcal{F}_{\text{iPAKE}}$ (full text) and $\mathcal{F}_{\text{siPAKE}}$ (grey text omitted)

it is allowed to change the reported identities. A successful OnlineTestPwd query allows the adversary to specify any identity, while a successful Impersonate query limits the choice to the impersonated party's real identity only. If any of the attacks fails, we still allow the adversary to control the reported identity, at the cost of causing each party to output an independent random key. Therefore, in the absence of a successful online attack, matching session keys indicate the reported identities are correct.

We additionally allow multiple OnlineTestPwd and Impersonate queries against the same session, as long as they succeed. This is achieved by accepting them on compromised sessions, not only fresh. Note that this permits at most one failed attempt per session, which has minor impact on security.

The $\mathcal{F}_{\text{iPAKE}}$ functionality is weaker than $\mathcal{F}_{\text{siPAKE}}$ in the sense that it permits pre-computation of OfflineTestPwd queries prior to party compromise. It is therefore only of interest when permitting more efficient constructions than its Strong counterpart. Indeed, we present a more efficient iPAKE protocol (4) realizing $\mathcal{F}_{\text{iPAKE}}$ in ROM using any cyclic group, while CRISP (5) requires bilinear groups for realizing $\mathcal{F}_{\text{siPAKE}}$ in GGM.

3.4 UC Modelling of Generic Group

The necessity of some non-black-box assumptions for proving compromise resilience in the UC framework has been previously observed (see [GMR06], [JKX18] and [BJX19]). In Hesse[Hes19] UC-realization of aPAKE is proved to be impossible under non-programmable ROM. In this work we rely on Generic Group Model assumption for proving CRISP.

The Generic Group Model (GGM) introduced by [Sho97] allows proving properties of algorithms, assuming the only permitted operations on group elements are the group operation and comparison. Hence a "generic group element" has no meaningful representation. Algorithms in GGM operate on encodings of elements, and may consult a group oracle which computes the group operation for two valid encodings, returning the encoded result. The group oracle declines queries for encodings not returned by some previous query.

Any cyclic group $\mathbb{G}$ of prime-order q with generator g can be viewed as $\{[x]_{\mathbb{G}} \mid x \in \mathbb{Z}_q\}$ with group operations $[x]_{\mathbb{G}} \odot [y]_{\mathbb{G}} = [x+y]_{\mathbb{G}}$ and $[x]_{\mathbb{G}} \oslash [y]_{\mathbb{G}} = [x-y]_{\mathbb{G}}$, unit element $[0]_{\mathbb{G}}$ and generator $[1]_{\mathbb{G}}$, using some encoding function $[\cdot]_{\mathbb{G}}\colon x \mapsto g^x$. In GGM we consider encoding functions carrying no further information about the group, e.g., encodings using random bit-strings or numbers in the range $\{0, \ldots, q-1\}$. This is in contrast to concrete groups which might have a meaningful encoding.

In order to prove CRISP's security under Universal Composition, we need to formalize GGM in terms of an ideal functionality $\mathcal{F}_{\text{GG}}$, similarly to using $\mathcal{F}_{\text{RO}}$ (depicted in Fig. 3) for proving protocols in ROM. Fig. 4 shows the basic GGM functionality $\mathcal{F}_{\text{GG}}$, which answers group operation queries (multiply/divide) on encoded elements.

Functionality $\mathcal{F}_{\text{RO}}$, parametrized by range E, interacting with parties $\{\mathcal{P}_i\}_{i=1}^n$ and adversary $\mathcal{S}$. ($\mathcal{P} \in \{\mathcal{P}_i\}_{i=1}^n \cup \{\mathcal{A}\}$)
Upon (Hash, sid, s) **from** $\mathcal{P}$**:**

- If there is no record $\langle \text{Hash}, s, h \rangle$:
  - Pick $h \xleftarrow{R} E$ and record $\langle \text{Hash}, s, h \rangle$
- Return h to $\mathcal{P}$.

Fig. 3: Random Oracle functionality $\mathcal{F}_{\text{RO}}$

Functionality $\mathcal{F}_{\text{GG}}$, parametrized by group order q, encoding set $\mathbb{E}$ ($|\mathbb{E}| \geq q$) and generator $g \in \mathbb{E}$, interacting with parties $\{\mathcal{P}_i\}_{i=1}^n$ and adversary $\mathcal{S}$. ($\mathcal{P} \in \{\mathcal{P}_i\}_{i=1}^n \cup \{\mathcal{A}\}$)
Initially, $S=\{1\}$, $[1]_{\mathbb{G}}=g$ and $[x]_{\mathbb{G}}$ is undefined for any other $x \in \mathbb{Z}_q$. Whenever $\mathcal{F}_{\text{GG}}$ references an undefined $[x]_{\mathbb{G}}$, set $[x]_{\mathbb{G}} \xleftarrow{\text{R}} \mathbb{E} \backslash S$ and insert $[x]_{\mathbb{G}}$ to S.

Upon (MULDIV, $sid, [x_1]_{\mathbb{G}}, [x_2]_{\mathbb{G}}, s$) **from** $\mathcal{P}$**:**
- $x \leftarrow x_1 + (-1)^s x_2 \mod q$
- Return $[x]_{\mathbb{G}}$ to $\mathcal{P}$

Fig. 4: Generic Group functionality $\mathcal{F}_{\text{GG}}$

For simplicity one can think of the set of encoding $\mathbb{E}=\mathbb{Z}_q$, so each exponent $x \in \mathbb{Z}_q$ is encoded as $[x]_{\mathbb{G}}=\xi$ for some $\xi \xleftarrow{\text{R}} \mathbb{Z}_q$, resulting in the encoding function being a random permutation over $\mathbb{Z}_q$, ensuring no information about oracle usage is disclosed between parties.

Note that although the group order q might be (exponentially) large, $\mathcal{F}_{\text{GG}}$ maps at most one new element per query. Also note the mapping is injective.

A bilinear group is a triplet of cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order q, with an efficiently computable bilinear map $\hat{e}:\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ satisfying the following requirements:
- **Bilinearity:** $\hat{e}(g_1^x, g_2^y) = \hat{e}(g_1, g_2)^{xy}$ for all $x, y \in \mathbb{Z}_q$.
- **Non-degeneracy:** $\hat{e}(g_1, g_2) \neq 1_T$.

where g_1, g_2 are generators for $\mathbb{G}_1, \mathbb{G}_2$ respectively. We also consider an efficiently computable isomorphism $\psi:\mathbb{G}_1 \to \mathbb{G}_2$ satisfying $\psi(g_1)=g_2$.

A hash to group (also referred to as Hash2Curve) is an efficiently computable hash function, modelled as random oracle, whose range is a group. For the bilinear setting, we consider the range $\mathbb{G}_1$.

In order to represent groups with pairing and hash into group, we suggest a modified functionality $\mathcal{F}_{\text{GGP}}$, depicted in Fig. 5, similar to the extension of GGM to bilinear groups by [BB04]. $\mathcal{F}_{\text{GGP}}$ can be queried MULDIV for each of $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$, and maintains separate encoding maps for each group. It introduces three new queries: (a) PAIRING to compute the bilinear pairing $\hat{e}$: $([x_1]_{\mathbb{G}_1}, [x_2]_{\mathbb{G}_2}) \mapsto [x_1 \cdot x_2]_{\mathbb{G}_T}$; (b) ISOMORPHISM to compute an isomorphism ψ, ψ^{-1} between $\mathbb{G}_1$ and $\mathbb{G}_2$: $[x]_{\mathbb{G}_1} \mapsto [x]_{\mathbb{G}_2}$, $[x]_{\mathbb{G}_2} \mapsto [x]_{\mathbb{G}_1}$; and (c) HASH which is a random oracle into $\mathbb{G}_1$: for each freshly queried string $s \in \{0,1\}^\star$ it picks a random exponent $x \xleftarrow{\text{R}} \mathbb{Z}_q^\star$, then returns its encoding $[x]_{\mathbb{G}_1}$.

We note that in some real-world scenarios, there is no efficiently computable isomorphism, in which case this query can be omitted (it is not required by CRISP). We still allow for ISOMORPHISM queries by the adversary to guarantee security even when such isomorphism exists.

Functionality $\mathcal{F}_{\text{GGP}}$, parametrized by group order q, encoding sets $\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_T$ ($|\mathbb{E}_j| \geq q$ for $j \in \{1,2,T\}$) and generators $g_1 \in \mathbb{E}_1$, $g_2 \in \mathbb{E}_2$, interacting with parties $\{\mathcal{P}_i\}_{i=1}^n$ and adversary $\mathcal{S}$. ($\mathcal{P} \in \{\mathcal{P}_i\}_{i=1}^n \cup \{\mathcal{A}\}$)
Initially, $S_1 = S_2 = \{1\}$, $S_T = \varnothing$, $[1]_{\mathbb{G}_1} = g_1$, $[1]_{\mathbb{G}_2} = g_2$ and $[x]_{\mathbb{G}_j}$ is undefined for any other $x \in \mathbb{Z}_q$ $j \in \{1,2,T\}$. Whenever $\mathcal{F}_{\text{GGP}}$ references an undefined $[x]_{\mathbb{G}_j}$, set $[x]_{\mathbb{G}_j} \xleftarrow{\text{R}} \mathbb{E} \backslash S_j$ and insert $[x]_{\mathbb{G}_j}$ to S_j.

Upon $(\text{MulDiv}, sid, j_{\in\{1,2,T\}}, [x_1]_{\mathbb{G}_j}, [x_2]_{\mathbb{G}_j}, s)$ **from** $\mathcal{P}$**:**
- Return $[x \leftarrow x_1 + (-1)^s x_2 \mod q]_{\mathbb{G}_j}$ to $\mathcal{P}$

Upon $(\text{Pairing}, sid, [x_1]_{\mathbb{G}_1}, [x_2]_{\mathbb{G}_2})$ **from** $\mathcal{P}$**:**
- Return $[x_T \leftarrow x_1 \cdot x_2 \mod q]_{\mathbb{G}_T}$ to $\mathcal{P}$

Upon $(\text{Isomorphism}, sid, j_{\in\{1,2\}}, [x]_{\mathbb{G}_j})$ **from** $\mathcal{S}$**:**
- Return $[x]_{\mathbb{G}_{3-j}}$ to $\mathcal{P}$

Upon (Hash, sid, s) **from** $\mathcal{P}$**:**
- If there is no record $\langle \text{HASH}, s, [x]_{\mathbb{G}_1} \rangle$:
  - ▷ pick $x \xleftarrow{\text{R}} \mathbb{Z}_q^\star$ and record $\langle \text{HASH}, s, [x]_{\mathbb{G}_1} \rangle$
- Return $[x]_{\mathbb{G}_1}$ to $\mathcal{P}$

Fig. 5: Generic Group with Pairing and Hash-to-Group functionality $\mathcal{F}_{\text{GGP}}$

4 iPAKE from IB-KA

Identity-Based Key-Agreement (IB-KA) protocols (also called Identity-Based Key-Exchange, IB-KE) allow any two parties in a multi-party setting to establish a shared key, while verifying each other's identity. They achieve this by relying upon a trusted third party called a Key Distribution Centre (KDC). During the *Setup Phase*, the KDC generates keys for each party, based on that party's identity. In the *Online Phase*, a pair of parties may communicate to derive a shared session key.

An important property of IB-KA protocols is Key Compromise Impersonation (KCI) resistance: the adversary can never impersonate an identity that does not belong to a compromised party. This is despite the fact that the adversary can inevitably impersonate any compromised party. Since KCI resistance is also mandated by iPAKE, we rely upon IB-KA to provide it.

We remark that non-interactive protocols, such as Identity-Based Non-Interactive Key-Exchange (IB-NIKE), cannot satisfy KCI resistance. This is since the adversary, having compromised some party $\mathcal{P}$, can follow $\mathcal{P}$'s steps in the protocol to derive a shared key with any other party. Hence, without interaction, an adversary can impersonate any party towards $\mathcal{P}$.

Fig. 6 depicts the IB-KA based iPAKE protocol. Note that the data flows and inputs to $\mathcal{F}_{\text{PAKE}}$ can be transmitted simultaneously, resulting in a single round of communication.

The construction is nearly identical to the IB-KA protocol from [FG10], with the following modifications:

- **KDC Simulation:** Instead of using a real KDC, each party $\mathcal{P}_i$ simulates the KDC's setup phase during its password file generation. This is achieved by taking the KDC private value y_i from $\mathcal{P}_i$'s password hash $H_1(\pi_i)$, where originally the KDC picked this value at random, and reused it for all parties.
- **PAKE Integration:** Another value, p_i, also taken from the password hash, is stored and used as input to a PAKE instance that runs in parallel. PAKE's output key α_i and the IB-KA transcript tr_i are appended to the hash invocation for generating the final key K_i.

Theorem 1. *If the Strong CDH assumption holds in $\mathbb{G}$, then the protocol in Fig. 6 UC realizes $\mathcal{F}_{\text{iPAKE}}$ in the $(\mathcal{F}_{\text{PAKE}}, \mathcal{F}_{\text{RO}})$-hybrid world.*

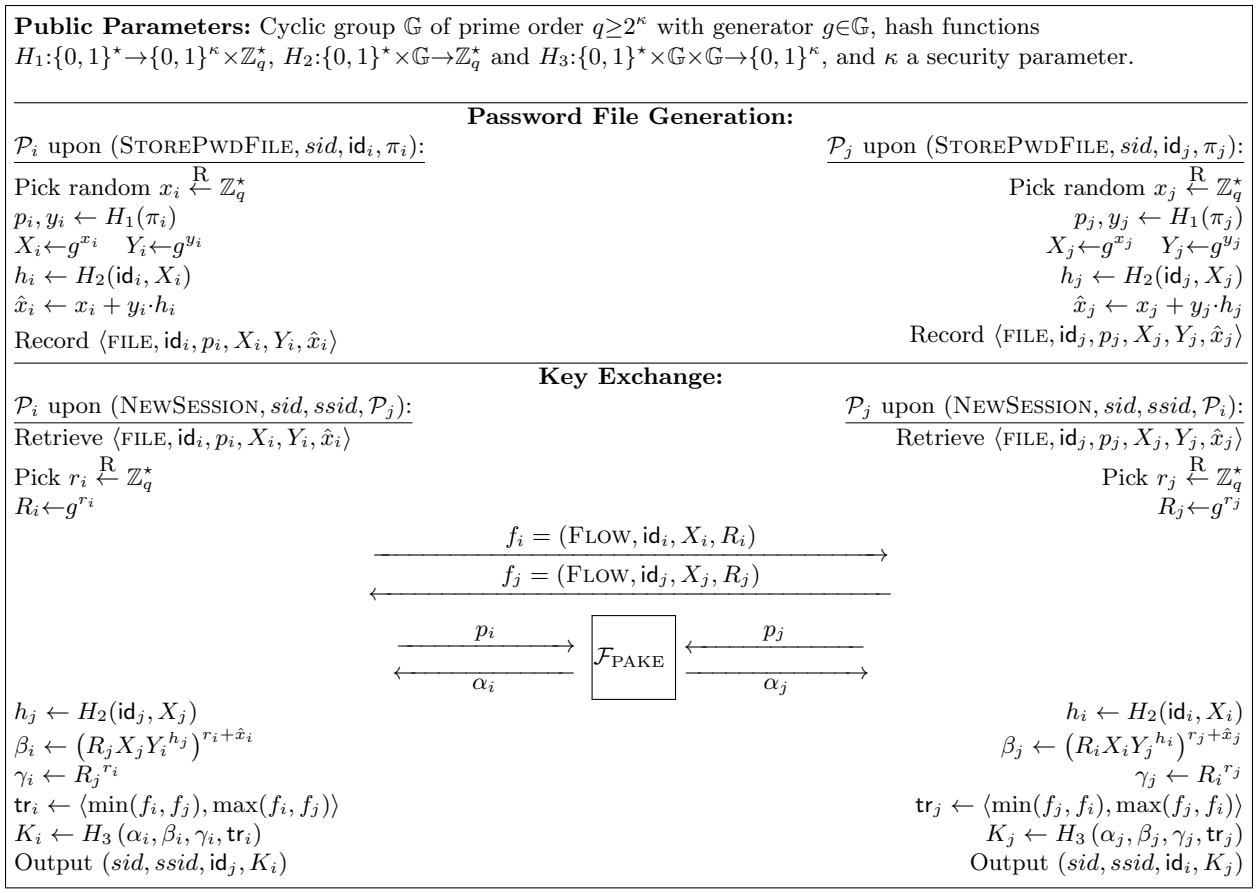


Fig. 6: IB-KA based iPAKE protocol

The proof can be found in the extended version of this paper. Note that H_1 corresponds to OfflineTestPwd, so it is advised to choose a computationally costly hash (see Section 6.1), and include *sid* (representing common parameters, e.g. user and service names) in H_1's input. We omitted *sid* for clarity.

5 CRISP

5.1 Protocol Description

CRISP is a compiler that transforms any PAKE into a compromise resilient, identity-based, and symmetric PAKE protocol. CRISP (defined in Fig. 7) is composed of the following phases:

1. **Public Parameters Generation:** In this phase, public parameters common to all parties are generated from a security parameter κ. These parameters include the bilinear groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ with hash to group functions H_1[3], H_2[4], and the PAKE protocol to be used.
2. **Password File Derivation:** In this phase, the user enters a password π_i and an identifier id_i for a party $\mathcal{P}_i$ (e.g., some device such as a personal computer, smartphone, server or access

[3] Since H_1 is invoked on the password, the note from Section 4 applies to it.

[4] hash-to-group functions (H_1 and H_2) can be realized by $\mathcal{F}_{\text{GGP}}$'s Hash queries using domain separation with different prefixes: $H_1(\pi)$ will query Hash using $s = 1||\pi$, and $H_2(\mathsf{id})$ will use $s = 2||\mathsf{id}$.

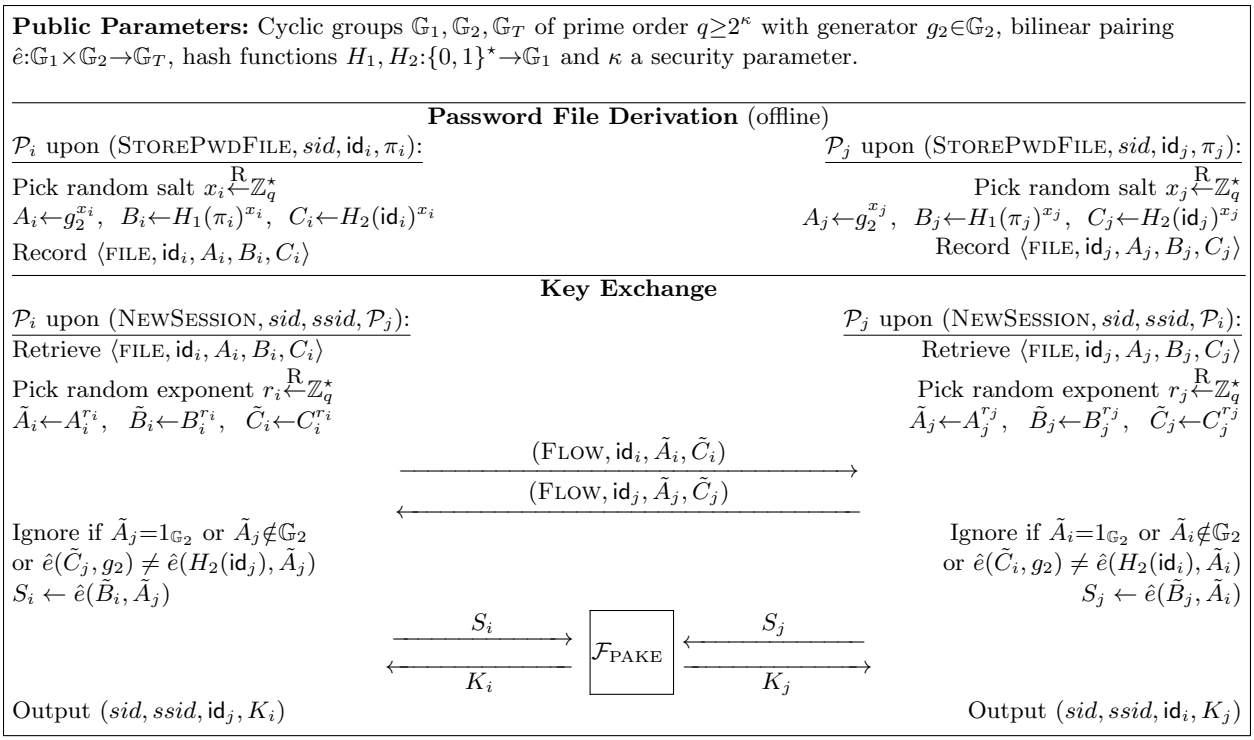


Fig. 7: CRISP protocol

point). A random salt is uniformly selected, and a password file is generated and stored by the party.

3. **Key Exchange:** In this phase, two parties, $\mathcal{P}_i$ and $\mathcal{P}_j$ engage in a sub-session to derive a shared key. This phase consists of three stages:
   (a) *Blinding.* Values from the password file are raised to the power of a randomly selected exponent. This stage can be performed once and re-used across sub-sessions (see subsection 6.3).
   (b) *Secret Exchange.* Using a single communication round (two messages), each party computes a secret value. These values depend on the generating party's password, and both parties' salt and blinding exponents.
   (c) *PAKE.* Both parties engage in a PAKE where they input their secret values as passwords to receive secure cryptographic keys.

5.2 Correctness

Honest parties $\mathcal{P}_i$, $\mathcal{P}_j$ compute a shared secret $S_i = S_j$ if and only if their password hashes match:

$$S_i = \hat{e}(\tilde{B}_i, \tilde{A}_j) = \hat{e}(H_1(\pi_i)^{x_i r_i}, g_2^{x_j r_j}) = \hat{e}(H_1(\pi_i), g_2)^{x_i r_i \cdot x_j r_j}$$
$$S_j = \hat{e}(\tilde{B}_j, \tilde{A}_i) = \hat{e}(H_1(\pi_j)^{x_j r_j}, g_2^{x_i r_i}) = \hat{e}(H_1(\pi_j), g_2)^{x_j r_j \cdot x_i r_i}$$

which reduces to the passwords themselves matching, given H_1 is injective on the password domain. Each party inputs its S_i as a password to $\mathcal{F}_{\text{PAKE}}$, which hands back the keys K_i, K_j satisfying (omitting collisions in H_1):

$$K_i = K_j \iff S_i = S_j \iff H_1(\pi_i) = H_1(\pi_j) \iff \pi_i = \pi_j$$

5.3 Intuition

Let us provide some intuition for the protocol by explaining the necessity of several components.

Blinding. The blinding stage perfectly hides the salt x_i (information theoretically) in the flow transmitted from $\mathcal{P}_i$, since $\langle \tilde{A}_i, \tilde{C}_i \rangle = \langle g_2^{\tilde{x}_i}, H_2(\mathsf{id}_i)^{\tilde{x}_i} \rangle$ for $\tilde{x}_i{=}x_i r_i$ which is a random element of $\mathbb{Z}_q^\star$. Blinding is required because transmitting the raw A_i value allows $\mathcal{A}$ to mount the following pre-computation attack:

$\mathcal{A}$ may compute the inverse map $B_{\pi'} \mapsto \pi'$ for any password guess π':

$$B_{\pi'} = \hat{e}(H_1(\pi'), A_i) = \hat{e}(H_1(\pi'), g_2)^{x_i}$$

Then after once compromising $\mathcal{P}_i$, use the map to lookup:

$$\hat{e}(B_i, g_2) = \hat{e}(H_1(\pi_i)^{x_i}, g_2) = \hat{e}(H_1(\pi_i), g_2)^{x_i}$$

Finding the correct $\pi'{=}\pi_i$ instantly. A similar attack would have also been possible if the value $\tilde{B}_i{=}B_i^{r_i}$ (or r_i) was disclosed to $\mathcal{A}$ upon compromise.

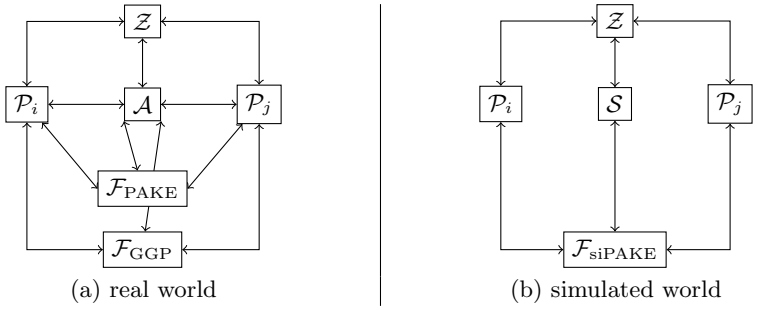
Symmetric PAKE. The key K_i should be derived from the secret S_i using $\mathcal{F}_{\text{PAKE}}$ and not some deterministic key derivation function. Consider the following attack:

Adversary $\mathcal{A}$ selects values $\tilde{A}'_j{=}g_2^{a'_j}$, $\tilde{C}'_j{=}H_2(\mathsf{id}_j)^{a'_j}$ using some private exponent a'_j. $\mathcal{A}$ can now guess a password π' and use $\tilde{A}_i$ (sent by an honest party $\mathcal{P}_i$) to compute the value $S' = \hat{e}(H_1(\pi')^{x_j}, \tilde{A}_i)$. Using S', $\mathcal{A}$ can derive a guess for the resulting key K' and test this key and password guess on encrypted messages sent by P_i. This can be repeated for multiple password guess without engaging in additional exchanges.

5.4 UC Security

Theorem 2. *Protocol CRISP as depicted in Fig. 7 UC-realizes $\mathcal{F}_{\text{siPAKE}}$ in the ($\mathcal{F}_{\text{PAKE}}$,$\mathcal{F}_{\text{GGP}}$)-hybrid world.*

We prove CRISP's UC-security by providing an ideal-world adversary $\mathcal{S}$, that simulates a real-world adversary $\mathcal{A}$ against CRISP, while only having access to the ideal functionality $\mathcal{F}_{\text{siPAKE}}$. The real and ideal world are shown in Fig. 8. The simulator $\mathcal{S}$ is detailed in Fig. 9, Fig. 10, Fig. 11 and Algorithm 1, but we first describe the strategy in high-level.



(a) real world (b) simulated world

Fig. 8: Depiction of real world running protocol CRISP with adversary $\mathcal{A}$ versus simulated world running the ideal protocol for $\mathcal{F}_{\text{siPAKE}}$ with adversary $\mathcal{S}$.

Simulator $\mathcal{S}$ proceeds as follows, interacting with environment $\mathcal{Z}$ and ideal functionality $\mathcal{F}_{\text{siPAKE}}$.
Initially, matrix M is empty, $S_1{=}S_2{=}\{1\}$, $S_T{=}\varnothing$, $[1]_{\mathbb{G}_1}{=}g_1$, $[1]_{\mathbb{G}_2}{=}g_2$ and $[F]_{\mathbb{G}_j}$ is undefined for any other polynomial F and $j{\in}\{1,2,T\}$. Whenever $\mathcal{S}$ references an undefined $[F]_{\mathbb{G}_j}$, set $[F]_{\mathbb{G}_j} \xleftarrow{\text{R}} \mathbb{E}_j \backslash S_j$ and insert $[F]_{\mathbb{G}_j}$ to S_j.

Upon (StealPwdFile, *sid*) **from** $\mathcal{Z}$ **towards** $\mathcal{P}_i$**:**
- Send (StealPwdFile, *sid*, $\mathcal{P}_i$) to $\mathcal{F}_{\text{siPAKE}}$
- If $\mathcal{F}_{\text{siPAKE}}$ returned "no password file":
  - ▷ Return this to $\mathcal{Z}$
- Otherwise, $\mathcal{F}_{\text{siPAKE}}$ returned $\langle$"password file stolen", $\mathsf{id}_i\rangle$
- Record $\langle$compromised, $\mathcal{P}_i$, $\mathsf{id}_i\rangle$
- Create variables $\mathtt{X}_i, \mathtt{Z}_i, \mathtt{I}_{\mathsf{id}_i}$ if necessary
- For each $\langle$compromised, $\mathcal{P}_j$, $\mathsf{id}_j\rangle$ with $\mathcal{P}_j{\neq}\mathcal{P}_i$:
  - ▷ Send (OfflineComparePwd, *sid*, $\mathcal{P}_i$, $\mathcal{P}_j$) to $\mathcal{F}_{\text{siPAKE}}$
  - ▷ If $\mathcal{F}_{\text{siPAKE}}$ returned "passwords match":
    - ◇ Merge variables $\mathtt{Z}_i$ and $\mathtt{Z}_j$
- Return $\langle \mathsf{id}_i, [\mathtt{X}_i]_{\mathbb{G}_2}, [\mathtt{X}_i\mathtt{Z}_i]_{\mathbb{G}_1}, [\mathtt{X}_i\mathtt{I}_{\mathsf{id}_i}]_{\mathbb{G}_1}\rangle$ to $\mathcal{Z}$

Upon (NewSession, *sid*, *ssid*, $\mathcal{P}_i$, $\mathcal{P}_j$, id_i) **from** $\mathcal{F}_{\text{siPAKE}}$**:**
- Create variables $\mathtt{X}_i, \mathtt{Z}_i, \mathtt{I}_{\mathsf{id}_i}, \mathtt{R}_{i,ssid}$ as necessary
- $f_i \leftarrow$ (Flow, id_i, $[\mathtt{X}_i\mathtt{R}_{i,ssid}]_{\mathbb{G}_2}$, $[\mathtt{X}_i\mathtt{I}_{\mathsf{id}_i}\mathtt{R}_{i,ssid}]_{\mathbb{G}_1}$)
- Send f_i to $\mathcal{Z}$ as $\mathcal{P}_i$ towards $\mathcal{P}_j$, and receive f_i' from $\mathcal{Z}$ towards $\mathcal{P}_j$
- Parse f_i' as (Flow, id', $[a']_{\mathbb{G}_2}$, $[c']_{\mathbb{G}_1}$)
- Ignore if $a'{=}0$ or $c' \neq a'{\cdot}\mathtt{I}_{\mathsf{id}'}$
- Record $\langle$sent, *ssid*, $\mathcal{P}_i$, $\mathcal{P}_j$, id', a', $c'\rangle$

Fig. 9: $\mathcal{S}$ simulating party compromise and session.

Upon (TestPwd, $sid||ssid, \mathcal{P}_i, [F]_{\mathbb{G}_T}$) **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{PAKE}}$**:**
- Retrieve $\langle$sent, *ssid*, $\mathcal{P}_j$, $\mathcal{P}_i$, id', a', $c'\rangle$
- For each $\langle$compromised, $\mathcal{P}_k$, $\mathsf{id}_k\rangle$ with $\mathsf{id}_k{=}\mathsf{id}'$:
  - ▷ If $\mathtt{Z}_k$ appears in F:
    - ◇ Send (Impersonate, *sid*, *ssid*, $\mathcal{P}_i$, $\mathcal{P}_k$) to $\mathcal{F}_{\text{siPAKE}}$
    - ◇ If $\mathcal{F}_{\text{siPAKE}}$ returned "correct guess": replace all $\mathtt{Z}_k$ with $\mathtt{Z}_i$ in F
- For each password π' queried by $H_1(\pi')$:
  - ▷ If $\mathtt{Y}_{\pi'}$ appears in F:
    - ◇ Send (OnlineTestPwd, *sid*, *ssid*, $\mathcal{P}_i$, π') to $\mathcal{F}_{\text{siPAKE}}$
    - ◇ If $\mathcal{F}_{\text{siPAKE}}$ returned "correct guess": replace all $\mathtt{Y}_{\pi'}$ with $\mathtt{Z}_i$ in F
- If $F = a'\mathtt{X}_i\mathtt{Z}_i\mathtt{R}_{i,ssid}$:
  - ▷ Return "correct guess" to $\mathcal{Z}$
- Otherwise:
  - ▷ Send (OnlineTestPwd, *sid*, *ssid*, $\mathcal{P}_i$, $\perp$) to $\mathcal{F}_{\text{siPAKE}}$
  - ▷ Return "wrong guess" to $\mathcal{Z}$

Upon (NewKey, $sid||ssid, \mathcal{P}_i, K'$) **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{PAKE}}$**:**
- Retrieve $\langle$sent, *ssid*, $\mathcal{P}_i$, $\mathcal{P}_j$, id_i', a_i', $c_i'\rangle$ and $\langle$sent, *ssid*, $\mathcal{P}_j$, $\mathcal{P}_i$, id_j', a_j', $c_j'\rangle$
- If $\nexists\, \alpha{\in}\mathbb{Z}_q^\star$ s.t. $a_i'{=}\alpha\mathtt{X}_i\mathtt{R}_{i,ssid}$ and $a_j'{=}\alpha\mathtt{X}_j\mathtt{R}_{j,ssid}$:
  - ▷ Send (OnlineTestPwd, *sid*, *ssid*, $\mathcal{P}_i$, $\perp$) to $\mathcal{F}_{\text{siPAKE}}$
- Send $\big($NewKey, *sid*, *ssid*, $\mathcal{P}_i$, id_j', $K'\big)$ to $\mathcal{F}_{\text{siPAKE}}$

Fig. 10: $\mathcal{S}$ simulating PAKE functionality $\mathcal{F}_{\text{PAKE}}$

The main challenge $\mathcal{S}$ faces is the unknown passwords assigned to parties by the environment $\mathcal{Z}$. To overcome this, $\mathcal{S}$ simulates the real-world $H_1(\pi_i) = [y_{\pi_i}]_{\mathbb{G}_1}$ using a formal variable (indeterminate) $\mathtt{Z}_i$ in the ideal-world: $H_1^\star(\pi_i) = [\mathtt{Z}_i]_{\mathbb{G}_1}$. Wherever the real world uses group encodings of

Upon $(\text{MulDiv}, sid, j_{\in\{1,2,T\}}, [F_1]_{\mathbb{G}_j}, [F_2]_{\mathbb{G}_j}, s)$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{GGP}}$**:**
- Return $[F_1 + (-1)^s \cdot F_2]_{\mathbb{G}_j}$ to $\mathcal{Z}$

Upon $(\text{Pairing}, sid, [F_1]_{\mathbb{G}_1}, [F_2]_{\mathbb{G}_2})$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{GGP}}$**:**
- $F_T \leftarrow F_1 \cdot F_2$
- Execute $\text{InsertRow}(v)$ on the coefficient vector v of F_T
- Return $[F_T]_{\mathbb{G}_T}$ to $\mathcal{Z}$

Upon $(\text{Isomorphism}, sid, j_{\in\{1,2\}}, [F]_{\mathbb{G}_j})$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{GGP}}$**:**
- Return $[F]_{\mathbb{G}_{3-j}}$ to $\mathcal{Z}$

Upon (Hash, sid, s) **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{GGP}}$**:**
- Return $\begin{cases} [\mathtt{Y}_\pi]_{\mathbb{G}_1} & s = 1||\pi \\ [\mathtt{I}_{\mathsf{id}}]_{\mathbb{G}_1} & s = 2||\mathsf{id} \end{cases}$ to $\mathcal{Z}$

Fig. 11: $\mathcal{S}$ simulating generic group functionality $\mathcal{F}_{\text{GGP}}$

exponents, $\mathcal{S}$ simulates them using encodings of polynomials with these formal variables: $[F]_{\mathbb{G}_j}$ for polynomial F.

This simulation technique, using formal variables for unknown values, is very common in GGM proofs. It "works" because $\mathcal{Z}$ is only able to detect equality of group elements, and group operations produce only linear combinations of the exponents. Two formally distinct polynomials $F_1 \neq F_2$ in the ideal world would only represent the same value in the real world in the case of a collision on some unknown value: $F_1(x) = F_2(x)$. Since these unknown values are uniformly selected over a large domain and the polynomials have low degrees, the probability of collisions is negligible.

We apply the technique for simulating several unknown values using these variables:

1. $\mathtt{X}_i$ represents party $\mathcal{P}_i$'s salt x_i.
2. $\mathtt{Y}_\pi$ represents the unknown logarithm y_π of $H_1(\pi) = g_1^{y_\pi}$.
3. $\mathtt{I}_{\mathsf{id}}$ represents the unknown logarithm ι_{id} of $H_2(\mathsf{id}) = g_1^{\iota_{\mathsf{id}}}$.
4. $\mathtt{R}_{i,ssid}$ represents party $\mathcal{P}_i$'s blinding value r_i in sub-session $ssid$.
5. $\mathtt{Z}_i$ is an alias for $\mathtt{Y}_{\pi_i}$ for party $\mathcal{P}_i$'s password π_i.

Note that some variables are created "on the fly" during the simulation. For example, upon every fresh $H_1(\pi)$ query $\mathcal{S}$ creates a new variable $\mathtt{Y}_\pi$.

Using these variables, $\mathcal{S}$ simulates the following:

- **Hash queries:** $H_1(\pi) = [\mathtt{Y}_\pi]_{\mathbb{G}_1}$ and $H_2(\mathsf{id}) = [\mathtt{I}_{\mathsf{id}}]_{\mathbb{G}_1}$.
- **Group operations:** $[F_1]_{\mathbb{G}_j} \odot [F_2]_{\mathbb{G}_j} = [F_1 + F_2]_{\mathbb{G}_j}$, $[F_1]_{\mathbb{G}_j} \oslash [F_2]_{\mathbb{G}_j} = [F_1 - F_2]_{\mathbb{G}_j}$, $\hat{e}([F_1]_{\mathbb{G}_1}, [F_2]_{\mathbb{G}_2}) = [F_1 \cdot F_2]_{\mathbb{G}_T}$, $\psi([F]_{\mathbb{G}_1}) = [F]_{\mathbb{G}_2}$ and $\psi^{-1}([F]_{\mathbb{G}_2}) = [F]_{\mathbb{G}_1}$.
- $\mathcal{P}_i$**'s password file:** $\langle \mathsf{id}_i, [\mathtt{X}_i]_{\mathbb{G}_2}, [\mathtt{X}_i\mathtt{Z}_i]_{\mathbb{G}_1}, [\mathtt{X}_i\mathtt{I}_{\mathsf{id}_i}]_{\mathbb{G}_1} \rangle$.
- **Flow from** $\mathcal{P}_i$**:** $(\text{Flow}, \mathsf{id}_i, [\mathtt{X}_i\mathtt{R}_{i,ssid}]_{\mathbb{G}_2}, [\mathtt{X}_i\mathtt{R}_{i,ssid}\mathtt{I}_{\mathsf{id}_i}]_{\mathbb{G}_1})$.

Variable Aliasing. Note that $\mathcal{S}$ uses both $\mathtt{Y}_\pi$ and $\mathtt{Z}_i$ variables: $\mathtt{Y}_\pi$ are used for simulating an evaluation of $H_1(\pi)$, while $\mathtt{Z}_i$ are used for simulating $\mathcal{P}_i$'s password file. Since $\mathtt{Y}_{\pi_i}$ and $\mathtt{Z}_i$ are distinct variables that might represent the same value in the real world, the simulation seems flawed. For instance, $\mathcal{Z}$ might ask $\mathcal{A}$ to compromise a party $\mathcal{P}_i$ and then evaluate $\hat{e}(B_i, g_2) = \hat{e}(H_1(\pi_i)^{x_i}, g_2)$ and $\hat{e}(H_1(\pi'), A_i) = \hat{e}(H_1(\pi'), g_2^{x_i})$. These encodings will be equal if and only if $\mathcal{Z}$ chose $\pi_i = \pi'$ (or it is a collision in H_1, which is found with negligible probability). Yet because of using the alias $\mathtt{Z}_i$, $\mathcal{S}$ would generate $\hat{e}(B_i, g_2) = \hat{e}([\mathtt{X}_i\mathtt{Z}_i], [1]_{\mathbb{G}_2}) = [\mathtt{X}_i\mathtt{Z}_i]_{\mathbb{G}_T}$ and $\hat{e}(H_1(\pi'), A_i) = \hat{e}([Y_{\pi'}]_{\mathbb{G}_1}, [\mathtt{X}_i]_{\mathbb{G}_2}) = [\mathtt{X}_i\mathtt{Y}_{\pi'}]_{\mathbb{G}_T}$ which are always different encodings.

Nevertheless, $\mathcal{S}$ is able to detect possible aliasing collisions: when two distinct polynomials, whose group encodings were sent to the environment $\mathcal{Z}$, become equal under substitution of $\mathtt{Z}_i$ with $\mathtt{Y}_{\pi'}$ (for some previously evaluated $H_1(\pi')$), $\mathcal{S}$ knows there will be a collision if $\pi_i{=}\pi'$. This condition can be tested by $\mathcal{S}$ using OfflineTestPwd queries, for a compromised party $\mathcal{P}_i$. When $\mathcal{F}_{\text{siPAKE}}$ replies "correct guess" to such query, $\mathcal{S}$ substitutes $\mathtt{Y}_{\pi'}$ for $\mathtt{Z}_i$ in all its data sets.

While we could have identified collisions across all $\mathcal{F}_{\text{GGP}}$ queries, we chose to limit OfflineTestPwd to only bilinear pairing evaluations (Pairing simulation), for better modelling of pre-computation resilience (see subsection 5.5). This implies that $\mathcal{S}$ needs to predict possible future collisions when simulating a pairing. This prediction is achieved by the polynomial matrix explained below.

function InsertRow(v)
  for all row w with pivot column j in M **do**
    $v \leftarrow v - v[j]{\cdot}w$
  $j \leftarrow$ SelectPivot(v)
  if $v = \vec{0}$ **then return**
  $v \leftarrow v/v[j]$
  for all row w in M **do**
    $w \leftarrow w - w[j]{\cdot}v$
  Insert row v with pivot column j to M

function SelectPivot(v)
  for all compromised party $\mathcal{P}_i$ with identifier id_i **do**
    for all passwords π' that were queried by $H_1(\pi')$ **do**
      $j_1 \leftarrow$ index of monomial $\mathtt{X}_i\mathtt{Y}_{\pi'}$
      $j_2 \leftarrow$ index of monomial $\mathtt{X}_i\mathtt{Y}_{\pi'}\mathtt{I}_{\mathsf{id}_i}$
      if $v[j_1]{\neq}0$ **or** $v[j_2]{\neq}0$ **then**
        Send (OfflineTestPwd, $sid, \mathcal{P}_i, \pi'$) to $\mathcal{F}_{\text{siPAKE}}$
        if $\mathcal{F}_{\text{siPAKE}}$ returned "wrong guess" **then**
          return $\begin{cases} j_1 & \textbf{if } v[j_1]{\neq}0 \\ j_2 & \textbf{otherwise} \end{cases}$
        Substitute variable $\mathtt{Z}_i$ with $\mathtt{Y}_{\pi'}$ in all polynomials
        Merge corresponding columns of M, v
  if some party $\mathcal{P}_i$ has been compromised **then**
    Send (OfflineTestPwd, $sid, \mathcal{P}_i, \perp$) to $\mathcal{F}_{\text{siPAKE}}$
  if $v \neq \vec{0}$ **then return** arbitrary column j having $v[j] \neq 0$

Algorithm 1: $\mathcal{S}$'s row reduction algorithm, using OfflineTestPwd queries

Polynomial Matrix. Throughout the simulation $\mathcal{S}$ maintains a matrix M whose rows correspond to polynomials in $\mathbb{G}_T$, and its columns to possible terms. A polynomial is represented in M by its coefficients stored in the appropriate columns. For example, if columns 1 to 3 correspond to terms $\mathtt{X}_i$, $\mathtt{X}_i\mathtt{Z}_i$ and $\mathtt{X}_i\mathtt{Y}_{\pi'}$ (respectively) then polynomial $F = 2\mathtt{X}_i\mathtt{Z}_i - 3\mathtt{X}_i\mathtt{Y}_{\pi'}$ will be represented in M by a row $(0, 2, -3)$.

Matrix M is extended during the simulation: when a new variable is introduced (e.g., when $\mathcal{A}$ issues a Hash query) new columns are added; and when a new polynomial is created in $\mathbb{G}_T$ by a Pairing query, another row is added to M, but using a row-reduction algorithm (see Algorithm 1) so the matrix is always kept in reduced row-echelon form. Note that when polynomials are created due to MulDiv operations in $\mathbb{G}_T$, $\mathcal{S}$ does not extend the table, as the created polynomial is by

definition a linear combination of others, so it would have been eliminated by the row-reduction algorithm. It is therefore clear that all polynomials created by $\mathcal{S}$ in $\mathbb{G}_T$ are linear combinations of the matrix rows (seen as polynomials).

When invoked by $\mathcal{A}$ to compute a pairing $\hat{e}([F_1]_{\mathbb{G}_1}, [F_2]_{\mathbb{G}_2})$, $\mathcal{S}$ first computes the product polynomial $F_T = F_1 \cdot F_2$, converts it to a coefficient vector V then applies the first step of row-reduction; that is, a linear combination of M's rows is added to V so to zero V's entries already selected as pivots for these rows. $\mathcal{S}$ then scans V for a non-zero entry corresponding to a term $\mathtt{X}_i\mathtt{Y}_{\pi'}$ (or $\mathtt{X}_i\mathtt{I}_{\mathsf{id}_i}\mathtt{Y}_{\pi'}$) for some compromised party $\mathcal{P}_i$ and a password guess π' (password guesses are taken from $\mathcal{A}$'s $H_1(\pi')$ queries). If such non-zero entry exists in V, $\mathcal{S}$ sends OfflineTestPwd query to $\mathcal{F}_{\text{siPAKE}}$ testing whether party $\mathcal{P}_i$ was assigned password π' (e.g., $\pi_i{=}\pi'$). If the guess failed, $\mathcal{S}$ chooses this as the pivot antry. Otherwise, $\mathcal{S}$ merges the variable $\mathtt{Z}_i$ with $\mathtt{Y}_{\pi'}$, and repeats the process until some test fails or no more entries of the specified form are non-zero in V. If $V{\neq}0$ and no pivot is selected, arbitrary non-zero entry is selected. $\mathcal{S}$ then applies the second step of row-reduction; that is $\mathcal{S}$ uses V to zero the entries of the selected pivot entry in other rows, and insert V as a new row to M. Finally, $\mathcal{S}$ proceeds as usual for group operations, choosing the encoding $[F_T]_{\mathbb{G}_T}$ using the original F_T (possibly having some variables merged).

Lemma 1. *The probability of collisions is negligible.*

Using the above lemma , we now prove CRISP's UC-security with respect to $\mathcal{F}_{\text{siPAKE}}$:

Proof (Theorem 2). For simplicity let us call the ($\mathcal{F}_{\text{PAKE}}$,$\mathcal{F}_{\text{GGP}}$)-hybrid world real world. For any real-world adversary $\mathcal{A}$ we describe an ideal world simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish between real-world execution of CRISP and a simulation in the ideal-world. As shown in [Can01], it suffices to prove this for a "dummy" adversary who merely passes all inputs to the environment and acts according to its instructions.

We remark that the depiction of CRISP ignored the impact of an active adversary. That is, the flow f_i transmitted by $\mathcal{P}_i$ might be received differently on $\mathcal{P}_j$. Here we denote incoming flows as f'_i (and values they carry as $\mathsf{id}'_i, \tilde{A}'_i, \tilde{C}'_i$) to account for adversarial modifications.

$$\xrightarrow{f_i = (\textsc{Flow}, \mathsf{id}_i, \tilde{A}_i, \tilde{C}_i)} \quad \xleftarrow{f'_j = (\textsc{Flow}, \mathsf{id}'_j, \tilde{A}'_j, \tilde{C}'_j)} \qquad \boxed{\mathcal{A}} \qquad \xleftarrow{f_j = (\textsc{Flow}, \mathsf{id}_j, \tilde{A}_j, \tilde{C}_j)} \quad \xrightarrow{f'_i = (\textsc{Flow}, \mathsf{id}'_i, \tilde{A}'_i, \tilde{C}'_i)}$$

Consider the simulator $\mathcal{S}$ as depicted in Fig. 9, Fig. 10 and Fig. 11. First we exclude collisions in the simulation, since by Lemma 1 those appear with negligible probability. Let us analyse $\mathcal{Z}$'s view in both the real world and the simulated world:

From Table 2 we can see that group elements observed by $\mathcal{Z}$ are encodings of polynomials in the simulated world and encodings of assignments to those polynomials in the real world. Since $\mathcal{Z}$ only observes encoded group elements, distinguishing between the worlds can only be achieved by polynomial collisions, i.e. the encodings of two polynomials differ $[F_1]_{\mathbb{G}_j} \neq [F_2]_{\mathbb{G}_j}$ while concrete values assigned to them in the real world (variable assignment $\vec{x}$) have the same encodings $[F_1(\vec{x})]_{\mathbb{G}_j} = [F_2(\vec{x})]_{\mathbb{G}_j}$. Since the encoding function is injective, this implies collisions $F_1{\neq}F_2$ while $F_1(\vec{x}){=}F_2(\vec{x})$. By Lemma 1 the probability for collisions in the simulation is negligible, so $\mathcal{Z}$ has negligible advantage in distinguishing between the encodings.

[5] We remark that $\mathcal{Z}$ does not observe S_i directly in TestPwd query, but rather the result of comparing its guess S' against S_i.

Query	Value	Real	Simulated
MulDiv	$\xi_1 \odot \xi_2$	$[a_1{+}a_2]_{\mathbb{G}_j}$	$[F_1{+}F_2]_{\mathbb{G}_j}$
	$\xi_1 \oslash \xi_2$	$[a_1{-}a_2]_{\mathbb{G}_j}$	$[F_1{-}F_2]_{\mathbb{G}_j}$
Pairing	$\hat{e}(\xi_1, \xi_2)$	$[a_1{\cdot}a_2]_{\mathbb{G}_T}$	$[F_1{\cdot}F_2]_{\mathbb{G}_T}$
Isomorphism	$\psi(\xi_1)$	$[a_1]_{\mathbb{G}_2}$	$[F_1]_{\mathbb{G}_2}$
	$\psi^{-1}(\xi_2)$	$[a_2]_{\mathbb{G}_1}$	$[F_2]_{\mathbb{G}_1}$
Hash	$H_1(\pi')$	$[y_{\pi'}]_{\mathbb{G}_1}$	$[\mathtt{Y}_{\pi'}]_{\mathbb{G}_1}$
	$H_2(\mathsf{id})$	$[\iota_{\mathsf{id}}]_{\mathbb{G}_1}$	$[\mathtt{I}_{\mathsf{id}}]_{\mathbb{G}_1}$
StealPwdFile	id_i	id_i	id_i
	$A_i = g_2^{x_i}$	$[x_i]_{\mathbb{G}_2}$	$[\mathtt{X}_i]_{\mathbb{G}_2}$
	$B_i = H_1(\pi_i)^{x_i}$	$[x_i y_{\pi_i}]_{\mathbb{G}_1}$	$[\mathtt{X}_i\mathtt{Z}_i]_{\mathbb{G}_1}$
	$C_i = H_2(\mathsf{id}_i)^{x_i}$	$[x_i \iota_{\mathsf{id}_i}]_{\mathbb{G}_1}$	$[\mathtt{X}_i\mathtt{I}_i]_{\mathbb{G}_1}$
Flow	id_i	id_i	id_i
	$\tilde{A}_i = {A_i}^{r_i}$	$[x_i r_i]_{\mathbb{G}_2}$	$[\mathtt{X}_i\mathtt{R}_{i,ssid}]_{\mathbb{G}_2}$
	$\tilde{C}_i = {C_i}^{r_i}$	$[x_i \iota_{\mathsf{id}_i} r_i]_{\mathbb{G}_1}$	$[\mathtt{X}_i\mathtt{I}_i\mathtt{R}_{i,ssid}]_{\mathbb{G}_1}$
TestPwd	$S_i = \hat{e}(\tilde{B}_i, \tilde{A}'_j)$ [5]	$[(x_i y_{\pi_i} r_i){\cdot}a'_j]_{\mathbb{G}_T}$	$[(\mathtt{X}_i\mathtt{Z}_i\mathtt{R}_{i,ssid}){\cdot}F'_j]_{\mathbb{G}_T}$

Table 2: Comparison of values viewed by $\mathcal{Z}$ in the real world versus the simulated world.

TestPwd answer. Although Table 2 refers to TestPwd query, it does not compare the responses of this query to $\mathcal{A}/\mathcal{Z}$. In the real world, this response is consistent with the state of the session: when the guess is correct ($S' = S_i$) the session becomes compromised and the response is "correct guess", while a wrong guess makes the session interrupted and causes "wrong guess" to be returned. However, when $\mathcal{S}$ simulates TestPwd there seems to be a path allowing the session to remain fresh, when neither Impersonate nor OnlineTestPwd queries are sent by $\mathcal{S}$ to $\mathcal{F}_{\text{siPAKE}}$, but the condition $F = a'\mathtt{X}_i\mathtt{Z}_i\mathtt{R}_{i,ssid}$ holds.

When $\mathcal{S}$ responds "correct guess" to a TestPwd query, $\mathcal{Z}$ provided a polynomial satisfying $F{=}a'_j\mathtt{X}_i\mathtt{Z}_i\mathtt{R}_{i,ssid}$. Recall that $\mathtt{Z_i}$ might only alias another variable $\mathtt{Z}_k$ (when $\pi_i{=}\pi_k$) or by $\mathtt{Y}_{\pi'}$ (when $\pi_i{=}\pi'$). If F contains $\mathtt{Y}_{\pi'}$ then $\mathcal{S}$ issued an OnlineTestPassword query, making the session compromised. Similar argument applies for $\mathtt{Z}_k$ where $\mathcal{P}_k$ has been compromised and having $\mathsf{id}_k{=}\mathsf{id}'$. Since $\mathcal{Z}$ only obtains polynomials with $\mathtt{Z}_k$ by compromising $\mathcal{P}_k$, we are left with the case that $\mathcal{P}_k$ has been compromised, but $\mathsf{id}_k{\neq}\mathsf{id}'$. However, in this case a'_j must contain $\mathtt{X}_k$ and therefore $c'_j{=}a'_j{\cdot}\mathtt{I}_{\mathsf{id}'}$ contains $\mathtt{X}_k{\cdot}\mathtt{I}_{\mathsf{id}'}$, which is a term $\mathcal{Z}$ cannot produce in $\mathbb{G}_1$. Thus, if $\mathcal{S}$ replies "correct guess" then the session becomes compromised in the simulated world, as well as in the real world.

If $\mathcal{S}$ answers "wrong guess" then either no queries were submitted by $\mathcal{S}$, or some query has failed and thus F contains a variable ($\mathtt{Y}_{\pi'}$ or $\mathtt{Z}_k$) that is not aliased by $\mathtt{Z}_i$. In both cases $S'{\neq}S_i$ in the real world and the session becomes interrupted. We conclude that after a TestPwd query the sessions of both the real and simulated worlds are in the same state, and the responses to $\mathcal{A}/\mathcal{S}$ are equal.

It is left to compare the outputs of parties in each world. In both worlds, the output consists of an identity and a session key: $\langle sid, ssid, \mathsf{id}, K_i\rangle$, which we will analyse separately.

Identity. The identity output by party $\mathcal{P}_i$ in the real world is id' taken from the incoming flow f'_j controlled by the adversary. In the real world, the identity is taken from the simulator's input to NewKey query. Since $\mathcal{S}$ uses the same id' in its query, we only need to show that this query is not ignored by $\mathcal{F}_{\text{siPAKE}}$ (i.e. that id' is allowed by the check in NewKey).

When the session is INTERRUPTED, no restriction is placed on the identity selected by $\mathcal{S}$. The same applies when the session is COMPROMISED due to a successful ONLINETESTPWD query. When an IMPERSONATE query caused the session to become COMPROMISED, only the impersonated identity is allowed, and indeed $\mathcal{S}$ verifies that $\mathsf{id}_k{=}\mathsf{id}'$ before impersonating party $\mathcal{P}_k$. When the session is FRESH, only the true identity of the peer party is permitted, but $\mathcal{S}$ uses id' as in the real world. Nevertheless, if $\mathsf{id}'{\neq}\mathsf{id}_j$ and $a'_j = \alpha \mathtt{X}_j \mathtt{R}_{j,ssid}$ ($\alpha{\in}\mathbb{Z}_q^\star$) then the condition

$$c'_j = a'_j \cdot \mathtt{I}_{\mathsf{id}'} = \alpha \mathtt{X}_j \mathtt{R}_{j,ssid} \cdot \mathtt{I}_{\mathsf{id}'}$$

could not have been fulfilled and the modified flow should have been ignored in both worlds.

Session Key. In the real world, K_i is party $\mathcal{P}_i$'s output of $\mathcal{F}_{\text{PAKE}}$. If the peer $\mathcal{P}_j$ is corrupted or $\mathcal{P}_i$'s session was COMPROMISED then $\mathcal{A}$'s input key K' to NEWKEY is selected. Otherwise, both parties receive the same randomly chosen key $K_i{=}K_j$ if they had the same input $S_i{=}S_j$ to NEWSESSION with FRESH sessions, or independent random keys otherwise.

In the simulated world, the key K_i selected by $\mathcal{F}_{\text{siPAKE}}$ for party $\mathcal{P}_i$ is $\mathcal{S}$'s input key K' to NEWKEY (decided by $\mathcal{Z}$) if the session is COMPROMISED or either party in the sub-session is corrupted. Otherwise, $\mathcal{F}_{\text{siPAKE}}$ generates the same random key for parties using a common password with FRESH sessions, or independent random keys otherwise.

If a session is COMPROMISED in the simulated world, then a TESTPWD query succeeded, and as shown above the session is COMPROMISED in the real-world as well.

If a session is FRESH in the simulated world then no TESTPWD query was sent, so it is also FRESH in the real world. Additionally, $a'_i{=}\alpha \mathtt{X}_i \mathtt{R}_{i,ssid}$ and $a'_j{=}\alpha \mathtt{X}_j \mathtt{R}_{j,ssid}$ ($\mathcal{S}$ will interrupt a session with modified flows, even if $\mathcal{A}$ would not send TESTPWD queries in the real world), so if the parties' passwords were identical $\pi_i{=}\pi_j$, then in the real world the inputs to $\mathcal{F}_{\text{PAKE}}$ must also be equal ($S_i{=}S_j$).

However, if a session is INTERRUPTED in the simulated world, it might be from a failing TESTPWD query, which caused the session to be INTERRUPTED in the real world as well, or because $\mathcal{S}$ sent ONLINETESTPWD with $\pi{=}\perp$ when handling NEWKEY query. This happens when the modified flows f'_i and f'_j are not using $a'_i{=}\alpha_i \mathtt{X}_i \mathtt{R}_{i,ssid}$ and $a'_j{=}\alpha_J \mathtt{X}_j \mathtt{R}_{j,ssid}$ with $\alpha_i{=}\alpha_j$. If the flows have this form with $\alpha_i{\neq}\alpha_j$, then

$$S_i = [\mathtt{X}_i \mathtt{Z}_i \mathtt{R}_{i,ssid} \cdot \alpha_j \mathtt{X}_j \mathtt{R}_{j,ssid}]_{\mathbb{G}_T} \neq [\mathtt{X}_j \mathtt{Z}_j \mathtt{R}_{j,ssid} \cdot \alpha_i \mathtt{X}_i \mathtt{R}_{i,ssid}]_{\mathbb{G}_T} = S_j$$

in the simulated world, regardless of $\mathtt{Z}_i{=}\mathtt{Z}_j$. Thus, in the real world $S_i{\neq}S_j$, since assignment collisions are negligible. If the modifications (a'_i and a'_j) do not take this form, then since there are no other polynomials with $\mathtt{R}_{i,ssid}$ and $\mathtt{R}_{j,ssid}$, $S_i{\neq}S_j$ in both real and ideal world (again due to assignment collisions being negligible).

We proceed to prove the lemmas:

Proof (Lemma 1). There are three types of possible collisions:

1. **Hash queries.** Since HASH responses are taken from the uniform distribution over $\mathbb{Z}_q^\star$, the probability of such collisions is bound by $\frac{q_H}{q-1}$, where q_H is the number of HASH queries (polynomial in κ) and $q \geq 2^\kappa$.
2. **Variable Aliasing.** By Lemma 2, there are no aliasing collisions in the simulation.

3. **Variable Assignment.** Polynomials created by $\mathcal{S}$ for elements in $\mathbb{G}_1$ and $\mathbb{G}_2$ have maximal degree 3. MulDiv and Isomorphism queries cannot increase the degree, and Pairing allows creating polynomials in $\mathbb{G}_T$ adding the input degrees. Therefore, the maximal degree of any polynomial whose encoding is observed by $\mathcal{Z}$ is 3+3=6.
Since in the real world the exponents (corresponding to variables in the simulated world) are taken from the uniform distribution over $\mathbb{Z}_q^\star$, the probability of assignment collisions $F_i(\vec{\mathtt{X}}) = F_j(\vec{\mathtt{X}})$ for some variable assignment $\vec{\mathtt{X}}$, is bound by:

$$\Pr_{\vec{\mathtt{X}} \xleftarrow{R} \mathbb{Z}_q^\star}\left[\exists_{i\neq j}\ F_i(\vec{\mathtt{X}}) = F_j(\vec{\mathtt{X}})\right] \leq \sum_{i\neq j} \Pr_{\vec{\mathtt{X}} \xleftarrow{R} \mathbb{Z}_q^\star}\left[(F_i - F_j)(\vec{\mathtt{X}}) = 0\right]$$
$$\leq \sum_{i\neq j} \frac{\deg(F_i - F_j)}{|\mathbb{Z}_q^\star|} \leq \binom{N}{2}\frac{6}{q-1}$$

which is negligible in κ, where N denotes the number of distinct polynomials created in the simulation.

Lemma 2. *There are no aliasing collisions in the simulation.*

Proof. Variable aliasing collisions take the form $\mathtt{Z}_i{=}\mathtt{Y}_{\pi_i}$, where π_i is the password assigned by the environment to party $\mathcal{P}_i$. They arise from defining separate formal variables to represent the logarithm of $H_1(\pi)$ for (a) each party $\mathcal{P}_i$'s password π_i (unknown to the simulator) and (b) each adversary invocation of H_1 on some password guess π'.

Note that this implies possible aliasing between parties: $\mathtt{Z}_i{=}\mathtt{Z}_j$ when both parties are assigned the same password: $\pi_i{=}\pi_j$.

Since the proof for Theorem 2 has already dealt with aliasing in TestPwd queries, it remains to show no collisions are possible for group encoding of elements. The following basic polynomials are accessible to the adversary after the corresponding queries:

1	public generator
$\mathtt{X}_i$ $\mathtt{X}_i{\cdot}\mathtt{I}_{\mathsf{id}_i}$ $\mathtt{X}_i{\cdot}\mathtt{Z}_i$	$\mathcal{F}_{\text{PAKE}}$'s StealPwdFile query
$\mathtt{X}_i{\cdot}\mathtt{R}_{i,ssid}$ $\mathtt{X}_i{\cdot}\mathtt{R}_{i,ssid}{\cdot}\mathtt{I}_{\mathsf{id}_i}$	Flow message from $\mathcal{P}_i$
$\mathtt{Y}_\pi$	$\mathcal{F}_{\text{GGP}}$'s Hash query for $H_1(\pi)$
$\mathtt{I}_{\mathsf{id}}$	$\mathcal{F}_{\text{GGP}}$'s Hash query for $H_2(\mathsf{id})$

Recall that polynomials in $\mathbb{G}_1, \mathbb{G}_2$ are simply linear combinations of these basic polynomials, and polynomials in $\mathbb{G}_T$ are linear combinations of their pairwise products. The only basic polynomial in which $\mathtt{Z}_i$ appears is $\mathtt{X}_i{\cdot}\mathtt{Z}_i$, which cannot collide (under aliases) with anything but $\mathtt{X}_i{\cdot}\mathtt{Y}_{\pi_i}$ or $\mathtt{X}_i{\cdot}\mathtt{Z}_j$. Since such polynomials are not given, no aliasing collisions are possible in $\mathbb{G}_1, \mathbb{G}_2$. Since $\mathbb{G}_T$ polynomials are combinations of products, only only linear combinations of the following basic collisions are possible under aliasing ($\mathtt{Z}_i{=}\mathtt{Y}_{\pi_i}$):

1. $(\mathtt{X}_i{\cdot}\mathtt{Z}_i)\cdot(1) = (\mathtt{X}_i)\cdot(\mathtt{Y}_{\pi'})$ where $\mathtt{Z}_i{=}\mathtt{Y}_{\pi'}$ ($\pi_i{=}\pi'$)
2. $(\mathtt{X}_i{\cdot}\mathtt{Z}_i)\cdot(\mathtt{I}_{\mathsf{id}'}) = (\mathtt{X}_i{\cdot}\mathtt{I}_{\mathsf{id}_i})\cdot(\mathtt{Y}_{\pi'})$ where $\mathtt{Z}_i{=}\mathtt{Y}_{\pi'}$ and $\mathsf{id}'{=}\mathsf{id}_i$
3. $(\mathtt{X}_i{\cdot}\mathtt{Z}_i)\cdot(\mathtt{X}_j) = (\mathtt{X}_j{\cdot}\mathtt{Z}_j)\cdot(\mathtt{X}_i)$ where $\mathtt{Z}_i{=}\mathtt{Z}_j$ ($\pi_i{=}\pi_j$)

4. $(\mathtt{X}_i \cdot \mathtt{Z}_i) \cdot (\mathtt{X}_j \cdot \mathtt{I}_{\mathsf{id}_j}) = (\mathtt{X}_j \cdot \mathtt{Z}_j) \cdot (\mathtt{X}_i \cdot \mathtt{I}_{\mathsf{id}_i})$ where $\mathtt{Z}_i = \mathtt{Z}_j$ and $\mathsf{id}_i = \mathsf{id}_j$

Recall that the simulator $\mathcal{S}$ issues OfflineComparePwd queries comparing the password of freshly compromised party $\mathcal{P}_i$ with those of previously compromised parties, therefore eliminating collisions of the form $\mathtt{Z}_i = \mathtt{Z}_j$ altogether. It is left to prove only for type 1 and 2 aliasing collisions.

Since every polynomial in $\mathbb{G}_T$ is a linear combination of F_T polynomials created in Pairing query, it is also a linear combination of matrix M's rows.

Matrix M created by $\mathcal{S}$ in Pairing queries is kept in row echelon form (see Algorithm 1), therefore each row r is represented by a pivot monomial P_r, corresponding to the pivot column holding 1. Consider a collision (under aliases):

$$0 = \sum \alpha_r F_r \ (\exists_r \alpha_r \neq 0)$$

where F_r is the polynomial corresponding to the r'th row. For every row r whose pivot P_r is non-collidable, the coefficient α_r must be 0, since by the row echelon form, pivots are unique. Therefore if $\alpha_r \neq 0$ for some row r, then the pivot P_r is collidable.

Recall that monomials containing $\mathtt{X}_i \mathtt{Y}_{\pi'}$ are only selected by $\mathcal{S}$ as pivots after an OfflineTestPwd query failed, implying that $\pi_i \neq \pi'$ and hence such monomials are not collidable. Therefore, for a row r with $\alpha_r \neq 0$ the pivot P_r must either be $\mathtt{X}_i \mathtt{Z}_i$ or $\mathtt{X}_i \mathtt{Z}_i \mathtt{I}_{\mathsf{id}_i}$ which collides with $\mathtt{X}_i \mathtt{Y}_{\pi_i}$ or $\mathtt{X}_i \mathtt{Y}_{\pi_i} \mathtt{I}_{\mathsf{id}_i}$ (respectively).

However, if there is a row r' that has non-zero coefficient for $\mathtt{X}_i \mathtt{Y}_{\pi_i}$ or $\mathtt{X}_i \mathtt{Y}_{\pi_i} \mathtt{I}_{\mathsf{id}_i}$, then $\mathcal{S}$ must have queried OfflineTestPwd for $\mathcal{P}_i$ with π_i, and this test must have succeeded, causing $\mathcal{S}$ to merge $\mathtt{Z}_i$ with $\mathtt{Y}_{\pi'}$. In this case $\alpha_r = 0$ since the pivot P_r is not collidable after the merge.

5.5 Pre-Computation Resilience

We resume by considering pre-computation resilience. As discussed earlier, the original UC framework does not limit the ideal-world adversary $\mathcal{S}$ from testing every possible password via OfflineTestPwd queries once compromising a party. This allows a very strong simulator who can instantly reconstruct the party's password once compromised with StealPwdFile. The solution is to bind offline tests with some real-world work, by keeping the environment aware of OfflineTestPwd queries in the ideal world and of the corresponding real-world computation. For instance, [JKX18] requires OPRF query for each tested password, while [BJX19] shows linear relation between number of offline tests and Generic Group operations.

In this work we will bind each ideal-world OfflineTestPwd query with a bilinear pairing computed (after a compromise) in the real-world using Pairing query to $\mathcal{F}_{\text{GGP}}$. We stress that it suffices to prove this for failed offline tests, since successful tests may happen at most once per compromised party's password. In real-life scenarios, where all parties share a single password, there might only be one successful offline test.

It can be easily observed that $\mathcal{S}$ never sends OfflineTestPwd queries, except when simulating $\mathcal{F}_{\text{GGP}}$'s Pairing query, where a sequence of such offline tests is sent to $\mathcal{F}_{\text{siPAKE}}$. It is also easy to see that this sequence ends when $\mathcal{F}_{\text{siPAKE}}$ replies with "Correct guess". If all tests are answered on the affirmative and some party $\mathcal{P}_i$ has been compromised, then $\mathcal{S}$ sends a final query with $\pi = \perp$ resulting in "Wrong guess" from $\mathcal{F}_{\text{siPAKE}}$.

Therefore there is a one-to-one mapping between bilinear pairings computed by the real-world adversary after a compromise, and OfflineTestPwd queries sent by the ideal-world adversary $\mathcal{S}$ when simulating those computations. As a result, an environment $\mathcal{Z}$ equipped with awareness

of failed offline tests (in the ideal-world) and of pairings (in the real-world) gains no advantage distinguishing these executions.

6 Computational Cost

The computational cost for our iPAKE compiler and CRISP are summarized in the following table:

		iPAKE	CRISP
Password File	Derivation	2H + 2E	2H + 3E
Key Exchange	Blinding	2E	3E
	Identity check		1H + 2P
	Key generation	2E + PAKE	1P + PAKE
Offline Test	Pre-Compromise	1H	1H
	Post-Compromise	0	1P

Here H, E, and P denote Hash (or Hash-to-Group), Exponentiation, and Pairing costs, respectively, and PAKE denotes the additional cost of the underlying PAKE used. We ignore the cost of group multiplications and the difference between Hash and Hash-to-Group.

6.1 Password Hardening for Pre-Compromise

Common password hardening techniques (e.g., PBKDF2 [MKR17], Argon2 [BDK16], and scrypt [PJ16]) are used in the process of deriving a key from a password to increase the cost of brute-force attacks. As mentioned in Section 2 both our iPAKE and CRISP protocols can use those techniques to increase the cost of the pre-compromise computation phase of the attack (pre-computation). In iPAKE we can use any of those hardening techniques to implement the hash function denoted as H_1. Similarly, in the CRISP protocol, we can use those techniques as the first step in implementing the Hash-to-Group function denoted as H_1. As those functions are only called once in the password file derivation phase, we can increase their cost without increasing the cost of the online phase of the protocol.

6.2 Password Hardening for Post-Compromise

In addition to the cost of the pre-compromise phase, the CRISP protocol also requires the attacker to perform a post-compromise phase. The offline test post-compromise cost above is taken from the lower bound proved in Section 5.5. This is also an upper bound for CRISP, since having compromised a password file, an adversary can check for any password guess π' if:

$$\hat{e}(B_i, g_2) \stackrel{?}{=} \hat{e}(H_1(\pi'), A_i)$$

The left-hand side can be computed once and re-used for different guesses. The right-hand side must be computed per-password, but the invocation of H_1 can be done prior to the compromise.

We stress that a pairing operation is preferred over exponentiation when considering the cost of an offline test. While the latter can be amortized, e.g., by using a window implementation, to the best of our knowledge, optimization for pairing with a fixed point only speed the computation by 37% [CS10]. Moreover, pairing requires more memory than simple point multiplication and is harder to accelerate using GPUs [PL13].

In order to increase the difficulty of offline tests (password hardening), we cannot use a method such as iterative hashing, as was done in [JKX18]. However, by using larger group size, we can increase the cost of each pairing, and hence of offline tests. Although coarse-grained, this allows some trade-off between resilience to compromise and computational complexity of CRISP.

6.3 CRISP Optimization

We can optimize the CRISP protocol in several ways to reduce the added computational cost and latency.

Blinding operation The blinding of the group elements from the password file requires three exponentiation. However, we can amortize this cost across multiple runs of the protocol. The blinding can be calculated once every period (e.g., every reboot of the devices or once an hour), and the same blinding value can be reused multiple times. The PAKE protocol will still return a fresh key for each run and provide forward secrecy. Moreover, we can calculate those blinding values offline, in preparation for a protocol. This does not reduce the overall computational cost but reduces the protocol's latency.

Identity Verification A substantial part of the added computational cost of the protocol is the identity verification that requires two pairing operations. There are two main options to optimize this cost:

1. Reducing latency – The verification does not affect the derived key or the subsequent messages. This implies we can continue with the protocol by sending the next message and postpone the verification for later, while we wait for the other party to respond. The total computational cost remains the same, but the latency (or running time) of the protocol is reduced.
2. Verification delegation – Any party that receives the protocol messages, can verify the identity appearing in it (verification is only based on the identity and blinded values). We consider the following scenario, where we have a broadcast network with many low-end devices (e.g., IoT devices) and one or more high-end devices (a controller or bridge). The bridge can perform the identity verification for all protocols in the network, and alert the user if any verification fails.

Number of Messages CRISP requires two additional messages compared to the underlying PAKE. We can trivially reduce this to one additional message. The first message remains the same. However, once receiving it, the other party can already derive the shared secret S_i and prepare the first PAKE message. Consequently, CRISP's second message can be combined with the first PAKE message, resulting in a single additional message, and again reducing the total latency of the protocol. As any PAKE protocol requires at least two (simultaneous) messages [KV11], we can implement a siPAKE protocol using only three (albeit sequential) messages.

7 Conclusions and Open Problems

In this paper, we have formalized the novel notions of iPAKE and siPAKE. We also introduced and proved the security of CRIPS protocol which realizes $\mathcal{F}_{\text{siPAKE}}$. Moreover, we have shown that for CRISP each offline password guess requires a computational cost equivalent to one pairing operation.

The following open problems arise:

Two message siPAKE protocol. In subsection 6.3, we showed how CRISP requires only three messages. As shown in section 4, iPAKE can be realized with only two messages. It is an open problem to either prove a lower bound of three messages or to implement a two message siPAKE.

Optimal bound on the cost of brute-force attack. In subsection 2.1 we have shown a black-box post-compromise brute-force attack on any PAKE protocol. The computational cost of the attack is *two* runs of the PAKE protocol for each offline password guess. However, brute-forcing currently known PAKE implementations requires a computational cost equivalent to only *one* run

of the protocol. It remains an open problem to either find a more efficient black-box attack or to implement a more resilient PAKE.

Fine grained password hardening. CRISP allows for a coarse-grained password hardening by changing the pairing group (e.g., using curves of larger size). How to allow for a fine-grained password hardening (e.g., such as iterative hashing) secure against pre-comptuation remains an open problem.

Acknowledgments

We thank Nir Bitansky, Ran Canetti, Ben Fisch, Hugo Krawczyk, Eylon Yogev for many helpful discussions and insightful ideas.

References

BB04. Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 56–73, 2004.

BDK16. Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *EuroS&P*, pages 292–302. IEEE, 2016.

Bed19. Adrian Bednarek. Password managers: Under the hood of secrets management. Retrieved 9 February 2020 from https://www.ise.io/casestudies/password-manager-hacking/, 2019.

BF18. Richard Barnes and Owen Friel. Usage of pake with tls 1.3. Internet-Draft draft-barnes-tls-pake-04, IETF Secretariat, July 2018. http://www.ietf.org/internet-drafts/draft-barnes-tls-pake-04.txt.

BJX19. Tatiana Bradley, Stanislaw Jarecki, and Jiayu Xu. Strong asymmetric PAKE based on trapdoor CKEM. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, pages 798–825, 2019.

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

CHK+05. Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 404–421, 2005.

CS10. Craig Costello and Douglas Stebila. Fixed argument pairings. In *LATINCRYPT*, volume 6212 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2010.

FG10. Dario Fiore and Rosario Gennaro. Identity-based key exchange protocols without pairings. *Trans. Comput. Sci.*, 10:42–77, 2010.

GMR06. Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2006.

Hes19. Julia Hesse. Separating standard and asymmetric password-authenticated key exchange. *IACR Cryptology ePrint Archive*, 2019:1064, 2019.

HZ10. D. Harkins and G. Zorn. Extensible authentication protocol (eap) authentication using only a password. RFC 5931, RFC Editor, August 2010.

JKX18. Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT (3)*, volume 10822 of *Lecture Notes in Computer Science*, pages 456–486. Springer, 2018.

Kre19. Brian Krebs. Facebook stored hundreds of millions of user passwords in plain text for years. Retrieved 9 February 2020 from https://krebsonsecurity.com/2019/03/facebook-stored-hundreds-of-millions-of-user-passwords-in-plain-text-for-years/, 2019.

KV11. Jonathan Katz and Vinod Vaikuntanathan. Round-optimal password-based authenticated key exchange. In *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2011.

MKR17. K. Moriarty, B. Kaliski, and A. Rusch. Pkcs #5: Password-based cryptography specification version 2.1. RFC 8018, RFC Editor, January 2017.

MSHH18. Nathaniel McCallum, Simo Sorce, Robbie Harwood, and Greg Hudson. Spake pre-authentication. Internet-Draft draft-ietf-kitten-krb-spake-preauth-06, IETF Secretariat, August 2018. http://www.ietf.org/internet-drafts/draft-ietf-kitten-krb-spake-preauth-06.txt.

NYHR05. C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The kerberos network authentication service (v5). RFC 4120, RFC Editor, July 2005. http://www.rfc-editor.org/rfc/rfc4120.txt.

PJ16. C. Percival and S. Josefsson. The scrypt password-based key derivation function. RFC 7914, RFC Editor, August 2016.

PL13. Shi Pu and Jyh-Charn Liu. EAGL: an elliptic curve arithmetic gpu-based library for bilinear pairing. In *Pairing*, volume 8365 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2013.

Sho97. Victor Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 256–266, 1997.

Wi-18. Wi-Fi Alliance. WPA3 specification version 1.0. Retrieved 6 April 2019 from https://www.wi-fi.org/file/wpa3-specification-v10, April 2018.

Wri16. Jordan Wright. How browsers store your passwords (and why you shouldnt let them). Retrieved 9 February 2020 from http://raidersec.blogspot.com/2013/06/how-browsers-store-your-passwords-and.html, 2016.

A Asymmetric PAKE Functionality

Fig. 12 shows the Strong Asymmetric PAKE functionality from [JKX18], in which only two parties engage: a server S and a user U. It introduces the concept of a password file, created for S upon StorePwdFile query and disclosed to the adversary upon adaptive corruption query StealPwdFile modelling a server compromise attack. Once a server's password file is obtained, the ideal-world adversary is able to mount an offline guessing attack using OfflineTestPwd queries, and an online impersonation attack using Impersonate query.

$\mathcal{F}_{\text{saPAKE}}$ encompasses the concept of sub-sessions: a single session corresponds to a single user account on the server, allowing many sub-sessions (identified by *ssid*) where the user and server reuse the same password file to establish independent random keys.

The asymmetry between user and server in this functionality is prominent: only OnlineTestPwd and NewKey queries consider a general party $\mathcal{P}$, while other queries explicitly mention either U or S. Even $\mathcal{F}_{\text{PAKE}}$'s NewSession query is split in $\mathcal{F}_{\text{saPAKE}}$ into UsrSession and SvrSession, since the user supplies a password for each session, while the server uses its password file.

Functionality $\mathcal{F}_{\text{saPAKE}}$, with security parameter κ, interacting with parties $\{U, S\}$ and an adversary $\mathcal{S}$.

Upon (StorePwdFile, sid, U, π_S) **from** S**:**
- If there is no record $\langle \text{file}, U, S, \cdot \rangle$:
  - ▷ record $\langle \text{file}, U, S, \pi_S \rangle$ and mark it uncompromised

Upon (StealPwdFile, sid, S) **from** $\mathcal{S}$**:**
- If there is a record $\langle \text{file}, U, S, \pi_i \rangle$:
  - ▷ mark it compromised
  - ▷ return "password file stolen" to $\mathcal{S}$
- else: return "no password file" to $\mathcal{S}$

Upon (OfflineTestPwd, sid, S, π') **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{file}, U, S, \pi_S \rangle$ marked compromised
- if $\pi_S = \pi'$: return "correct guess" to $\mathcal{S}$
- else: return "wrong guess" to $\mathcal{S}$

Upon (UsrSession, $sid, ssid, S, \pi_U$) **from** U**:**
- Send (UsrSession, $sid, ssid, U, S$) to $\mathcal{S}$
- If there is no record $\langle \text{session}, ssid, U, S, \cdot \rangle$:
  - ▷ record $\langle \text{session}, ssid, U, S, \pi_U \rangle$ and mark it fresh

Upon (SvrSession, $sid, ssid, U$) **from** S**:**
- Retrieve $\langle \text{session}, U, S, \pi_S \rangle$
- Send (SvrSession, $sid, ssid, S, U$) to $\mathcal{S}$
- If there is no record $\langle \text{session}, ssid, S, U, \cdot \rangle$:
  - ▷ record $\langle \text{session}, ssid, S, U, \pi_S \rangle$ and mark it fresh

Upon (OnlineTestPwd, $sid, ssid, \mathcal{P}, \pi'$) **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{session}, ssid, \mathcal{P}, \mathcal{P}', \pi_{\mathcal{P}} \rangle$ marked fresh
- if $\pi_{\mathcal{P}} = \pi'$: mark the session compromised and return "correct guess" to $\mathcal{S}$
- else: mark the session interrupted and return "wrong guess" to $\mathcal{S}$

Upon (Impersonate, $sid, ssid$) **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{session}, ssid, U, S, \pi_U \rangle$ marked fresh
- Retrieve $\langle \text{file}, U, S, \pi_S \rangle$ marked compromised
- If $\pi_U = \pi_S$: mark the session compromised and return "correct guess" to $\mathcal{S}$
- else: mark the session interrupted and return "wrong guess" to $\mathcal{S}$

Upon (NewKey, $sid, ssid, \mathcal{P}, K'$) **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{session}, ssid, \mathcal{P}, \mathcal{P}', \pi_{\mathcal{P}} \rangle$ not marked completed
- if it is marked compromised, or either $\mathcal{P}_i$ or $\mathcal{P}_j$ is corrupted: $K_{\mathcal{P}} \leftarrow K'$
- else if it is marked fresh and there is a record $\langle \text{key}, ssid, \mathcal{P}', \pi_{\mathcal{P}'}, K_{\mathcal{P}'} \rangle$ with $\pi_{\mathcal{P}} = \pi_{\mathcal{P}'}$: $K_{\mathcal{P}} \leftarrow K_{\mathcal{P}'}$
- else: pick $K_{\mathcal{P}} \xleftarrow{\text{R}} \{0,1\}^{\kappa}$
- if the session is marked fresh:
  - ▷ record $\langle \text{key}, ssid, \mathcal{P}, \pi_{\mathcal{P}}, K_{\mathcal{P}} \rangle$
- mark the session completed
- send $\langle ssid, K_{\mathcal{P}} \rangle$ to $\mathcal{P}$

Fig. 12: Strong Asymmetric PAKE functionality $\mathcal{F}_{\text{saPAKE}}$