# CHIP and CRISP: Protecting All Parties Against Compromise through Identity-Binding PAKEs

Cas Cremers[1] [0000-0003-0322-2293], Moni Naor[2], Shahar Paz[3], and Eyal Ronen(✉)[3] [0000-0002-6013-7426]

[1] CISPA Helmholtz Center for Information Security
cremers@cispa.de

[2] Faculty of Mathematics and Computer Science, Weizmann Institute of Science, Israel
moni.naor@weizmann.ac.il

[3] School of Computer Science, Tel Aviv University.
{eyal.ronen, shaharps}@cs.tau.ac.il

**Abstract.** Recent advances in password-based authenticated key exchange (PAKE) protocols can offer stronger security guarantees for globally deployed security protocols. Notably, the OPAQUE protocol [Eurocrypt2018] realizes Strong Asymmetric PAKE (saPAKE), strengthening the protection offered by aPAKE to compromised servers: after compromising an saPAKE server, the adversary still has to perform a full brute-force search to recover any passwords or impersonate users. However, (s)aPAKEs do not protect client storage, and can only be applied in the so-called *asymmetric* setting, in which some parties, such as servers, do not communicate with each other using the protocol.

Nonetheless, passwords are also widely used in *symmetric* settings, where a group of parties share a password and can all communicate (e.g., Wi-Fi with client devices, routers, and mesh nodes; or industrial IoT scenarios). In these settings, the (s)aPAKE techniques cannot be applied, and the state-of-the-art still involves handling plaintext passwords.

In this work, we propose the notions of *(strong) identity-binding PAKEs* that improve this situation: they protect against compromise of *any* party, and can also be applied in the symmetric setting. We propose counterparts to state-of-the-art security notions from the asymmetric setting in the UC model, and construct protocols that provably realize them. Our constructions bind the local storage of all parties to abstract identities, building on ideas from identity-based key exchange, but without requiring a third party.

Our first protocol, CHIP, generalizes the security of aPAKE protocols to all parties, forcing the adversary to perform a brute-force search to recover passwords or impersonate others. Our second protocol, CRISP, additionally renders any adversarial pre-computation useless, thereby offering saPAKE-like guarantees for all parties, instead of only the server.

We evaluate prototype implementations of our protocols and show that even though they offer stronger security for real-world use cases, their performance is in line with, or even better than, state-of-the-art protocols.

**Keywords:** Password authentication, PAKE, Symmetric PAKE, Compromise Resilience, Key Compromise Impersonation.

# 1 Introduction

Passwords are arguably the most widely deployed authentication method today, and are used in a vast range of applications from authentication on the internet (e.g., email and bank servers), wireless network encryption (e.g., Wi-Fi, Smart Homes, Industry 4.0), and enterprise network authentication (e.g., Kerberos [27], EAP-pwd [21]). Early password-based protocols allowed adversaries to verify password guesses offline against observed network traffic. To remedy this, Password Authenticated Key Exchange (PAKE) protocols were proposed, as first studied by Bellovin and Merritt [3]. PAKEs allow parties to negotiate a strong secret key based only on a shared and possibly low-entropy password, do not leak any information about the password to passive adversaries, and allow only an inevitable online password guess attack.

The traditional PAKE threat model does not include compromise of the local storage – notably, most PAKEs work in a way that requires the plaintext password to be available at both parties, including SPAKE-2 and WPA3's DragonFly/SAE. This implies that non-interactive parties such as servers, IoT devices, and wireless access points, need to store the password in plaintext. Compromising the database of these parties directly reveals the password. In the client-server model, this means that a server compromise allows the adversary to impersonate as the client or server towards either, or perform a MiTM attack. Moreover, because clients often re-use passwords across services, this enables credential stuffing.

To partially mitigate this threat, Bellovin and Merritt [4] proposed so-called asymmetric PAKEs (also known as aPAKEs, Augmented PAKEs, or V(erifier)-PAKEs) that make this much harder: the clients still need to provide the password in plaintext, but the verifying servers now only need to provide, and thus store, information that (a) is derived from the password using a one-way function, yet (b) allows establishing a shared key with a party that knows the password. Thus, compromising an aPAKE server does not allow the adversary to impersonate the client, and forces it to perform a brute-force attempt to extract the password.

## 1.1 Identity-binding PAKEs (iPAKE)

aPAKE protocols still have substantial limitations: they only protect the server, and perhaps more importantly, cannot be applied to settings that do not fall into the client-server model, e.g., where a password can be shared among group members that can communicate with all other members. Prime examples of such *symmetric* settings are found in wireless networking and IoT settings. For example, the globally deployed IEEE 802.11 Wi-Fi standard includes the WPA protocol, which uses network passwords to enable devices to automatically connect to routers, extenders, and mesh network nodes; crucially, all parties can automatically communicate with each other using the network password without any user input. This led the Wi-Fi alliance to base their latest WPA3 protocol [31] on a symmetric PAKE for mesh networks called Simultaneous Authentication of Equals (SAE) [20].

In such settings, asymmetric PAKEs cannot be applied, because protecting two parties using known aPAKE-server methods stops them from being able to communicate with each other: by construction, aPAKE's servers can only authenticate themselves to clients, not to other servers. Furthermore, because parties in common symmetric group settings operate without user input, they need to store the password in plaintext. E.g., Wi-Fi passwords are stored in plaintext on users' devices.

Hence, despite the many advances made over the years, all state-of-the-art PAKEs in the symmetric setting offer substantially weaker protection and no containment: compromising any party allows impersonation of any other party in the group, thus compromising the entire group.

In this work we address this gap by initiating the study and construction of so-called *identity-binding* PAKEs (iPAKE). We provide a UC-security definition that is the symmetric counterpart to aPAKE. We instantiate iPAKE with CHIP, a novel compiler from any PAKE to iPAKE. We leverage ideas from Identity-Based Key-Exchange to introduce abstract identities for each party, and effectively bind the locally stored password-derived data to these identities, while retaining the required key agreement functionality. Unlike Identity-Based Key-Exchange, we do not require a third party: instead, each party locally simulates the Key Distribution Center during the password file generation. Identities can be arbitrary bit strings, and could also encode functions or roles instead of the party's name, e.g., "server", "router", or "fire brigade chief", "Elon's third iPhone". Binding the locally stored password-derived data to identities is useful for many purposes, such as preventing reflection attacks, revocation of compromised or disposed of devices, network segmentation (i.e., which nodes may interact), permissions (e.g., prevent guest devices from configuring an access point), and authentic audit logs that allow anomaly detection and reliable retroactive damage assessment.

### 1.2 Strong Identity-binding PAKEs (siPAKE)

In 2018, Jarecki, Krawczyk, and Xu [23] strengthened the aPAKE notion by additionally requiring that an adversary gains no benefits from any pre-computations performed before a server compromise, thereby forcing it to do a full brute-force attack after the compromise. They named this notion *strong* asymmetric PAKE (saPAKE), and proposed the OPAQUE protocol to meet it. This has been widely regarded as a major step forward, and has led the Internet Engineering Task Force (IETF) to work towards standardizing OPAQUE and its use for TLS 1.3's password-based logins [7].

To provide similar protection against pre-computations, we strengthen iPAKE to *strong identity-binding* PAKEs (siPAKE), and provide a UC-security definition that is the symmetric counterpart to saPAKE. We instantiate siPAKE with CRISP, a novel compiler from any PAKE to siPAKE, that extends the protection provided by state-of-the-art saPAKE protocols [9, 23] to all parties.

We prove the correctness of both of our constructions, provide open source prototype implementations, and evaluate their efficiency.

### 1.3 Contributions

1. We initiate the study of *identity-binding PAKEs*, which offer additional security guarantees compared to their corresponding state-of-the-art aPAKE relatives. In particular:
   - Identity-binding PAKEs offer containment against compromise of any party, instead of only a specific subset such as servers.
   - Unlike aPAKEs, iPAKEs are symmetric and allow all parties to communicate with each other, and can therefore also be applied to settings such as IEEE 802.11's WPA (Wi-Fi).
2. We define the ideal functionality $\mathcal{F}_{\text{iPAKE}}$ for **identity-binding PAKE (iPAKE)** in the UC model, and construct the **CHIP** compiler that turns any symmetric PAKE into an iPAKE. CHIP offers aPAKE-like guarantees for all parties: the compromise of any party does not allow the adversary to impersonate another unless they perform a brute-force attack. We prove that CHIP is secure in the Programmable Random Oracle Model (ROM) under the Strong Diffie-Hellman assumption.
3. We define the ideal functionality $\mathcal{F}_{\text{siPAKE}}$ for **strong identity-binding PAKE (siPAKE)** in the UC model, and construct the **CRISP** compiler that turns any symmetric PAKE into an siPAKE. CRISP offers saPAKE/OPAQUE-like guarantees for *all* parties: to impersonate any other party after a compromise, the adversary's brute-force attack additionally cannot utilize any pre-computation in a useful manner. CRISP is based on a bilinear group with pairing and "Hash-to-Group", and we prove it secure in the Generic Group with Random Oracle Model (GGM+ROM).
4. We implemented prototypes of both our protocols. While our protocols offer substantial security benefits over existing state-of-the-art PAKEs for the symmetric setting, a performance benchmark (Section 8.4) that shows their performance is in line with, or even better than, state-of-the-art protocols.

Table 1 summarizes the different security notions and example protocols.

**Prototype implementations** We provide open source implementations of both protocols at `https://github.com/shapaz/CRISP`.

### 1.4 Structure of the Paper

We give background on the formalization of PAKEs in Section 2. We discuss various methods for compromise resilience in Section 3. In Section 4 we describe the notation and UC building blocks we use. We present our new ideal functionalities for iPAKE and siPAKE in Section 5. We introduce the CHIP compiler in Section 6 and the CRISP compiler in Section 7. In Section 8 we analyze the computational cost of running our protocols and the cost of the inevitable brute-force attack. We also propose several optimization to the protocol as well as performance benchmarks. We conclude and present open problems in Section 9.

We provide full proofs and more in the full version of our paper [15].

| Security notion | Example protocol | Post-compromise impersonation resistance | Secure against pre-computation |
|---|---|---|---|
| PAKE [13] | CPace [19] | ○ | ○ |
| aPAKE [17] | AuCPace [19] | ◐ | ○ |
| **iPAKE** (Section 4) | **CHIP** (Section 6) | ● | ○ |
| saPAKE [23] | OPAQUE [23] | ◐ | ◐ |
| **siPAKE** (Section 4) | **CRISP** (Section 7) | ● | ● |

Table 1: PAKE notions, example protocols, and security guarantees. ○ denotes the property is not provided; ◐ denotes that the property only holds for servers, and can *only* be applied to the asymmetric setting; and ● denotes that it is provided for all parties.

## 2 Related Work on Formalizing PAKE

Bellare, Pointcheval, and Rogaway [2] were the first to formalize the notion of PAKE. Canetti, Halevi, Katz, Lindell, and MacKenzie [13] formalized PAKE in the Universal Composability (UC) framework [11]. Their ideal functionality $\mathcal{F}_{\text{PAKE}}$ (originally denoted $\mathcal{F}_{\text{pwKE}}$) trades 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 (a.k.a. Augmented PAKEs or Verifier PAKEs) were formalized by Boyko, MacKenzie, and Patel [8]. They 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. To prevent servers from impersonating clients, it should be "hard" to directly extract the password from such a file. However, since we assume that the password domain is small, an attacker can run an *offline dictionary attack*, testing every possible password against the file until one is accepted. The best one can hope for is that password extraction time will be linear in the dictionary's size. Gentry, MacKenzie, and Ramzan [17] formalized an ideal functionality $\mathcal{F}_{\text{aPAKE}}$ in the UC framework, and presented a generic compiler from $\mathcal{F}_{\text{PAKE}}$ to $\mathcal{F}_{\text{aPAKE}}$.

The notion of Strong Asymmetric PAKE $\mathcal{F}_{\text{saPAKE}}$ by Jarecki, Krawczyk, and Xu [23] addresses an issue with the original $\mathcal{F}_{\text{aPAKE}}$, 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. When a server is compromised at a later point, the adversary can find the pre-image for the compromised hash value, retrieving the password immediately.

In summary, while (s)aPAKE protect against server compromise in the asymmetric setting, prior works did not address party compromise in the symmetric setting or client compromise (in the asymmetric setting).

## 3 Methods and limitations for compromise resilience

In compromise resilience of PAKE protocols, we consider two main parameters:

1. The computational cost of a brute-force attack to recover the original password, using the information stored on the device in the offline phase (i.e., in the password file).
2. The possibility of performing a trade-off between the pre-computation cost (performed before the compromise of the device) and the computation cost (performed after the compromise).

We assume the adversary holds a password dictionary that contains the right password, and a brute-force attack's computational cost is proportional to the size of that dictionary. Being a "machine-in-the-middle", our adversary may alter messages and exploit information sent in the online phase of the protocol, and might target multiple passwords used by different users.

We note that in practice, passwords are used across many types of devices. Some of these devices are directly controlled by (human) users, such as phones or laptops, which either don't store the password (e.g., user remembers) or store it protected by another interactive security mechanism (e.g., biometrics, password, PIN), thereby making the compromise of the password file harder. However, a large proportion of devices that share the same password have no such user interaction, such as internet routers, TVs, IoT devices, and drones; and compromising them thus can lead to revealing the unprotected password file.

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 [31], and the client-side for aPAKEs.
2. **Hashed password:** A one-way function of the password is stored in the password file. This option is only beneficial when using a high entropy password chosen from a password space that is too large to pre-compute. Otherwise, an adversary might hash every possible password and prepare a reverse lookup table from hash value to plain password, allowing password recovery in $O(1)$ time. This can be done once, amortizing the cost of the pre-computation over multiple password recoveries.
3. **Hashed password with 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 names. In this case, pre-computation is still possible, but amortization is prevented, since the pre-computation does not apply for different public identifiers. This protection is offered by some aPAKE protocols [19] and by our novel iPAKE protocol.

4. **Hashed password with 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 in the clear, as part of the PAKE protocol. As in the previous case, pre-computation before a compromise is possible, but only after the adversary eavesdrops to a PAKE protocol of the target device and learns the "salt". This is the case for the server side in some aPAKE protocols [19, 32].
5. **Hashed password with *secret* "salt":** In this case, the random "salt" is kept secret, which requires more intricate mechanisms than with the public salt, since it is no longer possible to send the salt in the clear. This approach prevents any pre-computation, and yields a level of protection that is offered by saPAKE for the servers in the asymmetric setting, and by our novel siPAKE protocol for all parties in any setting. The only remaining attack left for the adversary is a brute-force post-compromise attack, which is inevitable, as we show below.

**Inevitable Generic Post-compromise Brute-force Attack**

Post-compromise brute-force dictionary attacks are inevitable for any PAKE protocol. In the following attack, we assume that the correct password is in the dictionary and exploit the property that PAKE protocols fail to agree on a key when the participants have different passwords. The attack works by simulating a normal protocol run, where one party uses the compromised data, and the peer uses the password guess:

1. Retrieve a password file FILE from a compromised device.
2. For every password guess $\pi'$ in the dictionary:
   (a) Derive password file FILE$'$ according to the protocol specification's setup phase for the peer, using $\pi'$.
   (b) Use FILE and FILE$'$ to simulate both parties in a normal run of the PAKE protocol.
   (c) If the simulated parties negotiate the same key, $\pi'$ is the correct password for the compromised device.

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 for both parties. This generic attack provides an upper bound to the cost of the brute-force attack on any PAKE protocol. To increase the cost of the generic attack, we must also increase the computational cost of either password file derivation or running the online phase of the protocol. Note that the password file derivation can be done in pre-computation.

## 4 Notation and UC Building Blocks

In this section, we first introduce some notational convention and recall the symmetric PAKE functionality. We then introduce modelling of the random oracle model and the generic group model.

**Notation and conventions** Our notational conventions inherit from the PAKE and UC settings:

| | |
|---|---|
| $\pi$ | a password |
| $\mathsf{id}$ | some party's abstract identifier |
| $\mathcal{P}$ | a party interacting in either real or ideal world |
| $\kappa$ | a security parameter |
| $q$ | a large prime number $q \geq 2^\kappa$ |
| $\mathbb{Z}_q$ | the field of integers modulo $q$, $\mathbb{Z}_q^\star = \mathbb{Z}_q \backslash \{0\}$ |
| $x$ | an element of $\mathbb{Z}_q$ |
| $F$ | a polynomial in $\mathbb{Z}_q[X]$ |
| $\mathtt{X}$ | a formal variable in a polynomial (indeterminate) |
| $\mathbb{G}$ | a cyclic group of order $q$ |
| $[x]_\mathbb{G}$ | a member of group $\mathbb{G}$, identified by the exponent $x$ of some public generator $g{\in}\mathbb{G}$: $[x]_\mathbb{G} = g^x$ |
| $\{0,1\}^n$ | the set of binary strings of length $n$ |
| $\{0,1\}^\star$ | the set of binary strings of any length |
| $x \overset{\text{R}}{\leftarrow} S$ | sampling $x$ from uniform distribution over set $S$ |
| $x_{\in S}$ | restriction: $x$ must be an element of $S$ |
| $H$ | a hash function |
| $\hat{H}$ | a hash-to-group function |

Similar to existing asymmetric PAKE constructions analyzed in the UC framework, we use two levels of sessions:

| | |
|---|---|
| *sid* | identifies a static session, e.g., a group of parties communicating using the same shared password. (E.g., when instantiated in the Wi-Fi setting, this could be the Wi-Fi network identifier) |
| *ssid* | identifies a particular online exchange, i.e., a sub-session. |

**Symmetric PAKE Functionality** In Figure 1 we restate the symmetric PAKE functionality $\mathcal{F}_{\text{PAKE}}$ from Canetti et al. [13] (denoted $\mathcal{F}_{\text{pwKE}}$ there), incorporating the fix recommended by Abdalla et al. [1]. In our presentation of $\mathcal{F}_{\text{PAKE}}$, we explicitly record keys handed to parties in FRESH sessions using $\langle\text{KEY}, \dots\rangle$ records, which we will later use in our protocol proofs.

Whenever an ideal functionality is required to retrieve some record ("Retrieve $\langle\text{RECORD}, \dots\rangle$ ") but it cannot be found, the functionality is said to implicitly ignore the query.

### 4.1 UC Modelling of Random Oracle and Generic Group

The necessity of non-black-box assumptions for proving compromise resilience in the UC framework has been previously observed (see [17], [23] and [9]). Hesse [22] proved that UC-realization of aPAKE is impossible under non-programmable ROM. In this work we rely on programmable ROM for proving CHIP and on Generic Group Model for CRISP.

We model ROM in UC by allowing parties in the real world to access an ideal functionality $\mathcal{F}_{\text{RO}}$, depicted in Figure 2. Invocations of hash functions in the protocol are modelled as queries to $\mathcal{F}_{\text{RO}}$. The functionality acts as an oracle, answering fresh queries with independent random values, but consistent results

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

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

Upon $(\textsc{TestPwd}, sid, \mathcal{P}_i, \pi')$ from $\mathcal{S}$:
  ○ Retrieve $\langle \textsc{session}, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ 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 $\left(\textsc{NewKey}, sid, \mathcal{P}_i, K'_{\in\{0,1\}^\kappa}\right)$ from $\mathcal{S}$:
  ○ Retrieve $\langle \textsc{session}, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ not marked COMPLETED
  ○ If it is marked COMPROMISED: $K_i \leftarrow K'$
  ○ else if it is marked FRESH and there is a record $\langle \textsc{key}, \mathcal{P}_j, \pi_j, K_j \rangle$ with $\pi_i = \pi_j$: $K_i \leftarrow K_j$
  ○ otherwise: pick $K_i \overset{\text{R}}{\leftarrow} \{0,1\}^\kappa$
  ○ If the session is marked FRESH: record $\langle \textsc{key}, \mathcal{P}_i, \pi_i, K_i \rangle$
  ○ Mark the session COMPLETED and send $\langle sid, K_i \rangle$ to $\mathcal{P}_i$
```

Fig. 1: Symmetric PAKE functionality $\mathcal{F}_{\text{PAKE}}$ from [13] with the fix recommended by [1] and minor presentational modifications to simplify comparison.

```
Functionality $\mathcal{F}_{\text{RO}}$, parametrized by domain $D$ and range $E$, interacting with parties $\{\mathcal{P}_i\}_{i=1}^n$ and adversary $\mathcal{S}$.
Upon $(\textsc{Hash}, sid, s_{\in D})$ from $\mathcal{P} \in \{\mathcal{P}_i\}_{i=1}^n \cup \{\mathcal{S}\}$:
  ○ If there is no record $\langle \textsc{hash}, s, h \rangle$:
      ▷ Pick $h \overset{\text{R}}{\leftarrow} E$ and record $\langle \textsc{hash}, s, h \rangle$
  ○ Return $h$ to $\mathcal{P}$.
```

Fig. 2: Random Oracle functionality $\mathcal{F}_{\text{RO}}$

to repeated queries. The model is *programmable*, meaning that the simulator is able to view hash queries and program their results. The model is also *local*, meaning that every session has a separate independent $\mathcal{F}_{\text{RO}}$ machine. However, every HASH query is parametrized by a unique *sid*, effectively separating the hash domain. Consequently, a single global random oracle in the real world suffices to handle queries from multiple sessions.

The Generic Group Model (GGM), introduced by [30], 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.

Functionality $\mathcal{F}_{\mathrm{GG}}$, 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 $\mathcal{S}$.
Initially, $S{=}\{1\}$, $[1]_{\mathbb{G}}{=}g$ and $[x]_{\mathbb{G}}$ is undefined for any other $x{\in}\mathbb{Z}_q$. Whenever $\mathcal{F}_{\mathrm{GG}}$ references an undefined $[x]_{\mathbb{G}}$, set $[x]_{\mathbb{G}} \overset{\mathrm{R}}{\leftarrow} \mathbb{E}\backslash S$ and insert $[x]_{\mathbb{G}}$ to $S$.

**Upon** $\left(\textsc{MulDiv}, sid, [x_1]_{\mathbb{G}}, [x_2]_{\mathbb{G}}, s_{\in\{0,1\}}\right)$ **from** $\mathcal{P} \in \{\mathcal{P}_i\}_{i=1}^{n} \cup \{\mathcal{S}\}$**:**

- ○ $x \leftarrow x_1 + (-1)^s x_2 \mod q$
- ○ Return $[x]_{\mathbb{G}}$ to $\mathcal{P}$

Fig. 3: Generic Group functionality $\mathcal{F}_{\mathrm{GG}}$

Any cyclic group $\mathbb{G}$ of prime-order $q$ with generator $g$ can be viewed as $\{[x]_{\mathbb{G}} \,|\, x{\in}\mathbb{Z}_q\}$ 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 $[\cdot]_{\mathbb{G}}$: $x \mapsto g^x$. 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 $\mathcal{F}_{\mathrm{GG}}$. Figure 3 shows the basic GGM functionality $\mathcal{F}_{\mathrm{GG}}$, which answers group operation queries (multiply/divide) on encoded elements. As with $\mathcal{F}_{\mathrm{RO}}$, functionality $\mathcal{F}_{\mathrm{GG}}$ is both programmable and local. Unlike ROM, where local independent oracles can be created from a single global one, the same is not trivial with generic groups. The full version [15] deals with group reuse across instances of CRISP.

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 $[x]_{\mathbb{G}}{=}\xi{\in}\mathbb{Z}_q$, 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, $\mathcal{F}_{\mathrm{GG}}$ 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 $\psi{:}\mathbb{G}_2{\to}\mathbb{G}_1$ satisfying $\psi(g_2){=}g_1$.

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 $\mathbb{G}_2$.

In order to represent groups with pairing and hash into group, we suggest a modified functionality $\mathcal{F}_{\mathrm{GGP}}$, depicted in Figure 4, similar to the extension of GGM to bilinear groups by [6]. $\mathcal{F}_{\mathrm{GGP}}$ can be queried $\textsc{MulDiv}$ for each of $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$, and maintains separate encoding maps for each group. It introduces three

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 $\mathcal{S}$. Let $\mathfrak{P} = \{\mathcal{P}_i\}_{i=1}^n \cup \{\mathcal{S}\}$.
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 $[x]_{\mathbb{G}_j} \overset{\text{R}}{\leftarrow} \mathbb{E}\backslash S_j$ and insert $[x]_{\mathbb{G}_j}$ to $S_j$.

**Upon** $\left(\text{MulDiv}, \mathit{sid}, j_{\in\{1,2,T\}}, [x_1]_{\mathbb{G}_j}, [x_2]_{\mathbb{G}_j}, s_{\in\{0,1\}}\right)$ **from** $\mathcal{P} \in \mathfrak{P}$**:**

- ○ Return $[x \leftarrow x_1 + (-1)^s x_2 \mod q]_{\mathbb{G}_j}$ to $\mathcal{P}$

**Upon** $(\text{Pairing}, \mathit{sid}, [x_1]_{\mathbb{G}_1}, [x_2]_{\mathbb{G}_2})$ **from** $\mathcal{P} \in \mathfrak{P}$**:**

- ○ Return $[x_T \leftarrow x_1 \cdot x_2 \mod q]_{\mathbb{G}_T}$ to $\mathcal{P}$

**Upon** $\left(\text{Isomorphism}, \mathit{sid}, j_{\in\{1,2\}}, [x]_{\mathbb{G}_j}\right)$ **from** $\mathcal{S}$**:**

- ○ Return $[x]_{\mathbb{G}_{3-j}}$ to $\mathcal{P}$

**Upon** $(\text{Hash}, \mathit{sid}, s)$ **from** $\mathcal{P} \in \mathfrak{P}$**:**

- ○ If there is no record $\langle \text{HASH}, s, [x]_{\mathbb{G}_2} \rangle$:
  - ▷ pick $x \overset{\text{R}}{\leftarrow} \mathbb{Z}_q^\star$ and record $\langle \text{HASH}, s, [x]_{\mathbb{G}_2} \rangle$
- ○ Return $[x]_{\mathbb{G}_2}$ to $\mathcal{P}$

Fig. 4: Generic Group with Pairing and Hash-to-Group functionality $\mathcal{F}_{\text{GGP}}$

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 $\psi, \psi^{-1}$ between $\mathbb{G}_2$ and $\mathbb{G}_1$: $[x]_{\mathbb{G}_1}{\mapsto}[x]_{\mathbb{G}_1}$, $[x]_{\mathbb{G}_1}{\mapsto}[x]_{\mathbb{G}_2}$; and (c) Hash which is a random oracle into $\mathbb{G}_2$: for each freshly queried string $s{\in}\{0,1\}^\star$ it picks a random exponent $x{\overset{\text{R}}{\leftarrow}}\mathbb{Z}_q^\star$, then returns its encoding $[x]_{\mathbb{G}_2}$.

We note that there are groups for which only $\psi$ is efficiently computable but $\psi^{-1}$ is not, or even $\psi$ itself is inefficient. However, CRISP does not require these Isomorphism queries and they can be omitted for such groups. We state that equipping the adversary with Isomorphism queries guarantees security even when such isomorphism is found.

## 5 (Strong) Identity-binding PAKE Functionality

In Figure 5 we present the Identity-binding PAKE functionality $\mathcal{F}_{\text{iPAKE}}$ and the Strong Identity-binding 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 Asymmetric PAKE functionality $\mathcal{F}_{\text{aPAKE}}$ of [17] and Strong Asymmetric PAKE functionality $\mathcal{F}_{\text{saPAKE}}$ of [23] (found in the full version [15].

Informally speaking, our threat model includes the online adversary from traditional PAKEs. Additionally, we consider adversaries that may compromise parties in order to impersonate as other parties, e.g., compromise an IoT device to impersonate as the router or server. The strong form additionally considers adversaries that can perform large amounts of precomputation.

Compared to the asymmetric functionalities, our main addition is the notion of abstract identities ($\mathsf{id}_i$) assigned by the environment to parties, and reported

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

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

Upon $(\textsc{StealPwdFile}, sid, \mathcal{P}_i)$ from $\mathcal{S}$:
  ∘ If there is a record $\langle \textsc{file}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$:
      ▷ $\pi \leftarrow \begin{cases} \pi_i & \text{if there is a record } \langle \textsc{offline}, \mathcal{P}_i, \pi_i \rangle \\ \perp & \text{otherwise} \end{cases}$
      ▷ mark the file COMPROMISED and return $\big(\text{“password file stolen”}, \mathsf{id}_i, \pi\big)$ to $\mathcal{S}$
  ∘ otherwise: return “no password file” to $\mathcal{S}$

Upon $(\textsc{OfflineTestPwd}, sid, \mathcal{P}_i, \pi')$ from $\mathcal{S}$:
  ∘ Retrieve $\langle \textsc{file}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$
  ∘ If it is marked COMPROMISED:
      ▷ return “correct guess” to $\mathcal{S}$ if $\pi_i = \pi'$, and “wrong guess” otherwise
  ∘ otherwise: Record $\langle \textsc{offline}, \mathcal{P}_i, \pi' \rangle$

Upon $(\textsc{OfflineComparePwd}, sid, \mathcal{P}_i, \mathcal{P}_j)$ from $\mathcal{S}$:
  ∘ Retrieve $\langle \textsc{file}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$ and $\langle \textsc{file}, \mathcal{P}_j, \mathsf{id}_j, \pi_j \rangle$ both marked COMPROMISED
  ∘ Return “passwords match” to $\mathcal{S}$ if $\pi_i = \pi_j$, and “passwords differ” otherwise

Upon $(\textsc{NewSession}, sid, ssid, \mathcal{P}_j)$ from $\mathcal{P}_i$:
  ∘ Retrieve $\langle \textsc{file}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$ and send $(\textsc{NewSession}, ssid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{id}_i)$ to $\mathcal{S}$
  ∘ If there is no record $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, \cdot \rangle$:
      ▷ record $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ and mark it FRESH

Upon $(\textsc{OnlineTestPwd}, sid, ssid, \mathcal{P}_i, \pi')$ from $\mathcal{S}$:
  ∘ Retrieve $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ marked FRESH or COMPROMISED
  ∘ If $\pi_i = \pi'$: record $\langle \textsc{imp}, ssid, \mathcal{P}_i, \star \rangle$
  ∘ 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 $(\textsc{Impersonate}, sid, ssid, \mathcal{P}_i, \mathcal{P}_k)$ from $\mathcal{S}$:
  ∘ Retrieve $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ marked FRESH or COMPROMISED
  ∘ Retrieve $\langle \textsc{file}, \mathcal{P}_k, \mathsf{id}_k, \pi_k \rangle$ marked COMPROMISED
  ∘ If $\pi_i = \pi_k$: record $\langle \textsc{imp}, ssid, \mathcal{P}_i, \mathsf{id}_k \rangle$
  ∘ 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 $\big(\textsc{NewKey}, sid, ssid, \mathcal{P}_i, \mathsf{id}', K'_{\in\{0,1\}^\kappa}\big)$ from $\mathcal{S}$:
  ∘ Retrieve $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ not marked COMPLETED and $\langle \textsc{file}, \mathcal{P}_j, \mathsf{id}_j, \pi_j \rangle$
  ∘ Ignore the query if either the session is marked FRESH and $\mathsf{id}' \neq \mathsf{id}_j$, or it is COMPROMISED and $\langle \textsc{imp}, ssid, \mathcal{P}_i, \mathsf{id} \rangle$ is not recorded for both $\mathsf{id} \in \{\mathsf{id}', \star\}$
  ∘ If the session is marked COMPROMISED: $K_i \leftarrow K'$
  ∘ else if it is marked FRESH and there is a record $\langle \textsc{key}, ssid, \mathcal{P}_j, \pi_j, K_j \rangle$ with $\pi_i = \pi_j$: $K_i \leftarrow K_j$
  ∘ otherwise: pick $K_i \overset{\text{R}}{\leftarrow} \{0,1\}^\kappa$
  ∘ If the session is marked FRESH: record $\langle \textsc{key}, ssid, \mathcal{P}_i, \pi_i, K_i \rangle$
  ∘ Mark the session COMPLETED and send $\langle ssid, \mathsf{id}', K_i \rangle$ to $\mathcal{P}_i$
```

Fig. 5: Functionality $\mathcal{F}_{\text{iPAKE}}$ is defined by the full text (including grey text), and $\mathcal{F}_{\text{siPAKE}}$ is defined by the text excluding grey text.

to participating parties as output alongside the session key. Without them, a single party compromise would allow the adversary to compromise any sub-session by impersonating any other party or perform a MiTM attack. Having the functionality inform a party of its peer identity prevents such attacks.

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 $\mathcal{F}_{\text{aPAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$ 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.

Our functionalities introduce 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 Wi-Fi network), and hence such a query is both inevitable and non-beneficial for the adversary.

Notice the four types of records used by the functionalities:

1. $\langle \textbf{FILE}, \boldsymbol{\mathcal{P}_i}, \mathsf{id}_i, \boldsymbol{\pi_i} \rangle$ records represent password files created for each party $\mathcal{P}_i$, and are derived from its password $\pi_i$ and identity $\mathsf{id}_i$. Similar type of records exist in $\mathcal{F}_{\text{PAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$ (without identities) only for the server.
2. $\langle \textbf{SESSION}, \boldsymbol{ssid}, \boldsymbol{\mathcal{P}_i}, \boldsymbol{\mathcal{P}_j}, \mathsf{id}_i, \boldsymbol{\pi_i} \rangle$ 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 $\mathcal{F}_{\text{aPAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$, without identities.
3. $\langle \textbf{KEY}, \boldsymbol{ssid}, \boldsymbol{\mathcal{P}_i}, \boldsymbol{\pi_i}, \boldsymbol{K_i} \rangle$ records represent sub-session keys $K_i$ created for party $\mathcal{P}_i$ participating in sub-session *ssid* with password $\pi_i$, and whose session was not compromised or interrupted. These records were implicitly required in prior UC PAKE works [13, 17, 23], and appear here explicitly for clarity.
4. $\langle \textbf{IMP}, \boldsymbol{ssid}, \boldsymbol{\mathcal{P}_i}, \mathsf{id}' \rangle$ records represent "permissions" for the adversary to set the peer identity observed by party $\mathcal{P}_i$ in sub-session *ssid* to $\mathsf{id}'$. They are created when the adversary invokes one of the online attack queries OnlineTestPwd or Impersonate. The functionalities reject NewKey queries with non-permitted $\mathsf{id}'$. When $\mathsf{id}'{=}\star$ this record acts as a "wild card", permitting the adversary to select any identity.

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

5. $\langle \textbf{OFFLINE}, \boldsymbol{\mathcal{P}_i}, \boldsymbol{\pi'} \rangle$ records represent an offline-guess $\pi'$ 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., $\pi'{=}\pi_i$.

Identity 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, 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.

To simplify our UC simulator, we additionally allow both OnlineTestPwd and Impersonate queries against the same session, as long as they succeed[4]. 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 no 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 the more efficient CHIP protocol (Section 6) realizing $\mathcal{F}_{\text{iPAKE}}$ in ROM using any cyclic group, while CRISP (Section 7) requires bilinear groups for realizing $\mathcal{F}_{\text{siPAKE}}$ in GGM.

**Comparison to (s)aPAKE** The symmetric functionalities $\mathcal{F}_{\text{iPAKE}}$ and $\mathcal{F}_{\text{siPAKE}}$ offer security guarantees beyond their asymmetric counterparts: given a $\mathcal{F}_{\text{iPAKE}}$ (respectively, $\mathcal{F}_{\text{siPAKE}}$) functionality, it is trivial to realize the $\mathcal{F}_{\text{aPAKE}}$ (respectively, $\mathcal{F}_{\text{saPAKE}}$) functionality. The client party $U$ will be assigned identity "client" and will simply compute its password file on each session, when receiving UsrSession query from the environment. The server party $S$ will be identified as "server" and will have to verify its peer identity is "client". Nevertheless, we are not aware of any direct extension of $\mathcal{F}_{\text{aPAKE}}/\mathcal{F}_{\text{saPAKE}}$ to $\mathcal{F}_{\text{iPAKE}}/\mathcal{F}_{\text{siPAKE}}$.

**Sessions and identifiers** The distinction between a "static" session (identified by *sid*) and an "online" sub-session (identified by *ssid*) was inherited from $\mathcal{F}_{\text{aPAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$.

A static session represents a set of parties which are expected to communicate with each other, such as devices connected to the same Wi-Fi network (*sid* can be the network name). Normally, all such parties are configured with the same password. Otherwise, only parties with matching passwords will be able to derive a shared key. Since *sid* is selected locally, it is possible to have two unrelated networks configured with the same identifier (e.g., two home networks named "Miller"). As long as their passwords differ, there will not be any real impact on security; password files created for one network are unusable for the other.

An online sub-session is a specific run of the protocol between two parties of a static session. *ssid* is given as external input to the protocol in order to uniquely identify message flows within a sub-session among parties of the same static session. In many cases the transport layer's communication identifiers (e.g.,

[4] In fact, our relaxed functionality now allows for a stronger adversary that can submit as many such queries as it chooses. However, the first failed query interrupts the session, thus preventing subsequent queries. On the other hand, after a successful attack, the adversary has already compromised the session.

TCP/IP 5-tuple) suffice. If necessary, an additional communication round can be used to negotiate unique *ssid* (as in [19]).

# 6 The CHIP iPAKE protocol

## 6.1 Design motivation

When extending the protection of traditional PAKE to consider party compromise attacks, one might think of a trivial solution: simply store the hash of the password, and use this hash value in the PAKE, instead of the plain password. While this solves the problem of leaking the password upon party compromise, it does not protect from impersonation. Since hash values are not bound to any identity, a hash value stolen from a compromised party $\mathcal{P}_i$ can be used to impersonate any non-compromised party $\mathcal{P}_j$ towards anyone. This is known as a Key Compromise Impersonation (KCI) attack.

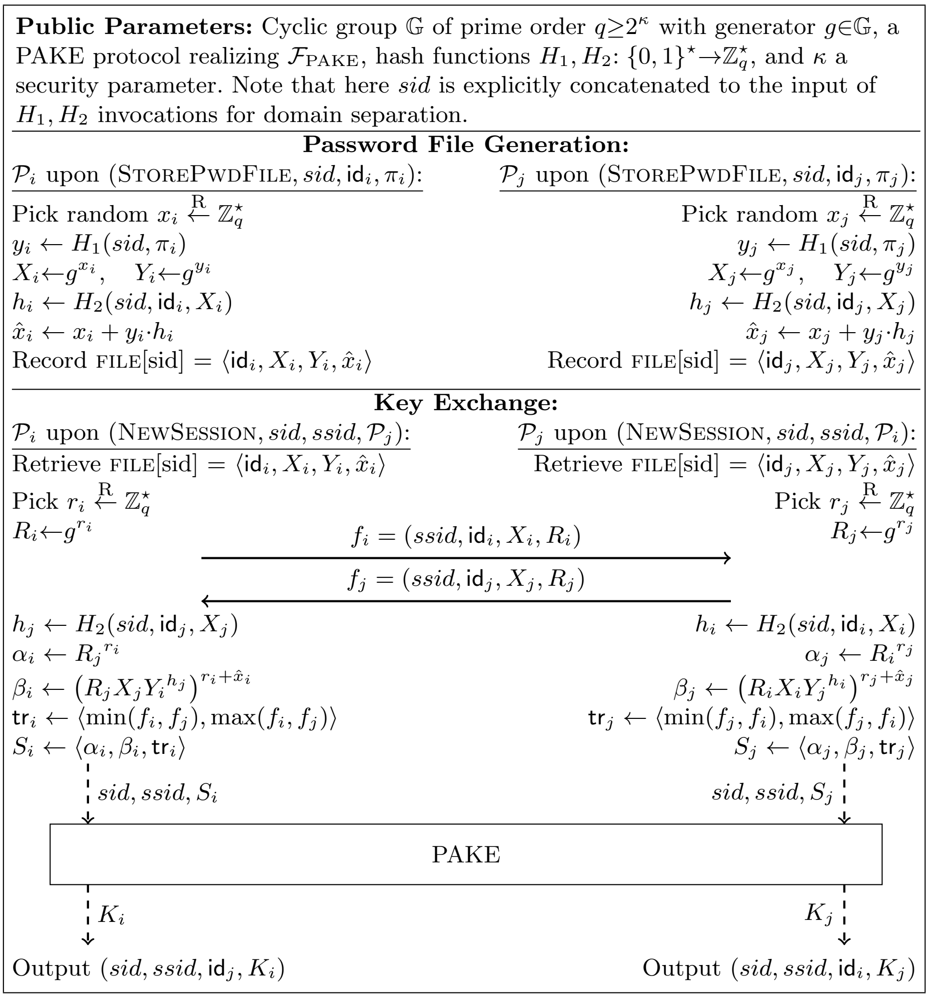
To protect against KCI attacks we need to bind those hash values to identities. However, KCI resistance is not trivial to achieve. For instance, if parties were to concatenate their identity to the password as input to a hash function: $h_i \leftarrow H(\mathsf{id}_i, \pi)$, there would be no simple means for party $\mathcal{P}_i$ knowing $h_i$ (but no longer $\pi$) to derive a shared key with another party $\mathcal{P}_j$ that only holds $h_j$.

One family of protocols that provides KCI resistance by design is Identity-Based Key-Exchange (IB-KE), introduced by Günther [18]. Unfortunately, IB-KE protocols require a trusted third party called Key Distribution Centre (KDC). The KDC is responsible for delivering identity-bound key material to other parties in a setup phase. In our setting, there is no trusted third party, only a password that is shared between the parties. To remove the KDC requirement, we modify the IB-KE protocol by allowing each party to *locally simulate the operation of the KDC*. To achieve this, we use the password hash as the KDC's secret data. This ensures that all parties with the same password are simulating "the same" KDC, i.e., using the same KDC secrets to derive password files.

Unfortunately, this construction might still be vulnerable to offline password guessing. Since an IB-KE protocol assumes the KDC secret to have high entropy, IB-KE protocols might send information that is dependent on this value. For instance, a certificate signed by the KDC secret key might be sent in the clear. With the KDC secrets being derived deterministically from a low entropy password, a passive eavesdropper might capture such a message then start an offline brute-force attack to find the correct password.

We solve this by considering IB-KE protocols with message flows independent from the KDC secrets. Specifically, we chose the Identity-Based Key-Agreement (IB-KA) protocol by Fiore and Gennaro [16]. IB-KA requires a single simultaneous communication round, is proven secure in the Canetti-Krawczyk model [12] under the strong Diffie-Hellman assumption, and provides weak Forward Secrecy (wFS) and KCI resistance.

A final issue with the construction is that the output key of IB-KA depends on the KDC secret. Recall that Forward Secrecy (ephemeral key secrecy after long-term keys are compromised) in IB-KA is not perfect but weak (i.e., only holds against passive adversaries), therefore an active adversary can modify the



**Public Parameters:** Cyclic group $\mathbb{G}$ of prime order $q \geq 2^\kappa$ with generator $g \in \mathbb{G}$, a PAKE protocol realizing $\mathcal{F}_{\text{PAKE}}$, hash functions $H_1, H_2$: $\{0,1\}^\star \to \mathbb{Z}_q^\star$, and $\kappa$ a security parameter. Note that here *sid* is explicitly concatenated to the input of $H_1, H_2$ invocations for domain separation.

**Password File Generation:**

$\mathcal{P}_i$ upon $(\text{StorePwdFile}, sid, \mathsf{id}_i, \pi_i)$:

Pick random $x_i \overset{R}{\leftarrow} \mathbb{Z}_q^\star$

$y_i \leftarrow H_1(sid, \pi_i)$

$X_i \leftarrow g^{x_i}, \quad Y_i \leftarrow g^{y_i}$

$h_i \leftarrow H_2(sid, \mathsf{id}_i, X_i)$

$\hat{x}_i \leftarrow x_i + y_i \cdot h_i$

Record $\text{FILE}[\text{sid}] = \langle \mathsf{id}_i, X_i, Y_i, \hat{x}_i \rangle$

$\mathcal{P}_j$ upon $(\text{StorePwdFile}, sid, \mathsf{id}_j, \pi_j)$:

Pick random $x_j \overset{R}{\leftarrow} \mathbb{Z}_q^\star$

$y_j \leftarrow H_1(sid, \pi_j)$

$X_j \leftarrow g^{x_j}, \quad Y_j \leftarrow g^{y_j}$

$h_j \leftarrow H_2(sid, \mathsf{id}_j, X_j)$

$\hat{x}_j \leftarrow x_j + y_j \cdot h_j$

Record $\text{FILE}[\text{sid}] = \langle \mathsf{id}_j, X_j, Y_j, \hat{x}_j \rangle$

**Key Exchange:**

$\mathcal{P}_i$ upon $(\text{NewSession}, sid, ssid, \mathcal{P}_j)$:

Retrieve $\text{FILE}[\text{sid}] = \langle \mathsf{id}_i, X_i, Y_i, \hat{x}_i \rangle$

Pick $r_i \overset{R}{\leftarrow} \mathbb{Z}_q^\star$

$R_i \leftarrow g^{r_i}$

$\mathcal{P}_j$ upon $(\text{NewSession}, sid, ssid, \mathcal{P}_i)$:

Retrieve $\text{FILE}[\text{sid}] = \langle \mathsf{id}_j, X_j, Y_j, \hat{x}_j \rangle$

Pick $r_j \overset{R}{\leftarrow} \mathbb{Z}_q^\star$

$R_j \leftarrow g^{r_j}$

$\xrightarrow{f_i = (ssid, \mathsf{id}_i, X_i, R_i)}$

$\xleftarrow{f_j = (ssid, \mathsf{id}_j, X_j, R_j)}$

$h_j \leftarrow H_2(sid, \mathsf{id}_j, X_j)$

$\alpha_i \leftarrow R_j{}^{r_i}$

$\beta_i \leftarrow \left(R_j X_j Y_j{}^{h_j}\right)^{r_i + \hat{x}_i}$

$\mathsf{tr}_i \leftarrow \langle \min(f_i, f_j), \max(f_i, f_j) \rangle$

$S_i \leftarrow \langle \alpha_i, \beta_i, \mathsf{tr}_i \rangle$

$h_i \leftarrow H_2(sid, \mathsf{id}_i, X_i)$

$\alpha_j \leftarrow R_i{}^{r_j}$

$\beta_j \leftarrow \left(R_i X_i Y_j{}^{h_i}\right)^{r_j + \hat{x}_j}$

$\mathsf{tr}_j \leftarrow \langle \min(f_j, f_i), \max(f_j, f_i) \rangle$

$S_j \leftarrow \langle \alpha_j, \beta_j, \mathsf{tr}_j \rangle$

$sid, ssid, S_i$

$sid, ssid, S_j$

PAKE

$K_i$

$K_j$

Output $(sid, ssid, \mathsf{id}_j, K_i)$

Output $(sid, ssid, \mathsf{id}_i, K_j)$

Fig. 6: CHIP protocol

incoming flow to party $\mathcal{P}_i$, then offline derive the resulting key from every possible password guess $\pi'$. Any subsequent usage of the key, e.g. for data authentication, would allow the adversary to test the password guesses and extract the correct session key. We resolve this by using the IB-KA output key as input to a symmetric PAKE, along with the transcript of the IB-KA.

Figure 6 depicts CHIP, which transforms any PAKE into an iPAKE using the modified IB-KA protocol [16], with the following changes:

- **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 replacing the KDC's randomly generated private value $y_i$ with the hash of $\mathcal{P}_i$'s password $H_1(sid, \pi_i)$.

- **PAKE Integration:** We use the output of IB-KA $(\alpha_i, \beta_i)$ alongside the IB-KA transcript $(\mathsf{tr}_i)$ as input to a PAKE instance. The output from this PAKE, $K_i$, is the resulting session key.

### 6.2 Correctness

The correctness of CHIP follows from the correctness of IB-KA. Parties $\mathcal{P}_i$, $\mathcal{P}_j$ compute the secret values $S_i$, $S_j$ respectively, where $S_i = \langle \alpha_i, \beta_i, \mathsf{tr}_i \rangle$. $S_i, S_j$ are converted to keys $K_i, K_j$ by inputting them to the PAKE. For honest parties:

$$\alpha_i = (g^{r_i})^{r_j} = (g^{r_j})^{r_i} = \alpha_j$$
$$\mathsf{tr}_i = \langle min(f_i, f_j), max(f_j, f_i) \rangle = \langle min(f_j, f_i), max(f_i, f_j) \rangle = \mathsf{tr}_j$$

Therefore, assuming $H_1(sid, \cdot)$ is injective on the password domain we get:

$$\beta_i = (R_j X_j Y_i^{h_j})^{r_i + \hat{x}_i} = g^{(r_j + x_j + y_i \cdot h_j) \cdot (r_i + x_i + y_i \cdot h_i)}$$
$$\beta_j = (R_i X_i Y_j^{h_i})^{r_j + \hat{x}_j} = g^{(r_i + x_i + y_j \cdot h_i) \cdot (r_j + x_j + y_j \cdot h_j)}$$
$$K_i{=}K_j \iff S_i{=}S_j \iff \beta_i{=}\beta_j \iff y_i{=}y_j \iff$$
$$H_1(sid, \pi_i){=}H_1(sid, \pi_j) \iff \pi_i{=}\pi_j$$

### 6.3 CHIP realizes $\mathcal{F}_{\text{iPAKE}}$

The IB-KA protocol, which CHIP is based upon, is proven secure in [16] under the strong DH assumption:

**Definition 1 (SDH).** *Let $\mathbb{G}$ be a group and $DDH(X, Y, Z)$ an oracle returning 1 if $Z = DH(X, Y)$ and 0 otherwise. The* Strong Diffie-Hellman (SDH) *assumption is said to hold in $\mathbb{G}$ if every PPT adversary $\mathcal{A}$ with oracle access $DDH$ has only negligible probability to compute the Diffie-Hellman result $DH(X, Y)$ for given inputs $X, Y \overset{R}{\leftarrow} \mathbb{G}$.*

The following theorem (proven in full version of the paper [15] states the security of CHIP as an iPAKE protocol in the UC framework.

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

### Proof Technique Intuition

To prove that CHIP UC-realizes $\mathcal{F}_{\text{iPAKE}}$ we need to show how CHIP can be simulated using $\mathcal{F}_{\text{iPAKE}}$. Here we provide some intuition for key aspects of our simulation and proof.

**Simulation of message flows.** One of the properties of IB-KA is that its flows are independent of the KDC secrets, which in our setting translates to being independent of the passwords. This has the side-effect of allowing us to easily simulate message flows.

**Simulating password files.** When a password hash is requested we employ the programmability of our ROM to set the hash value in correspondence with previously stolen password files. We use OfflineComparePwd to ensure consistency of generated hash values across parties with the same password. If a party is compromised after the hash is computed, we take advantage of OfflineTestPwd executed during Hash simulation to reveal the correct password of the party to be compromised, then simulate a password file with the known hash.

**Simulating TestPwd.** To extract a password guess from the environment's TestPwd input we consider all possible password hash values: If a previous $H_1(\pi')$ query outputs a value satisfying $\mathcal{Z}$'s input, we mount an OnlineTestPwd against $\mathcal{F}_{\text{iPAKE}}$ with $\pi'$; If a previously compromised password file contained a hash value satisfying the input, then we Impersonate that compromised party. It is possible that $\mathcal{Z}$'s guess was incorrect, in which case our attacks will also fail.

**Preserving KCI-resistance.** We state that despite simulating the KDC using a hash of a password, we preserve the KCI resistance property of IB-KE, as long as the password remains secret. That is, modelling the hash function applied to the password as a random oracle, the adversary has no access to the random value $H(\pi)$ until it queries the oracle with the correct password. Thus, the local generation of a password file under our modification is equivalent to a KDC generating key files, while $H(\pi)$ is not queried by the adversary.

### 6.4 The Cost of Brute-force Attack on CHIP

We note that in our proof, $H_1$ corresponds to OfflineTestPwd or the cost of a single password guess. Therefore, to increase the cost of a brute-force attack, it is advised to choose a computationally costly hash function (see Section 8.1).

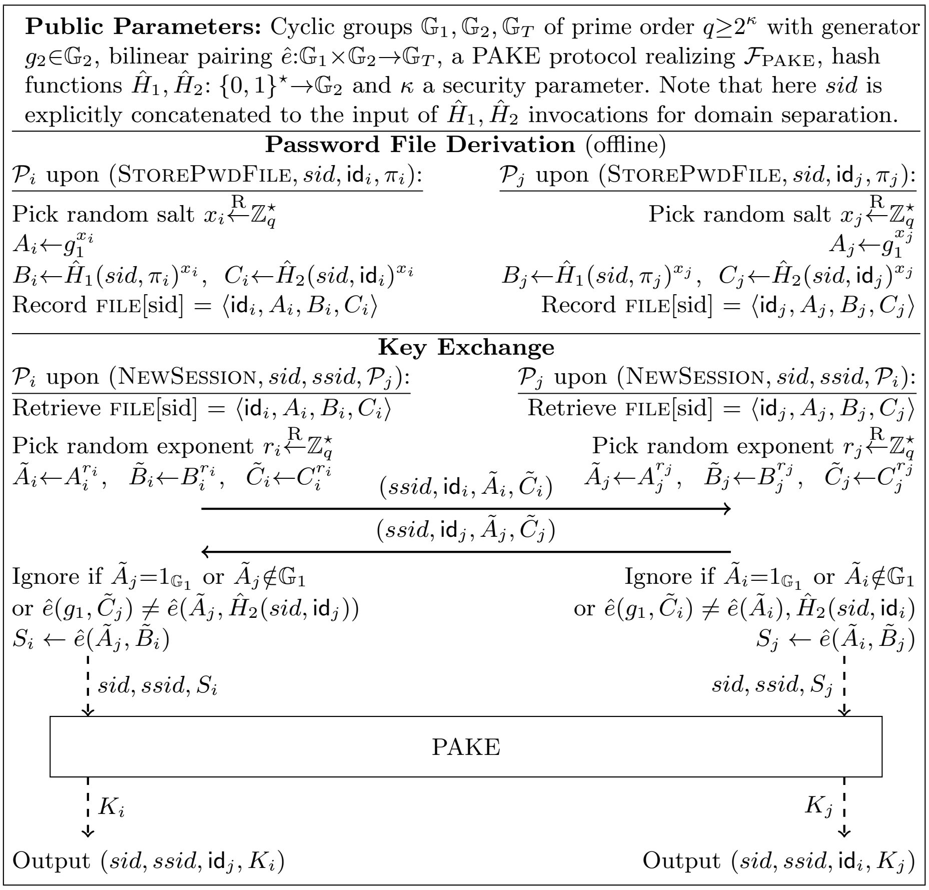
CHIP is vulnerable to pre-computation. CHIP's password files include the (unsalted) hash value $Y = g^y = g^{H_1(sid,\pi)}$. While extracting the password from a compromised file requires a brute-force attack, this property enables precomputation: if the adversary prepares a mapping $Y_{\pi'} \mapsto \pi'$ for each password guess $\pi'$ in advance for a specific $sid$, it can discover the correct password immediately after compromising a party. Our next protocol mitigates this.

## 7 The CRISP siPAKE protocol

### 7.1 Protocol Description

CRISP is a compiler that transforms any PAKE into a compromise resilient, identity-binding, and symmetric PAKE. CRISP (defined in Figure 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 $\kappa$. These parameters include the bilinear groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ with hash to group functions $\hat{H}_1$, $\hat{H}_2$, and the PAKE protocol to be used.
2. **Password File Derivation:** In this phase, the user enters a password $\pi_i$ and an identifier $\mathsf{id}_i$ for a party $\mathcal{P}_i$ (e.g., some device such as a personal computer, smartphone, server or access point). The party selects an independent and uniform random salt, and then derives and stores the password file.



**Public Parameters:** Cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order $q \geq 2^\kappa$ with generator $g_2 \in \mathbb{G}_2$, bilinear pairing $\hat{e}: \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, a PAKE protocol realizing $\mathcal{F}_{\text{PAKE}}$, hash functions $\hat{H}_1, \hat{H}_2$: $\{0,1\}^\star \to \mathbb{G}_2$ and $\kappa$ a security parameter. Note that here *sid* is explicitly concatenated to the input of $\hat{H}_1, \hat{H}_2$ invocations for domain separation.

**Password File Derivation** (offline)

$\mathcal{P}_i$ upon $(\textsc{StorePwdFile}, sid, \mathsf{id}_i, \pi_i)$:

Pick random salt $x_i \xleftarrow{R} \mathbb{Z}_q^\star$

$A_i \leftarrow g_1^{x_i}$

$B_i \leftarrow \hat{H}_1(sid, \pi_i)^{x_i}, \quad C_i \leftarrow \hat{H}_2(sid, \mathsf{id}_i)^{x_i}$

Record $\textsc{file}[\text{sid}] = \langle \mathsf{id}_i, A_i, B_i, C_i \rangle$

$\mathcal{P}_j$ upon $(\textsc{StorePwdFile}, sid, \mathsf{id}_j, \pi_j)$:

Pick random salt $x_j \xleftarrow{R} \mathbb{Z}_q^\star$

$A_j \leftarrow g_1^{x_j}$

$B_j \leftarrow \hat{H}_1(sid, \pi_j)^{x_j}, \quad C_j \leftarrow \hat{H}_2(sid, \mathsf{id}_j)^{x_j}$

Record $\textsc{file}[\text{sid}] = \langle \mathsf{id}_j, A_j, B_j, C_j \rangle$

**Key Exchange**

$\mathcal{P}_i$ upon $(\textsc{NewSession}, sid, ssid, \mathcal{P}_j)$:

Retrieve $\textsc{file}[\text{sid}] = \langle \mathsf{id}_i, A_i, B_i, C_i \rangle$

Pick random exponent $r_i \xleftarrow{R} \mathbb{Z}_q^\star$

$\tilde{A}_i \leftarrow A_i^{r_i}, \quad \tilde{B}_i \leftarrow B_i^{r_i}, \quad \tilde{C}_i \leftarrow C_i^{r_i}$

$\mathcal{P}_j$ upon $(\textsc{NewSession}, sid, ssid, \mathcal{P}_i)$:

Retrieve $\textsc{file}[\text{sid}] = \langle \mathsf{id}_j, A_j, B_j, C_j \rangle$

Pick random exponent $r_j \xleftarrow{R} \mathbb{Z}_q^\star$

$\tilde{A}_j \leftarrow A_j^{r_j}, \quad \tilde{B}_j \leftarrow B_j^{r_j}, \quad \tilde{C}_j \leftarrow C_j^{r_j}$

$\xrightarrow{(ssid, \mathsf{id}_i, \tilde{A}_i, \tilde{C}_i)}$

$\xleftarrow{(ssid, \mathsf{id}_j, \tilde{A}_j, \tilde{C}_j)}$

Ignore if $\tilde{A}_j = 1_{\mathbb{G}_1}$ or $\tilde{A}_j \notin \mathbb{G}_1$

or $\hat{e}(g_1, \tilde{C}_j) \neq \hat{e}(\tilde{A}_j, \hat{H}_2(sid, \mathsf{id}_j))$

$S_i \leftarrow \hat{e}(\tilde{A}_j, \tilde{B}_i)$

Ignore if $\tilde{A}_i = 1_{\mathbb{G}_1}$ or $\tilde{A}_i \notin \mathbb{G}_1$

or $\hat{e}(g_1, \tilde{C}_i) \neq \hat{e}(\tilde{A}_i), \hat{H}_2(sid, \mathsf{id}_i)$

$S_j \leftarrow \hat{e}(\tilde{A}_i, \tilde{B}_j)$

$sid, ssid, S_i$

$sid, ssid, S_j$

PAKE

$K_i$

$K_j$

Output $(sid, ssid, \mathsf{id}_j, K_i)$

Output $(sid, ssid, \mathsf{id}_i, K_j)$

Fig. 7: CRISP protocol

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 Section 8.3).
   (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.

The hash-to-group functions ($\hat{H}_1$ and $\hat{H}_2$) can be realized by $\mathcal{F}_{\text{GGP}}$'s $\textsc{Hash}$ queries using domain separation with different prefixes: $\hat{H}_1(sid, \pi)$ will query $\textsc{Hash}$ using $s = 1||\pi$, and $\hat{H}_2(sid, \mathsf{id})$ will use $s = 2||\mathsf{id}$.

We provide intuition by explaining the necessity of several components.

**Bilinear Pairing.** To protect against pre-computation attacks the password file cannot contain neither the plain password, nor its unsalted hash. Nevertheless, the classical salted hash method (e.g., $H(\pi, x)$ for a random salt $x$) guarantees pre-computation resistance, but cannot be used to derive a shared key across parties with independent salts, because the hashes have no structure to link them with each other, in the absence of the password during the online key exchange. Storing $\langle x, Y\rangle$ for a random $x$ and $Y{=}g^{H(\pi)\cdot x}$ is also vulnerable to pre-computation of a map $M$: $g^{H(\pi')} \mapsto \pi'$, then finding the password $\pi$ immediately with $M[Y^{1/x}]$.

In search of a construct that is both resilient to pre-computation and has some algebraic structure we considered $\langle X, Y\rangle$ for $X{=}g_1^x$, $Y{=}g_2^{H(\pi)\cdot x}$ and random $x$. This utilizes the oracle hashing scheme [10] $\langle X, X^{H(v)}\rangle$, which implies pre-computation resistance. The parties can then compute a shared value using bilinear pairing:

$$\hat{e}(X_i, Y_j) = \hat{e}(g_1^{x_i}, g_2^{H(\pi)\cdot x_j}) = \hat{e}(g_1, g_2)^{H(\pi)\cdot x_i \cdot x_j} = \hat{e}(g_1^{x_j}, g_2^{H(\pi)\cdot x_i}) = \hat{e}(X_j, Y_i)$$

**Hash-to-Group.** Although the $\langle X, Y\rangle$ construct from last paragraph satisfies pre-computation resistance, it has inherent asymmetry in the computation cost: while honest parties are required to run bilinear pairing to derive a shared key, an adversary that has stolen a password file can test passwords offline with a cost of one exponentiation per password guess. This is accomplished by pre-computing $h[\pi']{=}H(\pi')$, then after compromising a party testing whether $X^{h[\pi']} \stackrel{?}{=} \psi(Y)$ for each password guess $\pi'$. [5]

The similar approach selected for CRISP is $\langle X, Y\rangle$ for $X{=}g_1^x$, $Y{=}\hat{H}(\pi)^x$ and $x$ generated at random, using a hash-to-group function $\hat{H}$. This ensures that the exponent $e$ for $g_2^e{=}\hat{H}(\pi)$ is kept hidden, even from those who possess the password. Thus, the adversary is required to compute a bilinear pairing per password guess post compromise.

**Blinding.** The blinding stage perfectly hides the salt $x_i$ (information theoretically) in the first message transmitted from $\mathcal{P}_i$, since $\langle \tilde{A}_i, \tilde{C}_i\rangle = \langle g_1^{\tilde{x}_i}, \hat{H}_2(sid, \mathsf{id}_i)^{\tilde{x}_i}\rangle$ 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 a pre-computation attack. $\mathcal{A}$ may compute the inverse map $B_{\pi'}{\mapsto}\pi'$ for any password guess $\pi'$:

$$B_{\pi'} = \hat{e}(A_i, \hat{H}_1(sid, \pi')) = \hat{e}(g_1, \hat{H}_1(sid, \pi'))^{x_i}$$

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

$$\hat{e}(g_1, B_i) = \hat{e}(g_1, \hat{H}_1(sid, \pi_i)^{x_i}) = \hat{e}(g_1, \hat{H}_1(sid, \pi_i))^{x_i},$$

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

[5] Even without $\psi$, $\mathcal{A}$ can compute $X_T{=}\hat{e}(X, g_2)$ and $Y_T{=}\hat{e}(g_1, Y)$ with just two pairings, then test each password guess $\pi'$ using a single exponentiation: $X_T^{h[\pi']} \stackrel{?}{=} Y_T$.

**Symmetric PAKE.** The final key $K_i$ should be derived from the secret $S_i$ using a PAKE and not some deterministic key derivation function. The reason is the lack of perfect forward secrecy in the first message exchange, as explained for CHIP in Section 6.1. Concretely, consider the following attack:

Adversary $\mathcal{A}$ modifies the flow from $\mathcal{P}_j$ to $\mathcal{P}_i$ into $\tilde{A}'_j{=}g_1^{x'_j}$, $\tilde{C}'_j{=}\hat{H}_2(sid, \mathsf{id}_j)^{x'_j}$ using some arbirarily chosen exponent $x'_j$. $\mathcal{A}$ can now use $\tilde{A}_i$ (sent by an honest party $\mathcal{P}_i$) to compute the value $S[\pi'] = \hat{e}(\tilde{A}_i, \hat{H}_1(sid, \pi')^{x'_j})$ for any password guess $\pi'$. $\mathcal{A}$ can now derive a guess for the resulting key $K'$ and test this key against encrypted messages sent by $P_i$. A correct key implies the password guess was right. This can be repeated for multiple guesses without engaging in additional online exchanges.

**Generic group model.** As discussed in Section 4.1 we require a non-black-box assumption to prove pre-computation resilience, and "count" the number of operations required for an offline brute-force attack. Similarly to [9], we use GGM to bind each offline guess to a group operation. In our case, we bind it to the computationally expensive operation of pairing. This is explained in more detail in Section 7.4. CRISP is proven in *local* GGM. The full version [15] discuss how we can modify the functionality to allow the reuse of a single generic group for all CRISP instances. It also discusses the limitation on composing CRISP with other protocols sharing the same group (e.g., same bilinear curve).

### 7.2 Correctness

Honest parties $\mathcal{P}_i$, $\mathcal{P}_j$ compute the secrets $S_i$, $S_j$ respectively, which are used as inputs to $\mathcal{F}_{\mathrm{PAKE}}$ to get $K_i$, $K_j$. Assuming $\hat{H}_1(sid, \cdot)$ is injective on the password domain we get:

$$\begin{gathered}S_i = \hat{e}(\tilde{A}_j, \tilde{B}_i) = \hat{e}(g_1^{x_j r_j}, \hat{H}_1(sid, \pi_i)^{x_i r_i}) = \hat{e}(g_1, \hat{H}_1(sid, \pi_i))^{x_i r_i \cdot x_j r_j} \\ S_j = \hat{e}(\tilde{A}_i, \tilde{B}_j) = \hat{e}(g_1^{x_i r_i}, \hat{H}_1(sid, \pi_j)^{x_j r_j}) = \hat{e}(g_1, \hat{H}_1(sid, \pi_j))^{x_j r_j \cdot x_i r_i} \\ K_i{=}K_j \iff S_i{=}S_j \iff \hat{H}_1(sid, \pi_i){=}\hat{H}_1(sid, \pi_j) \iff \pi_i{=}\pi_j\end{gathered}$$

### 7.3 CRISP realizes $\mathcal{F}_{\mathrm{siPAKE}}$

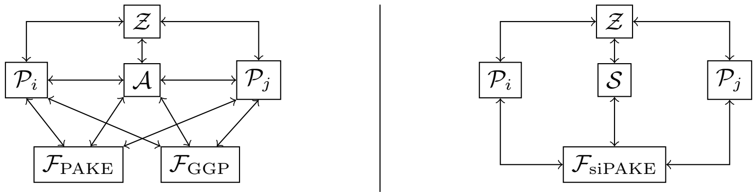
**Theorem 2.** *Protocol CRISP as depicted in Figure 7 UC-realizes $\mathcal{F}_{\mathrm{siPAKE}}$ in the ($\mathcal{F}_{\mathrm{PAKE}}$,$\mathcal{F}_{\mathrm{GGP}}$)-hybrid world.*

We give the full proof in the full version [15] and describe the high-level strategy below. In the UC proof, we omit *sid* from $\hat{H}_1$ and $\hat{H}_2$ for the sake of brevity.

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 $\mathcal{F}_{\mathrm{siPAKE}}$. We show the real and ideal worlds in Figure 8.

The main challenge for $\mathcal{S}$ is the unknown passwords assigned to parties by $\mathcal{Z}$. To overcome this, $\mathcal{S}$ simulates the real-world $\hat{H}_1(\pi_i) = [y_{\pi_i}]_{\mathbb{G}_2}$ using a formal variable (indeterminate) $\mathsf{Z}_i$ in the ideal-world: $\hat{H}_1^{\star}(\pi_i) = [\mathsf{Z}_i]_{\mathbb{G}_2}$. Wherever the real world uses group encodings of 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



(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 $\mathcal{S}$.

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.

To simulate several unknown values, we use these variables:

1. $\mathtt{X}_i$ represents party $\mathcal{P}_i$'s salt $x_i$.
2. $\mathtt{Y}_\pi$ represents the unknown exponent $y_\pi$ s.t. $\hat{H}_1(\pi){=}g_2^{y_\pi}$, for any password $\pi$.
3. $\mathtt{I}_{\mathsf{id}}$ represents the unknown exponent $\iota_{\mathsf{id}}$ s.t. $\hat{H}_2(\mathsf{id}) = g_2^{\iota_{\mathsf{id}}}$.
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 $\mathtt{Y}_{\pi_i}$, where $\pi_i$ is party $\mathcal{P}_i$'s password.

Note that some variables are created "on the fly" during the simulation. For example, upon every fresh $\hat{H}_1(\pi)$ query $\mathcal{S}$ creates a new variable $\mathtt{Y}_\pi$.

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

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

**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 $\hat{H}_1(\pi)$, 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 $\hat{e}(g_1, B_i) = \hat{e}(g_1, \hat{H}_1(\pi_i)^{x_i})$ and $\hat{e}(A_i, \hat{H}_1(\pi')) = \hat{e}(g_1^{x_i}, \hat{H}_1(\pi'))$. With overwhelming probability, these encodings will be equal if and only if $\mathcal{Z}$ chose $\pi_i{=}\pi'$, since collisions in $\hat{H}_1$ only occur with negligible probability. Yet because of using the alias $\mathtt{Z}_i$, $\mathcal{S}$ would generate $\hat{e}(g_1, B_i) = \hat{e}([1]_{\mathbb{G}_1}, [\mathtt{X}_i\mathtt{Z}_i]) = [\mathtt{X}_i\mathtt{Z}_i]_{\mathbb{G}_T}$ and $\hat{e}(A_i, \hat{H}_1(\pi')) = \hat{e}([\mathtt{X}_i]_{\mathbb{G}_1}, [Y_{\pi'}]_{\mathbb{G}_2}) = [\mathtt{X}_i\mathtt{Y}_{\pi'}]_{\mathbb{G}_T}$ 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

```
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($v$)
    sent $\leftarrow$ false
    for all compromised party $\mathcal{P}_i$ with identifier $\mathsf{id}_i$ do
        for all passwords $\pi'$ that were queried by $\hat{H}_1(\pi')$ 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 $\mathcal{F}_{\text{siPAKE}}$
                sent $\leftarrow$ true
                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 and sent=false 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

equal under substitution of $\mathtt{Z}_i$ with $\mathtt{Y}_{\pi'}$ (for some previously evaluated $\hat{H}_1(\pi')$), $\mathcal{S}$ 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 pairing evaluations (Pairing simulation), for better modelling of pre-computation resilience (see Section 7.4). 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.

**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 $\pi'$, where password guesses are taken from $\mathcal{A}$'s $\hat{H}_1(\pi')$ 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 $\pi'$ (i.e., $\pi_i{=}\pi'$). If the guess failed, $\mathcal{S}$ chooses this as the pivot entry. 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 merging some variables.

This completes the proof sketch; for further details we refer the full version [15].

### 7.4 Cost of offline brute-force attack on CRISP

In the full version of the paper [15] we provide a lower bound for the cost of offline brute-force attack. This is usually achieved by binding the offline tests OfflineTestPwd with some real-world work. For instance, [23] requires OPRF query for each tested password, while [9] shows linear relation between the number of offline tests and Generic Group operations. We bind each ideal-world OfflineTestPwd query with a bilinear pairing computed (after a compromise). In 8.2 we explain why binding to bilinear pairing is favorable compare to other group operations.

### 7.5 Primum Non Nocere - breakdown resilience of CRISP

Our CRISP compiler is based on pairing-friendly group and UC-realizes $\mathcal{F}_{\text{siPAKE}}$ assuming the Generic Group Model with pairing. However, we can show that CRISP preserves several important properties even when the *pairing-friendly* group's security is completely broken (e.g., discrete log is easy).

**Unconditional PAKE Security** First we consider the underlying symmetric PAKE's original properties. To show this, we are only concerned with the additional actions added before invoking the PAKE. Recall that the message added

| | | CHIP | CRISP |
|---|---|---|---|
| Password file derivation | | $2H + 2E$ | $2\hat{H} + 3E$ |
| Key exchange: | Blinding | $1E$ | $3E$ |
| | Identity check | $0$ | $1\hat{H} + 2P$ |
| | Key generation | $1H + 3E + \mathrm{PAKE}$ | $1P + \mathrm{PAKE}$ |

Table 3: Comparison of costly operations in CRISP and CHIP

by CRISP for party $\mathcal{P}_i$ is:

$$\mathsf{id}_i, \tilde{A}_i, \tilde{C}_i \ = \ \mathsf{id}_i, (g_1^{x_i})^{r_i}, (\hat{H}_2(sid, \mathsf{id}_i)^{x_i})^{r_i},$$

where $r_i$ and $x_i$ are random values. This message is thus completely independent of the password and does not leak any information about it. Also, we recall from Section 7.2 that the inputs to $\mathcal{F}_{\mathrm{PAKE}}$ $S_i, S_j$ are equal if and only if the passwords are equal (only assuming $\hat{H}_1$ is injective on the password domain). Thus, unless a party is compromised, the underlying PAKE properties (leaking no information of the password and allowing a single online guess) are preserved by CRISP.

**GGM-Free Password File Security** Recall that CRISP's password file for party $\mathcal{P}_i$ takes the following form: $\langle \textsc{file}, \mathsf{id}_i, A_i, B_i, C_i \rangle$ where only $B_i$ is derived from the password $\pi_i$ as $B_i{=}\hat{H}_1(\pi_i)^{x_i}$ with a random salt $x_i$. Hash-to-Group functions usually consist of a composition of a "conventional" hash function $H$ with a Map-to-Group function $F$: $\hat{H}_i(s) \leftarrow F(H_i(s))$. Therefore, the password file is derived from a "conventionally hashed" password $H_1(\pi_i)$ rather than the plain password. Thus, modelling $H_1$ as RO, to mount a brute-force attack against a compromised password file, the adversary has to evaluate $H_1$ on the each guess $\pi'$, regardless of group properties.

For example, with discrete log capabilities, the adversary can extract the salt $x_i$ from $A_i{=}g_1^{x_i}$. Assuming $F^{-1}$ is efficiently computable, they can extract:

$$F^{-1}(B_i^{1/x_i}) = F^{-1}(\hat{H}_1(\pi_i)^{x_i/x_i}) = F^{-1}(F(H_1(\pi_i))) = H_1(\pi_i)$$

However, a conventional hash computation is still required to test each password guess: $H_1(\pi') \stackrel{?}{=} H_1(\pi_i)$. Note that hash evaluation of guesses can be pre-computed. GGM is only used to prove that some work per guess (specifically, bilinear pairing) is required from the attacker post-compromise.

## 8 Computational Cost

The computational costs for CHIP and CRISP are summarized in Table 3 in terms of costly operations. In the table, we use $H$, $\hat{H}$, $E$, and $P$ to denote Hash, 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.

### 8.1 Password Hardening for Pre-Compromise

Common password hardening techniques (e.g., PBKDF2 [26], Argon2 [5], and scrypt [28]) are used in the process of deriving a key from a password to increase the cost of brute-force attacks. As mentioned in Section 3 both CHIP and CRISP protocols can use those techniques to increase the cost of the pre-compromise computation phase of the attack (pre-computation). In CHIP, we can use any of those hardening techniques to implement the hash function denoted as $H_1$. Similarly, in CRISP, we can use those techniques as the first step in implementing the Hash-to-Group function denoted as $\hat{H}_1$. 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.

### 8.2 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 mentioned above is taken from the lower bound proved in Section 7.4. This is also an upper bound for CRISP, since having compromised a password file, an adversary can check for any password guess $\pi'$ if:

$$\hat{e}(g_1, B_i) \stackrel{?}{=} \hat{e}(A_i, \hat{H}_1(sid, \pi'))$$

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 $\hat{H}_1$ 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 significantly amortized (e.g., by using a window implementation), to the best of our knowledge, only 37% speed-up can be achieved for pairing with a fixed point [14]. Moreover, pairing requires more memory than a simple point multiplication and is harder to accelerate using GPUs [29].

In OPAQUE [23], the difficulty of offline tests was increased by iterative hashing (password hardening). CRISP cannot benefit from this approach for post-compromise hardening, because the design does not allow the salt inside the hash. However, by using larger group sizes, we can increase the cost of each pairing and slow down offline tests. Although coarse-grained, this allows some trade-off between compromise resilience and computational complexity of CRISP.

### 8.3 CRISP Optimization

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

**Identity Verification** A substantial part of the added computational cost of the protocol is the identity verification that requires two pairing operations. We propose two 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

| | CPace | SAE | CHIP | OPAQUE | CRISP |
|---|---|---|---|---|---|
| CPU time (ms) | 0.2 | >1.3 | 0.6 | 0.6 | 4.1 |
| Communication rounds | 1 | 2 | 2 | 2 | 2 |
| Security notion | PAKE | *none* | iPAKE | saPAKE | siPAKE |

Table 4: Online performance comparison and proven security notions for PAKEs.

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, such as IoT devices, and one or more high-end devices, such as 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, but after 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 [25], we can implement CRISP using only three sequential messages. The same optimization applies to CHIP.

### 8.4 Performance Benchmark

We provide open source implementations for CHIP and CRISP. In both we rely on CPace [19] as the underlying symmetric PAKE. CHIP was implemented on top of Ristretto255 curve from the libsodium library (v1.0.18). CRISP uses the pairing friendly curve BLS12-381 from the MCL library (v1.22). Both curves are assumed to provide 128-bit of security strength. The source code is available at `https://github.com/shapaz/CRISP`.

In Table 4 we compare the online performance of CHIP and CRISP with those of other popular PAKE protocols, running on an i7-4790 processor. CPace and OPAQUE [23] were chosen by IETF CFRG as symmetric and asymmetric PAKEs (respectively) for usage with TLS 1.3, and are considered to be very efficient. SAE [20] is the underlying symmetric PAKE of Wi-Fi's WPA-3 and is designed to be supported by low-resource embedded devices. For measurements, our code implements both CPace and OPAQUE over Ristretto255. For SAE we used the official hostapd/wpa_supplicant. Note that although Wi-Fi's SAE was designed to be a PAKE, its security was never proven.

## 9 Conclusions and discussion

In this paper, we formalized the novel notions of iPAKE and siPAKE, that bring compromise resilience to all parties, and can also be applied in the symmetric setting. We presented CHIP, which we proved to UC-realize $\mathcal{F}_{\text{iPAKE}}$ under ROM. We also introduced CRISP, which we proved to realize $\mathcal{F}_{\text{siPAKE}}$ under GGM+ROM. Moreover, we have shown that each offline password guess for CRISP requires a computational cost equivalent to one pairing operation. Finally, we showed our protocols are practical and efficient.

**Deploying (s)iPAKE** Deploying (s)iPAKEs in practice could be done by, e.g., using CRISP or CHIP inside a Wi-Fi handshake, and choosing roles and device names ("Phone: Elon's third iPhone") as the identities, and requiring consistency between the reported identity and the identity in the handshake. A compromise of the phone would afterwards only allow the adversary to impersonate as this device identity, which would enable manual detection (e.g., a lost phone appearing as an access point) and facilitate allow/deny listing. Other application examples include IoT settings, where one could link role identities to capabilities, e.g., the window cannot instruct the garage door to open.

**Comparison of CRISP and CHIP** CHIP and CRISP both provide Password Authenticated Key Exchange with compromise resilience, and allow fine-grained password hardening by selecting computationally hard hash functions (Section 8.1). Parties running CHIP or CRISP only evaluate those hash functions once in the offline setup phase, which means that computationally costly variants can be chosen.

However, while CHIP realizes $\mathcal{F}_{\text{iPAKE}}$ providing "Hashed password with public identifiers" level of compromise resilience (Section 3), CRISP realizes $\mathcal{F}_{\text{siPAKE}}$, providing the more secure "Hashed password with secret salt" level. Thus, CRISP requires the adversary to pay an additional coarse-grained cost after party compromise (Section 8.2). CRISP's pre-computation resistance comes at a cost: CHIP is faster, requires standard assumptions, and can be implemented with simple group operations; CRISP, on the other hand, requires bilinear pairing and *local* GGM, and cannot be trivially composed with other protocols that share the same group.

Going forward with the concept of identity-binding PAKEs, we identify several remaining open problems:

**Two message protocol.** In Section 8.3, we showed how our protocols require only three messages. As shown in [25], PAKE 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 iPAKE or siPAKE protocol. To the best of our knowledge, there are no two message (s)aPAKE protocols. Jutla and Roy [24] propose a one-round aPAKE, but it seems that they require an additional message from the server before the protocol [23].

**Optimal bound on the cost of brute-force attack.** In Section 3 we showed a black-box post-compromise brute-force attack on any PAKE protocol. The computational cost of the attack is *two* runs (i.e., for both parties) of the PAKE protocol for each offline password guess. However, to the best of our knowledge, brute-forcing current PAKE implementations requires a computational cost equivalent to only *one* run of the protocol. It remains an open problem to find a more efficient black-box attack or to implement a more resilient PAKE.

**Fine-grained password hardening.** While both CHIP and CRISP allow for fine-grained password hardening, CRISP additionally provides coarse-grained post-compromise password hardening by enlarging the group (e.g., curves of larger size). Allowing fine-grained hardening (e.g., iterative hashing) while preserving pre-computation resistance for all parties remains an open problem.

## Acknowledgments

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

The second author is supported in part by grants from the Israel Science Foundation (no. 2686/20) and by the Simons Foundation Collaboration on the Theory of Algorithmic Fairness. Incumbent of the Judith Kleeman Professorial Chair. The fourth author is supported in part by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik ICRC, and Robert Bosch Technologies Israel Ltd. Member of the Check Point Institute for Information Security.

## Bibliography

[1] M. Abdalla, B. Haase, and J. Hesse. Security analysis of CPace. In *ASIACRYPT (4)*, 2021.

[2] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT*, 2000.

[3] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *IEEE Symposium on Security and Privacy*, 1992.

[4] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *ACM CCS*, 1993.

[5] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *EuroS&P*. IEEE, 2016.

[6] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *EUROCRYPT*, 2004.

[7] Daniel Bourdrez, Dr. Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-irtf-cfrg-opaque-08, Internet Engineering Task Force, March 2022. URL https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque-08.

[8] V. Boyko, P.D. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *EUROCRYPT*, 2000.

[9] Tatiana Bradley, Stanislaw Jarecki, and Jiayu Xu. Strong asymmetric PAKE based on trapdoor CKEM. In *CRYPTO*, 2019.

[10] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *CRYPTO*, 1997.

[11] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[12] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *EUROCRYPT*, 2001.

[13] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT*, 2005.

[14] Craig Costello and Douglas Stebila. Fixed argument pairings. In *LATINCRYPT*, 2010.

[15] Cas Cremers, Moni Naor, Shahar Paz, and Eyal Ronen. CHIP and CRISP: Protecting All Parties Against Compromise through Identity-Binding PAKEs. Cryptology ePrint Archive, 2020. https://eprint.iacr.org/2020/529.

[16] Dario Fiore and Rosario Gennaro. Identity-Based Key Exchange Protocols without Pairings. *Trans. Comput. Sci.*, 10:42–77, 2010.

[17] Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO*, 2006.

[18] Christoph G. Günther. An identity-based key-exchange protocol. In *EUROCRYPT*, 1989.

[19] B. Haase and B. Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019.

[20] D. Harkins. Simultaneous Authentication of Equals: A secure, password-based key exchange for mesh networks. In *2008 Second International Conference on Sensor Technologies and Applications*, 2008.

[21] D. Harkins and G. Zorn. Extensible Authentication Protocol (EAP) Authentication Using Only a Password. RFC 5931, August 2010.

[22] Julia Hesse. Separating symmetric and asymmetric password-authenticated key exchange. In *SCN*, 2020.

[23] S. Jarecki, H. Krawczyk, and J. Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, 2018.

[24] Charanjit S. Jutla and Arnab Roy. Smooth NIZK arguments. In *TCC*, 2018.

[25] Jonathan Katz and Vinod Vaikuntanathan. Round-optimal password-based authenticated key exchange. In *TCC*, 2011.

[26] K. Moriarty, B. Kaliski, and A. Rusch. PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018, January 2017.

[27] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120, July 2005.

[28] C. Percival and S. Josefsson. The scrypt Password-Based Key Derivation Function. RFC 7914, August 2016.

[29] Shi Pu and Jyh-Charn Liu. EAGL: An Elliptic Curve Arithmetic GPU-Based Library for Bilinear Pairing. In *Pairing*, 2013.

[30] Victor Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT*, 1997.

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

[32] Thomas D. Wu. The Secure Remote Password Protocol. In *NDSS*. The Internet Society, 1998.