CHIP and CRISP: Compromise Resilient Identity-based Symmetric PAKEs

Moni Naor[⋆], Shahar Paz[⋆⋆], and Eyal Ronen(✉)[⋆⋆⋆]

Abstract. Password Authenticated Key Exchange (PAKE) protocols allow parties to establish a shared key based only on the knowledge of a low entropy password. In this work, we propose a novel notion called “Identity-based PAKE” (iPAKE) – providing resilience against compromise of one or more parties. iPAKE protocols protect all parties in the symmetric setting, whereas in Asymmetric PAKE (aPAKE) only one party (a server) is protected w.r.t compromise. Binding each party to its identity prevents impersonation between devices with different roles and allows the revocation of compromised parties. We achieve this by using ideas from Identity-Based Key-Agreement (IB-KA) while using *only a low entropy password*, without requiring a trusted center.

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

We formalize the iPAKE and siPAKE notions in the Universally Composable (UC) framework. We propose CHIP: a compiler from PAKE to iPAKE using IB-KA and prove its UC-security in the Random Oracle Model (ROM). We then present CRISP: a construction of siPAKE from any PAKE using bilinear groups with “Hash2Curve”. We prove CRISP’s UC-security in the Generic Group Model (GGM) and show that each offline password guess requires at least one pairing operation.

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

[⋆] Department of Computer Science and Applied Mathematics, Weizmann Institute of Science. Supported in part by grant 950/16 from the Israel Science Foundation. This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No. 819702). Incumbent of the Judith Kleeman Professorial Chair. Email: `moni.naor@weizmann.ac.il`

[⋆⋆] School of Computer Science, Tel Aviv University. Email: `shaharps@tau.ac.il`

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

1 Introduction

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

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

Password Authenticated Key Exchange (PAKE) protocols were first studied by Bellovin and Merritt [BM92]. They allow parties to negotiate a strong secret key based only on the knowledge of a shared (and possibly low-entropy) password. PAKEs do not leak any information about the password to passive adversaries and allow only an inevitable online password guess attack.[1]

However, the notion of PAKE does not deal with the issue of long term password storage, leaving it open to compromise. Bellovin and Merrit [BM93] introduced Asymmetric PAKE (aPAKE) protocols, stipulating asymmetry in the setting, calling some participants "servers" and other "clients". Only servers are protected from compromise, by storing only some function (hash) of the password. This asymmetry does not fit all settings, as in practice it is quite common for passwords to be stored insecurely on devices/clients.[2].

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

1. *Compromise resilience.* Protect the password from compromise of any party.
2. *Impersonation prevention.* Authenticate the party's identity, as different parties can have different roles or permissions.
3. *Revocation.* Protect the network from compromised parties without changing the password.

Note that we also aim to prevent *pre-computation attacks*, where the adversary is able to perform most of the computation *before* a device is compromised. Their goal is that *after* compromising a device, the pre-computation may be used to extract the password rather quickly. In saPAKE protocols [JKX18,BJX19], such protection was provided only to servers, leaving clients vulnerable.

In this paper, we set out to answer the following question:

Can we provide the required security guarantees to all parties in the symmetric multi-party setting, using only a low entropy password and no trusted center?

[1] In recent years, a significant effort has been made to standardize the usage of PAKEs (e.g. Kerberos with SPAKE-2^+ [MSHH18], Wi-Fi with Dragonfly [Wi-18], and combining TLS with PAKEs [BF18]).

[2] Many users allow their browsers to save their password, exposing them to any program running under the same user [Wri16], which makes them an easy target for any malware. Moreover, there have been multiple breaches and security issues in many popular password managers [Bed19]

1.1 Our Contributions

In this work, we answer the question in the positive, by embedding aPAKE's notion of security into the symmetric setting, i.e. providing security to all parties. We propose a novel notion of "Identity Based" PAKE (iPAKE): In the setup phase, the user inputs the password, and it is derived together with the party's identity and a random "salt" using a "One Way Function". The resulting password file protects the password in case of compromise, while still allowing any two parties to create a secure channel and verify each other's identity.

Unlike traditional Identity-Based Key-Agreement solutions, *we do not require a KDC* (Key Distribution Center). The setup phase is done locally on the device, "simulating" the KDC using only the user's low-entropy password. After generating the party's password file, the password is discarded, as it is not needed for the key exchange and to protect it from compromise.

The fact that our protocols prevent impersonation and are able to verify the peer's identity allows us to implement very complex permission and revocation systems. For instance, consider a symmetric multi-party network with servers, administrators, managers, and workers. The two high-order bits of the party's identity can be set to indicate its class (i.e., server, administrator, manager, or worker). The lower-order bits will encode the party's unique identifier (e.g., MAC address). This allows a higher-level state machine to use the protocol to enforce different rules. For example, a server might block a connection with another server (as in aPAKE), only administrators might be given some permissions, etc.

Moreover, in the case of device compromise, a trusted party (e.g., the administrator) can send a revocation list containing the device's ID. After each protocol, the verified identity of the peer is checked against the revocation list. The connection is dropped in case the peer was revoked.[3]. Furthermore, audit logs can record parties' verified identity to keep a reliable track of events and the devices involved.

This work provides five main contributions:

1. We introduce the ideal functionality $\mathcal{F}_{\text{iPAKE}}$ that formalizes the notion of "Identity-based" PAKE (iPAKE) in the Universal Composability (UC) framework [Can01].
2. We also formalize the "Strong Identity-based" PAKE notion (siPAKE) in the ideal UC functionality $\mathcal{F}_{\text{siPAKE}}$. This functionality additionally protects from pre-computation attacks against *all* parties.
3. We suggest CHIP (Compromise Hindering Identity-based PAKE): an iPAKE protocol based on the Identity-Based Key-Agreement (IB-KA) protocol of Fiore and Gennaro [FG10] combined with any symmetric PAKE. We prove that CHIP UC-realizes the ideal functionality $\mathcal{F}_{\text{iPAKE}}$ in the Programmable Random Oracle Model (ROM) under the Strong Diffie-Hellman assumption. We also suggest a variant that includes key confirmation.
4. We present CRISP (Compromise Resilient Identity-based Symmetric PAKE): a compiler turning any PAKE protocol into an siPAKE protocol. It is based on a bilinear group with pairing and "Hash-to-Group". One of its key properties is that each post-compromise offline password guess by the adversary requires a computational cost equivalent to one pairing operation and that pre-computation is of no avail. We prove its UC-security in the Generic Group Model (GGM), realizing the ideal functionality of $\mathcal{F}_{\text{siPAKE}}$.

[3] Note that the revocation might be effective for only a limited time depending on the entropy of the password (see Section 3.1)

Protected Parties		none	server	all
Pre-computation	vulnerable	PAKE [CHK$^+$05]	aPAKE [GMR06]	iPAKE [Section 4]
	resilient	–	saPAKE [JKX18]	siPAKE [Section 4]

Table 1: PAKE notions under party compromise attack.

5. We provide an open-source implementation of both protocols at `https://github.com/shapaz/CRISP`. A performance benchmark (Section 7.4) shows only a small overhead over the efficient CPace [HL19] PAKE, and that our protocols are an order of magnitude faster than SAE [Har08], the symmetric PAKE of Wi-Fi's WPA-3.

Note that the notions of $\mathcal{F}_{\text{siPAKE}}$ and $\mathcal{F}_{\text{iPAKE}}$ are strictly stronger than $\mathcal{F}_{\text{saPAKE}}$ and $\mathcal{F}_{\text{aPAKE}}$, respectively. For a summary of the protection provided by the different notions see Table 1.

1.2 Structure of the Paper

Background on the formalization of PAKEs is given in section 2. We discuss various methods for compromise resilience in section 3. The ideal functionalities for the extensions of PAKE (including the UC Modelling of Generic Groups) are described in section 4. The CHIP protocol is introduced in section 5 and the CRISP protocol is described in section 6. In section 7 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. Conclusions and open problems are presented in section 8.

2 Formalizations of PAKE

Bellare et al. [BPR00] were the first to formalize the notion of PAKE. Canetti et al. [CHK$^+$05] formalized PAKE in the Universal Composability (UC) framework [Can01]. Their ideal functionality $\mathcal{F}_{\text{PAKE}}$ (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 (also called Augmented PAKE) were formalized by Boyko et al. [BMP00]. 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 extract the password from such a file. Since we assume that the password domain is small, an attacker can always run an *"offline dictionary attack"*, validating every possible password against the password file until one is accepted. Gentry et al. [GMR06] 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 formulation of Strong Asymmetric PAKE $\mathcal{F}_{\text{saPAKE}}$ [JKX18] 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, then once a server is compromised, simply find the pre-image for the compromised hash value to find the password immediately. Note that prior to this work, clients (and all parties in the symmetric case) were not protected in case of compromise.

3 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 that the adversary has a password dictionary that contains the real password, and the brute-force computational cost is proportional to the size of the dictionary. Our adversary can exploit information sent in the online phase of the protocol and might target multiple passwords used by different users.

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

1. **Plaintext password:** The password is stored as-is in the password file. No computation is required for password recovery. This is the case for the WPA3 protocol in Wi-Fi [Wi-18], and the client-side for 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 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 eavesdrop to a PAKE protocol of the target device and learns the "salt". This is the case for the server side in some aPAKE protocols.
5. **Hashed password with *secret* "salt":** In this case, the random "salt" is kept secret, and no pre-computation is possible. This level of protection is offered by saPAKE and our novel siPAKE protocol. A brute-force attack is inevitably possible post-compromise, as shown below.

3.1 Black Box 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 that the PAKE protocol fails if two different passwords are used by the two parties. The attack is straightforward:

1. Retrieve password file FILE_c from compromised device.
2. For every password π_i in the dictionary:
   (a) Derive password file FILE_i.
   (b) Use FILE_c and FILE_i to simulate both parties of the PAKE protocol.

(c) If the same key was negotiated by the simulated parties, π_i is the correct password.

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

4 Ideal Functionalities

4.1 Notation

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

The following subsections depict several ideal functionalities. Whenever a functionality is required to retrieve some record ("Retrieve $\langle$RECORD$, \ldots \rangle$ ") but it cannot be found, the functionality is said to implicitly ignore the query.

4.2 Symmetric PAKE Functionality

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

4.3 (Strong) Identity-based PAKE Functionality

In Fig. 2 we present the Identity-based PAKE functionality $\mathcal{F}_{\mathrm{iPAKE}}$ and the Strong Identity-based PAKE functionality $\mathcal{F}_{\mathrm{siPAKE}}$. Essentially, they preserve the symmetry of $\mathcal{F}_{\mathrm{PAKE}}$ while adopting the notion of password files and party compromise from the Asymmetric PAKE functionality $\mathcal{F}_{\mathrm{aPAKE}}$ of [GMR06] and Strong Asymmetric PAKE functionality $\mathcal{F}_{\mathrm{saPAKE}}$ of [JKX18] (found in Appendix C). Note that the symmetric functionalities $\mathcal{F}_{\mathrm{iPAKE}}$ and $\mathcal{F}_{\mathrm{siPAKE}}$ are strictly stronger than their asymmetric counterparts: given a $\mathcal{F}_{\mathrm{iPAKE}}$ (respectively, $\mathcal{F}_{\mathrm{siPAKE}}$) functionality, it is trivial to realize the $\mathcal{F}_{\mathrm{aPAKE}}$ (respectively, $\mathcal{F}_{\mathrm{saPAKE}}$) functionality. The user party U simply computes its password file on each session, when receiving UsrSession query from the environment. However, we are not aware of any direct extension of $\mathcal{F}_{\mathrm{aPAKE}}/\mathcal{F}_{\mathrm{saPAKE}}$ to $\mathcal{F}_{\mathrm{iPAKE}}/\mathcal{F}_{\mathrm{siPAKE}}$.

Note that $\mathcal{F}_{\mathrm{iPAKE}}/\mathcal{F}_{\mathrm{siPAKE}}$ follow $\mathcal{F}_{\mathrm{aPAKE}}/\mathcal{F}_{\mathrm{siPAKE}}$ in using two levels of sessions: a static session (identified by sid) representing a certain network of parties (e.g. a home Wi-Fi network) and a sub-session (identified by $ssid$) representing a particular online exchange.

```
Functionality $\mathcal{F}_{\text{PAKE}}$, with security parameter $\kappa$, interacting with parties $\{\mathcal{P}_i\}_{i=1}^n$ and an adversary $\mathcal{S}$.
Upon (NewSession, $sid, \mathcal{P}_j, \pi_i$) from $\mathcal{P}_i$:
  ◦ Send (NewSession, $sid, \mathcal{P}_i, \mathcal{P}_j$) to $\mathcal{S}$
  ◦ If there is no record $\langle$session$, \mathcal{P}_i, \mathcal{P}_j, \cdot, \cdot\rangle$:
      ▷ record $\langle$session$, \mathcal{P}_i, \mathcal{P}_j, \pi_i\rangle$ and mark it fresh
Upon (TestPwd, $sid, \mathcal{P}_i, \pi'$) from $\mathcal{S}$:
  ◦ Retrieve $\langle$session$, \mathcal{P}_i, \mathcal{P}_j, \pi_i\rangle$ marked fresh
  ◦ If $\pi_i = \pi'$: mark the session compromised and return "correct guess" to $\mathcal{S}$
  ◦ otherwise: mark the session interrupted and return "wrong guess" to $\mathcal{S}$
Upon (NewKey, $sid, \mathcal{P}_i, K'$) from $\mathcal{S}$:
  ◦ Retrieve $\langle$session$, \mathcal{P}_i, \mathcal{P}_j, \pi_i\rangle$ not marked completed
  ◦ If it is marked compromised, or either $\mathcal{P}_i$ or $\mathcal{P}_j$ is corrupted: $K_i \leftarrow K'$
  ◦ else if it is marked fresh and there is a record $\langle$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 $\langle$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 $\mathcal{F}_{\text{PAKE}}$

Our main addition (relative to the asymmetric functionalities) is the notion of identities (id_i) assigned by the environment to parties. Without them, a single party compromise would allow the adversary to compromise any sub-session by impersonating that party. Having the functionality inform a party of its peer identity prevents the attack. See Section 1.1 for an example of how verified identities can improve security.

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), thus such a query is both inevitable and non-beneficial for the adversary.

Notice the four types of records used by the functionalities:

1. **$\langle$FILE$, \mathcal{P}_i, \mathsf{id}_i, \pi_i\rangle$** records represent password files created for each party $\mathcal{P}_i$, and are derived from its password π_i and identity id_i. Similar type of records exist in $\mathcal{F}_{\text{PAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$ (without identities) only for the server.
2. **$\langle$SESSION$, ssid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{id}_i, \pi_i\rangle$** records represent party $\mathcal{P}_i$'s view of a sub-session with identifier $ssid$ between $\mathcal{P}_i$ and $\mathcal{P}_j$. Similar type of records exist in $\mathcal{F}_{\text{aPAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$, without identities.
3. **$\langle$KEY$, ssid, \mathcal{P}_i, \pi_i, K_i\rangle$** records represent sub-session keys K_i created for party $\mathcal{P}_i$ participating in sub-session $ssid$ with password π_i, and whose session was not compromised or interrupted.

```
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 $\langle \textsc{file}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$:
      ▷ $\pi \leftarrow \begin{cases} \pi_i & \text{if there is a record } \langle \textsc{offline}, \mathcal{P}_i, \pi' \rangle \text{ with } \pi' = \pi_i \\ \perp & \text{otherwise} \end{cases}$
      ▷ mark the file COMPROMISED and return $\langle$"password file stolen"$, \mathsf{id}_i, \pi \rangle$ to $\mathcal{S}$
  ◦ otherwise: return "no password file" to $\mathcal{S}$

Upon $(\textsc{OfflineTestPwd}, sid, \mathcal{P}_i, \pi')$ 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
  ◦ 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
  ◦ 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 $(\textsc{NewKey}, sid, ssid, \mathcal{P}_i, \mathsf{id}', K')$ from $\mathcal{S}$:
  ◦ Retrieve $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, \pi_i \rangle$ not marked COMPLETED and $\langle \textsc{file}, \mathcal{P}_j, \mathsf{id}_j, \pi_j \rangle$
  ◦ If $\mathcal{P}_i$ is honest: ignore the query if either the session is marked FRESH and $\mathsf{id}' \neq \mathsf{id}_j$, or it is COMPROMISED and
    $\langle \textsc{imp}, ssid, \mathcal{P}_i, \mathsf{id} \rangle$ is not recorded for both $\mathsf{id} \in \{\mathsf{id}', \star\}$
  ◦ If the session is marked COMPROMISED, or either $\mathcal{P}_i$ or $\mathcal{P}_j$ is corrupted: $K_i \leftarrow K'$
  ◦ 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. 2: Functionalities $\mathcal{F}_{\text{iPAKE}}$ (full text) and $\mathcal{F}_{\text{siPAKE}}$ (grey text omitted)

These records were also implicitly created in $\mathcal{F}_{\mathrm{PAKE}}$ and $\mathcal{F}_{\mathrm{saPAKE}}$, but appear here explicitly for the sake of 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 id'. When $\mathsf{id}'{=}\star$ this record acts as a "wild card", permitting the adversary to select any identity.

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

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

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.

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 $\mathcal{F}_{\mathrm{iPAKE}}$ functionality is weaker than $\mathcal{F}_{\mathrm{siPAKE}}$ 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 5) realizing $\mathcal{F}_{\mathrm{iPAKE}}$ in ROM using any cyclic group, while CRISP (Section 6) requires bilinear groups for realizing $\mathcal{F}_{\mathrm{siPAKE}}$ in GGM.

4.4 UC Modelling of Random Oracle and Generic Group

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

We model programmable ROM in UC by allowing parties in the real world to access an ideal functionality $\mathcal{F}_{\mathrm{RO}}$ (depicted in Fig. 3). Invocations of hash functions in the protocol are modelled as queries to $\mathcal{F}_{\mathrm{RO}}$. The functionality acts as an oracle, answering fresh queries with independent random values, but consistent results to repeated queries.

The Generic Group Model (GGM), introduced by [Sho97], allows proving properties of algorithms, assuming the only permitted operations on group elements are the group operation and comparison. Hence a "generic group element" has no meaningful representation. Algorithms in GGM operate on encodings of elements, and may consult a group oracle which computes the group

[4] In fact, we allow the adversary to submit as many such queries as it chooses. However, a failed query interrupts the session, thus preventing subsequent queries. On the other hand, after a successful attack the adversary has already compromised the session.

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

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

```
Functionality $\mathcal{F}_{\text{GG}}$, parametrized by group order $q$, encoding set $\mathbb{E}$ ($|\mathbb{E}| \geq q$) and generator $g \in \mathbb{E}$, interacting with parties $\{\mathcal{P}_i\}_{i=1}^n$ and adversary $\mathcal{S}$. ($\mathcal{P} \in \{\mathcal{P}_i\}_{i=1}^n \cup \{\mathcal{A}\}$)
Initially, $S=\{1\}$, $[1]_{\mathbb{G}}=g$ and $[x]_{\mathbb{G}}$ is undefined for any other $x \in \mathbb{Z}_q$. Whenever $\mathcal{F}_{\text{GG}}$ references an undefined $[x]_{\mathbb{G}}$, set $[x]_{\mathbb{G}} \overset{R}{\leftarrow} \mathbb{E} \setminus S$ and insert $[x]_{\mathbb{G}}$ to $S$.
Upon (MulDiv, $sid, [x_1]_{\mathbb{G}}, [x_2]_{\mathbb{G}}, s$) from $\mathcal{P}$:
  ∘ $x \leftarrow x_1 + (-1)^s x_2 \mod q$
  ∘ Return $[x]_{\mathbb{G}}$ to $\mathcal{P}$
```

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

operation for two valid encodings, returning the encoded result. The group oracle declines queries for encodings not returned by some previous query.

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

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

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

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

A bilinear group is a triplet of cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order q, with an efficiently computable bilinear map $\hat{e}: \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ satisfying the following requirements:

- **Bilinearity:** $\hat{e}(g_1^x, g_2^y) = \hat{e}(g_1, g_2)^{xy}$ for all $x, y \in \mathbb{Z}_q$.
- **Non-degeneracy:** $\hat{e}(g_1, g_2) \neq 1_T$.

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

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

In order to represent groups with pairing and hash into group, we suggest a modified functionality $\mathcal{F}_{\text{GGP}}$, depicted in Fig. 5, similar to the extension of GGM to bilinear groups by [BB04]. $\mathcal{F}_{\text{GGP}}$ can be queried MulDiv for each of $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$, and maintains separate encoding maps

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

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

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

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

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

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

for each group. It introduces three new queries: (a) Pairing to compute the bilinear pairing $\hat{e}$: $([x_1]_{\mathbb{G}_1}, [x_2]_{\mathbb{G}_2}) \mapsto [x_1 \cdot x_2]_{\mathbb{G}_T}$; (b) Isomorphism to compute an isomorphism ψ, ψ^{-1} between $\mathbb{G}_1$ and $\mathbb{G}_2$: $[x]_{\mathbb{G}_1} \mapsto [x]_{\mathbb{G}_2}$, $[x]_{\mathbb{G}_2} \mapsto [x]_{\mathbb{G}_1}$; and (c) Hash which is a random oracle into $\mathbb{G}_1$: for each freshly queried string $s \in \{0,1\}^\star$ it picks a random exponent $x \overset{\text{R}}{\leftarrow} \mathbb{Z}_q^\star$, then returns its encoding $[x]_{\mathbb{G}_1}$.

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

5 The CHIP iPAKE protocol

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

An important property of IB-KA protocols is Key Compromise Impersonation (KCI) resistance: Although the adversary can inevitably impersonate any compromised party, she can never impersonate the identity of other parties. We rely upon IB-KA to provide the KCI resistance mandated by iPAKE.[5]

Fig. 6 depicts CHIP, our IB-KA based iPAKE protocol. The construction is nearly identical to the IB-KA protocol from [FG10], with the following modifications:

- **KDC Simulation:** Instead of using a real KDC, each party $\mathcal{P}_i$ simulates the KDC's setup phase during its password file generation. This is achieved by replacing the KDC's randomly generated private value y_i with the hash of $\mathcal{P}_i$'s password $H_1(\pi_i)$.

[5] We remark that non-interactive protocols, such as Identity-Based Non-Interactive Key-Exchange (IB-NIKE), cannot satisfy KCI resistance. After compromising some party $\mathcal{P}$, the adversary can follow $\mathcal{P}$'s steps in the protocol to derive the constant shared keys with any other party. Using these keys, the adversary can impersonate any party towards $\mathcal{P}$.

Public Parameters: Cyclic group $\mathbb{G}$ of prime order $q \geq 2^{\kappa}$ with generator $g \in \mathbb{G}$, hash functions H_1, H_2: $\{0,1\}^{\star} \to \mathbb{Z}_q^{\star}$, and κ a security parameter. Note that *sid* is concatenated to the input of H_1, H_2 invocations, omitted for the sake of brevity.

Password File Generation:

$\mathcal{P}_i$ upon $(\textsc{StorePwdFile}, sid, \mathsf{id}_i, \pi_i)$:	$\mathcal{P}_j$ upon $(\textsc{StorePwdFile}, sid, \mathsf{id}_j, \pi_j)$:
Pick random $x_i \overset{R}{\leftarrow} \mathbb{Z}_q^{\star}$	Pick random $x_j \overset{R}{\leftarrow} \mathbb{Z}_q^{\star}$
$y_i \leftarrow H_1(\pi_i)$	$y_j \leftarrow H_1(\pi_j)$
$X_i \leftarrow g^{x_i}, \quad Y_i \leftarrow g^{y_i}$	$X_j \leftarrow g^{x_j}, \quad Y_j \leftarrow g^{y_j}$
$h_i \leftarrow H_2(\mathsf{id}_i, X_i)$	$h_j \leftarrow H_2(\mathsf{id}_j, X_j)$
$\hat{x}_i \leftarrow x_i + y_i \cdot h_i$	$\hat{x}_j \leftarrow x_j + y_j \cdot h_j$
Record $\textsc{file}[\text{sid}] = \langle \mathsf{id}_i, X_i, Y_i, \hat{x}_i \rangle$	Record $\textsc{file}[\text{sid}] = \langle \mathsf{id}_j, X_j, Y_j, \hat{x}_j \rangle$

Key Exchange:

$\mathcal{P}_i$ upon $(\textsc{NewSession}, sid, ssid, \mathcal{P}_j)$:		$\mathcal{P}_j$ upon $(\textsc{NewSession}, sid, ssid, \mathcal{P}_i)$:
Retrieve $\textsc{file}[\text{sid}] = \langle \mathsf{id}_i, X_i, Y_i, \hat{x}_i \rangle$		Retrieve $\textsc{file}[\text{sid}] = \langle \mathsf{id}_j, X_j, Y_j, \hat{x}_j \rangle$
Pick $r_i \overset{R}{\leftarrow} \mathbb{Z}_q^{\star}$		Pick $r_j \overset{R}{\leftarrow} \mathbb{Z}_q^{\star}$
$R_i \leftarrow g^{r_i}$		$R_j \leftarrow g^{r_j}$
	$\xrightarrow{f_i = (\textsc{Flow}, \mathsf{id}_i, X_i, R_i)}$	
	$\xleftarrow{f_j = (\textsc{Flow}, \mathsf{id}_j, X_j, R_j)}$	
$\alpha_i \leftarrow {R_j}^{r_i}$		$\alpha_j \leftarrow {R_i}^{r_j}$
$h_j \leftarrow H_2(\mathsf{id}_j, X_j)$		$h_i \leftarrow H_2(\mathsf{id}_i, X_i)$
$\beta_i \leftarrow \left(R_j X_j {Y_j}^{h_j}\right)^{r_i + \hat{x}_i}$		$\beta_j \leftarrow \left(R_i X_i {Y_j}^{h_i}\right)^{r_j + \hat{x}_j}$
$\mathsf{tr}_i \leftarrow \langle \min(f_i, f_j), \max(f_i, f_j) \rangle$		$\mathsf{tr}_j \leftarrow \langle \min(f_j, f_i), \max(f_j, f_i) \rangle$
	$\xrightarrow{\alpha_i, \beta_i, \mathsf{tr}_i} \boxed{\mathcal{F}_{\text{PAKE}}} \xleftarrow{\alpha_j, \beta_j, \mathsf{tr}_j}$	
	$\xleftarrow[K_i]{} \boxed{\mathcal{F}_{\text{PAKE}}} \xrightarrow[K_j]{}$	
Output $(sid, ssid, \mathsf{id}_j, K_i)$		Output $(sid, ssid, \mathsf{id}_i, K_j)$

Fig. 6: CHIP protocol

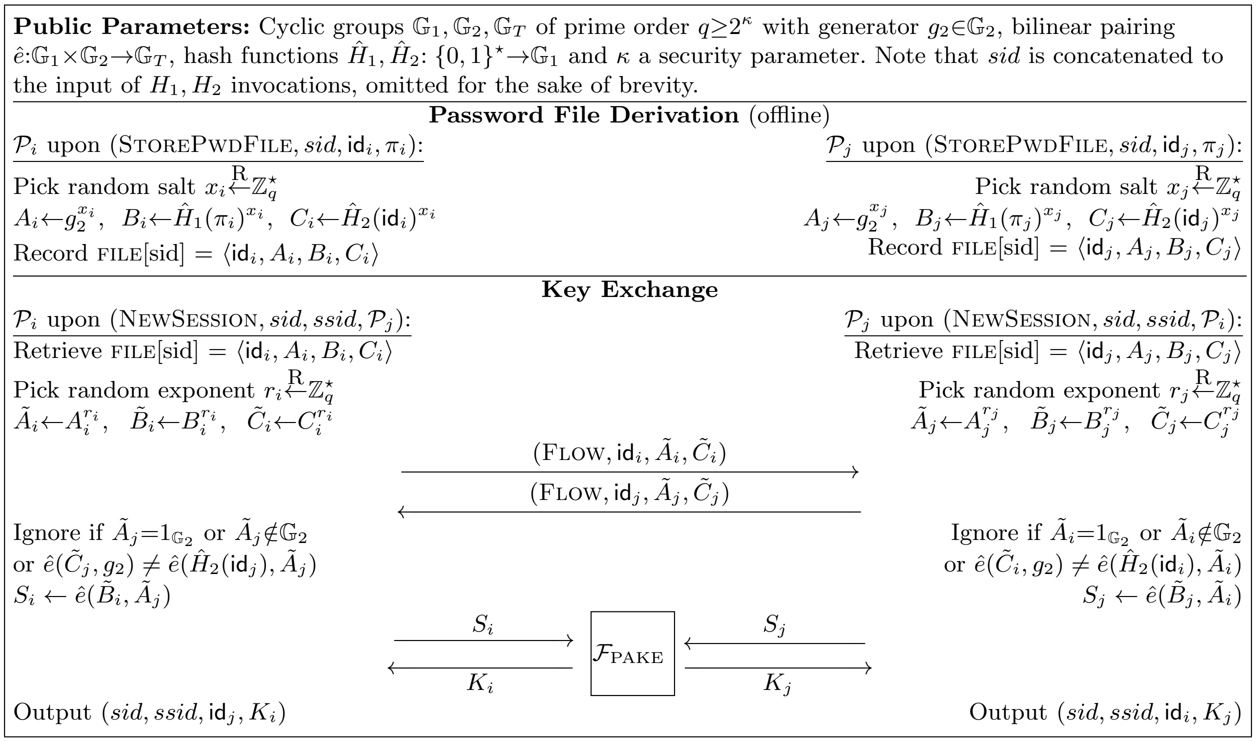
- **PAKE Integration:** We use the output of the IB-KA (α_i, β_i) alongside the IB-KA transcript (tr_i) as input to a PAKE instance. The output from this PAKE, K_i, is the resulting session key. This is necessary to prevent an offline attack against the session key.[6] We note that PAKE would not have been necessary had the IB-KA provided Perfect Forward Secrecy (PFS).

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

The proof can be found in Appendix A. Note that H_1 corresponds to $\textsc{OfflineTestPwd}$, so it is advised to choose a computationally costly hash (see Section 7.1), and include *sid* (representing common parameters, e.g. user and service names) in H_1's input. We omitted *sid* for clarity.

For another variant of this construction refer to Appendix B.

[6] An active adversary $\mathcal{A}$ can pick the values $X_j = g^{x_j}, R_j = g^{r_j}$ and send them to $\mathcal{P}_i$ as $\mathcal{P}_j$. Had the session key been $H_3(\alpha_i, \beta_i)$ as in the original IB-KA, the adversary could have computed the resulting key for each possible password guess. Exploiting any usage of the session key (e.g., for encrypting data) the adversary can find the correct guess in an offline brute-force attack. PAKE eliminates the attack by allowing the adversary at most one online guess against the IB-KA values α_i, β_i.



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 $\hat{e}: \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, hash functions $\hat{H}_1, \hat{H}_2$: $\{0,1\}^\star \to \mathbb{G}_1$ and κ a security parameter. Note that sid is concatenated to the input of H_1, H_2 invocations, omitted for the sake of brevity.

Password File Derivation (offline)

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

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

$A_i \leftarrow g_2^{x_i}$, $B_i \leftarrow \hat{H}_1(\pi_i)^{x_i}$, $C_i \leftarrow \hat{H}_2(\mathsf{id}_i)^{x_i}$

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

$A_j \leftarrow g_2^{x_j}$, $B_j \leftarrow \hat{H}_1(\pi_j)^{x_j}$, $C_j \leftarrow \hat{H}_2(\mathsf{id}_j)^{x_j}$

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

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

$\mathcal{P}_j$ 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 $r_j \overset{R}{\leftarrow} \mathbb{Z}_q^\star$

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

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

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

Ignore if $\tilde{A}_j = 1_{\mathbb{G}_2}$ or $\tilde{A}_j \notin \mathbb{G}_2$
or $\hat{e}(\tilde{C}_j, g_2) \neq \hat{e}(\hat{H}_2(\mathsf{id}_j), \tilde{A}_j)$
$S_i \leftarrow \hat{e}(\tilde{B}_i, \tilde{A}_j)$

Ignore if $\tilde{A}_i = 1_{\mathbb{G}_2}$ or $\tilde{A}_i \notin \mathbb{G}_2$
or $\hat{e}(\tilde{C}_i, g_2) \neq \hat{e}(\hat{H}_2(\mathsf{id}_i), \tilde{A}_i)$
$S_j \leftarrow \hat{e}(\tilde{B}_j, \tilde{A}_i)$

$$\begin{tikzcd} \mathcal{P}_i \arrow[r, shift left, "S_i"] & \boxed{\mathcal{F}_{\text{PAKE}}} \arrow[l, shift left, "K_i"] \arrow[r, shift right, "K_j"'] & \mathcal{P}_j \arrow[l, shift right, "S_j"'] \end{tikzcd}$$

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

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

Fig. 7: CRISP protocol

6 The CRISP siPAKE protocol

6.1 Protocol Description

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

1. **Public Parameters Generation:** In this phase, public parameters common to all parties are generated from a security parameter κ. These parameters include the bilinear groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ with hash to group functions $\hat{H}_1$[7], $\hat{H}_2$[8], and the PAKE protocol to be used.
2. **Password File Derivation:** In this phase, the user enters a password π_i and an identifier id_i for a party $\mathcal{P}_i$ (e.g., some device such as a personal computer, smartphone, server or access point). The party selects an independent and uniform random salt, and then derives and stores the password file.
3. **Key Exchange:** In this phase, two parties, $\mathcal{P}_i$ and $\mathcal{P}_j$ engage in a sub-session to derive a shared key. This phase consists of three stages:
   (a) *Blinding.* Values from the password file are raised to the power of a randomly selected exponent. This stage can be performed once and re-used across sub-sessions (see subsection 7.3).
   (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.

[7] Since $\hat{H}_1$ is invoked on the password, the note from Section 5 applies to it.

[8] 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: $\hat{H}_1(\pi)$ will query $\textsc{Hash}$ using $s = 1||\pi$, and $\hat{H}_2(\mathsf{id})$ will use $s = 2||\mathsf{id}$.

(c) *PAKE.* Both parties engage in a PAKE where they input their secret values as passwords to receive secure cryptographic keys.

6.2 Correctness

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

$$\begin{array}{c} S_i = \hat{e}(\tilde{B}_i, \tilde{A}_j) = \hat{e}(\hat{H}_1(\pi_i)^{x_i r_i}, g_2^{x_j r_j}) = \hat{e}(\hat{H}_1(\pi_i), g_2)^{x_i r_i \cdot x_j r_j} \\ S_j = \hat{e}(\tilde{B}_j, \tilde{A}_i) = \hat{e}(\hat{H}_1(\pi_j)^{x_j r_j}, g_2^{x_i r_i}) = \hat{e}(\hat{H}_1(\pi_j), g_2)^{x_j r_j \cdot x_i r_i} \\ K_i{=}K_j \iff S_i{=}S_j \iff \hat{H}_1(\pi_i){=}\hat{H}_1(\pi_j) \iff \pi_i{=}\pi_j \end{array}$$

6.3 Intuition

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

Blinding. The blinding stage perfectly hides the salt x_i (information theoretically) in the flow transmitted from $\mathcal{P}_i$, since $\langle \tilde{A}_i, \tilde{C}_i \rangle = \langle g_2^{\tilde{x}_i}, \hat{H}_2(\mathsf{id}_i)^{\tilde{x}_i} \rangle$ for $\tilde{x}_i{=}x_i r_i$ which is a random element of $\mathbb{Z}_q^\star$. Blinding is required because transmitting the raw A_i value allows $\mathcal{A}$ to mount a pre-computation attack. $\mathcal{A}$ may compute the inverse map $B_{\pi'} {\mapsto} \pi'$ for any password guess π':

$$B_{\pi'} = \hat{e}(\hat{H}_1(\pi'), A_i) = \hat{e}(\hat{H}_1(\pi'), g_2)^{x_i}$$

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

$$\hat{e}(B_i, g_2) = \hat{e}(\hat{H}_1(\pi_i)^{x_i}, g_2) = \hat{e}(\hat{H}_1(\pi_i), g_2)^{x_i}$$

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

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

Adversary $\mathcal{A}$ selects values $\tilde{A}'_j{=}g_2^{a'_j}$, $\tilde{C}'_j{=}\hat{H}_2(\mathsf{id}_j)^{a'_j}$ using some private exponent a'_j. $\mathcal{A}$ can now guess a password π' and use $\tilde{A}_i$ (sent by an honest party $\mathcal{P}_i$) to compute the value $S' = \hat{e}(\hat{H}_1(\pi')^{x_j}, \tilde{A}_i)$. Using S', $\mathcal{A}$ can derive a guess for the resulting key K' and test this key and password guess on encrypted messages sent by P_i. This can be repeated for multiple password guess without engaging in additional exchanges.

6.4 UC Security

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

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

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

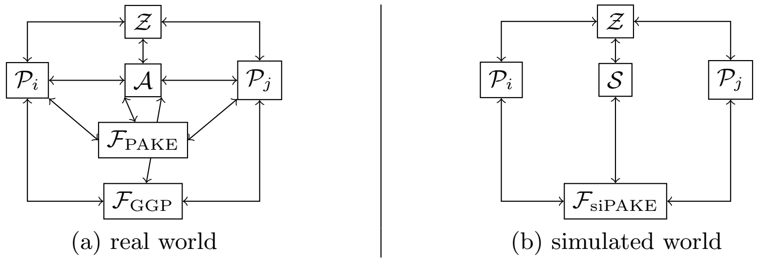


Fig. 8: Depiction of real world running protocol CRISP with adversary $\mathcal{A}$ versus simulated world running the ideal protocol for $\mathcal{F}_{\mathrm{siPAKE}}$ with adversary $\mathcal{S}$.

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

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

1. $\mathtt{X}_i$ represents party $\mathcal{P}_i$'s salt x_i.
2. $\mathtt{Y}_\pi$ represents the unknown logarithm y_π of $\hat{H}_1(\pi) = g_1^{y_\pi}$.
3. $\mathtt{I}_{\mathsf{id}}$ represents the unknown logarithm ι_{id} of $\hat{H}_2(\mathsf{id}) = g_1^{\iota_{\mathsf{id}}}$.
4. $\mathtt{R}_{i,ssid}$ represents party $\mathcal{P}_i$'s blinding value r_i in sub-session $ssid$.
5. $\mathtt{Z}_i$ is an alias for $\mathtt{Y}_{\pi_i}$ for party $\mathcal{P}_i$'s password π_i.

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

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

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

Variable Aliasing. Note that $\mathcal{S}$ uses both $\mathtt{Y}_\pi$ and $\mathtt{Z}_i$ variables: $\mathtt{Y}_\pi$ are used for simulating an evaluation of $\hat{H}_1(\pi)$, 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 $\hat{e}(B_i, g_2) = \hat{e}(\hat{H}_1(\pi_i)^{x_i}, g_2)$ and $\hat{e}(\hat{H}_1(\pi'), A_i) = \hat{e}(\hat{H}_1(\pi'), g_2^{x_i})$. These encodings will be equal if and only if $\mathcal{Z}$ chose $\pi_i = \pi'$ (or it is a collision in $\hat{H}_1$, which is found with negligible probability). Yet because of using the alias $\mathtt{Z}_i$, $\mathcal{S}$ would generate $\hat{e}(B_i, g_2) = \hat{e}([\mathtt{X}_i\mathtt{Z}_i], [1]_{\mathbb{G}_2}) = [\mathtt{X}_i\mathtt{Z}_i]_{\mathbb{G}_T}$ and $\hat{e}(\hat{H}_1(\pi'), A_i) = \hat{e}([Y_{\pi'}]_{\mathbb{G}_1}, [\mathtt{X}_i]_{\mathbb{G}_2}) = [\mathtt{X}_i\mathtt{Y}_{\pi'}]_{\mathbb{G}_T}$ which are always different encodings.

Nevertheless, $\mathcal{S}$ is able to detect possible aliasing collisions: when two distinct polynomials, whose group encodings were sent to the environment $\mathcal{Z}$, become equal under substitution of $\mathtt{Z}_i$ with $\mathtt{Y}_{\pi'}$ (for some previously evaluated $\hat{H}_1(\pi')$), $\mathcal{S}$ knows there will be a collision if $\pi_i = \pi'$. This condition can be tested by $\mathcal{S}$ using $\textsc{OfflineTestPwd}$ queries, for a compromised party $\mathcal{P}_i$. When $\mathcal{F}_{\mathrm{siPAKE}}$ replies "correct guess" to such query, $\mathcal{S}$ substitutes $\mathtt{Y}_{\pi'}$ for $\mathtt{Z}_i$ in all its data sets.

```
Simulator $\mathcal{S}$ proceeds as follows, interacting with environment $\mathcal{Z}$ and ideal functionality $\mathcal{F}_{\text{siPAKE}}$.
Initially, matrix $M$ is empty, $S_1{=}S_2{=}\{1\}$, $S_T{=}\varnothing$, $[1]_{\mathbb{G}_1}{=}g_1$, $[1]_{\mathbb{G}_2}{=}g_2$ and $[F]_{\mathbb{G}_j}$ is undefined for any other polynomial
$F$ and $j{\in}\{1,2,T\}$. Whenever $\mathcal{S}$ references an undefined $[F]_{\mathbb{G}_j}$, set $[F]_{\mathbb{G}_j} \overset{\text{R}}{\leftarrow} \mathbb{E}_j \backslash S_j$ and insert $[F]_{\mathbb{G}_j}$ to $S_j$.
Upon (StealPwdFile, $sid$) from $\mathcal{Z}$ towards $\mathcal{P}_i$:
  ∘ Send (StealPwdFile, $sid, \mathcal{P}_i$) to $\mathcal{F}_{\text{siPAKE}}$
  ∘ If $\mathcal{F}_{\text{siPAKE}}$ returned "no password file":
      ▷ Return this to $\mathcal{Z}$
  ∘ Otherwise, $\mathcal{F}_{\text{siPAKE}}$ returned $\langle$"password file stolen", $\mathsf{id}_i\rangle$
  ∘ Record $\langle$compromised$, \mathcal{P}_i, \mathsf{id}_i\rangle$
  ∘ Create variables $\mathtt{X}_i, \mathtt{Z}_i, \mathtt{I}_{\mathsf{id}_i}$ if necessary
  ∘ For each $\langle$compromised$, \mathcal{P}_j, \mathsf{id}_j\rangle$ with $\mathcal{P}_j{\neq}\mathcal{P}_i$:
      ▷ Send (OfflineComparePwd, $sid, \mathcal{P}_i, \mathcal{P}_j$) to $\mathcal{F}_{\text{siPAKE}}$
      ▷ If $\mathcal{F}_{\text{siPAKE}}$ returned "passwords match":
          ◇ Merge variables $\mathtt{Z}_i$ and $\mathtt{Z}_j$
  ∘ Return $\langle \mathsf{id}_i, [\mathtt{X}_i]_{\mathbb{G}_2}, [\mathtt{X}_i\mathtt{Z}_i]_{\mathbb{G}_1}, [\mathtt{X}_i\mathtt{I}_{\mathsf{id}_i}]_{\mathbb{G}_1}\rangle$ to $\mathcal{Z}$
Upon (NewSession, $sid, ssid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{id}_i$) from $\mathcal{F}_{\text{siPAKE}}$:
  ∘ Create variables $\mathtt{X}_i, \mathtt{Z}_i, \mathtt{I}_{\mathsf{id}_i}, \mathtt{R}_{i,ssid}$ as necessary
  ∘ $f_i \leftarrow$ (Flow, $\mathsf{id}_i, [\mathtt{X}_i\mathtt{R}_{i,ssid}]_{\mathbb{G}_2}, [\mathtt{X}_i\mathtt{I}_{\mathsf{id}_i}\mathtt{R}_{i,ssid}]_{\mathbb{G}_1}$)
  ∘ Send $f_i$ to $\mathcal{Z}$ as $\mathcal{P}_i$ towards $\mathcal{P}_j$ , and receive $f_i'$ from $\mathcal{Z}$ towards $\mathcal{P}_j$
  ∘ Parse $f_i'$ as (Flow, $\mathsf{id}', [a']_{\mathbb{G}_2}, [c']_{\mathbb{G}_1}$)
  ∘ Ignore if $a'{=}0$ or $c' \neq a'{\cdot}\mathtt{I}_{\mathsf{id}'}$
  ∘ Record $\langle$sent$, ssid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{id}', a', c'\rangle$
```

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

```
Upon (TestPwd, $sid||ssid, \mathcal{P}_i, [F]_{\mathbb{G}_T}$) from $\mathcal{Z}$ towards $\mathcal{F}_{\text{PAKE}}$:
  ∘ Retrieve $\langle$sent$, ssid, \mathcal{P}_j, \mathcal{P}_i, id', a', c'\rangle$
  ∘ For each $\langle$compromised$, \mathcal{P}_k, \mathsf{id}_k\rangle$ with $\mathsf{id}_k{=}\mathsf{id}'$:
      ▷ If $\mathtt{Z}_k$ appears in $F$:
          ◇ Send (Impersonate, $sid, ssid, \mathcal{P}_i, \mathcal{P}_k$) to $\mathcal{F}_{\text{siPAKE}}$
          ◇ If $\mathcal{F}_{\text{siPAKE}}$ returned "correct guess": replace all $\mathtt{Z}_k$ with $\mathtt{Z}_i$ in $F$
  ∘ For each password $\pi'$ queried by $\hat{H}_1(\pi')$:
      ▷ If $\mathtt{Y}_{\pi'}$ appears in $F$:
          ◇ Send (OnlineTestPwd, $sid, ssid, \mathcal{P}_i, \pi'$) to $\mathcal{F}_{\text{siPAKE}}$
          ◇ If $\mathcal{F}_{\text{siPAKE}}$ returned "correct guess": replace all $\mathtt{Y}_{\pi'}$ with $\mathtt{Z}_i$ in $F$
  ∘ If $F = a'\mathtt{X}_i\mathtt{Z}_i\mathtt{R}_{i,ssid}$:
      ▷ Return "correct guess" to $\mathcal{Z}$
  ∘ Otherwise:
      ▷ Send (OnlineTestPwd, $sid, ssid, \mathcal{P}_i, \perp$) to $\mathcal{F}_{\text{siPAKE}}$
      ▷ Return "wrong guess" to $\mathcal{Z}$
Upon (NewKey, $sid||ssid, \mathcal{P}_i, K'$) from $\mathcal{Z}$ towards $\mathcal{F}_{\text{PAKE}}$:
  ∘ Retrieve $\langle$sent$, ssid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{id}_i', a_i', c_i'\rangle$ and $\langle$sent$, ssid, \mathcal{P}_j, \mathcal{P}_i, \mathsf{id}_j', a_j', c_j'\rangle$
  ∘ If $\nexists\, \alpha{\in}\mathbb{Z}_q^\star$ s.t. $a_i'{=}\alpha\mathtt{X}_i\mathtt{R}_{i,ssid}$ and $a_j'{=}\alpha\mathtt{X}_j\mathtt{R}_{j,ssid}$:
      ▷ Send (OnlineTestPwd, $sid, ssid, \mathcal{P}_i, \perp$) to $\mathcal{F}_{\text{siPAKE}}$
  ∘ Send (NewKey, $sid, ssid, \mathcal{P}_i, \mathsf{id}_j', K'$) to $\mathcal{F}_{\text{siPAKE}}$
```

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

While we could have identified collisions across all $\mathcal{F}_{\text{GGP}}$ queries, we chose to limit OfflineTest-Pwd to only bilinear pairing evaluations (Pairing simulation), for better modelling of pre-computation

Upon $\left(\textsc{MulDiv}, sid, j_{\in\{1,2,T\}}, [F_1]_{\mathbb{G}_j}, [F_2]_{\mathbb{G}_j}, s\right)$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{GGP}}$**:**

- Return $[F_1 + (-1)^s \cdot F_2]_{\mathbb{G}_j}$ to $\mathcal{Z}$

Upon $(\textsc{Pairing}, sid, [F_1]_{\mathbb{G}_1}, [F_2]_{\mathbb{G}_2})$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{GGP}}$**:**

- $F_T \leftarrow F_1 \cdot F_2$
- Execute $\textsc{InsertRow}(v)$ on the coefficient vector v of F_T
- Return $[F_T]_{\mathbb{G}_T}$ to $\mathcal{Z}$

Upon $\left(\textsc{Isomorphism}, sid, j_{\in\{1,2\}}, [F]_{\mathbb{G}_j}\right)$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{GGP}}$**:**

- Return $[F]_{\mathbb{G}_{3-j}}$ to $\mathcal{Z}$

Upon $(\textsc{Hash}, sid, s)$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{GGP}}$**:**

- Return $\begin{cases} [\mathtt{Y}_\pi]_{\mathbb{G}_1} & s = 1||\pi \\ [\mathtt{I}_{\mathsf{id}}]_{\mathbb{G}_1} & s = 2||\mathsf{id} \end{cases}$ to $\mathcal{Z}$

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

resilience (see subsection 6.5). This implies that $\mathcal{S}$ needs to predict possible future collisions when simulating a pairing. This prediction is achieved by the polynomial matrix explained below.

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

function SelectPivot($v$)
    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 $(\textsc{OfflineTestPwd}, sid, \mathcal{P}_i, \pi')$ 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 $(\textsc{OfflineTestPwd}, sid, \mathcal{P}_i, \perp)$ 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 $\textsc{OfflineTestPwd}$ queries

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

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

When invoked by $\mathcal{A}$ to compute a pairing $\hat{e}([F_1]_{\mathbb{G}_1}, [F_2]_{\mathbb{G}_2})$, $\mathcal{S}$ first computes the product polynomial $F_T = F_1 \cdot F_2$, converts it to a coefficient vector V then applies the first step of row-reduction; that is, a linear combination of M's rows is added to V so to zero V's entries already selected as pivots for these rows. $\mathcal{S}$ then scans V for a non-zero entry corresponding to a term $\mathtt{X}_i\mathtt{Y}_{\pi'}$ (or $\mathtt{X}_i\mathtt{I}_{\mathsf{id}_i}\mathtt{Y}_{\pi'}$) for some compromised party $\mathcal{P}_i$ and a password guess π' (password guesses are taken from $\mathcal{A}$'s $\hat{H}_1(\pi')$ 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 π' (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 having some variables merged).

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

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

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

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

$$\begin{tikzcd}[column sep=huge] {} \arrow[r, shift left, "{f_i = (\textsc{Flow}, \mathsf{id}_i, \tilde{A}_i, \tilde{C}_i)}"] & \boxed{\mathcal{A}} \arrow[l, shift left, "{f'_j = (\textsc{Flow}, \mathsf{id}'_j, \tilde{A}'_j, \tilde{C}'_j)}"] \arrow[r, shift right, "{f'_i = (\textsc{Flow}, \mathsf{id}'_i, \tilde{A}'_i, \tilde{C}'_i)}"'] & {} \arrow[l, shift right, "{f_j = (\textsc{Flow}, \mathsf{id}_j, \tilde{A}_j, \tilde{C}_j)}"'] \end{tikzcd}$$

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

Query	Value	Real	Simulated
MulDiv	$\xi_1 \odot \xi_2$	$[a_1{+}a_2]_{\mathbb{G}_j}$	$[F_1{+}F_2]_{\mathbb{G}_j}$
	$\xi_1 \oslash \xi_2$	$[a_1{-}a_2]_{\mathbb{G}_j}$	$[F_1{-}F_2]_{\mathbb{G}_j}$
Pairing	$\hat{e}(\xi_1, \xi_2)$	$[a_1{\cdot}a_2]_{\mathbb{G}_T}$	$[F_1{\cdot}F_2]_{\mathbb{G}_T}$
Isomorphism	$\psi(\xi_1)$	$[a_1]_{\mathbb{G}_2}$	$[F_1]_{\mathbb{G}_2}$
	$\psi^{-1}(\xi_2)$	$[a_2]_{\mathbb{G}_1}$	$[F_2]_{\mathbb{G}_1}$
Hash	$\hat{H}_1(\pi')$	$[y_{\pi'}]_{\mathbb{G}_1}$	$[\mathtt{Y}_{\pi'}]_{\mathbb{G}_1}$
	$\hat{H}_2(\mathsf{id})$	$[\iota_{\mathsf{id}}]_{\mathbb{G}_1}$	$[\mathtt{I}_{\mathsf{id}}]_{\mathbb{G}_1}$
StealPwdFile	id_i	id_i	id_i
	$A_i = g_2^{x_i}$	$[x_i]_{\mathbb{G}_2}$	$[\mathtt{X}_i]_{\mathbb{G}_2}$
	$B_i = \hat{H}_1(\pi_i)^{x_i}$	$[x_i y_{\pi_i}]_{\mathbb{G}_1}$	$[\mathtt{X}_i \mathtt{Z}_i]_{\mathbb{G}_1}$
	$C_i = \hat{H}_2(\mathsf{id}_i)^{x_i}$	$[x_i \iota_{\mathsf{id}_i}]_{\mathbb{G}_1}$	$[\mathtt{X}_i \mathtt{I}_{\mathsf{id}_i}]_{\mathbb{G}_1}$
Flow	id_i	id_i	id_i
	$\tilde{A}_i = {A_i}^{r_i}$	$[x_i r_i]_{\mathbb{G}_2}$	$[\mathtt{X}_i \mathtt{R}_{i,ssid}]_{\mathbb{G}_2}$
	$\tilde{C}_i = {C_i}^{r_i}$	$[x_i \iota_{\mathsf{id}_i} r_i]_{\mathbb{G}_1}$	$[\mathtt{X}_i \mathtt{I}_{\mathsf{id}_i} \mathtt{R}_{i,ssid}]_{\mathbb{G}_1}$
TestPwd	$S_i = \hat{e}(\tilde{B}_i, \tilde{A}'_j)$ [9]	$[(x_i y_{\pi_i} r_i){\cdot}a'_j]_{\mathbb{G}_T}$	$[(\mathtt{X}_i \mathtt{Z}_i \mathtt{R}_{i,ssid}){\cdot}F'_j]_{\mathbb{G}_T}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We proceed to prove the lemmas:

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

1. **Hash queries.** Since Hash responses are taken from the uniform distribution over $\mathbb{Z}_q^\star$, the probability of such collisions is bound by $\frac{q_H}{q-1}$, where q_H is the number of Hash queries (polynomial in κ) and $q \geq 2^\kappa$.
2. **Variable Aliasing.** By Lemma 2, there are no aliasing collisions in the simulation.
3. **Variable Assignment.** Polynomials created by $\mathcal{S}$ for elements in $\mathbb{G}_1$ and $\mathbb{G}_2$ have maximal degree 3. MulDiv and Isomorphism queries cannot increase the degree, and Pairing allows creating polynomials in $\mathbb{G}_T$ adding the input degrees. Therefore, the maximal degree of any polynomial whose encoding is observed by $\mathcal{Z}$ is 3+3=6.
   Since in the real world the exponents (corresponding to variables in the simulated world) are taken from the uniform distribution over $\mathbb{Z}_q^\star$, the probability of assignment collisions $F_i(\vec{\mathtt{X}}) = F_j(\vec{\mathtt{X}})$ for some variable assignment $\vec{\mathtt{X}}$, is bound by:

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

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

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

Proof. Variable aliasing collisions take the form $\mathtt{Z}_i = \mathtt{Y}_{\pi_i}$, where π_i is the password assigned by the environment to party $\mathcal{P}_i$. They arise from defining separate formal variables to represent the logarithm of $\hat{H}_1(\pi)$ for (a) each party $\mathcal{P}_i$'s password π_i (unknown to the simulator) and (b) each adversary invocation of $\hat{H}_1$ on some password guess π'.

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

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

1	public generator
$\mathtt{X}_i$ $\mathtt{X}_i \cdot \mathtt{I}_{\mathsf{id}_i}$ $\mathtt{X}_i \cdot \mathtt{Z}_i$	$\mathcal{F}_{\text{PAKE}}$'s StealPwdFile query
$\mathtt{X}_i \cdot \mathtt{R}_{i,ssid}$ $\mathtt{X}_i \cdot \mathtt{R}_{i,ssid} \cdot \mathtt{I}_{\mathsf{id}_i}$	Flow message from $\mathcal{P}_i$
$\mathtt{Y}_\pi$	$\mathcal{F}_{\text{GGP}}$'s Hash query for $\hat{H}_1(\pi)$
$\mathtt{I}_{\mathsf{id}}$	$\mathcal{F}_{\text{GGP}}$'s Hash query for $\hat{H}_2(\mathsf{id})$

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

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

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

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

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

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

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

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

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

6.5 Pre-Computation Resilience

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

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

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

Therefore there is a one-to-one mapping between bilinear pairings computed by the real-world adversary after a compromise, and OfflineTestPwd queries sent by the ideal-world adversary $\mathcal{S}$ when simulating those computations. As a result, an environment $\mathcal{Z}$ equipped with awareness of failed offline tests (in the ideal-world) and of pairings (in the real-world) gains no advantage distinguishing these executions.

6.6 Group Reuse Across Sessions

The formalization of $\mathcal{F}_{\text{GGP}}$ requires the session identifier *sid* to be supplied as parameter on each operation, implying that each *sid* session has a dedicated independent group oracle. Such a requirement is easy to achieve with random oracles using domain separation, by prefixing any hash input with *sid*. However, domain separation is not trivially achieved with generic groups.

Nonetheless, the security of our protocol can be maintained even if the same group oracle is reused between sessions. The only requirement is for $\mathcal{F}_{\text{GGP}}$'s Hash query (hash to group) to be independent for each *sid* session. This preserves the validity of our simulator, since it ensures that there are no variable aliasing between the sessions (and other types of collisions occur with negligible probability). For readability we do not provide the modified version of the functionalities and UC proof. This will require adding a global group session (identified by *gid*) to determine the group, and add another layer of complexity to the notation and proof.

7 Computational Cost

The computational costs for CHIP and CRISP are summarized in the following table:

		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	$1H + 3E + \text{PAKE}$	$1P + \text{PAKE}$
Offline Test	Pre-Compromise	$1H$	$1\hat{H}$
	Post-Compromise	0	$1P$

Here H, $\hat{H}$, E, and P 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.

7.1 Password Hardening for Pre-Compromise

Common password hardening techniques (e.g., PBKDF2 [MKR17], Argon2 [BDK16], and scrypt [PJ16]) are used in the process of deriving a key from a password to increase the cost of brute-force attacks. As mentioned in Section 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 the CRISP protocol, 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.

7.2 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 6.5. This is also an upper bound for CRISP, since having compromised a password file, an adversary can check for any password guess π' if:

$$\hat{e}(B_i, g_2) \stackrel{?}{=} \hat{e}(\hat{H}_1(\pi'), A_i)$$

The left-hand side can be computed once and re-used for different guesses. The right-hand side must be computed per-password, but the invocation of $\hat{H}_1$ 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 [CS10]. Moreover, pairing requires more memory than a simple point multiplication and is harder to accelerate using GPUs [PL13].

In order to increase the difficulty of offline tests (password hardening), we cannot use a method such as iterative hashing, as was done in [JKX18]. However, by using larger group 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.

7.3 CRISP Optimization

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

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

Identity Verification A substantial part of the added computational cost of the protocol is the identity verification that requires two pairing operations. 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 wait for the other party to respond. The total computational cost remains the same, but the latency (or running time) of the protocol is reduced.
2. Verification delegation – Any party that receives the protocol messages, can verify the identity appearing in it (verification is only based on the identity and blinded values). We consider the

following scenario, where we have a broadcast network with many low-end devices (e.g., IoT devices) and one or more high-end devices (a controller or bridge). The bridge can perform the identity verification for all protocols in the network, and alert the user if any verification fails.

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

7.4 Performance Benchmark

We implemented CHIP and CRISP protocols, with groups over Ristretto-255 and BN-254 elliptic curves, using libsodium and MCL libraries. The source code is available at `https://github.com/shapaz/CRISP`.

In Table 3 we compare their online performance to those of other popular PAKE protocols, running on an i7-4790 processor. CPace [HL19] is the chosen IETF CFRG's candidates PAKE for usage with TLS 1.3 and is considered to be very efficient. SAE [Har08] is the underlying symmetric PAKE of Wi-Fi's WPA-3 and is designed to be supported by low-resource embedded devices.

	CPace	SAE	CHIP	CRISP
Communication[1]	2/1	3/2	3/2	3/2
CPU time (ms)	0.2	>10	0.5	0.6[2]

Table 3: Online performance comparison of PAKEs.

[1] Shown as #messages/#rounds.
[2] Excluding blinding and identity verification. See Section 7.3.

8 Conclusions and Open Problems

In this paper, we have formalized the novel notions of iPAKE and siPAKE. 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. 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.

The following open problems arise:

Two message protocol. In subsection 7.3, we showed how our protocols require only three messages. As shown in [KV11], 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.

Optimal bound on the cost of brute-force attack. In subsection 3.1 we have shown a black-box post-compromise brute-force attack on any PAKE protocol. The computational cost of the attack is *two* runs of the PAKE protocol for each offline password guess. However, 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 either find a more efficient black-box attack or to implement a more resilient PAKE.

Fine-grained password hardening. CRISP allows for a coarse-grained password hardening by changing the pairing group (e.g., using curves of larger size). Allowing fine-grained password hardening (e.g., iterative hashing) while preserving pre-computation resilience remains an open problem.

Acknowledgments

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

References

BB04. Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *EUROCRYPT*, 2004.

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

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

BF18. Richard Barnes and Owen Friel. Usage of pake with tls 1.3. Internet-Draft draft-barnes-tls-pake-04, IETF Secretariat, July 2018.

BJX19. Tatiana Bradley, Stanislaw Jarecki, and Jiayu Xu. Strong asymmetric PAKE based on trapdoor CKEM. In *CRYPTO*, 2019.

BM92. Steven M. Bellovin and Michael Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *IEEE Symposium on Security and Privacy*, 1992.

BM93. Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *ACM CCS*. ACM, 1993.

BMP00. Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using diffie-hellman. In *EUROCRYPT*, 2000.

BPR00. Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT*, 2000.

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

CHK$^+$05. Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT*, 2005.

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

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

GMR06. Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO*, 2006.

Har08. 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 (sensorcomm 2008)*, pages 839–844, 2008.

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

HL19. Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019, 2019.

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

JKX18. Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, 2018.

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

KV11. Jonathan Katz and Vinod Vaikuntanathan. Round-optimal password-based authenticated key exchange. In *TCC*, 2011.

MKR17. K. Moriarty, B. Kaliski, and A. Rusch. Pkcs #5: Password-based cryptography specification version 2.1. RFC 8018, RFC Editor, January 2017.
MSHH18. Nathaniel McCallum, Simo Sorce, Robbie Harwood, and Greg Hudson. Spake pre-authentication. Internet-Draft draft-ietf-kitten-krb-spake-preauth-06, IETF Secretariat, August 2018.
NYHR05. C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The kerberos network authentication service (v5). RFC 4120, RFC Editor, July 2005.
PJ16. C. Percival and S. Josefsson. The scrypt password-based key derivation function. RFC 7914, RFC Editor, August 2016.
PL13. Shi Pu and Jyh-Charn Liu. EAGL: an elliptic curve arithmetic gpu-based library for bilinear pairing. In *Pairing*, 2013.
Sho97. Victor Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT*, 1997.
Wi-18. Wi-Fi Alliance. WPA3 specification version 1.0. Retrieved 6 April 2019 from `https://www.wi-fi.org/file/wpa3-specification-v10`, April 2018.
Wri16. Jordan Wright. How browsers store your passwords (and why you shouldnt let them). Retrieved 9 February 2020 from `http://raidersec.blogspot.com/2013/06/how-browsers-store-your-passwords-and.html`, 2016.

A UC Security of the CHIP Protocol

We remark that the underlying IB-KA protocol satisfies the following properties:

1. **Weak Forward Secrecy:** A passive adversary has negligible probability in computing the session key K, even if any party is later compromised.
2. **KCI Resistance** Given that in session *sid* party $\mathcal{P}_i$ outputs $\langle \mathsf{id}, K \rangle$ and no party with identity id was compromised, the probability that an adversary can compute K is negligible. This property is proved in [FG10] based on the Strong CDH assumption (roughly speaking, assuming CDH problem remains hard given a DDH oracle).
3. **KDC Independent Flows:** The data exchanged between parties on each session is independent of the KDC secrets.

Before proving CHIP, we first explain why KCI resistance is preserved under our modifications. We stress that until $H_1(\pi_i)$ is queried by the adversary, the random oracle $\mathcal{F}_{\mathrm{RO}}$ acts like the original KDC, holding the secret random exponent y_i. Thus, the original property is preserved while $H_1(\pi_i)$ is not queried, and indeed the proof below only relies upon KCI resistance under this condition.

Following is the proof for UC security of CHIP protocol from Fig. 6, as stated in Theorem 1.

Proof (Theorem 1).
Let $\mathcal{A}$ be the dummy adversary running in the $(\mathcal{F}_{\mathrm{PAKE}}, \mathcal{F}_{\mathrm{RO}})$-hybrid world (will be referred to as "ideal world" from now on). Consider the simulator $\mathcal{S}$ depicted in Fig. 12 and Fig. 13, running in the ideal world and let $\mathcal{Z}$ be a PPT environment. The goal is to show that $\mathcal{Z}$'s views in both worlds are computationally indistinguishable.

First we observe that in the real world x_i and r_i are independent of the password, allowing $\mathcal{S}$ to perfectly simulate these values. It is also easy to see that the identity id_i sent by $\mathcal{S}$ in StealPwdFile query and flow comes from $\mathcal{F}_{\mathrm{iPAKE}}$ and thus matches the identity in the real world. Using the same calculations as a real party, $\mathcal{S}$ is therefore able to perfectly mimic that party in Flow. For StealPwdFile, however, $\mathcal{S}$ lacks the knowledge of the password, which is necessary for deriving y_i.

If $H_1(\pi_i)$ was requested by $\mathcal{Z}$ prior to compromising $\mathcal{P}_i$ via StealPwdFile, then $\mathcal{S}$ has issued an OfflineTestPwd query towards $\mathcal{F}_{\mathrm{iPAKE}}$ for $\mathcal{P}_i$ with password π_i. This query succeeded, causing $\mathcal{F}_{\mathrm{iPAKE}}$ to create an $\langle$offline, ...$\rangle$ record, ensuring that the correct π_i is returned from $\mathcal{F}_{\mathrm{iPAKE}}$ in response to StealPwdFile query. $\mathcal{S}$ can therefore generate y_i using $H_1[\pi_i]$ exactly like a real

```
Initially, pick $x_i \overset{R}{\leftarrow} \mathbb{Z}_q^\star$ for each party $\mathcal{P}_i$ and $H_i[\cdot]$ is undefined for $i = \{1,2\}$. Whenever $\mathcal{S}$ references an undefined
hash value $H_i[\cdot]$, set $H_1[\pi'] \overset{R}{\leftarrow} \mathbb{Z}_q^\star$, $H_2[\mathsf{id}, X] \overset{R}{\leftarrow} \mathbb{Z}_q^\star$.
Upon $(\textsc{StealPwdFile}, sid)$ from $\mathcal{Z}$ towards $\mathcal{P}_i$:
  ∘ Send $(\textsc{StealPwdFile}, sid, \mathcal{P}_i)$ to $\mathcal{F}_{\text{iPAKE}}$
  ∘ If $\mathcal{F}_{\text{siPAKE}}$ returned "no password file":
      ▷ return this to $\mathcal{Z}$
  ∘ Otherwise, $\mathcal{F}_{\text{iPAKE}}$ returned $\langle$"password file stolen"$, \mathsf{id}_i, \pi_i\rangle$
  ∘ If $\pi_i \neq \bot$:  set $y_i \leftarrow H_1[\pi_i]$
  ∘ Otherwise, pick $y_i \overset{R}{\leftarrow} \mathbb{Z}_q^\star$
  ∘ For each $\langle\textsc{compromised}, \mathcal{P}_k, \cdot, y_k\rangle$:
      ▷ Send $(\textsc{OfflineComparePwd}, sid, \mathcal{P}_i, \mathcal{P}_k)$ to $\mathcal{F}_{\text{iPAKE}}$
      ▷ If $\mathcal{F}_{\text{iPAKE}}$ returned "passwords match": set $y_i \leftarrow y_k$
  ∘ Record $\langle\textsc{compromised}, \mathcal{P}_i, \mathsf{id}_i, y_i\rangle$
  ∘ $X_i \leftarrow g^{x_i}$,    $Y_i \leftarrow g^{y_i}$
  ∘ $h_i \leftarrow H_2[\mathsf{id}_i, X_i]$
  ∘ $\hat{x}_i \leftarrow x_i + h_i \cdot y_i$
  ∘ Return $\langle\textsc{file}, \mathsf{id}_i, X_i, Y_i, \hat{x}_i\rangle$ to $\mathcal{Z}$
Upon $(\textsc{Hash}, sid, s)$ from $\mathcal{Z}$ towards $\mathcal{F}_{\text{RO}}$:
  ∘ If $s = \langle 1, \pi'\rangle$: for each party $\mathcal{P}_i$:
      ▷ Send $(\textsc{OfflineTestPwd}, sid, ssid, \mathcal{P}_i, \pi')$ to $\mathcal{F}_{\text{iPAKE}}$
      ▷ If $\mathcal{F}_{\text{iPAKE}}$ replied "correct guess":
          ◇ Retrieve $\langle\textsc{compromised}, \mathcal{P}_i, \cdot, y_i\rangle$
          ◇ set $H_1[\pi'] \leftarrow y_i$
  ∘ Return to $\mathcal{Z}$ $\begin{cases} H_1[\pi'] & s = \langle 1, \pi'\rangle \\ H_2[\mathsf{id}, X] & s = \langle 2, \mathsf{id}, X\rangle \end{cases}$
```

Fig. 12: Simulator $\mathcal{S}$ in the offline part

party. If, on the other hand, $H_1(\pi_i)$ has not been queried by the time $\mathcal{P}_i$ is compromised, then the value y_i is generated by $\mathcal{S}$ in StealPwdFile and saved in a $\langle\textsc{compromised}, \dots\rangle$ record. If $\mathcal{Z}$ will later choose to compute $H_1(\pi_i)$, $\mathcal{S}$ will detect this from a "correct guess" response to its OfflineTestPwd query, and will set $H_1[\pi_i]$ using the recorded y_i. Also notice that in this case $\mathcal{S}$ utilizes OfflineComparePwd queries to match the value y_i with y_k of a previously compromised party $\mathcal{P}_k$, when $\pi_i = \pi_k$.

Because $\mathcal{S}$ acts exactly like a random oracle in Hash queries, and since we have already considered the password file obtained by StealPwdFile and the values of Flow, we are left with $\mathcal{F}_{\text{PAKE}}$'s TestPwd and NewKey queries.

TestPwd. Consider TestPwd query's output. If in the real world $\mathcal{F}_{\text{PAKE}}$ returns "correct guess" then $\alpha' = \alpha_i$, $\beta' = \beta_i$ and $\mathsf{tr}' = \mathsf{tr}_i$. In the simulated world, $\mathcal{S}$ easily tests α' and tr', since α_i and tr_i are independent of the password and can be simulated. To check β' $\mathcal{S}$ has to extract the guess for y', either from $H_1(\pi')$ queried earlier, or from a compromised party's y_k.

If the correct password has been previously queried by $H_1(\pi_i)$, then $\mathcal{S}$ will compute $\beta(H_1(\pi_i))$ exactly like a real-world party $\mathcal{P}_i$. In this case the comparison with β' must succeed, and $\mathcal{S}$ will issue an OnlineTestPwd query for π_i, which will result in $\mathcal{F}_{\text{iPAKE}}$'s session being compromised and "correct guess" being returned. However, if $H_1(\pi_i)$ has not been queried yet, then KCI resistance is preserved, and thus $\mathcal{Z}$ could have only provided $\alpha' = \alpha_i$ and $\beta' = \beta_i$ if some compromised party $\mathcal{P}_k$ has $\mathsf{id}_k = \mathsf{id}'_j$ and $\pi_k = \pi_i$. In this case $\mathcal{S}$ will find $\beta(y_k) = \beta(H_1(\pi_k)) = \beta(H_1(\pi_i)) = \beta_i = \beta'$ and will

```
**Upon** $(\textsc{NewSession}, sid, ssid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{id}_i)$ **from** $\mathcal{F}_{\text{iPAKE}}$:
  ∘ Ignore if there is a record $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, \cdot\rangle$
  ∘ Pick $r_i \overset{\text{R}}{\leftarrow} \mathbb{Z}_q^\star$
  ∘ Record $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, r_i\rangle$ and mark it FRESH
  ∘ $X_i \leftarrow g^{x_i}, \quad R_i \leftarrow g^{r_i}$
  ∘ $f_i \leftarrow (\textsc{Flow}, \mathsf{id}_i, X_i, R_i)$
  ∘ Send $f_i$ to $\mathcal{Z}$ as $\mathcal{P}_i$ towards $\mathcal{P}_j$ and receive $f_i'$ from $\mathcal{Z}$ towards $\mathcal{P}_j$
  ∘ Parse $f_i'$ as $(\textsc{Flow}, \mathsf{id}_i', X_i', R_i')$
  ∘ Record $\langle \textsc{sent}, ssid, \mathcal{P}_i, (\mathsf{id}_i, X_i, R_i)\rangle$ and $\langle \textsc{recv}, ssid, \mathcal{P}_j, (\mathsf{id}_i', X_i', R_i')\rangle$
**Upon** $(\textsc{TestPwd}, sid, ssid, \mathcal{P}_i, \alpha', \beta', \mathsf{tr}')$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{PAKE}}$:
  ∘ Retrieve $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, r_i\rangle$ marked FRESH
  ∘ Retrieve $\langle \textsc{send}, ssid, \mathcal{P}_i, f_i\rangle$ and $\langle \textsc{recv}, ssid, \mathcal{P}_i, f_j' = (\mathsf{id}_j', X_j', R_j')\rangle$
  ∘ Set $\alpha_i \leftarrow R_j'^{r_i}, \quad h_i \leftarrow H_2[\mathsf{id}_i, X_i], \quad h_j \leftarrow H_2[\mathsf{id}_j', X_j']$
  ∘ $\mathsf{tr}_i \leftarrow \langle \min(f_i, f_j'), \max(f_i, f_j')\rangle$
  ∘ Define $\beta(y)$: $y \mapsto (R_j' X_j' g^{y\cdot h_j})^{r_i + x_i + y\cdot h_i}$
  ∘ If $\alpha_i = \alpha'$ and $\mathsf{tr}_i = \mathsf{tr}'$:
      ▹ If there is an entry $H_1[\pi'] = y'$ with $\beta(y') = \beta'$:
          ⋄ Send $(\textsc{OnlineTestPwd}, sid, ssid, \mathcal{P}_i, \pi')$ to $\mathcal{F}_{\text{iPAKE}}$
      ▹ Otherwise, if there is a record $\langle \textsc{compromised}, \mathcal{P}_k, \mathsf{id}_k, y_k\rangle$ with $\beta(y_k) = \beta'$ and $\mathsf{id}_k = \mathsf{id}_j'$:
          ⋄ Send $(\textsc{Impersonate}, sid, ssid, \mathcal{P}_i, \mathcal{P}_k)$ to $\mathcal{F}_{\text{iPAKE}}$
  ∘ If no other query was sent:
      ▹ Send $(\textsc{OnlineTestPwd}, sid, ssid, \mathcal{P}_i, \perp)$ to $\mathcal{F}_{\text{iPAKE}}$
  ∘ Mark the session COMPROMISED if $\mathcal{F}_{\text{iPAKE}}$ replied "correct guess" or INTERRUPTED otherwise
  ∘ Forward $\mathcal{F}_{\text{iPAKE}}$'s response to $\mathcal{Z}$
**Upon** $(\textsc{NewKey}, sid, ssid, \mathcal{P}_i, \alpha')$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{PAKE}}$:
  ∘ Retrieve $\langle \textsc{session}, ssid, \mathcal{P}_i, \mathcal{P}_j, r_i\rangle$ not marked COMPLETED
  ∘ Retrieve $\langle \textsc{sent}, ssid, \mathcal{P}_j, f_j\rangle$ and $\langle \textsc{recv}, ssid, \mathcal{P}_i, f_j' = (\mathsf{id}_j', \cdot, \cdot)\rangle$
  ∘ if the session is FRESH and $f_j \neq f_j'$:
      ▹ Send $(\textsc{OnlineTestPwd}, sid, ssid, \mathcal{P}_i, \perp)$ to $\mathcal{F}_{\text{iPAKE}}$
  ∘ Mark the session COMPLETED
  ∘ Send $(\textsc{NewKey}, sid, ssid, \mathcal{P}_i, K', \mathsf{id}_j')$ to *to* $\mathcal{F}_{\text{iPAKE}}$
```

Fig. 13: Simulator $\mathcal{S}$ in the online part

issue an IMPERSONATE query for $\mathcal{P}_k$, which will also result in $\mathcal{F}_{\text{iPAKE}}$'s session being COMPROMISED and "correct guess" to be returned.

In the other direction, if the TESTPWD query succeeded in the simulated world, then $\mathcal{S}$ found that $\alpha' = \alpha_i$, $tr' = \mathsf{tr}_i$ and either $\beta' = \beta(H_1[\pi'])$ or $\beta' = \beta(y_k)$. In the first case, $\mathcal{S}$ received "correct guess" in response to an ONLINETESTPWD query with π', implying $\pi_i = \pi'$ and thus $\beta' = \beta(H_1(\pi_i))$. In the latter case, $\mathcal{S}$ received "correct guess" from $\mathcal{F}_{\text{iPAKE}}$ after issuing an IMPERSONATE query, implying $\pi_i = \pi_k$ and thus $\beta' = \beta(H_1(\pi_k)) = \beta(H_1(\pi_i))$. This ensures that in both cases the real-world adversary would also get "correct guess" from $\mathcal{F}_{\text{PAKE}}$.

Note that since in both worlds TESTPWD query either results in "correct guess" with the session being marked COMPROMISED, or "wrong guess" and the session becoming INTERRUPTED, then both TESTPWD's output and the session state are equivalent in both worlds.

We now consider party $\mathcal{P}_i$'s output, which consists of a session key K_i and an identity id.

Identity. It is easy to see that $\mathcal{S}$ uses id'_j that was received in the modified flow f'_j, as the identity for $\mathcal{F}_{\text{iPAKE}}$'s NewKey. In the real world, an honest party $\mathcal{P}_i$ will also use this identity for its output. Therefore, we only need to show that $\mathcal{F}_{\text{iPAKE}}$ allows this identity as input.

When the session is interrupted $\mathcal{F}_{\text{iPAKE}}$ does not limit the selection of the identity at all. When the session is fresh $\mathcal{S}$ checks if $\mathcal{Z}$ asked to make any change in the flow f_j from $\mathcal{P}_j$ to $\mathcal{P}_i$. If it did not, then $\mathsf{id}'_j = \mathsf{id}_j$ and $\mathcal{F}_{\text{iPAKE}}$ will accept this identity. Otherwise, $\mathcal{S}$ sends an OnlineTestPwd query with $\perp$ as password, which will make the session interrupted in $\mathcal{F}_{\text{iPAKE}}$, and as a result will allow $\mathcal{S}$ to choose any identity.

When the session is compromised, $\mathcal{S}$ must have succeeded in either an OnlineTestPwd or an Impersonate query earlier. If the former was queried, then $\mathcal{F}_{\text{iPAKE}}$ allows $\mathcal{S}$ to choose any identity. Otherwise, it was a successful impersonation of party $\mathcal{P}_k$ towards $\mathcal{P}_i$. As shown above for TestPwd query, thanks to KCI resistance, this only happens when $\mathsf{id}_k = \mathsf{id}'_j$, which $\mathcal{F}_{\text{iPAKE}}$ permits.

Session Key. Finally, consider the output key K_i. Recall that when NewKey is requested by the environment, $\mathcal{F}_{\text{iPAKE}}$'s session in the simulated world has the same state as $\mathcal{F}_{\text{PAKE}}$ in the real world. Since $\mathcal{F}_{\text{iPAKE}}$ and $\mathcal{F}_{\text{PAKE}}$ use the same logic for selecting the session key (either $\mathcal{Z}$'s K', a previous K_j or a fresh random K_i), it seems clear that $\mathcal{Z}$ cannot distinguish between these keys. However, when the session is fresh and a change in the flow from $\mathcal{P}_j$ to $\mathcal{P}_i$ is detected ($f_j \neq f'_j$) $\mathcal{S}$ sends an OnlineTestPwd to $\mathcal{F}_{\text{iPAKE}}$ making the session interrupted (this was necessary for setting the identity, as explained above). Nevertheless, in this case the real parties observe different transcripts ($\mathsf{tr}_i \neq \mathsf{tr}_j$) and thus they provide $\mathcal{F}_{\text{PAKE}}$ with different inputs. Therefore, $\mathcal{F}_{\text{PAKE}}$ will provide $\mathcal{P}_i$ with random key K_i (since its session is fresh) regardless of its password. As for $\mathcal{P}_j$'s key, it is only affected by $\mathcal{P}_i$'s session state when $\mathcal{P}_j$'s own session is fresh, in which case it will be assigned an independent random key K_j. Recall that $\mathcal{S}$ made $\mathcal{P}_i$'s session in $\mathcal{F}_{\text{iPAKE}}$ interrupted, so in the simulated world too $\mathcal{P}_i$ and $\mathcal{P}_j$'s session keys will be randomly chosen.

Since with astonishing probability $\mathcal{Z}$ cannot distinguish between the real and ideal world views, CHIP UC-realizes $\mathcal{F}_{\text{iPAKE}}$ as stated.

B CHIP Variant With Key Confirmation

In Section 5 we showed how to combine an IB-KA protocol together with any symmetric PAKE to construct the CHIP protocol. The construction was sequential; at first the parties executed the IB-KA protocol, and only then were they able to engage in a PAKE, using the negotiated shared values (α_i and β_i). One might wonder if these two communication rounds can be merged by simultaneously executing both IB-KA and PAKE.

To run PAKE in parallel to IB-KA, the input to PAKE cannot depend on the IB-KA output. Instead, during the password file generation phase we derive two independent values from the password (instead of just one): $y_i, p_i \leftarrow H_1(\pi_i)$. As before, y_i is used to simulate the KDC's private key. The new value p_i is added to the password file, and will be provided as input to PAKE. Finally, both keys of IB-KA and PAKE should be combined in the derivation of the session key.

Unfortunately, this construction does not provide Perfect Forward Secrecy (PFS). Assume there is a set of parties that share the same password. Once an adversary has compromised a party, it can actively interfere in sessions between any two parties. When the correct password is later guessed, the adversary will find the keys to all those sessions. Recall that the IB-KA protocol only guarantees Weak Forward Secrecy (wFS not PFS), i.e. past sessions in which the adversary $\mathcal{A}$ was active are vulnerable when long term keys are compromised. In our settings, guessing the correct password after a session has ended allows $\mathcal{A}$ to find the IB-KA key. In order to find the final session key, $\mathcal{A}$ also has to succeed in a TestPwd against the PAKE. However, since the correct input is p_i, which is common to all parties (with the same password), the adversary only needs to have had compromised in advance a single party $\mathcal{P}_k$ (with $\pi_k = \pi_i$) so it can use its $p_k = p_i$ and bypass the PAKE.

We remark that this attack is possible due to the imperfect forward secrecy of IB-KA. Thus, we can eliminate it by adding PFS to the scheme. We augment our construction with an explicit key confirmation, and include the transcripts in the session key derivation. Intuitively, this prevents the aforementioned attack by requiring the adversary to find the correct password *during the session* to pass the key confirmation. The transcripts are included to prevent honest parties from agreeing on a session key in presence of an active adversary. In this case, even the impersonated party will not be able to complete the key confirmation. Unconfirmed, the resulting key will never be used, and the adversary will gain nothing from finding it later, when the password is guessed.

Although adding a key confirmation results in a two-round iPAKE protocol, with which we have started, we stress that in many real-life scenarios (such as TLS) an explicit key confirmation exists anyway, and so the added communication cost is only one round.

The complete CHIP variant described above is depicted in Fig. 14. Note that it combines the transcripts of both IB-KA and the underlying PAKE[10]. This requires a slight modification of $\mathcal{F}_{\text{PAKE}}$, to make it output the transcript together with the session key, as was done in [GMR06].

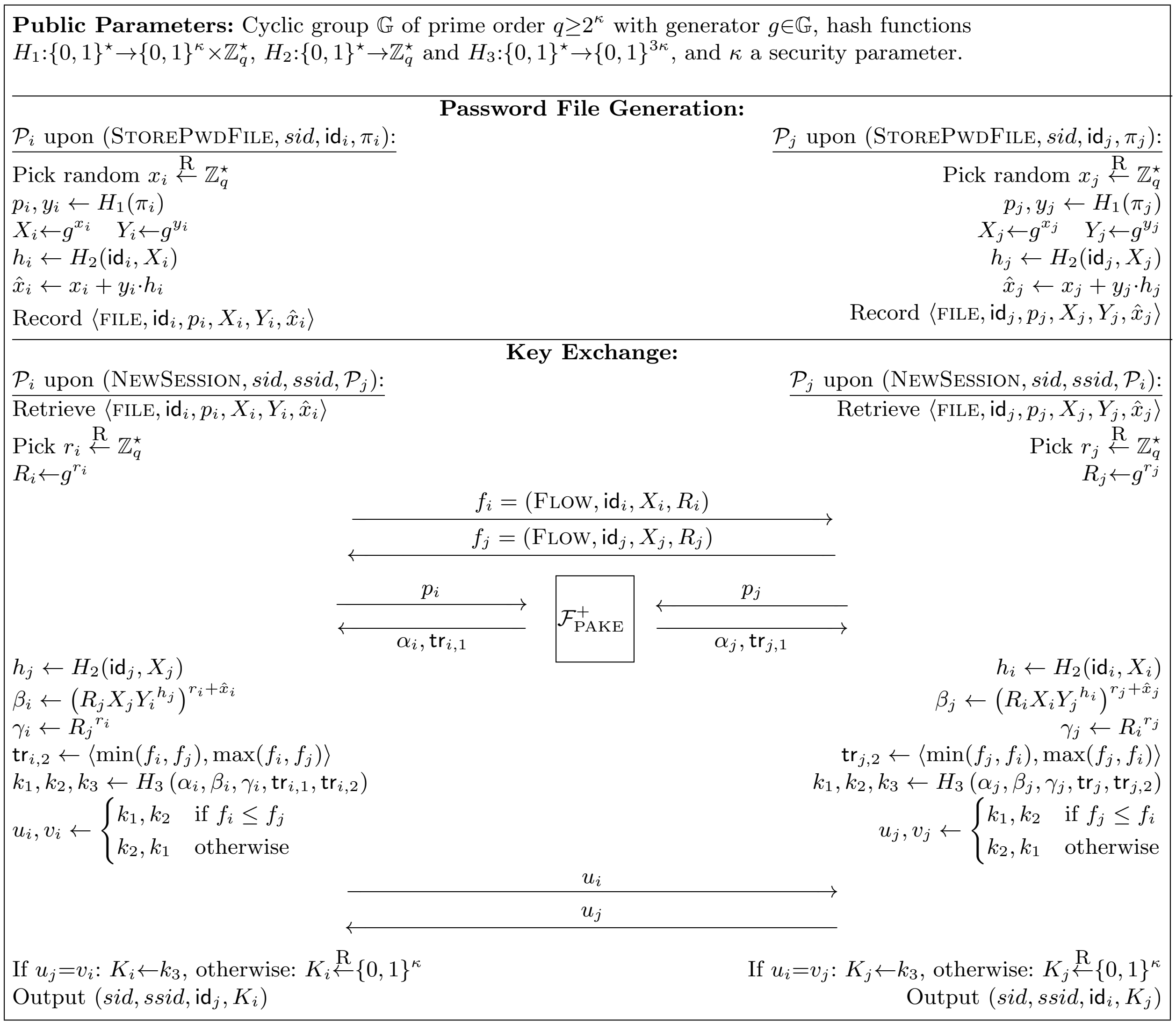
C Asymmetric PAKE Functionality

Fig. 15 shows the Strong Asymmetric PAKE functionality from [JKX18], in which only two parties engage: a server S and a user U. It introduces the concept of a password file, created for S upon StorePwdFile query and disclosed to the adversary upon adaptive corruption query StealPwdFile modelling a server compromise attack. Once a server's password file is obtained, the ideal-world adversary is able to mount an offline guessing attack using OfflineTestPwd queries, and an online impersonation attack using Impersonate query.

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

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

[10] The PAKE's transcript is necessary for simulating the case where the adversary uses a compromised p_k value to set the PAKE key, but does not modify the IB-KA flows. In this case the adversary cannot compute the session key, but decides whether the parties output matching or different keys.



Public Parameters: Cyclic group $\mathbb{G}$ of prime order $q \geq 2^\kappa$ with generator $g \in \mathbb{G}$, hash functions $H_1 : \{0,1\}^\star \to \{0,1\}^\kappa \times \mathbb{Z}_q^\star$, $H_2 : \{0,1\}^\star \to \mathbb{Z}_q^\star$ and $H_3 : \{0,1\}^\star \to \{0,1\}^{3\kappa}$, and κ a security parameter.

Password File Generation:

$\mathcal{P}_i$ upon $(\textsc{StorePwdFile}, sid, \mathsf{id}_i, \pi_i)$:	$\mathcal{P}_j$ upon $(\textsc{StorePwdFile}, sid, \mathsf{id}_j, \pi_j)$:
Pick random $x_i \overset{R}{\leftarrow} \mathbb{Z}_q^\star$	Pick random $x_j \overset{R}{\leftarrow} \mathbb{Z}_q^\star$
$p_i, y_i \leftarrow H_1(\pi_i)$	$p_j, y_j \leftarrow H_1(\pi_j)$
$X_i \leftarrow g^{x_i} \quad Y_i \leftarrow g^{y_i}$	$X_j \leftarrow g^{x_j} \quad Y_j \leftarrow g^{y_j}$
$h_i \leftarrow H_2(\mathsf{id}_i, X_i)$	$h_j \leftarrow H_2(\mathsf{id}_j, X_j)$
$\hat{x}_i \leftarrow x_i + y_i \cdot h_i$	$\hat{x}_j \leftarrow x_j + y_j \cdot h_j$
Record $\langle \text{FILE}, \mathsf{id}_i, p_i, X_i, Y_i, \hat{x}_i \rangle$	Record $\langle \text{FILE}, \mathsf{id}_j, p_j, X_j, Y_j, \hat{x}_j \rangle$

Key Exchange:

$\mathcal{P}_i$ upon $(\textsc{NewSession}, sid, ssid, \mathcal{P}_j)$:	$\mathcal{P}_j$ upon $(\textsc{NewSession}, sid, ssid, \mathcal{P}_i)$:
Retrieve $\langle \text{FILE}, \mathsf{id}_i, p_i, X_i, Y_i, \hat{x}_i \rangle$	Retrieve $\langle \text{FILE}, \mathsf{id}_j, p_j, X_j, Y_j, \hat{x}_j \rangle$
Pick $r_i \overset{R}{\leftarrow} \mathbb{Z}_q^\star$	Pick $r_j \overset{R}{\leftarrow} \mathbb{Z}_q^\star$
$R_i \leftarrow g^{r_i}$	$R_j \leftarrow g^{r_j}$

$$\xrightarrow{f_i = (\textsc{Flow}, \mathsf{id}_i, X_i, R_i)}$$
$$\xleftarrow{f_j = (\textsc{Flow}, \mathsf{id}_j, X_j, R_j)}$$

$$\begin{tikzcd} \mathcal{P}_i \arrow[r, shift left, "p_i"] & \boxed{\mathcal{F}^{+}_{\text{PAKE}}} \arrow[l, shift left, "{\alpha_i, \mathsf{tr}_{i,1}}"] \arrow[r, shift left, "{\alpha_j, \mathsf{tr}_{j,1}}"] & \mathcal{P}_j \arrow[l, shift left, "p_j"] \end{tikzcd}$$

$h_j \leftarrow H_2(\mathsf{id}_j, X_j)$	$h_i \leftarrow H_2(\mathsf{id}_i, X_i)$
$\beta_i \leftarrow \left(R_j X_j {Y_i}^{h_j}\right)^{r_i + \hat{x}_i}$	$\beta_j \leftarrow \left(R_i X_i {Y_j}^{h_i}\right)^{r_j + \hat{x}_j}$
$\gamma_i \leftarrow {R_j}^{r_i}$	$\gamma_j \leftarrow {R_i}^{r_j}$
$\mathsf{tr}_{i,2} \leftarrow \langle \min(f_i, f_j), \max(f_i, f_j) \rangle$	$\mathsf{tr}_{j,2} \leftarrow \langle \min(f_j, f_i), \max(f_j, f_i) \rangle$
$k_1, k_2, k_3 \leftarrow H_3(\alpha_i, \beta_i, \gamma_i, \mathsf{tr}_{i,1}, \mathsf{tr}_{i,2})$	$k_1, k_2, k_3 \leftarrow H_3(\alpha_j, \beta_j, \gamma_j, \mathsf{tr}_j, \mathsf{tr}_{j,2})$
$u_i, v_i \leftarrow \begin{cases} k_1, k_2 & \text{if } f_i \leq f_j \\ k_2, k_1 & \text{otherwise} \end{cases}$	$u_j, v_j \leftarrow \begin{cases} k_1, k_2 & \text{if } f_j \leq f_i \\ k_2, k_1 & \text{otherwise} \end{cases}$

$$\xrightarrow{u_i}$$
$$\xleftarrow{u_j}$$

If $u_j = v_i$: $K_i \leftarrow k_3$, otherwise: $K_i \overset{R}{\leftarrow} \{0,1\}^\kappa$	If $u_i = v_j$: $K_j \leftarrow k_3$, otherwise: $K_j \overset{R}{\leftarrow} \{0,1\}^\kappa$
Output $(sid, ssid, \mathsf{id}_j, K_i)$	Output $(sid, ssid, \mathsf{id}_i, K_j)$

Fig. 14: CHIP variant with key confirmation

```
Functionalities $\mathcal{F}_{\text{aPAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$, with security parameter $\kappa$, interacting with parties $\{U, S\}$ and an adversary $\mathcal{S}$.

Upon $(\textsc{StorePwdFile}, sid, U, \pi_S)$ from $S$:
  ◦ If there is no record $\langle \textsc{file}, U, S, \cdot \rangle$:
      ▷ record $\langle \textsc{file}, U, S, \pi_S \rangle$ and mark it UNCOMPROMISED

Upon $(\textsc{StealPwdFile}, sid, S)$ from $\mathcal{S}$:
  ◦ If there is a record $\langle \textsc{file}, U, S, \pi_S \rangle$:
      ▷ mark it COMPROMISED
      ▷ $\pi \leftarrow \begin{cases} \pi_S & \text{if there is a record } \langle \textsc{offline}, \pi' \rangle \text{ with } \pi'{=}\pi_S \\ \perp & \text{otherwise} \end{cases}$
      ▷ return $\langle$ "password file stolen" $, \pi \rangle$ to $\mathcal{S}$
  ◦ else: return "no password file" to $\mathcal{S}$

Upon $(\textsc{OfflineTestPwd}, sid, S, \pi')$ from $\mathcal{S}$:
  ◦ Retrieve $\langle \textsc{file}, U, S, \pi_S \rangle$
  ◦ If it is marked COMPROMISED:
      ▷ if $\pi_S = \pi'$: return "correct guess" to $\mathcal{S}$
      ▷ else: return "wrong guess" to $\mathcal{S}$
  ◦ otherwise: Record $\langle \textsc{offline}, \pi' \rangle$

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

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

Upon $(\textsc{OnlineTestPwd}, sid, ssid, \mathcal{P}, \pi')$ from $\mathcal{S}$:
  ◦ Retrieve $\langle \textsc{session}, ssid, \mathcal{P}, \mathcal{P}', \pi_{\mathcal{P}} \rangle$ marked FRESH
  ◦ if $\pi_{\mathcal{P}} = \pi'$: mark the session COMPROMISED and return "correct guess" to $\mathcal{S}$
  ◦ else: mark the session INTERRUPTED and return "wrong guess" to $\mathcal{S}$

Upon $(\textsc{Impersonate}, sid, ssid)$ from $\mathcal{S}$:
  ◦ Retrieve $\langle \textsc{session}, ssid, U, S, \pi_U \rangle$ marked FRESH
  ◦ Retrieve $\langle \textsc{file}, U, S, \pi_S \rangle$ marked COMPROMISED
  ◦ If $\pi_U = \pi_S$: mark the session COMPROMISED and return "correct guess" to $\mathcal{S}$
  ◦ else: mark the session INTERRUPTED and return "wrong guess" to $\mathcal{S}$

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

Fig. 15: Asymmetric PAKE functionality $\mathcal{F}_{\text{aPAKE}}$ (full text) and Strong Asymmetric PAKE functionality $\mathcal{F}_{\text{saPAKE}}$ (grey text omitted)