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

Version 2.0* – March 2021

Cas Cremers**, Moni Naor***, Shahar Paz†, and Eyal Ronen(✉)‡

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

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

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

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

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

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

* We provide an overview of the main differences between versions in Appendix E.

** CISPA Helmholtz Center for Information Security. Email: `cremers@cispa.de`

*** Department of Computer Science and Applied Mathematics, Weizmann Institute of Science. l. Supported in part by grants from the Israel Science Foundation (no. 950/15 and 2686/20) and by the Simons Foundation Collaboration on the Theory of Algorithmic Fairness. Also received funding from the European Research Council (ERC) under the European Union's 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

Passwords are arguably the most widely deployed authentication method today, and are used in a vast range of applications from authentication on the internet (e.g., email and bank servers), wireless network encryption (e.g., Wi-Fi, Smart Homes, Industry 4.0), and enterprise network authentication (e.g., Kerberos [24] and EAP-pwd [17]). Early password-based protocols allowed adversaries to verify password guesses offline against observed network traffic. To remedy this, Password Authenticated Key Exchange (PAKE) protocols were proposed, as first studied by Bellovin and Merritt [2]. PAKEs allow parties to negotiate a strong secret key based only on the knowledge of a shared and possibly low-entropy password, do not leak any information about the password to passive adversaries, and allow only an inevitable online password guess attack.

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

To partially mitigate this threat, Bellovin and Merritt [3] proposed so-called asymmetric PAKEs (also known as aPAKEs, Augmented PAKEs, or V(erifier)-PAKEs) that make this much harder: the clients still need to provide the password in plaintext, but the verifying servers now only need to provide, and thus store, information that (a) is derived from the password using a one-way function, yet (b) allows establishing a shared key with a party that knows the password. Thus, compromising an aPAKE server does not allow the adversary to impersonate the client, and forces it to perform a brute-force attempt to extract the password.

1.1 Identity-binding PAKEs (iPAKE)

aPAKE protocols still have substantial limitations: they only protect the server, and perhaps more importantly, cannot be applied to settings that do not fall into the client-server model, e.g., where a password can be shared among group members that can communicate with all other members. Prime examples of such *symmetric* settings are found in wireless networking and IoT settings. For example, the globally deployed IEEE 802.11 Wi-Fi standard includes the WPA protocol, which uses network passwords to enable devices to automatically connect to routers, extenders, and mesh network nodes; crucially, all parties can automatically communicate with each other using the network password without any user input. This led the Wi-Fi alliance to base their latest WPA3 protocol [28] on a symmetric PAKE for mesh networks called Simultaneous Authentication of Equals (SAE) [16].

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

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

In this work we address this gap by initiating the study and construction of so-called *identity-binding* PAKEs (iPAKE). We provide a UC-security definition that is the symmetric counterpart to aPAKE. We instantiate iPAKE with CHIP, a novel compiler from any PAKE to iPAKE. We leverage ideas from Identity-Based Key-Exchange to introduce abstract identities for each party, and effectively bind the locally stored password-derived data to these identities, while retaining the required key agreement functionality. Identities can be arbitrary bit strings, and could also encode functions or roles instead of the party's name, e.g., "server", "router", or "fire brigade chief", "Elon's third iPhone". Unlike Identity-Based Key-Exchange, we do not require a third party: instead, each party locally simulates the Key Distribution Center during the password file generation.

1.2 Strong Identity-binding PAKEs (siPAKE)

In 2018, Jarecki, Krawczyk, and Xu [19] strengthened the aPAKE notion by additionally requiring that an adversary gains no benefits from any pre-computations performed before a server compromise, thereby forcing it to do a full brute-force attack after the compromise. They named this notion *strong* asymmetric PAKE (saPAKE), and proposed the OPAQUE protocol to meet it. This has been widely regarded as a major step forward, and has led the Internet Engineering Task Force (IETF) to work towards standardizing OPAQUE and its use for TLS 1.3's password-based logins [22].

To provide similar protection against pre-computations, we strengthen iPAKE to *strong identity-binding* PAKEs (siPAKE), and provide a UC-security definition that is the symmetric counterpart to saPAKE. We instantiate siPAKE with CRISP, a novel compiler from any PAKE to siPAKE, that extends the protection provided by state-of-the-art saPAKE protocols [7, 19] to all parties.

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

1.3 Contributions

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

Security notion		Example protocol		Post-compromise impersonation resistance	Secure against pre-computation
PAKE	[10]	CPace	[15]	○	○
aPAKE	[13]	AuCPace	[15]	◐	○
iPAKE	(Section 4)	**CHIP**	(Section 5)	●	○
saPAKE	[19]	OPAQUE	[19]	◐	◐
siPAKE	(Section 4)	**CRISP**	(Section 6)	●	●

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

(Section 7.4) that shows their performance is in line with, or even better than, state-of-the-art protocols.

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

1.4 Structure of the Paper

We give background on the formalization of PAKEs in Section 2. We discuss various methods for compromise resilience in Section 3. We describe the ideal functionalities for the extensions of PAKE (including the UC Modelling of Generic Groups) in Section 4. We introduce the CHIP compiler in Section 5 and the CRISP compiler 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. We conclude and present open problems in Section 8.

We provide full proofs and further reference material in the supplementary appendix. In particular, we give the full proof for CHIP in Appendix A and provide a variant with key confirmation in Appendix B. We give the full proof for CRISP in Appendix C. For reference, we recall the (strong) asymmetric PAKE functionalities in Appendix D.

2 Formalizations of PAKE

Bellare, Pointcheval, and Rogaway [1] were the first to formalize the notion of PAKE. Canetti, Halevi, Katz, Lindell, and MacKenzie [10] formalized PAKE in the Universal Composability (UC) framework [9]. Their ideal functionality $\mathcal{F}_{\mathrm{PAKE}}$ (originally denoted $\mathcal{F}_{\mathrm{pwKE}}$) trades each party's password with a randomly chosen key for the session, only allowing the adversary an online attack where a single guess may be made to some party's password.

Asymmetric PAKE (aPAKE) protocols (a.k.a. Augmented PAKEs or Verifier PAKEs) were formalized by Boyko, MacKenzie, and Patel [6]. 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, MacKenzie, and Ramzan [13] formalized an ideal functionality $\mathcal{F}_{\mathrm{aPAKE}}$ in the UC framework, and presented a generic compiler from $\mathcal{F}_{\mathrm{PAKE}}$ to $\mathcal{F}_{\mathrm{aPAKE}}$.

The notion of Strong Asymmetric PAKE $\mathcal{F}_{\text{saPAKE}}$ by Jarecki, Krawczyk, and Xu [19] addresses an issue with the original $\mathcal{F}_{\text{aPAKE}}$, that allowed a pre-computation attack: password guesses could have been submitted before a server compromise. Most of the computational work could have been done prior to the actual compromise of the password file, allowing "instantaneous" password recovery upon compromise. For example, the attacker can pre-compute the hash value for all passwords in a given dictionary in advance. When a server is compromised at a later point, the adversary can find the pre-image for the compromised hash value, retrieving the password immediately.

Thus, prior to this work, there was no protection against compromise in the symmetric setting for any party, and no protection for clients in the asymmetric setting.

3 Methods and limitations for compromise resilience

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

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

We assume 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 [28], 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, which requires more intricate mechanisms than with the public salt, since it is no longer possible to send the salt in the clear. This approach prevents any pre-computation, and yields a level of

protection that is offered by saPAKE for the servers in the asymmetric setting, and by our novel siPAKE protocol for all parties in any setting. The only remaining attack left for the adversary is a brute-force post-compromise attack, which is inevitable, as we show below.

Black box brute-force attack after compromise

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

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

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

4 Ideal Functionalities

In this section, we first introduce some notational convention and recall the symmetric PAKE functionality. We then introduce our ideal functionalities for (strong) identity-binding PAKEs, and finally our modelling of the random oracle model and the generic group model.

Notation and conventions Our notational conventions inherit from the PAKE and UC settings.

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

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

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

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

Symmetric PAKE Functionality In Figure 1 we restate the symmetric PAKE functionality $\mathcal{F}_{\mathrm{PAKE}}$ from [10] (denoted $\mathcal{F}_{\mathrm{pwKE}}$ there). In our presentation of $\mathcal{F}_{\mathrm{PAKE}}$, we explicitly record keys handed to parties using $\langle\textsc{key},\dots\rangle$ records, which we will later use in our protocol proofs.

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

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

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

Upon $(\textsc{NewKey}, sid, \mathcal{P}_i, K')$ **from** $\mathcal{S}$**:**
- Retrieve $\langle\textsc{session}, \mathcal{P}_i, \mathcal{P}_j, \pi_i\rangle$ 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 $\langle\textsc{key}, \mathcal{P}_j, \pi_j, K_j\rangle$ with $\pi_i = \pi_j$: $K_i \leftarrow K_j$
- otherwise: pick $K_i \xleftarrow{\mathrm{R}} \{0,1\}^\kappa$
- If the session is marked FRESH: record $\langle\textsc{key}, \mathcal{P}_i, \pi_i, K_i\rangle$
- Mark the session COMPLETED and send $\langle sid, K_i\rangle$ to $\mathcal{P}_i$

Fig. 1: Symmetric PAKE functionality $\mathcal{F}_{\mathrm{PAKE}}$ from [10] with minor presentational modifications to simplify comparison.

4.1 (Strong) Identity-binding PAKE Functionality

In Figure 2 we present the Identity-binding PAKE functionality $\mathcal{F}_{\mathrm{iPAKE}}$ and the Strong Identity-binding 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 [13] and Strong Asymmetric PAKE functionality $\mathcal{F}_{\mathrm{saPAKE}}$ of [19] (found in Appendix D).

Compared to the asymmetric functionalities, our main addition is the notion of abstract 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 any other party or perform a MiTM attack. Having the functionality inform a party of its peer identity prevents such attacks.

For symmetry, we restored the notation of parties as $\{\mathcal{P}_i\}_{i=1}^n$: All parties invoke StorePwdFile before starting a session and all use the password file instead of providing a password when starting a session; UsrSession query was eliminated, and SvrSession was renamed NewSession as in

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

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

Upon $(\text{STEALPWDFILE}, sid, \mathcal{P}_i)$ **from** $\mathcal{S}$**:**
- If there is a record $\langle \text{FILE}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$:
  - ▷ $\pi \leftarrow \begin{cases} \pi_i & \text{if there is a record } \langle \text{OFFLINE}, \mathcal{P}_i, \pi' \rangle \text{ with } \pi'{=}\pi_i \\ \perp & \text{otherwise} \end{cases}$
  - ▷ 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 $(\text{OFFLINETESTPWD}, sid, \mathcal{P}_i, \pi')$ **from** $\mathcal{S}$**:**
- Retrieve $\langle \text{FILE}, \mathcal{P}_i, \mathsf{id}_i, \pi_i \rangle$
- If it is marked COMPROMISED:
  - ▷ return "correct guess" to $\mathcal{S}$ if $\pi_i = \pi'$, and "wrong guess" otherwise
- otherwise: Record $\langle \text{OFFLINE}, \mathcal{P}_i, \pi' \rangle$

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

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

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

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

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

Fig. 2: Functionality $\mathcal{F}_{\text{iPAKE}}$ is defined by the full text including grey text), and $\mathcal{F}_{\text{siPAKE}}$ is defined by the text excluding grey text.

$\mathcal{F}_{\text{PAKE}}$. We also parametrized queries on $\mathcal{P}_i$ and $\mathcal{P}_j$ where $\mathcal{F}_{\text{aPAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$ omitted them, since in the symmetric setting those queries may be applied to several parties, e.g., StealPwdFile applying to any party. On the other hand, we omit $\mathcal{P}_j$ from StorePwdFile; in our setting a password file is derived for each party independently, and is not bound to specific peers.

Our functionalities introduce a new query OfflineComparePwd, allowing the adversary to test whether two stolen password files correspond to the same password. In the real world, such attack is always possible by an adversary simulating the protocol for those parties, and comparing the resulting keys. We argue that in most real-world settings, all parties of the same session use the same password (e.g., devices connecting to the same Wi-Fi network), and hence such a query is both inevitable and non-beneficial for the adversary.

Notice the four types of records used by the functionalities:

1. $\langle \textbf{FILE}, \boldsymbol{\mathcal{P}_i}, \mathsf{id}_i, \boldsymbol{\pi_i} \rangle$ records represent password files created for each party $\mathcal{P}_i$, and are derived from its password 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. $\langle \textbf{SESSION}, \boldsymbol{ssid}, \boldsymbol{\mathcal{P}_i}, \boldsymbol{\mathcal{P}_j}, \mathsf{id}_i, \boldsymbol{\pi_i} \rangle$ records represent party $\mathcal{P}_i$'s view of a sub-session with identifier $ssid$ between $\mathcal{P}_i$ and $\mathcal{P}_j$. Similar type of records exist in $\mathcal{F}_{\text{aPAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$, without identities.
3. $\langle \textbf{KEY}, \boldsymbol{ssid}, \boldsymbol{\mathcal{P}_i}, \boldsymbol{\pi_i}, \boldsymbol{K_i} \rangle$ records represent sub-session keys K_i created for party $\mathcal{P}_i$ participating in sub-session $ssid$ with password π_i, and whose session was not compromised or interrupted. These records were also implicitly created in $\mathcal{F}_{\text{PAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$, but appear here explicitly for the sake of clarity.
4. $\langle \textbf{IMP}, \boldsymbol{ssid}, \boldsymbol{\mathcal{P}_i}, \mathsf{id}' \rangle$ 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, $\mathcal{F}_{\text{iPAKE}}$ 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.

To simplify our UC simulator, we additionally allow both OnlineTestPwd and Impersonate queries against the same session, as long as they succeed[1]. 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.

[1] 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.

The $\mathcal{F}_{\text{iPAKE}}$ functionality is weaker than $\mathcal{F}_{\text{siPAKE}}$ in the sense that it permits pre-computation of OfflineTestPwd queries prior to party compromise. It is therefore only of interest when permitting more efficient constructions than its Strong counterpart. Indeed, we present the more efficient CHIP protocol (Section 5) realizing $\mathcal{F}_{\text{iPAKE}}$ in ROM using any cyclic group, while CRISP (Section 6) requires bilinear groups for realizing $\mathcal{F}_{\text{siPAKE}}$ in GGM.

Comparison to (s)aPAKE The symmetric functionalities $\mathcal{F}_{\text{iPAKE}}$ and $\mathcal{F}_{\text{siPAKE}}$ are strictly stronger than their asymmetric counterparts: given a $\mathcal{F}_{\text{iPAKE}}$ (respectively, $\mathcal{F}_{\text{siPAKE}}$) functionality, it is trivial to realize the $\mathcal{F}_{\text{aPAKE}}$ (respectively, $\mathcal{F}_{\text{saPAKE}}$) functionality. The user party U simply computes its password file on each session, when receiving UsrSession query from the environment. Nevertheless, we are not aware of any direct extension of $\mathcal{F}_{\text{aPAKE}}/\mathcal{F}_{\text{saPAKE}}$ to $\mathcal{F}_{\text{iPAKE}}/\mathcal{F}_{\text{siPAKE}}$.

4.2 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 [13], [19] and [7]). Hesse [18] proved UC-realization of aPAKE 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 ROM in UC by allowing parties in the real world to access an ideal functionality $\mathcal{F}_{\text{RO}}$, depicted in Figure 3. Invocations of hash functions in the protocol are modelled as queries to $\mathcal{F}_{\text{RO}}$. The functionality acts as an oracle, answering fresh queries with independent random values, but consistent results to repeated queries. The model is *programmable*, meaning that the simulator is able to view hash queries and program their results. The model is also *local*, meaning that every session has a separate independent $\mathcal{F}_{\text{RO}}$ machine. However, every Hash query is parametrized by a unique *sid*, effectively separating the hash domain. Consequently, a single global random oracle in the real world suffices to handle queries from multiple sessions.

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

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

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

The Generic Group Model (GGM), introduced by [27], allows proving properties of algorithms, assuming the only permitted operations on group elements are the group operation and comparison. Hence a "generic group element" has no meaningful representation. Algorithms in GGM operate on encodings of elements, and may consult a group oracle which computes the group operation for two valid encodings, returning the encoded result. The group oracle declines queries for encodings not returned by some previous query.

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

In order to prove CRISP's security under Universal Composition, we need to formalize GGM in terms of an ideal functionality $\mathcal{F}_{\mathrm{GG}}$. Figure 4 shows the basic GGM functionality $\mathcal{F}_{\mathrm{GG}}$, which answers group operation queries (multiply/divide) on encoded elements. As with $\mathcal{F}_{\mathrm{RO}}$, functionality $\mathcal{F}_{\mathrm{GG}}$ is both programmable and local. Unlike ROM, where local independent oracles can be created from a single global one, the same is not trivial with generic groups. Section 6.6 deals with group reuse across instances of CRISP.

For simplicity one can think of the set of encoding $\mathbb{E}{=}\mathbb{Z}_q$, so each exponent $x{\in}\mathbb{Z}_q$ is encoded as $[x]_{\mathbb{G}}{=}\xi$ 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, $\mathcal{F}_{\mathrm{GG}}$ maps at most one new element per query. Also note the mapping is injective.

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

Upon $\left(\textsc{MulDiv}, sid, [x_1]_{\mathbb{G}}, [x_2]_{\mathbb{G}}, s_{\in\{1,-1\}}\right)$ **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 $\mathcal{F}_{\mathrm{GG}}$

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 $\mathcal{F}_{\mathrm{GGP}}$, depicted in Figure 5, similar to the extension of GGM to bilinear groups by [5]. $\mathcal{F}_{\mathrm{GGP}}$ can be queried $\textsc{MulDiv}$ for each of $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$, and maintains separate encoding maps for each group. It introduces three new queries: (a) $\textsc{Pairing}$ 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) $\textsc{Isomorphism}$ 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) $\textsc{Hash}$ which is a random oracle into $\mathbb{G}_1$: for each freshly queried string $s{\in}\{0,1\}^\star$ it picks a random exponent $x \overset{R}{\leftarrow} \mathbb{Z}_q^\star$, 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 $\textsc{Isomorphism}$ queries by the adversary to guarantee security even when such isomorphism exists.

5 The CHIP iPAKE protocol

When extending the protection of traditional PAKE to consider party compromise attacks, one might think of a trivial solution: simply store the hash of the password, and use this hash value in

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

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

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

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

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

the PAKE, instead of the plain password. While this solves the problem of leaking the password upon party compromise, it does not protect from impersonation. Since hash values are not bound to any identity, a hash value stolen from a compromised party $\mathcal{P}_i$ can be used to impersonate any non-compromised party $\mathcal{P}_j$ towards anyone. This is known as a Key Compromise Impersonation (KCI) attack.

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

One family of protocols providing KCI resistance by design is Identity-Based Key-Exchange (IB-KE) that were first introduced by Günther [14]. Unfortunately, IB-KE protocols require a trusted third party called Key Distribution Centre (KDC). The KDC is responsible for delivering identity-bound key material to other parties in a setup phase. In our setting, there is no trusted third party, only a password that is shared between the parties. To remove the requirement for a KDC, we modify the IB-KE protocol by allowing each party to *locally simulate the operation of the KDC*. To achieve this, we use the password hash as the KDC's secret data. This ensures that all parties with the same password are simulating "the same" KDC, i.e., using the same KDC secrets to derive password files.

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

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

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

Password File Generation:

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

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

Fig. 6: CHIP protocol

password, a passive eavesdropper might capture such a message then start an offline brute-force attack to find the correct password.

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

A final issue with the construction is that the output key of IB-KA depends on the KDC secret. Since IB-KA only provides wFS (as opposed to Perfect Forward Secrecy, PFS), an active adversary can modify the incoming flow to party $\mathcal{P}_i$, then offline derive the resulting key from every possible password guess π'. Any subsequent usage of the key, e.g. for data authentication, would allow the adversary to test the password guesses and extract the correct session key. We resolve this by using the IB-KA output key as input to a symmetric PAKE, along with the transcript of IB-KA.

Figure 6 depicts CHIP, which transforms any PAKE into an iPAKE using the modified IB-KA protocol [12], with the following changes:

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

5.1 Correctness

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

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

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

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

5.2 CHIP realizes $\mathcal{F}_{\text{iPAKE}}$

The following theorem states the security of CHIP as an iPAKE protocol in the UC framework.

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

We give the full proof in Appendix A, and we provide variant of CHIP with explicit key verification in Appendix B. Because H_1 corresponds to OfflineTestPwd, it is advised to choose a computationally costly hash. We return to this issue in Section 7.1.

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

6 The CRISP siPAKE protocol

6.1 Protocol Description

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

1. **Public Parameters Generation:** In this phase, public parameters common to all parties are generated from a security parameter κ. These parameters include the bilinear groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ with hash to group functions $\hat{H}_1$, $\hat{H}_2$, and the PAKE protocol to be used.
2. **Password File Derivation:** In this phase, the user enters a password 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.

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 $\hat{H}_1, \hat{H}_2 \colon \{0,1\}^\star \to \mathbb{G}_1$ and κ a security parameter. Note that here sid is explicitly concatenated to the input of $\hat{H}_1, \hat{H}_2$ invocations for domain separation.

Password File Derivation (offline)

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

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

$A_i \leftarrow g_2^{x_i}$

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

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

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

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

$A_j \leftarrow g_2^{x_j}$

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

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

Key Exchange

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$sid, ssid, S_i \to \mathcal{F}_{\textsc{pake}} \leftarrow sid, ssid, S_j$

$\mathcal{F}_{\textsc{pake}} \to K_i$, $\mathcal{F}_{\textsc{pake}} \to K_j$

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

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

Fig. 7: CRISP protocol

3. **Key Exchange:** In this phase, two parties, $\mathcal{P}_i$ and $\mathcal{P}_j$ engage in a sub-session to derive a shared key. This phase consists of three stages:
   (a) *Blinding.* Values from the password file are raised to the power of a randomly selected exponent. This stage can be performed once and re-used across sub-sessions (see Section 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.
   (c) *PAKE.* Both parties engage in a PAKE where they input their secret values as passwords to receive secure cryptographic keys.

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

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 $\mathcal{F}_{\mathrm{PAKE}}$ to get K_i, K_j. Assuming $\hat{H}_1(sid, \cdot)$ is injective on the password domain we get:

$$S_i = \hat{e}(\tilde{B}_i, \tilde{A}_j) = \hat{e}(\hat{H}_1(sid, \pi_i)^{x_i r_i}, g_2^{x_j r_j}) = \hat{e}(\hat{H}_1(sid, \pi_i), g_2)^{x_i r_i \cdot x_j r_j}$$

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

6.3 Intuition

We provide intuition by explaining the necessity of several components.

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

In search of a construct that is both resilient to pre-computation and has some algebraic structure we considered $\langle X, Y\rangle$ for $X{=}g_2^x$, $Y{=}g_1^{H(\pi)\cdot x}$ and random x. This utilizes the oracle hashing scheme [8] $\langle X, X^{H(v)}\rangle$, which implies pre-computation resistance. The parties can then compute a shared value using bilinear pairing:

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

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

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

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

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

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

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

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

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 $\tilde{A}'_j{=}g_2^{a'_j}$, $\tilde{C}'_j{=}\hat{H}_2(sid, \mathsf{id}_j)^{a'_j}$ 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 $S' =$

$\hat{e}(\hat{H}_1(sid, \pi')^{x_j}, \tilde{A}_i)$. 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.

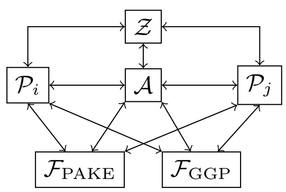
Generic group model. As discussed in Section 4.2 we require a non-black-box assumption to prove pre-computation resilience, and "count" the number of operations required for an offline brute-force attack. Similarly to [7], we use GGM to bind each offline guess to a group operation. In our case, we bind it to the computationally expensive operation of pairing. This is explained in more detail in Section 6.5.

6.4 CRISP realizes $\mathcal{F}_{\text{siPAKE}}$

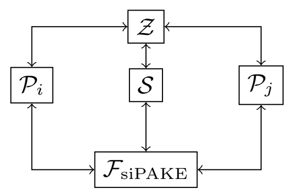
Theorem 2. *Protocol CRISP as depicted in Figure 7 UC-realizes $\mathcal{F}_{\text{siPAKE}}$ in the ($\mathcal{F}_{\text{PAKE}}$,$\mathcal{F}_{\text{GGP}}$)-hybrid world.*

We give the full proof in Appendix C and describe the high-level strategy below. Note that in the UC proof, we omit sid from $\hat{H}_1$ and $\hat{H}_2$ for the sake of brevity.

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



(a) real world



(b) simulated world

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

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

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 $\hat{H}_2(\mathsf{id}) = g_1^{\iota_{\mathsf{id}}}$.

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

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

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

Upon $(\textsc{TestPwd}, sid||ssid, \mathcal{P}_i, [F]_{\mathbb{G}_T})$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{PAKE}}$**:**
- Retrieve $\langle\textsc{sent}, ssid, \mathcal{P}_j, \mathcal{P}_i, id', a', c'\rangle$
- For each $\langle\textsc{compromised}, \mathcal{P}_k, \mathsf{id}_k\rangle$ with $\mathsf{id}_k{=}\mathsf{id}'$:
  - ▷ If $\mathtt{Z}_k$ appears in F:
    - ◇ Send $(\textsc{Impersonate}, sid, ssid, \mathcal{P}_i, \mathcal{P}_k)$ to $\mathcal{F}_{\text{siPAKE}}$
    - ◇ If $\mathcal{F}_{\text{siPAKE}}$ returned "correct guess": replace all $\mathtt{Z}_k$ with $\mathtt{Z}_i$ in F
- For each password π' queried by $\hat{H}_1(\pi')$:
  - ▷ If $\mathtt{Y}_{\pi'}$ appears in F:
    - ◇ Send $(\textsc{OnlineTestPwd}, sid, ssid, \mathcal{P}_i, \pi')$ 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 $(\textsc{OnlineTestPwd}, sid, ssid, \mathcal{P}_i, \perp)$ to $\mathcal{F}_{\text{siPAKE}}$
  - ▷ Return "wrong guess" to $\mathcal{Z}$

Upon $(\textsc{NewKey}, sid||ssid, \mathcal{P}_i, K')$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{PAKE}}$**:**
- Retrieve $\langle\textsc{sent}, ssid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{id}_i', a_i', c_i'\rangle$ and $\langle\textsc{sent}, ssid, \mathcal{P}_j, \mathcal{P}_i, \mathsf{id}_j', a_j', c_j'\rangle$
- 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 $(\textsc{OnlineTestPwd}, sid, ssid, \mathcal{P}_i, \perp)$ to $\mathcal{F}_{\text{siPAKE}}$
- Send $\big(\textsc{NewKey}, sid, ssid, \mathcal{P}_i, \mathsf{id}_j', K'\big)$ to $\mathcal{F}_{\text{siPAKE}}$

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

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.

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

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

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

Upon (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 $\mathcal{F}_{\text{GGP}}$

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:** $\langle \mathsf{id}_i, [\mathtt{X}_i]_{\mathbb{G}_2}, [\mathtt{X}_i\mathtt{Z}_i]_{\mathbb{G}_1}, [\mathtt{X}_i\mathtt{I}_{\mathsf{id}_i}]_{\mathbb{G}_1}\rangle$.
- **First message from $\mathcal{P}_i$:** $(\mathsf{id}_i, [\mathtt{X}_i\mathtt{R}_{i,ssid}]_{\mathbb{G}_2}, [\mathtt{X}_i\mathtt{R}_{i,ssid}\mathtt{I}_{\mathsf{id}_i}]_{\mathbb{G}_1})$.

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 $\hat{e}(\hat{H}_1(\pi'), A_i) = \hat{e}(\hat{H}_1(\pi'), g_2^{x_i})$. With overwhelming probability, these encodings will be equal if and only if $\mathcal{Z}$ chose $\pi_i{=}\pi'$, since collisions in $\hat{H}_1$ only occur with negligible probability. Yet because of using the alias $\mathtt{Z}_i$, $\mathcal{S}$ would generate $\hat{e}(B_i, g_2) = \hat{e}([\mathtt{X}_i\mathtt{Z}_i], [1]_{\mathbb{G}_2}) = [\mathtt{X}_i\mathtt{Z}_i]_{\mathbb{G}_T}$ 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 OFFLINETESTPWD queries, for a compromised party $\mathcal{P}_i$. When $\mathcal{F}_{\text{siPAKE}}$ replies "correct guess" to such query, $\mathcal{S}$ substitutes $\mathtt{Y}_{\pi'}$ for $\mathtt{Z}_i$ in all its data sets.

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

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

```
function InsertRow(v)
    for all row w with pivot column j in M do
        v ← v − v[j]·w
    j ← SelectPivot(v)
    if v = 0⃗ then return
    v ← v/v[j]
    for all row w in M do
        w ← w − w[j]·v
    Insert row v with pivot column j to M

function SelectPivot(v)
    sent ← false
    for all compromised party P_i with identifier id_i do
        for all passwords π′ that were queried by Ĥ_1(π′) do
            j_1 ← index of monomial X_i Y_π′
            j_2 ← index of monomial X_i Y_π′ I_id_i
            if v[j_1]≠0 or v[j_2]≠0 then
                Send (OfflineTestPwd, sid, P_i, π′) to F_siPAKE
                sent ← true
                if F_siPAKE returned "wrong guess" then
                    return { j_1   if v[j_1]≠0
                           { j_2   otherwise
                Substitute variable Z_i with Y_π′ in all polynomials
                Merge corresponding columns of M, v
    if some party P_i has been compromised and sent=false then
        Send (OfflineTestPwd, sid, P_i, ⊥) to F_siPAKE
    if v ≠ 0⃗ then return arbitrary column j having v[j] ≠ 0
```

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

$\mathtt{X}_i$, $\mathtt{X}_i\mathtt{Z}_i$ 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', where password guesses are taken from $\mathcal{A}$'s $\hat{H}_1(\pi')$ queries. If such non-zero entry exists in V, $\mathcal{S}$ sends OfflineTestPwd query to $\mathcal{F}_{\mathrm{siPAKE}}$ 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 merging some variables.

This completes the proof sketch; for further details we refer to Appendix C.

6.5 Cost of off-line brute-force attack

We now show that the cost of an off-line brute-force attack is at least one pairing per guess. 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, [19] requires OPRF query for each tested password, while [7] shows linear relation between number of offline tests and Generic Group operations.

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.

Note 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

As explained in Section 4.2, CRISP is proved in *local* generic group model. Locality of GGM implies that any instance of CRISP requires a dedicated independent group in the real world. Therefore, the proof does not hold when two protocol instances share the same group. While a similar requirement for CHIP's ROM is achieved with domain separation (prepending *sid* to any hash input), it is unclear how to achieve this for groups.

Instead, we propose to modify the functionality $\mathcal{F}_{\text{siPAKE}}$ to explicitly describe multiple instances with the same group. A similar approach was used in [13] to support multiple aPAKE sub-sessions under a single server setup. Likewise, we suggest a higher-level of global session that determines a shared group, identified by *gid*. Under this global session many CRISP instances may run. Each instance refers to a different network and identified by unique *sid*. Some parties may be associated (by invoking StorePwdFile) with the same *sid*, marking them members of the same network. We will forbid queries involving parties from different networks. Finally, a third level of sub-sessions will correspond to online key-exchanges performed by parties, and identified by *ssid*.

We do not provide the modified proof and functionalities, for the sake of readability. Mostly, a *gid* argument needs to be added to queries, and many data structures should be indexed by *sid*. The key change is that $\mathcal{F}_{\text{GGP}}$'s HASH query should include an *sid* parameter. As opposed to other group operations, *hash-to-group can enjoy domain separation.* This allows us to create separate $\mathtt{Y}_\pi$ variables for each network based on *sid*, to prevent variable aliasing across sessions.

This approach allows us to extend the use of a single generic group for multiple CRISP instances. It does not provide composition with other protocols using the same group. As a result, protocols composed with CRISP (either higher-level or the underlying symmetric PAKE) should not share its group. Note that CRISP uses a bilinear group, which would not be selected for most other protocols regardless of the above recommendation.

6.7 Primum Non Nocere — Unconditional PAKE Security

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

To show this, we are only concerned with the additional actions added before invoking the PAKE. Recall that the message added by CRISP for party $\mathcal{P}_i$ is:

$$\mathsf{id}_i, \tilde{A}_i, \tilde{C}_i \;=\; \mathsf{id}_i, (g_2^{x_i})^{r_i}, (\hat{H}_2(sid, \mathsf{id}_i)^{x_i})^{r_i},$$

where r_i and x_i are random values. This message is thus completely independent of the password and does not leak any information about it. Also, we recall from Section 6.2 that the inputs to $\mathcal{F}_{\text{PAKE}}$ S_i, S_j are equal if and only if the passwords are equal (only assuming $\hat{H}_1$ is injective on the password domain). Thus, unless a party is compromised, the underlying PAKE properties (leaking no information of the password and allowing a single online guess) are preserved by CRISP.

7 Computational Cost

The computational costs for CHIP and CRISP are summarized in Table 2 in terms of costly operations. In the table, we use 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.

		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	$1H + 3E + \text{PAKE}$	$1P + \text{PAKE}$

Table 2: Comparison of costly operations in CRISP and CHIP

7.1 Password Hardening for Pre-Compromise

Common password hardening techniques (e.g., PBKDF2 [23], Argon2 [4], and scrypt [25]) 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:

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

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 [11]. Moreover, pairing requires more memory than a simple point multiplication and is harder to accelerate using GPUs [26].

In OPAQUE [19], the difficulty of offline tests was increased by iterative hashing (password hardening). CRISP cannot benefit from this approach, because the design does not allow the salt inside the hash. However, by using larger group sizes, we can increase the cost of each pairing and slow down offline tests. Although coarse-grained, this allows some trade-off between compromise resilience and computational complexity of CRISP.

7.3 CRISP Optimization

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

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

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

Number of Messages CRISP requires two additional messages compared to the underlying PAKE. We can trivially reduce this to one additional message. The first message remains the same. 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 [21], we can implement CRISP using only three sequential messages. The same optimization also applies to CHIP.

7.4 Performance Benchmark

We provide open source implementations for CHIP and CRISP. In both we rely on CPace [15] as the underlying symmetric PAKE. CHIP was implemented on top of Ristretto255 curve from the libsodium library (v1.0.18). CRISP uses the pairing friendly curve BLS12-381 from the MCL library (v1.22). Both curves are assumed to provide 128-bit of security strength. The source code is available at `https://github.com/shapaz/CRISP`.

In Table 3 we compare the online performance of CHIP and CRISP with those of other popular PAKE protocols, running on an i7-4790 processor. CPace and OPAQUE [19] were chosen by IETF CFRG as symmetric and asymmetric PAKEs (respectively) for usage with TLS 1.3, and are considered to be very efficient. SAE [16] is the underlying symmetric PAKE of Wi-Fi's WPA-3 and is designed to be supported by low-resource embedded devices. For measurements, our code implements both CPace and OPAQUE over Ristretto255. For SAE we used the official hostapd/wpa_supplicant. Note that although Wi-Fi's SAE was designed to be a PAKE, its security was never proven.

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

Table 3: Online performance comparison and proven security notions for PAKEs.

8 Conclusions and discussion

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

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

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

Two message protocol. In Section 7.3, we showed how our protocols require only three messages. As shown in [21], PAKE can be realized with only two messages. It is an open problem to either prove a lower bound of three messages or to implement a two message iPAKE or siPAKE protocol. To the best of our knowledge, there are no two message (s)aPAKE protocols. Jutla and Roy [20] propose a one-round aPAKE, but it seems that they require an additional message from the server before the protocol [19].

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

Acknowledgments

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

Bibliography

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

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

[3] 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.

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

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

[6] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *EUROCRYPT*, 2000.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. draft-irtf-cfrg-opaque-03: The OPAQUE Asymmetric PAKE Protocol. Internet-draft, Internet Engineering Task Force, February 2021. https://tools.ietf.org/html/draft-irtf-cfrg-opaque-03 (work in progress).

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

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

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

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

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

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

A Proof of Theorem 1: CHIP UC-realizes $\mathcal{F}_{\text{iPAKE}}$

Before proceeding with the proof for CHIP, we recall 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 [12] 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.

Note that in the UC proof, we omit *sid* from H_1 and H_2 for the sake of brevity.

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 $\mathcal{F}_{\text{RO}}$ 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.

Initially, pick $x_i \xleftarrow{R} \mathbb{Z}_q^\star$ 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 $H_1[\pi'] \xleftarrow{R} \mathbb{Z}_q^\star$, $H_2[\mathsf{id}, X] \xleftarrow{R} \mathbb{Z}_q^\star$.

Upon $(\text{StealPwdFile}, sid)$ **from** $\mathcal{Z}$ **towards** $\mathcal{P}_i$**:**
- Send $(\text{StealPwdFile}, sid, \mathcal{P}_i)$ 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 $\pi_i \neq \perp$: set $y_i \leftarrow H_1[\pi_i]$
- Otherwise, pick $y_i \xleftarrow{R} \mathbb{Z}_q^\star$
- For each $\langle \text{compromised}, \mathcal{P}_k, \cdot, y_k\rangle$:
  - ▷ Send $(\text{OfflineComparePwd}, sid, \mathcal{P}_i, \mathcal{P}_k)$ to $\mathcal{F}_{\text{iPAKE}}$
  - ▷ If $\mathcal{F}_{\text{iPAKE}}$ returned "passwords match": set $y_i \leftarrow y_k$
- Record $\langle \text{compromised}, \mathcal{P}_i, \mathsf{id}_i, y_i\rangle$
- $X_i \leftarrow g^{x_i}$, $\quad Y_i \leftarrow g^{y_i}$
- $h_i \leftarrow H_2[\mathsf{id}_i, X_i]$
- $\hat{x}_i \leftarrow x_i + h_i \cdot y_i$
- Return $\langle \text{file}, \mathsf{id}_i, X_i, Y_i, \hat{x}_i\rangle$ to $\mathcal{Z}$

Upon (Hash, sid, s) **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{RO}}$**:**
- If $s = \langle 1, \pi'\rangle$: for each party $\mathcal{P}_i$:
  - ▷ Send $(\text{OfflineTestPwd}, sid, ssid, \mathcal{P}_i, \pi')$ to $\mathcal{F}_{\text{iPAKE}}$
  - ▷ If $\mathcal{F}_{\text{iPAKE}}$ replied "correct guess":
    - ◇ Retrieve $\langle \text{compromised}, \mathcal{P}_i, \cdot, y_i\rangle$
    - ◇ 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

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

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 $\langle \text{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, \cdot \rangle$
- Pick $r_i \xleftarrow{\text{R}} \mathbb{Z}_q^\star$
- Record $\langle \text{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, r_i \rangle$ and mark it FRESH
- $X_i \leftarrow g^{x_i}, \quad R_i \leftarrow g^{r_i}$
- $f_i \leftarrow (\mathsf{id}_i, X_i, R_i)$
- 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 $(\mathsf{id}_i', X_i', R_i')$
- Record $\langle \text{SENT}, ssid, \mathcal{P}_i, (\mathsf{id}_i, X_i, R_i) \rangle$ and $\langle \text{RECV}, ssid, \mathcal{P}_j, (\mathsf{id}_i', X_i', R_i') \rangle$

Upon $(\textsc{TestPwd}, sid, ssid, \mathcal{P}_i, \alpha', \beta', \mathsf{tr}')$ **from** $\mathcal{Z}$ **towards** $\mathcal{F}_{\text{PAKE}}$**:**
- Retrieve $\langle \text{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, r_i \rangle$ marked FRESH
- Retrieve $\langle \text{SEND}, ssid, \mathcal{P}_i, f_i \rangle$ and $\langle \text{RECV}, ssid, \mathcal{P}_i, f_j' = (\mathsf{id}_j', X_j', R_j') \rangle$
- 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 $\langle \text{COMPROMISED}, \mathcal{P}_k, \mathsf{id}_k, y_k \rangle$ 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 $\langle \text{SESSION}, ssid, \mathcal{P}_i, \mathcal{P}_j, r_i \rangle$ not marked COMPLETED
- Retrieve $\langle \text{SENT}, ssid, \mathcal{P}_j, f_j \rangle$ and $\langle \text{RECV}, ssid, \mathcal{P}_i, f_j' = (\mathsf{id}_j', \cdot, \cdot) \rangle$
- 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 $\left(\textsc{NewKey}, sid, ssid, \mathcal{P}_i, K', \mathsf{id}_j'\right)$ to to $\mathcal{F}_{\text{iPAKE}}$

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

Proof (Theorem 1).

Let $\mathcal{A}$ be the dummy adversary running in the $(\mathcal{F}_{\text{PAKE}}, \mathcal{F}_{\text{RO}})$-hybrid world (will be referred to as "ideal world" from now on). Consider the simulator $\mathcal{S}$ depicted in Figure 12 and Figure 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 appearing in the message flow comes from $\mathcal{F}_{\text{iPAKE}}$ 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's flow f_i. 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 $\mathcal{F}_{\text{iPAKE}}$ for $\mathcal{P}_i$ with password π_i. This query succeeded, causing $\mathcal{F}_{\text{iPAKE}}$ to create an $\langle \text{OFFLINE}, \dots \rangle$ record, ensuring that the correct π_i is returned from $\mathcal{F}_{\text{iPAKE}}$ in response to STEALPWDFILE query. $\mathcal{S}$ can therefore generate y_i using $H_1[\pi_i]$ exactly like a real 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 $\langle$compromised, …$\rangle$ 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 in the message flows, 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 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 the 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): $y_i, p_i \leftarrow H_1(sid, \pi_i)$. 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 Figure 14. Note that it combines the transcripts of both IB-KA and the underlying PAKE[2]. 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 [13].

C Proof of Theorem 2: CRISP UC-realizes $\mathcal{F}_{\text{siPAKE}}$

We first introduce a helper lemma to modularize the proof. The main proof considers the case of aliasing collisions in TestPwd; the following lemma excludes all other collisions.

Lemma 1. *Except with negligible probability, there are no collisions in the simulation outside of aliasing collisions in* TestPwd.

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 [9], 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.

$$\begin{array}{ccc} \xrightarrow{f_i = (\mathsf{id}_i, \tilde{A}_i, \tilde{C}_i)} & & \xleftarrow{f_j = (\mathsf{id}_j, \tilde{A}_j, \tilde{C}_j)} \\ & \boxed{\mathcal{A}} & \\ \xleftarrow[f_j' = (\mathsf{id}_j', \tilde{A}_j', \tilde{C}_j')]{} & & \xrightarrow[f_i' = (\mathsf{id}_i', \tilde{A}_i', \tilde{C}_i')]{} \end{array}$$

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

From Table 4 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.[4] 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.

[2] 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.

[3] 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.

[4] In the UC proof, we omit *sid* from $\hat{H}_1$ and $\hat{H}_2$ for the sake of brevity.

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. Note that here sid is explicitly concatenated to the input of H_1, H_2 invocations for domain separation.

Password File Generation:

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

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

$sid, ssid, p_i \;\to\; \mathcal{F}^{+}_{\text{PAKE}} \;\leftarrow\; sid, ssid, p_j$

$\alpha_i, \mathsf{tr}_{i,1} \;\leftarrow\; \mathcal{F}^{+}_{\text{PAKE}} \;\to\; \alpha_j, \mathsf{tr}_{j,1}$

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

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

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

Fig. 14: CHIP variant with key confirmation

TestPwd answer. Although Table 4 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.

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	$S_i = \hat{e}(\tilde{B}_i, \tilde{A}'_j)$ [3]	$[(x_i y_{\pi_i} r_i){\cdot}a'_j]_{\mathbb{G}_T}$	$[(\mathtt{X}_i \mathtt{Z}_i \mathtt{R}_{i,ssid}){\cdot}F'_j]_{\mathbb{G}_T}$

Table 4: Comparison of values viewed by $\mathcal{Z}$ in the real world versus the simulated world.

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 from Page 19 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. A similar argument applies for $\mathtt{Z}_k$ where $\mathcal{P}_k$ has been compromised and having $\mathsf{id}_k{=}\mathsf{id}'$. Since $\mathcal{Z}$ only obtains polynomials with $\mathtt{Z}_k$ by compromising $\mathcal{P}_k$, we are left with the case that $\mathcal{P}_k$ has been compromised, but $\mathsf{id}_k{\neq}\mathsf{id}'$. However, in this case a'_j must contain $\mathtt{X}_k$ and therefore $c'_j{=}a'_j{\cdot}\mathtt{I}_{\mathsf{id}'}$ contains $\mathtt{X}_k{\cdot}\mathtt{I}_{\mathsf{id}'}$, which is a term $\mathcal{Z}$ cannot produce in $\mathbb{G}_1$. Thus, if $\mathcal{S}$ replies "correct guess" then the session becomes compromised in the simulated world, as well as in the real world.

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

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

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

When the session is interrupted, no restriction is placed on the identity selected by $\mathcal{S}$. The same applies when the session is compromised due to a successful OnlineTestPwd query. When an Impersonate query caused the session to become compromised, only the impersonated identity is allowed, and indeed $\mathcal{S}$ verifies that $\mathsf{id}_k{=}\mathsf{id}'$ before impersonating party $\mathcal{P}_k$. When the session is fresh, only the true identity of the peer party is permitted, but $\mathcal{S}$ uses id' as in the real world.

Nevertheless, if $\mathsf{id}'{\neq}\mathsf{id}_j$ and $a'_j = \alpha \mathtt{X}_j \mathtt{R}_{j,ssid}$ ($\alpha{\in}\mathbb{Z}_q^\star$) then the condition

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

could not have been 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 $\pi{=}\perp$ when handling NEWKEY query. This happens when the modified flows f'_i and f'_j are not using $a'_i{=}\alpha_i \mathtt{X}_i \mathtt{R}_{i,ssid}$ and $a'_j{=}\alpha_J \mathtt{X}_j \mathtt{R}_{j,ssid}$ with $\alpha_i{=}\alpha_j$. If the flows have this form with $\alpha_i{\neq}\alpha_j$, then

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

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

We proceed to prove the lemma. We prove Lemma 1 by deferring a specific subcase to Lemma 2.

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:

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

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

Lemma 2. *Except with negligible probability, there are no aliasing collisions in the simulation outside of* TestPwd.

Proof (Proof of Lemma 2). 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 lemma does not consider 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
$\mathtt{X}_i{\cdot}\mathtt{R}_{i,ssid}$ $\mathtt{X}_i{\cdot}\mathtt{R}_{i,ssid}{\cdot}\mathtt{I}_{\mathsf{id}_i}$	message flow 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.

D Asymmetric PAKE Functionality

Figure 15 shows the Strong Asymmetric PAKE functionality from [19], 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.

Functionalities $\mathcal{F}_{\text{aPAKE}}$ and $\mathcal{F}_{\text{saPAKE}}$, with security parameter κ, 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 $K_{\mathcal{P}} \stackrel{\text{R}}{\leftarrow} \{0,1\}^\kappa$
- 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)

E Major differences between versions

Version 1.0, May 2020 Initial upload.

- Introduced iPAKE and siPAKE to protect all parties against compromise.
- Formalized $\mathcal{F}_{\text{iPAKE}}$ and $\mathcal{F}_{\text{siPAKE}}$ as UC functionalities.
- Presented an iPAKE construction from IB-KA.
- Presented CRISP and proved it realizes $\mathcal{F}_{\text{siPAKE}}$ in GGM.

Version 1.1, July 2020 Some improvements:

- Fixed the iPAKE protocol and proved it realizes $\mathcal{F}_{\text{iPAKE}}$ in ROM.
- Added a variant of that protocol which includes explicit key confirmation with the same number of messages.

Version 1.2, October 2020 Additional improvements:

- Reworked introduction and compromise resilience methods.
- Named the iPAKE protocol "CHIP".
- Added prototype implementation and benchmark for CHIP and CRISP.
- Justify group reuse across sessions.

Version 2.0, March 2021 Major reworking of the paper.

- Completely reworked motivation, problem positioning, and contributions. Improved the explanation of the difference to related approaches throughout the paper.
- Added explicit comparisons to aPAKE and saPAKE protocols in terms of security and application scope.
- Provided more intuition for ideal functionalities and protocol design choices in general. Clarified protocol diagram flows with distinction between message flows and functionality I/O, removing potentially confusing flow tags, and made session identifiers and inputs to $\mathcal{F}_{\text{PAKE}}$ explicit.
- For CHIP, substantially expanded on its underlying design choices and construction, and added explicit correctness statement.
- For CRISP, added explanation on which properties require GGM, and which ones are provided unconditionally. Improved accuracy in the phrasing of the lemmas and their role in the proof.
- New benchmarking results that are more representative and include more related protocols; also added explicit expensive operations counts for our protocols.
- Added paragraph on practical deployment in conclusions.