This is the full version of the extended abstract which appears in
Proceedings of the Cryptographers Track at the RSA Conference CT-RSA 11
(February 14–18, 2011, San Francisco, CA, USA)
A. Kiayias Ed. Springer-Verlag, LNCS 6558, pages 142–160.

Contributory Password-Authenticated Group Key Exchange with Join Capability

Michel Abdalla[1], Céline Chevalier[2], Louis Granboulan[3], and David Pointcheval[1]

[1] ENS/CNRS/INRIA, Paris, France
[2] LSV – ENS Cachan/CNRS/INRIA[†], France
[3] EADS, France

Abstract. Password-based authenticated group key exchange allows any group of users in possession of a low-entropy secret key to establish a common session key even in the presence of adversaries. In this paper, we propose a new generic construction of password-authenticated group key exchange protocol from any two-party password-authenticated key exchange with explicit authentication. Our new construction has several advantages when compared to existing solutions. First, our construction only assumes a common reference string and does not rely on any idealized models. Second, our scheme enjoys a simple and intuitive security proof in the universally composable framework and is optimal in the sense that it allows at most one password test per user instance. Third, our scheme also achieves a strong notion of security against insiders in that the adversary cannot bias the distribution of the session key as long as one of the players involved in the protocol is honest. Finally, we show how to easily extend our protocol to the dynamic case in a way that the costs of establishing a common key between two existing groups is significantly smaller than computing a common key from scratch.

1 Introduction

Password-authenticated key exchange (PAKE) allows any two parties in possession of a short (*i.e.*, low-entropy) secret key to establish a common session key even in the presence of an adversary. Since its introduction by Bellovin and Merritt [13], PAKE has become an important cryptographic primitive due to its simplicity and ease of use, which does not rely on expensive public-key infrastructures or high-entropy secret keys.

In the universally composable (UC) framework [17], the authors of [19] show how their new model (based on the ideal functionality $\mathcal{F}_{\mathsf{pwKE}}$) relates to previous PAKE models, such as [11] or [7]. In particular, they show that any protocol that realizes $\mathcal{F}_{\mathsf{pwKE}}$ is also a secure password-authenticated key-exchange protocol in the model of [11]. Other works in the UC framework include [23] and [25], where the authors study static corruptions without random oracles as well.

In this paper, we consider password-authenticated key exchange in the group setting (GPAKE) where the number of users involved in the computation of a common session key can be large. With few exceptions (*e.g.*, [1]), most protocols in this setting are built from scratch and are quite complex. Among these protocols, we can clearly identify two types of protocols: constant-round protocols (*e.g.*, [8, 14, 5]) and those whose number of communication rounds depends on the number of users involved in the protocol execution (*e.g.*, [15]). Since constant-round protocols are generally easier to implement and less susceptible to synchronization problems when the number of user increases, we focus our attention on these protocols. More precisely, we build upon the works of Abdalla, Catalano, Chevalier, and Pointcheval [5] and Abdalla, Bohli, González Vasco, and Steinwandt [1] and propose a new generic compiler which converts any two-party password-authenticated key exchange protocol into a password-authenticated group key exchange protocol. Like [1], our protocol relies on a common reference string (CRS) which seems to be a reasonable assumption when one uses a public software, that is somewhat "trusted". This is also a necessary assumption for realizing PAKE schemes in the UC framework as shown by [19]. Like [5], our protocol achieves a strong

[†] Work done while being at Télécom ParisTech, Paris, France.

notion of contributiveness in the UC framework. In particular, even if it can control all the network communications, the adversary cannot bias the key as long as one of the players involved in the protocol is honest. We indeed assume that all the communications are public, and such a network can be seen as a (non-reliable) broadcast channel, controlled by the adversary: the latter can delay, block, alter and/or replay messages. Players thus do not necessarily all receive the same messages. Since the adversary can block messages, we have to assume timeouts for each round. As a consequence, denial-of-service attacks are possible, but these are out of the scope of this paper.

Contributions. There are three main contributions in this paper. The first one regards the optimality of the security, which only allows one password test per subgroup. As mentioned in [5] and in Barak *et al.* [9], without any strong authentication mechanisms, which is the case in the password-based scenario, the adversary can always partition the players into disjoint subgroups and execute independent sessions of the protocol with each subgroup, playing the role of the other players. As a result, an adversary can always use each one of these partitions to test the passwords used by each subgroup. Since this attack is unavoidable, this is the best security guarantee that we can hope for. In contrast, the protocol in [5] required an additional password test for each user in the group.

The second contribution is the construction itself, which astutely combines several techniques: it applies the Burmester-Desmedt technique [16] to any secure two-party PAKE achieving (mutual) explicit authentication in the UC framework. The key idea used by our protocol is that, in addition to establishing pairwise keys between any pair of users in the ring, each user also chooses an additional random secret value to be used in the session key generation. In order to achieve the contributiveness property, our protocol enforces these random secret values to be chosen independently so that the final session key will be uniformly distributed as long as one of the players is honest. In order to prove our protocol secure in the UC framework, we also make use of a particular randomness extractor, which possesses a type of partial invertibility property which we use in the proof. The proof of security assumes the existence of a common reference string and does not rely on any idealized model. We note that UC-secure authenticated group key exchange protocols with contributiveness were already known [24, 5], but they either relied on idealized models [5] or were not applicable to the password-based scenario [24].

Our final contribution is to show how to extend our protocol to the dynamic case, with forward-secrecy, so that the cost of merging two subgroups is relatively small in comparison to generating a *new and independent* common group key from scratch. This is because given two subgroups, each with its own subgroup key, we only need to execute two instances of the PAKE protocol in order to merge these two groups and generate a new group key. Note that, if one were to compute a common group key from scratch, the number of PAKE executions would be proportional to the number of users in the group. Since the PAKE execution is the most computationally expensive part of the protocol, our new merge protocol significantly improves upon the trivial solution.

2 UC Two-Party PAKE

Notations and Security Model. We denote by k the security parameter. An event is said to be negligible if it happens with probability less than the inverse of any polynomial in k. If X is a finite set, $x \stackrel{R}{\leftarrow} X$ indicates the process of selecting x uniformly and at random in X (we thus implicitly assume that X can be sampled efficiently).

Throughout this paper, we assume basic familiarity with the universal composability framework. The interested reader is referred to [17, 19] for details. The model considered in this paper is the UC framework with joint state proposed by Canetti and Rabin [20] (the CRS will be in the joint state).

Given a functionality $\mathcal{F}$, the split functionality $s\mathcal{F}$ proceeds as follows:
Initialization:

- Upon receiving (Init, sid) from party P_i, send $(\mathsf{Init}, sid, P_i)$ to the adversary.
- Upon receiving a message $(\mathsf{Init}, sid, P_i, G, H, sid_H)$ from $\mathcal{A}$, where $H \subset G$ are sets of party identities, check that P_i has already sent (Init, sid) and that for all recorded $(H', sid_{H'})$, either $H = H'$ and $sid_H = sid_{H'}$ or H and H' are disjoint and $sid_H \neq sid_{H'}$. If so, record the pair (H, sid_H), send $(\mathsf{Init}, sid, sid_H)$ to P_i, and invoke a new functionality $(\mathcal{F}, sid_H)$ denoted as $\mathcal{F}_H$ on the group G and with set of initially honest parties H.

Computation:

- Upon receiving (Input, sid, m) from party P_i, find the set H such that $P_i \in H$ and forward m to $\mathcal{F}_H$.
- Upon receiving $(\mathsf{Input}, sid, P_j, H, m)$ from $\mathcal{A}$, such that $P_j \notin H$, forward m to $\mathcal{F}_H$ as if coming from P_j (it will be ignored if $P_j \notin G$ for the functionality $\mathcal{F}_H$).
- When $\mathcal{F}_H$ generates an output m for party $P_i \in H$, send m to P_i. If the output is for $P_j \notin H$ or for the adversary, send m to the adversary.

Fig. 1. Split Functionality $s\mathcal{F}$

In this paper, we consider adaptive adversaries which are allowed to arbitrarily corrupt players at any moment during the execution of the protocol, thus getting complete access to their internal memory. In a real execution of the protocol, this is modeled by letting the adversary $\mathcal{A}$ obtain the password and the internal state of the corrupted player. Moreover, $\mathcal{A}$ can arbitrarily modify the player's strategy. In an ideal execution of the protocol, the simulator $\mathcal{S}$ gets the corrupted player's password and has to simulate its internal state in a way that remains consistent to what was already provided to the environment.

Split Functionalities. Without any strong authentication mechanisms, the adversary can always partition the players into disjoint subgroups and execute independent sessions of the protocol with each subgroup, playing the role of the other players. Such an attack is unavoidable since players cannot distinguish the case in which they interact with each other from the case where they interact with the adversary. The authors of [9] addressed this issue by proposing a new model based on *split functionalities* which guarantees that this attack is the only one available to the adversary.

The split functionality is a generic construction based upon an ideal functionality. The original definition was for protocols with a fixed set of participants. Since our goal is to deal with dynamic groups, not known in advance, we let the adversary not only split the honest players into subsets H in each execution of the protocol, but also specify the players it will control. The functionality will thus start with the actual list of players in G, where H is the subgroup of the honest players in this execution. Note that H is the subset of the *initially* honest players, which can later get corrupted in case we consider adaptive adversaries. The restriction of the split functionality is to have disjoint sets H, since it models the fact that the adversary splits the honest players in several concurrent but independent executions of the protocol. The new description can be found on Figure 1. In the initialization stage, the adversary adaptively chooses disjoint subsets H of the honest parties (with a unique session identifier that is fixed for the duration of the protocol) together with the lists G of the players for each execution. More precisely, the protocol starts with a session identifier sid. Then, the initialization stage generates some random values which, combined together and with sid, create the new session identifier sid', shared by all parties which have received the same values – that is, the parties of the disjoint subsets. The important point here is that the subsets create a *partition* of the declared honest players, thus forbidding communication among the subsets. During the computation, each subset H activates a separate instance of the functionality $\mathcal{F}$ on the group G. All these functionality instances are independent: The executions of the protocol for each subset H can only be related in the way the adversary chooses the inputs of the players it controls.

The parties $P_i \in H$ provide their own inputs and receive their own outputs (see the first item of "computation" in Figure 1), whereas the adversary plays the role of all the parties $P_j \notin H$, but in G (see the second item).

UC 2-PAKE Protocols. Canetti *et al.* first proposed in [19] the ideal functionality for universally composable two-party password-based key exchange (2-PAKE), along with the first protocol to achieve such a level of security. This protocol is based on the Gennaro-Lindell extension of the KOY protocol [26, 22], and is not known to achieve adaptive security.

Later on, Abdalla *et al.* proposed in [4] an improvement of the ideal functionality, adding client authentication, which provides a guarantee to the server that when it accepts a key, the latter is actually known to the expected client. They also give a protocol realizing this functionality, and secure against adaptive corruptions, in the random oracle model. More recently, they presented another protocol in [6], based on the Gennaro-Lindell protocol, secure against adaptive corruptions in the standard model, but with no explicit authentication.

Mutual Authentication. Our generic compiler from a 2-PAKE to a GPAKE, that we present in Section 4, achieves security against static (*resp.* adaptive) adversaries, depending on the level of security achieved by the underlying 2-PAKE. Furthermore, the 2-PAKE needs to achieve mutual authentication. For the sake of completeness, we give here the modifications of the ideal functionality to capture this property: both client authentication and server authentication. Furthermore, to be compatible with the GPAKE functionality, we use the split functionality model. For the 2-PAKE, this model is equivalent to the use of $\mathsf{TestPwd}$ queries in the functionality. They both allow the adversary to test the password of a player (a dictionary attack) either by explicitly asking a $\mathsf{TestPwd}$ query, or by playing with this player. More precisely, an adversary willing to test the password of a player will play on behalf of its partner, with the trial password: If the execution succeeds, the password is correct. Finally, the 2-PAKE functionality with mutual authentication $\mathcal{F}_{\mathsf{PAKE}}^{MA}$, presented in Figure 2, is very close to the GPAKE functionality, see Section 3. As in the GPAKE one, we added the contributiveness property. Note that the protocols mentioned earlier can realize this functionality given very small modifications.

3 UC Group PAKE

We give here a slightly modified version of the ideal functionality for GPAKE presented in [5], by suppressing the $\mathsf{TestPwd}$ queries, which was left as an open problem in [5], since their protocol could not be proven without them. Our new functionality thus models the optimal security level: the adversary can test only one password per subgroup (split functionality). This is the same improvement as done in another context between [2] and [3]. Furthermore, the players in [5] were assumed to share the same passwords. We consider here a more general scenario where each user P_i owns a pair of passwords $(\mathsf{pw}_i^L, \mathsf{pw}_i^R)$, each one shared with one of his neighbors, P_{i-1} and P_{i+1}, when players are organized around a ring. This is a quite general scenario since it covers the case of a unique common password: for each user, we set $\mathsf{pw}_i^L = \mathsf{pw}_i^R$. The ring structure is also general enough since a centralized case could be converted into a ring, where the center is duplicated between the users. Recall that thanks to the use of the split functionality, the GPAKE functionality invoked knows the group of the players, as well as the order among them. The following description is strongly based on that of [5].

Contributory Protocols. As in [5], we consider a stronger corruption model against insiders than the one proposed by Katz and Shin in [27]: in the latter model, one allows the adversary to choose the session key as soon as there is one corruption; as in the former case, in this paper we consider the notion of contributiveness, which guarantees the distribution of the session keys to be

The functionality $\mathcal{F}_{\mathsf{PAKE}}^{MA}$ is parameterized by a security parameter k, and the parameter $t \in \{1, 2\}$ of the contributiveness. It maintains a list L initially empty of values of the form $((\mathsf{sid}, P_k, P_l, \mathsf{pw}, \mathsf{role}), *)$ and interacts with an adversary $\mathcal{S}$ and dynamically determined parties P_i and P_j via the following queries:

- **Initialization.**
  Upon receiving a query $(\mathsf{NewSession}, \mathsf{sid}, P_i, \mathsf{pw}, \mathsf{role})$ from $P_i \in \mathcal{H}$:
  - Send $(\mathsf{NewSession}, \mathsf{sid}, P_i, \mathsf{role})$ to $\mathcal{S}$.
  - If this is the first NewSession query, or if it is the second NewSession query and there exists a record $((\mathsf{sid}, P_j, P_i, \mathsf{pw}', \overline{\mathsf{role}}), \mathsf{fresh}) \in L$, then record $((\mathsf{sid}, P_i, P_j, \mathsf{pw}, \mathsf{role}), \mathsf{fresh})$ in L. If it is the second NewSession query, record the tuple $(\mathsf{sid}, \mathsf{ready})$.
- **Key Generation.** Upon receiving a message $(\mathsf{sid}, \mathsf{ok}, \mathsf{sk})$ from $\mathcal{S}$ where there exists a recorded tuple $(\mathsf{sid}, \mathsf{ready})$, then, denote by n_c the number of corrupted players, and
  - If P_i and P_j have the same password and $n_c < t$, choose $\mathsf{sk}' \in \{0,1\}^k$ uniformly at random and store $(\mathsf{sid}, \mathsf{sk}')$. Next, mark the records $((\mathsf{sid}, P_i, P_j, \mathsf{pw}_i, \mathsf{role}), *)$ and $((\mathsf{sid}, P_j, P_i, \mathsf{pw}_j, \overline{\mathsf{role}}), *)$ complete.
  - If P_i and P_j have the same passwords and $n_c \geq t$, store $(\mathsf{sid}, \mathsf{sk})$. Next, mark both the records $((\mathsf{sid}, P_i, P_j, \mathsf{pw}_i, \mathsf{role}), *)$ and $((\mathsf{sid}, P_j, P_i, \mathsf{pw}_j, \overline{\mathsf{role}}), *)$ complete.
  - In any other case, store the result $(\mathsf{sid}, \mathsf{error})$ and mark the records $((\mathsf{sid}, P_i, P_j, \mathsf{pw}_i, \mathsf{role}), *)$ and $((\mathsf{sid}, P_j, P_i, \mathsf{pw}_j, \overline{\mathsf{role}}), *)$ error.

  When the key is set, report the result (either error or complete) to $\mathcal{S}$.
- **Key Delivery.** Upon receiving a message $(\mathsf{deliver}, \mathsf{b}, \mathsf{sid}, P_i)$ from $\mathcal{S}$, then if $P_i \in \mathcal{H}$ and there is a recorded tuple (sid, α) where $\alpha \in \{0,1\}^k \cup \{\mathsf{error}\}$, send (sid, α) to P_i if b equals yes or $(\mathsf{sid}, \mathsf{error})$ if b equals no.
- **Player Corruption.** If $\mathcal{S}$ corrupts $P_i \in \mathcal{H}$ where there is a recorded tuple $((\mathsf{sid}, P_i, P_j, \mathsf{pw}_i, \mathsf{role}), *)$, then reveal pw_i to $\mathcal{S}$. If there also is a recorded tuple $(\mathsf{sid}, \mathsf{sk})$, that has not yet been sent to P_i, then send $(\mathsf{sid}, \mathsf{sk})$ to $\mathcal{S}$.

Fig. 2. Functionality $\mathcal{F}_{\mathsf{PAKE}}^{MA}$

random as long as there are enough honest participants in the session: the adversary cannot bias the distribution unless it controls a large number of players. Namely, this notion formally defines the difference between a key distribution system and a key agreement protocol. More precisely, a protocol is said to be (t, n)-contributory if the group consists of n people and if the adversary cannot bias the key as long as it has corrupted (strictly) less than t players. The authors of [5] achieved $(n/2, n)$-contributiveness in an efficient way, and even $(n-1, n)$-contributiveness by running parallel executions of the protocol. We claim that our proposed protocol directly achieves (n, n)-contributiveness (or full-contributiveness), which means that the adversary cannot bias the key as long as there is at least one honest player in the group. Note that this definition remains very general: letting $t = 1$, we get back to the case in which $\mathcal{A}$ can set the key when it controls at least one player, as in [19].

Ideal Functionality for GPAKE with Mutual Authentication. We assume that every player owns two passwords $(\mathsf{pw}_i^L, \mathsf{pw}_i^R)$, and that for all i, $\mathsf{pw}_i^R = \mathsf{pw}_{i-1}^L$. Our functionality builds upon that presented in [5]. In particular, note that the functionality is not in charge of providing the passwords to the participants. Rather we let the environment do this. As already pointed out in [19], such an approach allows to model, for example, the case where some users may use the same password for different protocols and, more generally, the case where passwords are chosen according to some arbitrary distribution (*i.e.*, not necessarily the uniform one). Moreover, notice that allowing the environment to choose the passwords guarantees forward secrecy, basically for free. More generally, this approach allows to preserve security[1] even in those situations where the password is used (by the same environment) for other purposes.

[1] By "preserved" here we mean that the probability of breaking the scheme is basically the same as the probability of guessing the password.

Since we consider the (improved) split functionality model, the functionality is parameterized by an ordered group $\mathsf{Pid} = \{P_1, \ldots, P_n\}$, dynamically defined, consisting of all the players involved in the execution (be they real players or players controlled by the adversary). Thus, we note that it is unnecessary to impose that the players give this value Pid when notifying their interest to join an execution via a $\mathsf{NewSession}$ query, as was done in [5]. This additional simplification has some interest in practice, since the players do not always know the exact number of players involved, but rather a common characteristic (such as a Facebook group).

We thus denote by n the number of players involved (that is, the size of Pid) and assume that every player starts a new session of the protocol with input $(\mathsf{NewSession}, \mathsf{sid}, P_i, (\mathsf{pw}_i^L, \mathsf{pw}_i^R))$, where P_i is the identity of the player and $(\mathsf{pw}_i^L, \mathsf{pw}_i^R)$ its passwords. Once all the players in Pid, sharing the same sid, have sent their notification message, $\mathcal{F}_{\mathsf{GPAKE}}$ informs the adversary that it is ready to start a new session of the protocol.

In principle, after the initialization stage is over, all the players are ready to receive the session key. However the functionality waits for $\mathcal{S}$ to send an "ok" message before proceeding. This allows $\mathcal{S}$ to decide the exact moment when the key should be sent to the players and, in particular, it allows $\mathcal{S}$ to choose the exact moment when corruptions should occur (for instance $\mathcal{S}$ may decide to corrupt some party P_i before the key is sent but after P_i decided to participate to a given session of the protocol, see [27]). One could imagine to get rid of this query and ask the functionality to generate the session key when the adversary asks the first delivery query, but it is easier to deal with the corruptions with the choice made here (which is the same as in [27]). Once the functionality receives a message $(\mathsf{sid}, \mathsf{ok}, \mathsf{sk})$ from $\mathcal{S}$, it proceeds to the key generation phase. This is done as in [5], except that, instead of checking whether the players all share the same passwords, $\mathcal{F}_{\mathsf{GPAKE}}$ checks whether the neighbors (the group is assumed to be ordered) share the same password. If all the players share the same passwords as their neighbors and less than t players are corrupted, $\mathcal{F}_{\mathsf{GPAKE}}$ chooses a key sk' uniformly and at random in the appropriate key space. If all the players share the same passwords as their neighbors but t or more players are corrupted, then the functionality allows $\mathcal{S}$ to fully determine the key by letting $\mathsf{sk}' = \mathsf{sk}$. In all the remaining cases no key is established.

This definition of the $\mathcal{F}_{\mathsf{GPAKE}}$ functionality deals with corruptions of players in a way quite similar to that of $\mathcal{F}_{\mathsf{GPAKE}}$ in [27], in the sense that if the adversary has corrupted some participants, it may determine the session key, but here only if there are enough corrupted players. Notice however that $\mathcal{S}$ is given such power only before the key is actually established. Once the key is set, corruptions allow the adversary to know the key but not to choose it.

In any case, after the key generation, the functionality informs the adversary about the result, meaning that the adversary is informed on whether a key was actually established or not. In particular, this means that the adversary is also informed on whether the players use compatible passwords or not: in practice, the adversary can learn whether the protocol succeeded or not by simply monitoring its execution (if the players follow the communication or stop it). Finally the key is sent to the players according to the schedule chosen by $\mathcal{S}$. This is formally modeled by means of key delivery queries. We assume that, as always in the UC framework, once $\mathcal{S}$ asks to deliver the key to a player, the key is sent immediately.

Notice that, the mutual authentication indeed means that if one of the players terminates with a session key (not an error), then all players share the key material; but, it doesn't mean that they all successfully terminated. Indeed, we cannot assume that all the flows are correctly forwarded by the adversary: it can modify just one flow, or at least omit to deliver one flow. This attack, called *denial of service*, is modeled in the functionality by the key delivery: the adversary can choose whether it wants the player to receive or not the good key/messages simply with the help of the keyword b set to yes or no.

The functionality $\mathcal{F}_{\mathsf{GPAKE}}$ is parameterized by a security parameter k, and the parameter t of the contributiveness. It interacts with an adversary $\mathcal{S}$ and an ordered set of parties $\mathsf{Pid} = \{P_1, \ldots, P_n\}$ via the following queries:

- **Initialization.** Upon receiving $(\mathsf{NewSession}, \mathsf{sid}, P_i, (\mathsf{pw}_i^L, \mathsf{pw}_i^R))$ from player P_i for the first time, record $(\mathsf{sid}, P_i, (\mathsf{pw}_i^L, \mathsf{pw}_i^R))$, mark it fresh, and send (sid, P_i) to $\mathcal{S}$.
  If there are already $n-1$ recorded tuples $(\mathsf{sid}, P_j, (\mathsf{pw}_j^L, \mathsf{pw}_j^R))$ for players $P_j \in \mathsf{Pid} \setminus \{P_i\}$, then record $(\mathsf{sid}, \mathsf{ready})$ and send it to $\mathcal{S}$.
- **Key Generation.** Upon receiving a message $(\mathsf{sid}, \mathsf{ok}, \mathsf{sk})$ from $\mathcal{S}$ where there exists a recorded tuple $(\mathsf{sid}, \mathsf{ready})$, then, denote by n_c the number of corrupted players, and
  - If for all i, $\mathsf{pw}_i^R = \mathsf{pw}_{i+1}^L$ and $n_c < t$, choose $\mathsf{sk}' \in \{0,1\}^k$ uniformly at random and store $(\mathsf{sid}, \mathsf{sk}')$. Next, for all $P_i \in \mathsf{Pid}$ mark the record $(\mathsf{sid}, P_i, (\mathsf{pw}_i^L, \mathsf{pw}_i^R))$ $\mathsf{complete}$.
  - If for all i, $\mathsf{pw}_i^R = \mathsf{pw}_{i+1}^L$ and $n_c \geq t$, store $(\mathsf{sid}, \mathsf{sk})$. Next, for all $P_i \in \mathsf{Pid}$ mark $(\mathsf{sid}, P_i, (\mathsf{pw}_i^L, \mathsf{pw}_i^R))$ $\mathsf{complete}$.
  - In any other case, store $(\mathsf{sid}, \mathsf{error})$. For all $P_i \in \mathsf{Pid}$ mark the record $(\mathsf{sid}, P_i, (\mathsf{pw}_i^L, \mathsf{pw}_i^R))$ error.

  When the key is set, report the result (either error or $\mathsf{complete}$) to $\mathcal{S}$.
- **Key Delivery.** Upon receiving a message $(\mathsf{deliver}, \mathsf{b}, \mathsf{sid}, P_i)$ from $\mathcal{S}$, then if $P_i \in \mathsf{Pid}$ and there is a recorded tuple (sid, α) where $\alpha \in \{0,1\}^k \cup \{\mathsf{error}\}$, send (sid, α) to P_i if b equals yes or $(\mathsf{sid}, \mathsf{error})$ if b equals no.
- **Player Corruption.** If $\mathcal{S}$ corrupts $P_i \in \mathsf{Pid}$ where there is a recorded tuple $(\mathsf{sid}, P_i, (\mathsf{pw}_i^L, \mathsf{pw}_i^R))$, then reveal $(\mathsf{pw}_i^L, \mathsf{pw}_i^R)$ to $\mathcal{S}$. If there also is a recorded tuple $(\mathsf{sid}, \mathsf{sk})$, that has not yet been sent to P_i, then send $(\mathsf{sid}, \mathsf{sk})$ to $\mathcal{S}$.

Fig. 3. Functionality $\mathcal{F}_{\mathsf{GPAKE}}$

4 Scheme

Intuition. The main idea of our protocol is to apply the Burmester-Desmedt technique [16] to any secure two-party PAKE achieving (mutual) explicit authentication in the UC framework. More precisely, the players execute such a protocol in flows $(2a)$ and $(2b)$ (see Figure 4) and use the obtained value in flows (3) and (4) as in a classical Burmester-Desmedt-based protocol.

The split functionality is emulated thanks to the first flow, where the players engage in their signature verification key, as well as the elements used for the splitting part of the two-party protocols. They are then (after the dotted line in the figure) partitioned according to the values they received during this first round.

Finally, the contributiveness is ensured by the following trick: In addition to establishing pairwise keys between any two pair of neighbors, the players also choose on their own a random secret value K_i, which will also be used in the session key generation. An important point is that these values are chosen independently thanks to the commitment between flows $(2a)$ and $(2b)$. This will ensure the session key to be uniformly distributed as long as at least one player is honest.

Building Blocks. We assume to be given a universally composable two-party password-based authenticated key exchange with mutual authentication $\mathsf{2PAKE}$, achieving or not security against adaptive corruptions. This key exchange is assumed (as defined by the ideal functionality) to give as output a uniformly distributed random string. Due to the mutual authentication, this protocol results in an error message in case it does not succeed: Either the two players end with the same key, or they end with an error. Note, however, that one player can have a key while the other is still waiting since the adversary can retain a message: This is a denial-of-service attack, since a timeout will stop the execution of the protocol. Mutual authentication guarantees that the players cannot end with two different keys.

Let $(\mathsf{SKG}, \mathsf{Sign}, \mathsf{Verify})$ be a one-time signature scheme, SKG being the signature key generation, Sign the signing algorithm and Verify the verifying algorithm. Note that we do not require a *strong* one-time signature: Here, the adversary is allowed to query the signing oracle at most once, and should not be able to forge a signature on a *new* message.

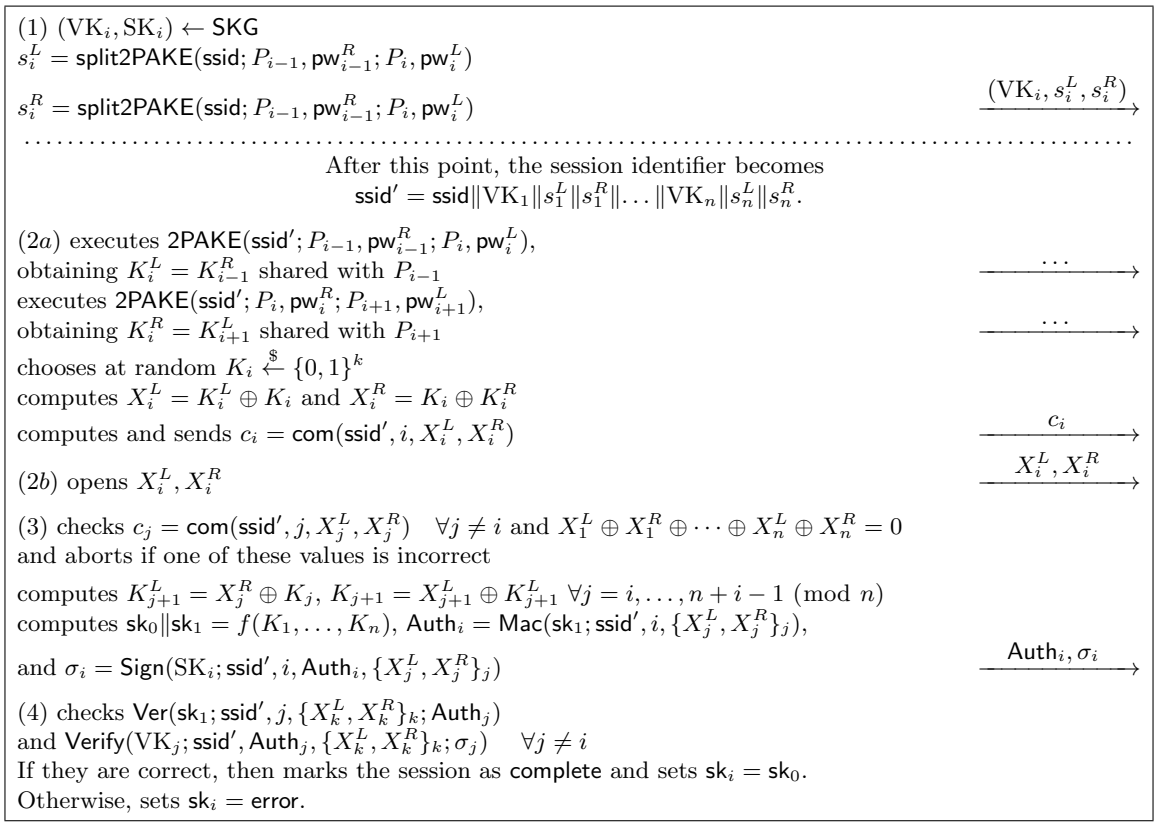

Fig. 4. Description of the protocol for player P_i, with index i and passwords pw_i^L and pw_i^R

Let $(\mathsf{Mac}, \mathsf{Ver})$ be a message authentication code scheme, Mac being the authenticating algorithm and Ver the verifying algorithm. A pseudo-random function could be used, since this is a secure MAC [10].

As usual, we will need a randomness extractor, in order to generate the final session key, as well as an authentication key (for the key confirmation round, guaranteed by a Mac computation). But because of the UC framework, and the definition of the functionality, in the case of a corrupted player, the adversary will learn all the inputs of the extractor, chosen by the players, and the session key chosen by the functionality as well. We will thus have to be able to choose the inputs for the honest players so that they lead to the expected output. We thus use a specific randomness extractor, with a kind of partial invertibility: we consider a finite field $\mathbb{F} = \mathbb{F}_q$. The function

$$\begin{aligned} f : (\mathbb{F}^* \times \ldots \times \mathbb{F}^*) \times (\mathbb{F} \times \ldots \times \mathbb{F}) &\to \mathbb{F} \\ (\alpha_1, \ldots, \alpha_n \quad ; \quad h_1, \ldots, h_n) &\mapsto \sum \alpha_i h_i \end{aligned}$$

is a randomness extractor from tuples $(h_1, \ldots, h_n) \in \mathbb{F}^n$ where at least one h_i is uniformly distributed and independent of the others. Since it can be shown as a universal hash function, using similar techniques to [21], if we consider any distribution $\mathcal{D}_i$ on $\mathbb{F}^n$, for which the distribution $\{h_i | (h_1, \ldots, h_n) \leftarrow \mathcal{D}_i\}$ is the uniform distribution in $\mathbb{F}$, then the distributions

$$(\alpha_1, \ldots, \alpha_n, f(\alpha_1, \ldots, \alpha_n; h_1, \ldots, h_n)), \qquad (\alpha_1, \ldots, \alpha_n) \xleftarrow{\$} \mathbb{F}^{*n}, (h_1, \ldots, h_n) \leftarrow \mathcal{D}_i$$

$$(\alpha_1, \ldots, \alpha_n, U), \qquad (\alpha_1, \ldots, \alpha_n) \xleftarrow{\$} \mathbb{F}^{*n}, U \xleftarrow{\$} \mathbb{F}$$

are perfectly indistinguishable. The tuple $(\alpha_1, \ldots, \alpha_n)$ is the public key of the randomness extractor, and it is well-known that it can be fixed in the CRS [28], with a loss of security linear in the number of queries. Since n might not be fixed in advance, we can use a pseudo-random generator that generates the sequence $\alpha_1, \ldots$, from a key k in the CRS. Anyway, we generically use f as the variable input-length randomness extractor in the following. As said above, we will have to invert f to adapt the input of an honest user to the expected session key: for a fixed key, some fixed inputs $I_i = (h_1, \ldots, \hat{h_i} \ldots, h_n) \in \mathbb{F}^{n-1}$ (possibly all but one, here h_i), and the output U, the function $g_i(I_i, U)$ completes the input so that the output by f is U. With our function f, we have $g_i(I_i, U) = (U - \sum_{j \neq i} \alpha_j h_j)/\alpha_i$.

Finally, we will also need a commitment scheme. In addition to being hiding and binding, we will require it to be extractable, equivocable and non-malleable, such as those of [18, 1, 6]. Even if this latter commitment is only *conditionally* extractable, this will not matter here since the commitment will be opened later: The user cannot try to cheat otherwise the protocol stops. Note that the extractable property allows the simulator to obtain the values committed to by the adversary, the equivocable property allows him to open his values to something consistent with them, and the non-malleable property ensures that when $\mathcal{A}$ sees a commitment, he is not able to construct another one with a related distribution. Because of extractability and equivocability, both the hiding and the binding properties are computational only.

Description of the Protocol. For the sake of completeness, we describe the case where each player owns two different passwords (pw_i^L and pw_i^R), and each pair of neighbors (while the ring is set) shares a common password ($\mathsf{pw}_i^R = \mathsf{pw}_{i+1}^L$). The case where the players all share the same password is easily derived from here, by letting $\mathsf{pw}_i^L = \mathsf{pw}_i^R$. Note that both cases will UC-emulate the GPAKE functionality presented earlier.

We do not assume that the members of the actual group are known in advance. Then one has to imagine a system of timeouts after which the participants consider that no one else will notify its interest in participating to the protocol, and continue the execution. Once the players are known, we order them using a public pre-determined technique (*e.g.*, the alphabetical order on the first flow). Then, for the sake of simplicity we rename the players actually participating $P_1, \ldots, P_n$ according to this order.

Furthermore, such timeouts will also be useful in Flow $(2a)$ in case a player has aborted earlier, in order to avoid other players to wait for it indefinitely. After a certain amount of time has elapsed, the participants should consider that the protocol has failed and abort. Such a synchronization step is useful for the contributiveness, see later on.

Informally, and omitting the details, the algorithm (see Figure 4) can be described as follows: First, each player applies SKG to generate a pair $(\mathrm{SK}_i, \mathrm{VK}_i)$ of signature keys, and sends the value VK_i. They also engage in two two-party key exchange protocols with each of their neighbors: We denote $\mathsf{split2PAKE}$ the corresponding first flow of this protocol, used for the split functionality. The players will be split after this round according to the values received. At this point, the session identifier becomes $\mathsf{ssid}' = \mathsf{ssid} \| \mathrm{VK}_1 \| s_1^L \| s_1^R \| \ldots \| \mathrm{VK}_n \| s_n^L \| s_n^R$ (more details follow). We stress that the round $(2a)$ does not begin until all commitments have been received. In this round, the players open to the values committed.

In round $(2a)$, the players check the commitments received (and abort if one of them is incorrect). Next, player P_i chooses at random a bitstring K_i. It also gets involved into two $\mathsf{2PAKE}$ protocols, with each of its neighbors P_{i-1} and P_{i+1}, and the passwords pw_i^L and pw_i^R, respectively. This creates two random strings: $K_i^L = \mathsf{2PAKE}(\mathsf{ssid}'; P_{i-1}, \mathsf{pw}_{i-1}^R; P_i, \mathsf{pw}_i^L)$, shared with P_{i-1}, and $K_i^R = \mathsf{2PAKE}(\mathsf{ssid}'; P_i, \mathsf{pw}_i^R; P_{i+1}, \mathsf{pw}_{i+1}^L)$, shared with P_{i+1}. It finally computes $X_i^L = K_i^L \oplus K_i$ and $X_i^R = K_i \oplus K_i^R$ and commits to these values.

Pictorially, the situation can be summarized as follows:

$$\begin{array}{cccc} P_{i-1}(\mathsf{pw}_{i-1}^R) & P_i(\mathsf{pw}_i^L) & P_i(\mathsf{pw}_i^R) & P_{i+1}(\mathsf{pw}_{i+1}^L) \\ & K_{i-1}^R = K_i^L \quad K_i \xleftarrow{\$} \{0,1\}^k \quad K_i^R = K_{i+1}^L & \\ X_{i-1}^R & X_i^L = K_i^L \oplus K_i \quad X_i^R = K_i \oplus K_i^R & X_{i+1}^L \end{array}$$

where $X_{i-1}^R = K_{i-1} \oplus K_{i-1}^R = K_{i-1} \oplus K_i^L$ and $X_{i+1}^L = K_{i+1}^L \oplus K_{i+1} = K_i^R \oplus K_{i+1}$.

Once P_i has received all these commitments (again, we stress that no player begins this round before having received all the commitments previously sent), it opens to the values committed (round $(2b)$).

In round (3), the players check the commitments received (and abort if one of them is incorrect). Next, player P_i iteratively computes all the K_j's required to compute the session keys $\mathsf{sk}_0 \| \mathsf{sk}_1$ and the key confirmation $\mathsf{Auth}_i = \mathsf{Mac}(\mathsf{sk}_1; \mathsf{ssid}', i, \{X_j^L, X_j^R\}_j)$. It also signs this authenticator along with all the commitments received in the previous flow.

Finally, in round (4), after having checked the authenticators and the signatures, the players mark their session as $\mathsf{complete}$ (or abort if one of these values is incorrect) and set their session key $\mathsf{sk}_i = \mathsf{sk}_0$.

Remarks. As soon as a value received by a player P_i doesn't match with the expected value, it aborts, setting the key $\mathsf{sk}_i = \mathsf{error}$. In particular, every player P_i checks the commitments $c_j = \mathsf{com}(\mathsf{ssid}', j, X_j^L, X_j^R)$, the signatures $\sigma_j = \mathsf{Sign}(\mathrm{SK}_j; \mathsf{ssid}', \mathsf{Auth}_j, \{X_k^L, X_k^R\}_k)$, and finally the key confirmations $\mathsf{Auth}_j = \mathsf{Mac}(\mathsf{sk}_1; \mathsf{ssid}', j, \{X_k^L, X_k^R\}_k)$. This enables the protocol to achieve mutual authentication.

The protocol also realizes the split functionality due to the two following facts: First, the players are partitioned according to the values VK_j and $\mathsf{split2PAKE}$ they received during the first round (*i.e.*, before the dotted line in Figure 4). All the VK_i are shared among them and their session identifier becomes $\mathsf{ssid}' = \mathsf{ssid} \| \mathrm{VK}_1 \| s_1^L \| s_1^R \| \ldots \| \mathrm{VK}_n \| s_n^L \| s_n^R$. Furthermore, in round 3, the signature added to the authentication flow prevents the adversary from being able to change an X_i^L or X_i^R to another value. Since the session identifier ssid' is included in all the commitments, and in the latter signature, only players in the same subset can accept and conclude with a common key.

Then, the contributory property is ensured by the following trick: At the beginning of each flow, the players wait until they have received all the other values of the previous flow before sending their new one. This is particularly important between $(2a)$ and $(2b)$. Thanks to the commitments sent in this flow, it is impossible for a player to compute its values X_i^L and X_i^R once it has seen the others: Every player has to commit its values at the same time as the others, and cannot make them depend on the other values sent by the players (recall that the commitment is non-malleable). This disables it from being able to bias the key (more details can be found in the proof, see Appendix A).

Finally we point out that, in our proof of security, we don't need to assume that the players erase any ephemeral value before the end of the computation of the session key.

Our Main Theorem. Let $\widehat{s\mathcal{F}_{GPAKE}}$ be the multi-session extension of the split functionality $s\mathcal{F}_{\mathsf{GPAKE}}$.

Theorem 1 *Assuming that the protocol* $\mathsf{2PAKE}$ *is a universally composable two-party password-based authenticated key exchange with mutual authentication secure against adaptive (*resp. *static) corruptions,* $(\mathsf{SKG}, \mathsf{Sign}, \mathsf{Verify})$ *a one-time signature scheme,* com *a non-malleable, extractable and equivocable commitment scheme,* $(\mathsf{Mac}, \mathsf{Ver})$ *a message authentication code scheme, and* f *a randomness extractor as defined earlier, the protocol presented in Figure 4 securely realizes* $\widehat{s\mathcal{F}_{GPAKE}}$ *in the CRS model, in the presence of adaptive (*resp. *static) adversaries, and is fully-contributory.*

5 Functionality for Merging two Groups or Joining a Member to a Group

We present in this section the functionalities to join a member (or a group of members) to a group already constituted. We assume the existence of a "host", that is, a member of the former group inviting the new player, and a "guest", which is the invited player, or a particular player in the invited group. These two players are assumed to share a common password.

Two cases may happen: Either we ask the whole group to participate, or only the host and the guest. We describe the former case, highlighting later on the (small) differences with the latter one. In any case, all the participants (the guest, the host, and the members of the former group) receive the new key.

The functionality comes with two variants: the forward-secure one, and the non-forward-secure one. In the latter case, the players not only receive the new key, but also the old key of the former group. This captures the fact that the invited players can learn the former session key of the first group (and the hosts can learn the former key of the second group), and thus the message exchanged before their invitation to join the group. Everything else works roughly as in the GPAKE functionality. Note that we consider the split functionality model and that no $\mathsf{TestPwd}$ is allowed to the adversary in this functionality.

Merging two Groups. $n + m$ PLAYERS, WITH FORWARD SECRECY. In this variant, all the players participate, that is all the members of the former groups $P_1, \ldots, P_n$ and $P'_1, \ldots, P'_m$ (including a host in the first group and a guest in the second one). The group Pid is thus of size $n + m$.

The model proposes $(t, n+m)$-contributiveness ($t \in \{1, n+m\}$), which means that the adversary can bias the key if it corrupts at least t persons. Otherwise, the new key is completely unpredictable.

The players first begin by noticing their interest in participating to an execution, by sending a $\mathsf{NewSession}$ query: the guest mentions the host inviting it, the former key of its group and the password it uses. The host mentions the same elements. The members of both groups mention the old key of their former group. When all of them have sent their queries, the functionality notifies the adversary that the players are ready to receive their key. As in the GPAKE functionality, the adversary is granted the right to choose the key if it has corrupted t or more players. Otherwise, the functionality chooses it at random. Similarly, once the key is set, the functionality waits for the adversary to deliver the new key to all the players, either active or passive. This functionality is described Figure 5.

$n + m$ PLAYERS, WITHOUT FORWARD SECRECY. Everything works as before, except that the functionality also sends the former keys of both groups to all the players.

2 PLAYERS, WITH FORWARD SECRECY. In this variant, only the two players host and guest participate in the join phase: the other members of the group only act passively (they learn in the end the new session key). The $\mathsf{NewSession}$ queries for the members is suppressed, and the last query of the initialization phase is modified as follows:

- If there are two recorded tuples $(\mathsf{sid}, P'_j, \mathsf{guest}, \mathsf{pw}', \mathsf{sk}'_j)$ and $(\mathsf{sid}, P_i, \mathsf{host}, \mathsf{pw}, \mathsf{sk}_i)$, then record the tuple $(\mathsf{sid}, \mathsf{Pid}, \mathsf{ready})$ and send it to $\mathcal{S}$.

Furthermore, in the key generation phase, the parameter of the contributiveness is now $t \in \{1, 2\}$ (the adversary can corrupt the host or the guest). Everything remains as before.

2 PLAYERS, WITHOUT FORWARD SECRECY. Everything works as in the previous case, except that the functionality also sends the former keys to all the players, and not only the new one.

Adding a single member. The variants (2 or $n + 1$ players, with or without forward secrecy) of the functionality capturing the join of a single player to a group $(P_1, \ldots, P_n)$ can be easily deduced from those described above. The main difference is that we get rid of the key of the former second group in the $\mathsf{NewSession}$-query of the invited person: $(\mathsf{NewSession}, \mathsf{sid}, P', \mathsf{guest}, \mathsf{pw}')$.

The functionality $\mathcal{F}_{\mathsf{GPAKEmerge}}$ is parameterized by a security parameter k, and the parameter t of the contributiveness. It interacts with an adversary $\mathcal{S}$ and a set of parties $\mathsf{Pid} = \{P_1, \ldots, P_n, P'_1, \ldots, P'_m\}$ via the following queries:

- **Initialization.**
  - Upon receiving a message $(\mathsf{NewSession}, \mathsf{sid}, P'_j, \mathsf{guest}, \mathsf{pw}', \mathsf{sk}'_j)$ from player P'_j for the first time, record $(\mathsf{sid}, P'_j, \mathsf{guest}, \mathsf{pw}', \mathsf{sk}'_j)$, mark it fresh, and send $(\mathsf{sid}, P'_j, \mathsf{guest})$ to $\mathcal{S}$. Ignore any subsequent query of the form $(\mathsf{NewSession}, \mathsf{sid}, *, \mathsf{guest}, *, *)$.
  - Upon receiving a message $(\mathsf{NewSession}, \mathsf{sid}, P_i, \mathsf{host}, \mathsf{pw}, \mathsf{sk}_i)$ from player P_i for the first time, record $(\mathsf{sid}, P_i, \mathsf{host}, \mathsf{pw}, \mathsf{sk}_i)$, mark it fresh, and send $(\mathsf{sid}, P_i, \mathsf{host})$ to $\mathcal{S}$. Ignore any subsequent query of the form $(\mathsf{NewSession}, \mathsf{sid}, *, \mathsf{host}, *, *)$.
  - Upon receiving $(\mathsf{NewSession}, \mathsf{sid}, Q_j, \mathsf{member}, \mathsf{key}_j)$ from player $Q_j = P_j$ or $Q_j = P'_j$ for the first time, with $\mathsf{key} = \mathsf{sk}$ or $\mathsf{key} = \mathsf{sk}'$, record $(\mathsf{sid}, Q_j, \mathsf{member}, \mathsf{key}_j)$, mark it fresh, and send $(\mathsf{sid}, Q_j, \mathsf{member})$ to $\mathcal{S}$.
  - If there are already $|\mathsf{Pid}| - 2$ recorded tuples of the form $(\mathsf{sid}, Q_k, \mathsf{member}, \mathsf{key}_k)$ and two recorded tuples $(\mathsf{sid}, P'_j, \mathsf{guest}, \mathsf{pw}', \mathsf{sk}'_j)$ and $(\mathsf{sid}, P_i, \mathsf{host}, \mathsf{pw}, \mathsf{sk}_i)$, then record $(\mathsf{sid}, \mathsf{Pid}, \mathsf{ready})$ and send it to $\mathcal{S}$.
- **Key Generation.** Upon receiving a message $(\mathsf{sid}, \mathsf{Pid}, \mathsf{ok}, \mathsf{sk})$ from $\mathcal{S}$ where there exists a recorded tuple $(\mathsf{sid}, \mathsf{Pid}, \mathsf{ready})$,
  - If all $P_j \in \mathsf{Pid}$ have the same key, as well as all $P'_j \in \mathsf{Pid}$, the host and the guest share the same pw, and strictly less than t layers have been corrupted among the active players, choose $\mathsf{sk}' \in \{0,1\}^k$ uniformly at random and store $(\mathsf{sid}, \mathsf{Pid}, \mathsf{sk}')$. Next, for all $Q_j \in \mathsf{Pid}$ mark the record $(\mathsf{sid}, Q_j, *, *)$ or $(\mathsf{sid}, Q_j, *, *, *)$ $\mathsf{complete}$.
  - If all $P_j \in \mathsf{Pid}$ have the same key, as well as all $P'_j \in \mathsf{Pid}$, the host and the guest share the same pw, but more than t players are corrupted, store $(\mathsf{sid}, \mathsf{Pid}, \mathsf{sk})$. Next, for all $Q_j \in \mathsf{Pid}$ mark the record $(\mathsf{sid}, P_j, *, *)$ or $(\mathsf{sid}, P_j, *, *, *)$ $\mathsf{complete}$.
  - In any other case, store $(\mathsf{sid}, \mathsf{Pid}, \mathsf{error})$. For all $Q_j \in \mathsf{Pid}$ mark the record $(\mathsf{sid}, Q_j, *, *)$ or $(\mathsf{sid}, Q_j, *, *, *)$ error.

  When the key is set, report the result (either error or $\mathsf{complete}$) to $\mathcal{S}$.
- **Key Delivery.** Upon receiving a message $(\mathsf{deliver}, \mathsf{b}, \mathsf{sid}, Q_j)$ from $\mathcal{S}$, then if $Q_j \in \mathsf{Pid}$ and there is a recorded tuple $(sid, \mathsf{Pid}, \alpha)$ where $\alpha \in \{0,1\}^k \cup \{\mathsf{error}\}$, send $(\mathsf{sid}, \mathsf{Pid}, \alpha)$ to Q_i if b equals yes or $(\mathsf{sid}, \mathsf{Pid}, \mathsf{error})$ if b equals no.
- **Player Corruption.** If $\mathcal{S}$ corrupts $Q_j \in \mathsf{Pid}$ where there is a recorded tuple $(\mathsf{sid}, Q_j, *, *, \mathsf{pw})$ of role guest or host, then reveal pw to $\mathcal{S}$. If there also is a recorded $(\mathsf{sid}, \mathsf{Pid}, \mathsf{sk})$, that has not yet been sent to Q_j, then send $(\mathsf{sid}, \mathsf{Pid}, \mathsf{sk})$ to $\mathcal{S}$.

Fig. 5. Functionality $\mathcal{F}_{\mathsf{GPAKEmerge}}$

6 Merging two Groups

Since the case in which a single user joins an existing group is a particular case of merging two groups, we concentrate on the latter more general case. Let $G = \{P_1, \ldots, P_n\}$ and $G' = \{P'_1, \ldots, P'_m\}$ be two groups which have already created two group session keys via the protocol described in Section 4. Using the same notations, we assume that each player P_k in G has kept in memory its own private value K_k as well as all the public values $\{X_1^L, X_1^R, \ldots, X_n^L, X_n^R\}$. Similarly, assume that each player P'_ℓ in G' has kept in memory its own private value K'_ℓ as well as all the public values $\{X_1'^L, X_1'^R, \ldots, X_m'^L, X_m'^R\}$.

In other words, we ask each player to keep in memory all the values necessary to the computation of the group's session key. Remarkably, note that they only have to keep a single private value; All the other values are public, and can be kept publicly in a single place accessible to the players.

The goal of our dynamic merge protocol is to allow the computation of a joint group session key between G and G', without asking the whole new group $G \cup G'$ to start a key-exchange protocol from scratch. In addition, the protocol we describe here has two nice properties: First, it does not increase the memory requirements of each player. Second, it is done in such a way that the situation of each player after the merge protocol is the same as its situation before it. That way, future join or merge protocols can easily take place iteratively without any change.

For sake of simplicity, we first describe a basic version of our protocol, in which only one representative of each group participates in the new exchange of messages between the two groups. Clearly, this version is not fully contributory since only two participants take place in the protocol. We then show how to extend it into a fully contributory protocol, in which all $n + m$ participants will take part in the message exchange.

Basic Version. Let P_i and P'_j denote the particular members of G and G' that are acting as the representative of these groups. Only these two participants will take part in the merge protocol. In order to construct a session key for the new group, these two players are assumed to share a common password, denoted as pw for P_i and pw' for P'_j. The situation is summarized in Figure 6, where the upper part (1) represents the former second group, with the values computed during the execution of the GPAKE protocol, and the lower part (2) represents the former first group, with the values computed during the execution of the GPAKE protocol. The hatched lines represent the abandoned "links". Indeed, both P_i and P'_j will erase their values K_i and K'_j and create two new connections between them, thus creating the new group

$$G'' = \{P_1, \ldots, P_{i-1}, P_i, P'_j, P'_{j+1}, \ldots, P'_m, P'_1, \ldots, P'_{j-1}, P'_j, P_i, P_{i+1}, \ldots, P_n\}$$

These connections are represented vertically in the middle part (2) of the figure. We stress that during the merge protocol, no value is computed in parts (1) and (2). The core of the protocol is part (3).

The protocol is described in Figures 7 and 8 with the notations of Figure 6. Informally, the merge protocol consists in the execution of a simplified GPAKE protocol with the whole group G", but the interesting point is that only P_i and P'_j participate and exchange messages, executing two $\mathsf{2PAKE}$ protocols, instead of the $n+m-1$ that would be necessary for an execution from scratch with this new group. Merging two groups is thus much more efficient. The two executions are performed once for the left part of (3) in Figure 6, and once for the right part. For P_i and P'_j, the steps are similar to those of a normal GPAKE protocol execution. Additionnaly, P_i and P'_j have to broadcast the necessary (old) values X_k^L, X_k^R and $X_l'^L, X_l'^R$ to the other members of each subgroup, to enable them derive the new key. These other players only participate passively, listening to broadcasts so as to learn the values needed to compute the new key of the merged group.

This merge protocol is thus only partially contributory since P_i and P'_j are the only players participating and exchanging messages. Furthermore, it is not forward-secure since the players of both groups become able to compute the former key of the other group thanks to the values broadcasted by P_i and P'_j. Also note that we could simplify this protocol by merging the commitments, signatures and MACs, doing only one for each player. But we chose to keep the protocol symmetric, the values $\tilde{x}$ representing roughly the unnecessary values (of the vanishing players, see the next paragraph) and the values $\overline{x}$ representing roughly the needed values.

We claim that after this execution, the players will find themselves in a similar situation than after a normal GPAKE protocol. For the moment, this is not the case since P_i and P'_j appear twice in the ring (see Figure 6). For both of them, we have to get rid of one instance of the player. To this aim, once this protocol is executed, P_i "vanishes" on the left part of (3) in Figure 6, letting the player P_{i-1} with a new value X_{i-1}^R equal to $X_{i-1}^R \oplus \widetilde{X_i^L}$ and the player P'_j with a new value $\overline{X_j'^L}$ equal to $\overline{X_j'^L} \oplus \widetilde{X_i^R}$. The new $\mathsf{2PAKE}$-value shared between them is $\widetilde{K_i}$. The same thing happens on the right part of (3) in Figure 6: P'_j vanishes, letting the player P'_{j-1} with the new value $X_{j-1}'^R$ equal to $X_{j-1}'^R \oplus \widetilde{X_j'^L}$ and P_i with the new value $\overline{X_i^L}$ equal to $\widetilde{X_j'^R} \oplus \overline{X_i^L}$. The new $\mathsf{2PAKE}$-value shared between them is $\widetilde{K'_j}$. This way, it is as if the players P_i and P'_j had only participated once in the new protocol: P_i between P'_{j-1} and P_{i+1}, and P'_j between P_{i-1} and P'_{j+1}. Finally, we will only

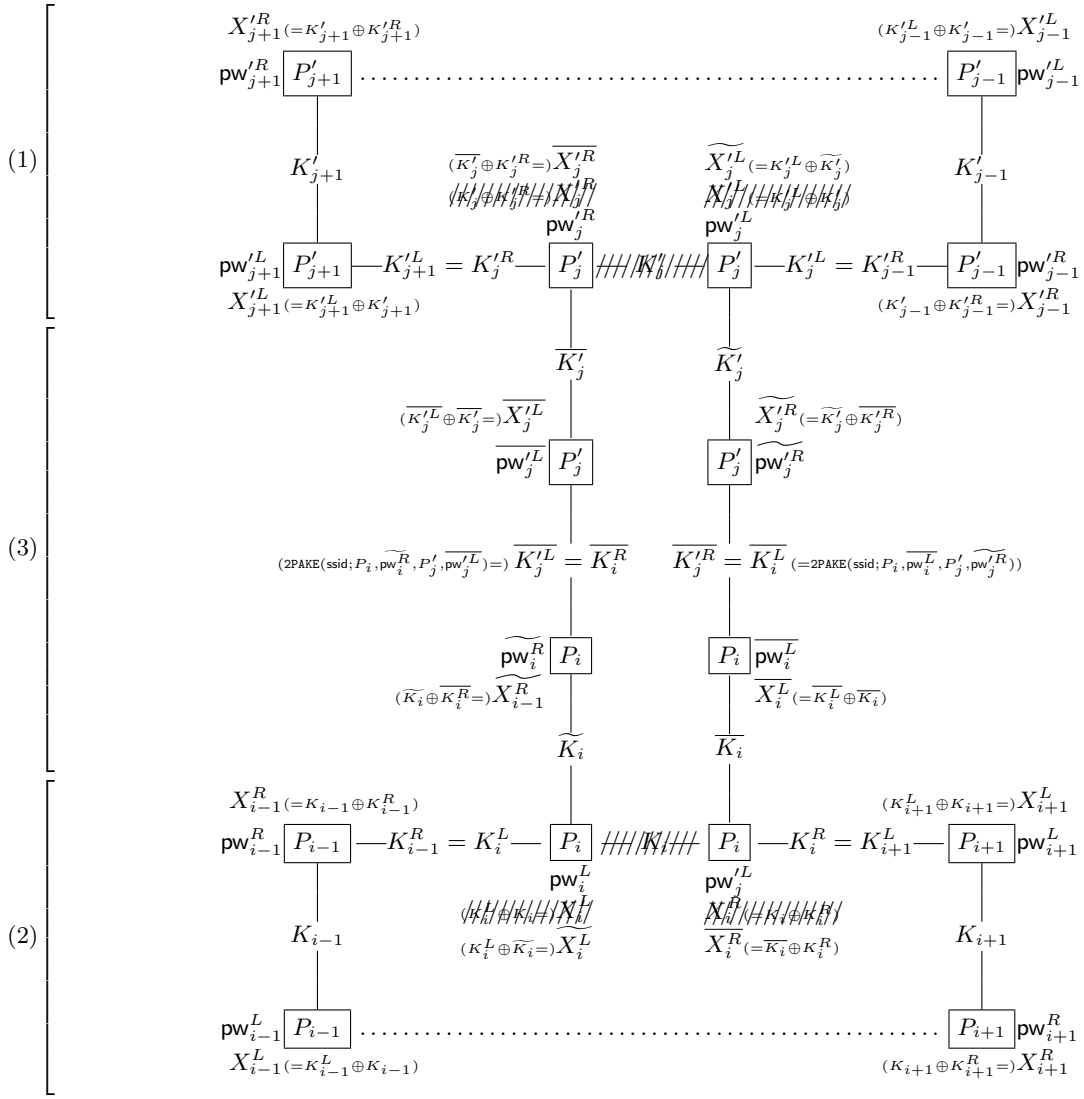

Fig. 6. Merging two Groups: (1) represents the former group $(P'_1, P'_2, \ldots, P'_m)$; (2) represents the former group $(P_1, P_2, \ldots, P_n)$; (3) is the proper merge protocol, between the inviter P_i and the invited P'_j.

need to keep the following values: $\overline{K'_j}$ secretly for P'_j, $\overline{K_i}$ secretly for P_i, and $X^R_{i-1} = X^R_{i-1} \oplus \widetilde{X^L_i}$, $\overline{X'^L_j} = \overline{X'^L_j} \oplus \widetilde{X^R_i}$, $X'^R_{j-1} = X'^R_{j-1} \oplus \widetilde{X'^L_j}$ and $\overline{X^L_i} = \widetilde{X'^R_j} \oplus \overline{X^L_i}$ publicly. The values of the rest of the group remain unchanged. This will allow to do another join of merge iteratively. Pictorially, this leads to the new following situation:

– The left part of (3) in Figure 6 without P_i becomes

$$\begin{array}{cccc} P_{i-1}(\mathsf{pw}^R_{i-1}) & & P'_j(\overline{\mathsf{pw}'^L_j}) & P'_j(\mathsf{pw}'^R_j) \quad P'_{j+1}(\mathsf{pw}'^L_{j+1}) \\ & \widetilde{K_i} & & \overline{K'_j} \quad K'^R_j = K'^L_{j+1} \\ X^R_{i-1} \oplus \widetilde{X^L_i} = K_{i-1} \oplus \widetilde{K_i} & & \overline{X'^L_j} \oplus \widetilde{X^R_i} = \widetilde{K_i} \oplus K'_j & \overline{X'^R_j} \quad X'^L_{j+1} \end{array}$$

with $\widetilde{K_i}, \overline{K'_j} \xleftarrow{\$} \{0,1\}^k$, $\overline{X'^R_j} = \overline{K'_j} \oplus K'^R_j$ and $X'^L_{j+1} = K'^L_{j+1} \oplus K'_{j+1}$.

– The right part of (3) in Figure 6 without P'_j (with $\widetilde{K'_j}, \overline{K_i} \xleftarrow{\$} \{0,1\}^k$, $\overline{X_i^R} = \overline{K_i} \oplus K_i^R$ and $X_{i+1}^L = K_{i+1}^L \oplus K_{i+1}$) becomes:

$$\begin{array}{cccc} P'_{j-1}(\mathsf{pw}'^R_{j-1}) & & P_i(\overline{\mathsf{pw}_i^L}) & P_i(\mathsf{pw}_i^R) \quad P_{i+1}(\mathsf{pw}_{i+1}^L) \\ & \widetilde{K'_j} & & \overline{K_i} \quad K_i^R = K_{i+1}^L \\ X'^R_{j-1} \oplus \widetilde{X'^R_j} = K'_{j-1} \oplus \widetilde{K'_j} & & \overline{X_i^L} \oplus \widetilde{X'^R_j} = \overline{K_i} \oplus \widetilde{K'_j} & \overline{X_i^R} \quad X_{i+1}^L \end{array}$$

– Again, all the other values of the rest of the group remain unchanged.

Forward-Secure Fully-Contributory Protocol. The scheme presented in the previous section does not provide forward secrecy since the players in one group learn enough information to compute the previous key of the other group. It is also not fully contributory because P_i and P'_j are the only players to actively participate in the merge protocol: they have full control over the value of the new group session key. In order to achieve these goals, we make two significant changes to the above protocol.

In order to guarantee full contributiveness, we will require that all the players of each group (except for the host P_i and the guest P'_j, which will keep behaving as before) use new fresh values K_k and K'_l when computing the new group session key. Since all the parties need to learn the values chosen by the other parties in a secure way, this change will require additional rounds of communication in the merge protocol (but no additional executions of the 2PAKE protocol). More precisely, we modify the protocol of Figures 7 and 8 so that all the players of each group participate in the later phases of the protocol.

– All the players send flows ($1a$) and ($1c$).
– In ($2a$), all the players choose a random K_k or K'_l, compute the values X_k^L, X_k^R or $X_l'^L, X_l'^R$ and commit to them.
– In ($2b$), all the players open their commitments. Note that the host and the guest no longer need to broadcast the values X_k^L, X_k^R or $X_l'^L, X_l'^R$ since they are now sent by the other players.
– Every player participates in (3) and (4).

In order to achieve forward security, we will change the way in which we compute the values K_i^L and K_i^R in step ($2a$) of original GPAKE protocol in Figure 4 and in the merge protocol in Figure 7. Instead of setting the output of the 2PAKE protocol to be K_i^L and K_i^R in Figures 4 and 7, we will first use these outputs as the initial seed or state of a forward-secure stateful pseudorandom generator [12] and then use this state to generate the actual values K_i^L and K_i^R as well as the next state. More precisely, let GEN be a stateful generator and let GEN:next be the next step algorithm, which on input the current state, outputs a pseudorandom bit string along with the next state (please refer to [12] for the precise definitions). In step ($2a$) of Figures 4 and 7, we first set the output of the two executions of 2PAKE protocol to St_i^L and St_i^R. Next, we run GEN:next on input St_i^L and St_i^R to obtain K_i^L and K_i^R, respectively, and the updated values of St_i^L and St_i^R. After the update, the previous values of St_i^L and St_i^R should be securely deleted.

Since the change above also applies to the original GPAKE protocol in Figure 4, it is important to say that the latter will remain secure as long as GEN meets the standard security notion for a stateful generator, which guarantees that its output will look random as long as its input state is chosen uniformly at random. Moreover, if GEN is also forward-secure, then the new merge protocol will also be forward secure if we assume that erasures are possible. This is because, after the deletion of the previous states, all the values generated by GEN will look random and independent of each other to any party who does not know the value of these previous states. As a result, it is straight-forward to adapt the proof of Theorem 1 to prove the following theorem:

Theorem 2 *Assuming that 2PAKE is a universally composable two-party password-based authenticated key exchange secure against adaptive (*resp. *static) corruptions,* (SKG, Sign, *Verify*) *is a one-time signature scheme,* com *is a non-malleable, extractable and equivocable commitment scheme,* (*Mac*, *Ver*) *is a message authentication code scheme, GEN is a forward-secure pseudorandom generator, and f is a randomness extractor as defined earlier, the protocol presented in Figure 7 and 8 with the modifications proposed above securely realizes $\widehat{sF}_{\mathsf{GPAKEjoin}}$ in the standard model, in the presence of adaptive (*resp. *static) adversaries, and is fully-contributory, assuming secure erasures.*

7 Implementation Considerations

The protocols that have been described above were designed for their security properties, and for the quality of the proof of security. When it comes to practical implementations, some additional considerations have to be made.

Definition of the Group. We will consider a use case where the participants to the GPAKE are already members of a chat room, which is the communication means used to broadcast messages. The protocol has to be resistant to the fact that some members of the chat room are idle and will not participate to the GPAKE, and also that some members of the chat room might have difficulties to participate because of connectivity issues: this is thus a nice property the functionality (granted the split functionality) does not need to know the list of participants in advance.

Therefore, instead of ending the initialization phase when a number n of participants is reached (as in previous protocols), we end the initialization phase at the initiative of any of the participants or a timeout. From a practical point of view, it means that in the algorithm of Figure 4, going to step (2a) does not need that all commitments are received, on the opposite, these commitments will be used to dynamically define the group after a certain time, possibly defined by a timeout: the members of the chat room that have sent their commitments.

Another practical issue is the ordering on the ring, which defines the neighbors of each participant. Since the group is not known in advance, this ordering will be defined from the commitments sent in (1): *e.g.*, the alphabetical order.

Authentication within the Group. As explained in the description of the protocol, is accepted as a member of the group anyone that shares a password with another member of the group. This is the best authentication that can be achieved for a GPAKE because a unique shared key is generated for the group. But after the protocol execution, each user owns a pair (SK_i, VK_i) of signing/verification key. It can be used by each participant to sign his/her own messages, to avoid that one participant impersonates another. But then, a (multi-time) signature scheme has to be used, with some formatting constraint to avoid collisions between the use for the GPAKE protocol and the signature of a message.

Removal of one Participant. This protocol provides the functionality of adding members to the group, but does not provide the functionality of removing members. Indeed, while there is a possibility of telling two participants apart (cf. previous paragraph) there is no possibility of truly authenticating a participant. Only the alias (the signing keys) is known.

A functionality that could be implemented is the ban of a participant identified by his/her alias, *e.g.*, because this participant has sent inappropriate messages. However, because all the random K_i are known at step (3), it is necessary to generate new random values that are not known by the banned participant. Therefore, the recommended way to remove one participant from a group is to start again the GPAKE protocol with shared passwords that are not known by this participant.

Acknowledgments

This work was supported in part by the French ANR-07-SESU-008-01 PAMPA Project.

References

1. Michel Abdalla, Jens-Matthias Bohli, Maria Isabel Gonzalez Vasco, and Rainer Steinwandt. (Password) authenticated key establishment: From 2-party to group. In Salil P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 499–514. Springer, February 2007.
2. Michel Abdalla, Xavier Boyen, Céline Chevalier, and David Pointcheval. Distributed public-key cryptography from weak secrets. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 139–159. Springer, March 2009.
3. Michel Abdalla, Xavier Boyen, Céline Chevalier, and David Pointcheval. Strong cryptography from weak secrets: Building efficient pke and ibe from distributed passwords in bilinear groups. In *AFRICACRYPT '10: Proceedings of the 3rd International Conference on Cryptology in Africa*, Berlin, Heidelberg, 2010. Springer-Verlag.
4. Michel Abdalla, Dario Catalano, Céline Chevalier, and David Pointcheval. Efficient two-party password-based key exchange protocols in the UC framework. In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 335–351. Springer, April 2008.
5. Michel Abdalla, Dario Catalano, Céline Chevalier, and David Pointcheval. Password-authenticated group key agreement with adaptive security and contributiveness. In *AFRICACRYPT '09: Proceedings of the 2nd International Conference on Cryptology in Africa*, pages 254–271, Berlin, Heidelberg, 2009. Springer-Verlag.
6. Michel Abdalla, Céline Chevalier, and David Pointcheval. Smooth projective hashing for conditionally extractable commitments. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, Lecture Notes in Computer Science, pages 671–689. Springer, August 2009.
7. Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. Password-based authenticated key exchange in the three-party setting. In Serge Vaudenay, editor, *PKC 2005: 8th International Workshop on Theory and Practice in Public Key Cryptography*, volume 3386 of *Lecture Notes in Computer Science*, pages 65–84. Springer, January 2005.
8. Michel Abdalla and David Pointcheval. A scalable password-based group key exchange protocol in the standard model. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 332–347. Springer, December 2006.
9. Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 361–377. Springer, August 2005.
10. Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, 2000.
11. Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 139–155. Springer, May 2000.
12. Mihir Bellare and Bennet S. Yee. Forward-security in private-key cryptography. In Marc Joye, editor, *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18. Springer, April 2003.
13. Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, May 1992.
14. Jens-Matthias Bohli, Maria Isabel Gonzalez Vasco, and Rainer Steinwandt. Password-authenticated constant-round group key establishment with a common reference string. Cryptology ePrint Archive, Report 2006/214, 2006. `http://eprint.iacr.org/`.
15. Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Dynamic group Diffie-Hellman key exchange under standard assumptions. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 321–336. Springer, April / May 2002.
16. Mike Burmester and Yvo Desmedt. A secure and scalable group key exchange system. *Information Processing Letters*, 94(3):137–143, May 2005.
17. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145. IEEE Computer Society Press, October 2001.
18. Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, August 2001.
19. Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 404–421. Springer, May 2005.

20. Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281. Springer, August 2003.
21. Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.
22. Rosario Gennaro and Yehuda Lindell. A framework for password-based authenticated key exchange. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 524–543. Springer, May 2003. `http://eprint.iacr.org/2003/032.ps.gz`.
23. Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 142–159. Springer, August 2006.
24. M. Choudary Gorantla, Colin Boyd, and Juan Manuel González Nieto. Universally composable contributory group key exchange. In Wanqing Li, Willy Susilo, Udaya Kiran Tupakula, Reihaneh Safavi-Naini, and Vijay Varadharajan, editors, *ASIACCS 09: 4th Conference on Computer and Communications Security*, pages 146–156. ACM Press, March 2009.
25. Adam Groce and Jonathan Katz. A new framework for efficient password-based authenticated key exchange. In *ACM CCS 10*, Berlin, Heidelberg, 2010. Springer-Verlag.
26. Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient password-authenticated key exchange using human-memorable passwords. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 475–494. Springer, May 2001.
27. Jonathan Katz and Ji Sun Shin. Modeling insider attacks on group key-exchange protocols. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, *ACM CCS 05: 12th Conference on Computer and Communications Security*, pages 180–189. ACM Press, November 2005.
28. V. Shoup. *A Computational Introduction to Number Theory and Algebra.* Cambridge University Press, Cambridge, 2005.

A Proof of Theorem 1

A.1 Idea of the Proof

We only prove here the adaptive case, the static case being the same, and easier because one does not need to take care of the corruptions.

We need to construct, for any real-world adversary $\mathcal{A}$ (interacting with real parties running the protocol), an ideal-world adversary $\mathcal{S}$ (interacting with dummy parties and the functionality $\widehat{s\mathcal{F}_{\mathsf{GPAKE}}}$) such that no environment $\mathcal{Z}$ can distinguish between an execution with $\mathcal{A}$ in the real world and $\mathcal{S}$ in the ideal world with non-negligible probability.

We assume the existence of a simulator $\mathcal{S}_{\mathsf{PAKE}}$ (interacting with dummy parties and the functionality $\widehat{s\mathcal{F}^{\mathsf{MA}}_{\mathsf{PAKE}}}$) such that, for any real-world adversary $\mathcal{A}$ (interacting with real parties running the protocol), no environment $\mathcal{Z}$ can distinguish between an execution of 2PAKE with $\mathcal{A}$ in the real world and $\mathcal{S}_{\mathsf{PAKE}}$ in the ideal world with non-negligible probability.

We incrementally define a sequence of games starting from the one describing a real execution of the protocol in the real world, and ending up with game $\mathbf{G_7}$ which we prove to be indistinguishable with respect to the ideal experiment. The key point will be $\mathbf{G_5}$. $\mathbf{G_0}$ is the real-world game. In $\mathbf{G_1}$, we start by simulating the CRS, allowing the simulator to know the extractability trapdoor for the commitments. Then, in $\mathbf{G_2}$, he can thus extract the values X_j^L's and X_j^R's committed to by the adversary in Step $(2a)$. In $\mathbf{G_3}$, $\mathcal{S}$ completely simulates the CRS, knowing also the equivocability trapdoor for the commitments. From this moment on, he simulates all the commitments and makes them become equivocable. The commitment remains binding and is also hiding. In $\mathbf{G_4}$, $\mathcal{S}$ rejects non-oracle-generated signed authenticators sent by players still honest. In $\mathbf{G_5}$, $\mathcal{S}$ deals with the case where the players remain honest up to the beginning of the protocol 2PAKE. In this case, he uses the simulator $\mathcal{S}_{\mathsf{PAKE}}$ to simulate 2PAKE without using the passwords of the players. As a side note, $\mathcal{S}$ does not need these passwords anymore for the all simulation. In $\mathbf{G_6}$, $\mathcal{S}$ deals with the case where there have been some corruptions before the beginning of the protocol 2PAKE. Finally, we

show that $\mathbf{G_7}$, in which we only replace the hybrid queries by the real ones, is indistinguishable from the ideal game.

To this aim, we first describe four hybrid queries that are going to be used in the games. The PAKESamePwd and GPAKESamePwd queries check if the players share the same password, without disclosing it. The PAKEDelivery and GPAKEDelivery queries provide the player with the session key. In some games, the simulator has actually access to the honest players, and thus to their passwords (and always knows the passwords committed by the adversary for corrupted users granted the ideal tweakable cipher). In such a case, these queries can be easily implemented by letting $\mathcal{S}$ look at these passwords. When the players are entirely simulated, $\mathcal{S}$ will replace the queries above with the Key Generation and Key Delivery queries to the ideal functionalities.

Following [19], we say that a flow is *oracle-generated* if it was sent by an honest player (our simulation) and arrives without any alteration to the player it was meant to. We say it is *non-oracle-generated* otherwise, that is either if it was sent by an honest player and modified by the adversary, or if it was sent by a corrupted player or a player impersonated by the adversary: in all these cases, we say that the sender is an *attacked* player. In brief, our simulation controls the random coins of oracle-generated flows, whereas the adversary may control them in non-oracle-generated flows.

Note that since we consider the split functionality, the players have been partitioned in sets according to what they received during the very first flows $(1a)$ and $(1b)$. In the following, we can thus assume that all the players have received the same $(1a)$ (and the same $(1b)$ for each pair participating in the same 2PAKE), under the binding property of the commitment. Oracle-generated flows $(1a)$ have been sent by players that will be considered honest in this session, whereas non-oracle-generated flows have been sent by the adversary, the corresponding players are thus assumed corrupted from the beginning of the session, since the adversary has chosen the password. Note that a player considered honest in the whole protocol can be considered corrupted in at least one of the 2PAKE protocols it participates to: In such a case, the (honest) player will abort after the 2PAKE, being unable to compute its values X_i^L and X_i^R (at least one of the values K_i^L and K_i^R have been computed by the adversary and are thus unknown to the player).

Finally, note that if $(2a)$ is oracle-generated, then $(2b)$ must be oracle-generated also with overwhelming probability, due to the commitment, as above. As a result, we set $\mathsf{sk}_i = \mathsf{error}$ whenever an inconsistency is noted by a player (incorrect commitment opening, or invalid signature). The latter then aborts its execution.

Adaptive Corruptions. Most of the values will be chosen at random during the simulation. As soon as a player gets corrupted, the simulator recovers its password, and has to provide the adversary with all the internal state of the player in a consistent way with respect to the view of the adversary, and even the environment. This is simplified with the use of a 2PAKE providing mutual authentication: This way, either a couple of players end with a shared key, or they end with an error. In the latter case, all the players abort the GPAKE protocol.

A.2 Description of the Simulator

Simulator: Session Initialization. The aim of the first flows $(1a)$ and $(1b)$ is to create the subsets H of players involved in the same protocol execution (see the split functionality, Section 2 and Figure 1).

More precisely, in flow $(1a)$, $\mathcal{S}$ chooses the values $(\mathrm{SK}_i, \mathrm{VK}_i)$ on behalf of the honest players at random, then computes and sends the commitments c_i to $\mathcal{A}$. The environment initializes a session for each honest (dummy) player, which is modeled by the Init queries sent to the split functionality $s\mathcal{F}_{\mathsf{GPAKE}}$. From this point, the players P_i are ordered according to some order on their index i. For sake of simplicity, we denote them as $P_1, \ldots, P_n$ in all the remaining.

The adversary (from the view of the c_i of the honest players) makes his decision about the subgroups he wants to make for the GPAKE protocol: he sends c_i on behalf of the players he wants to impersonate (they will become corrupted from the beginning of the session). We then define the H sets according to the received $\{c_j\}$: the honest players that have received the same $\{c_j\}$ (possibly modified by the adversary) are in the same subgroup H. The simulator forwards these sets H (which make a partition of all the players) along with the total group to the split functionality. The latter then initializes ideal functionalities with sid_H, for each subgroup H: all the players in the same session received and thus use the same $\{c_j\}$. We can then focus on a specific session $\mathsf{ssid}' = sid_H$ for some set H.

The environment gets back this split, via the dummy players, and then sends the NewSession queries on behalf of the latter, according to the appropriate sid_H. The simulator has to do the same on behalf of the players impersonated by $\mathcal{A}$, but he still does not their password: This is the aim of flow ($1b$).

Next, in flow ($1b$), $\mathcal{S}$ asks $\mathcal{S}_{\mathsf{PAKE}}$ to send the first flow of the 2PAKE protocol on behalf of the honest players (those which have not been impersonated by the adversary in the first flow). The environment then initializes a session for each honest (dummy) player, which is modeled by the Init queries sent to the split functionality $s\mathcal{F}^{MA}_{\mathsf{PAKE}}$. From the view of this flow, the adversary chooses the subgroups (of size 1 or 2) he wants to make for the 2PAKE protocol, and sends this flow on behalf of the players he wants to impersonate (they will become corrupted from the beginning of the session). We then define the H' sets according to the received messages. The simulator forwards these sets H' (which make a partition of all the players) along with the total group to the corresponding instances of the $s\mathcal{F}^{MA}_{\mathsf{PAKE}}$ functionality. They then initializes ideal functionalities $\mathcal{F}^{MA}_{\mathsf{PAKE}}$ with sid_H.

The environment gets back this split, via the dummy players, and then sends the NewSession queries on behalf of the latter, according to the appropriate sid_H. More precisely, if $\mathcal{Z}$ has sent the following NewSession query for P_i to $\mathcal{F}^{MA}_{\mathsf{PAKE}}$, $(\mathsf{NewSession}, \mathsf{sid}_H, P_i, (\mathsf{pw}_i^L, \mathsf{pw}_i^R))$, he sends $(\mathsf{NewSession}, \mathsf{sid}_{H'}, P_i, \mathsf{pw}_i^L)$ and $(\mathsf{NewSession}, \mathsf{sid}_{H''}, P_i, \mathsf{pw}_i^R)$ to the 2PAKE ideal functionalities corresponding respectively to its left or right neighbor.

The simulator asks $\mathcal{S}_{\mathsf{PAKE}}$ to send such NewSession queries on behalf of the players impersonated by $\mathcal{A}$, and he also asks him to construct from them the NewSession query to send the the GPAKE functionality. More precisely, from $(\mathsf{NewSession}, \mathsf{sid}_{H'}, P_i, \mathsf{pw}_i^L)$ and $(\mathsf{NewSession}, \mathsf{sid}_{H''}, P_i, \mathsf{pw}_i^R)$, $\mathcal{S}_{\mathsf{PAKE}}$ constructs $(\mathsf{NewSession}, \mathsf{sid}_H, P_i, (\mathsf{pw}_i^L, \mathsf{pw}_i^R))$ and hands it to $\mathcal{S}$, who can now send it to the GPAKEfunctionality.

Note that if a player is considered honest for the GPAKE execution and dishonest for at leat one 2PAKE execution, it will fail to learn one of its subkeys, say K_i^L, and thus abort during flow ($2a$). Furthermore, the adversary will player on behalf of any player considered dishonest for the GPAKE execution, so that it will necessarily be considered dishonest for the 2PAKE executions. We can thus suppose that the status of the player (honest or corrupted) is the same in both protocols.

Simulator: Main Idea. In a nutshell, the simulation of the remaining of the protocol will either be random or dealt with with the help of $\mathcal{S}_{\mathsf{PAKE}}$. First, the simulator chooses at random the first flow of the protocol, computes honestly the commitment, and he asks $\mathcal{S}_{\mathsf{PAKE}}$ to choose at random the second one. Then, he relies on $\mathcal{S}_{\mathsf{PAKE}}$ again in order to simulate the protocol 2PAKE and learns from it whether the passwords of the players are compatible (they all receive session keys shared with their neighbors) or not (they receive an error) – recall that we assume mutual authentication for the 2PAKE protocol. It would be possible to get rid of this assumption, but the proof would need TesTPwd queries, such as in [5]. He aborts if one of these protocols fails. Otherwise, he chooses values K_i and K_i^L at random, and sets $K_i^R = K_{i+1}^L$.

The key point of the simulation consists in sending coherent X_i^L's and X_i^R's, whose values completely determine the session key. To this aim, we consider two cases. First, if the players are all honest, everything is done honestly. Next, if there are some corrupted players, we will need that the commitments are simulated using the extractability and equivocability trapdoors. This way, the simulator learns the values X_i^L and X_i^R sent by the corrupted players and asks for a GPAKEDelivery key, giving him its value (through a corrupted player). Since there is at least one honest player in the execution, he can thus modify his value K_i, thus modifying also X_i^L and X_i^R, and equivocating the commitment, in order to remain consistent with the commitment already sent. It would be possible to get rid of the equivocability property, but we chose not to impose the simulator to play last. In case of corruption, $\mathcal{S}$ learns the password, and can give everything in a consistent way to the adversary. If a player was corrupted before sending the X_i^L's and X_i^R's, the values sent to the adversary will give him the correct session key. And if no player was corrupted before, they will give him a random session key. This exactly corresponds to the way the functionality deals with corruptions.

If a session aborts or terminates, $\mathcal{S}$ reports it to $\mathcal{A}$. If the session terminates with a session key sk, then $\mathcal{S}$ makes a Key Delivery call to $\widehat{\mathcal{F}}_{GPAKE}$, specifying the session key. But recall that unless enough players are corrupted, $\widehat{\mathcal{F}}_{GPAKE}$ will ignore the key specified by $\mathcal{S}$, and thus we do not have to bother with the key in these cases.

A.3 Description of the Games

Game $\boldsymbol{G}_0$: Game $\boldsymbol{G}_0$ is the real game.

Game $\boldsymbol{G}_1$: From this game on, we allow the simulator to program the common reference string, allowing it to know the extractability trapdoor for the commitment scheme.

Game $\boldsymbol{G}_2$: This game is almost the same as the previous one. The only difference is that $\mathcal{S}$ always extracts the values X_i^L's and X_i^R's committed to by the adversary (without taking advantage of the knowledge for the moment) whenever the latter has corrupted one of the parties. We allow the simulator to abort whenever this extraction fails (this happens when the adversary generates a commitment which is valid for two or more values). Due to the binding property of the commitment, the probability that the adversary achieves such a thing is negligible. This shows that $\boldsymbol{G}_2$ and $\boldsymbol{G}_1$ are indistinguishable. Note that this reduction is linear in the length of the password.

Game $\boldsymbol{G}_3$: In this game, $\mathcal{S}$ now totally programs the CRS. Since it is indistinguishable, this does not change anything compared to the previous game. However, $\mathcal{S}$ now knows the equivocability trapdoor of the commitment scheme and simulate the commitments such that they are both extractable and equivocable. Note that since the commitment is *hiding*, this does not change the view of an environment. In addition, the commitment remains binding (even if the adversary as access to equivocable commitments). Finally, $\boldsymbol{G}_3$ and $\boldsymbol{G}_2$ are indistinguishable.

Game $\boldsymbol{G}_4$: In this game, we reject any signed-authenticator that is non-oracle-generated whereas the player is still honest: the adversary does not know the signing key, and thus cannot correctly sign the authenticator. One can easily show that a difference between $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ would lead to an attack against the one-time signature scheme.

Game $\boldsymbol{G}_5$: In this game, we formally modify the way we simulate the honest players. The simulator does not know their passwords anymore. In round 1, $\mathcal{S}$ generates honestly a pair $(\mathrm{SK}_i, \mathrm{VK}_i)$ of signature and verification keys, and computes honestly the associated commitment c_i. He then runs the simulator $\mathcal{S}_{\mathsf{PAKE}}$, asking him to generate the messages sent in flows $(1b)$. He finally opens

honestly the commitments, checks them, and aborts if one of them is incorrect. Any corruption up to this moment can be dealt with easily, with the help of $\mathcal{S}_{\mathsf{PAKE}}$ for the 2PAKE part, and by giving the honestly-computed values for the rest. We now face two cases: If all the flows were oracle-generated up this moment, we require $\mathcal{S}$ to simulate the end of the execution of the protocol on behalf of all the players. Otherwise, he simply follows the protocol as above (we show in the following game how to deal with this case.

At the beginning of $(2a)$, before executing the 2PAKE protocols, the players are thus all supposed honest. $\mathcal{S}$ asks $\mathcal{S}_{\mathsf{PAKE}}$ to simulate these executions on behalf of all the players. In particular, he asks PAKESamePwd queries, learning whether the 2PAKE succeeded or not. In the latter case, the players abort the game (soon followed by the other players, when the latter will not receive some c'_j's values). If the 2PAKE succeeded, we face two cases.

If there was no corruption during the 2PAKE protocols, the simulator chooses at random the values K_i and K_i^L on behalf of each player, setting $K_i^R = K_{i+1}^L$. The computation of the commitments c'_i is then done honestly. If there was a corruption during the 2PAKE protocols, $\mathcal{S}$ learns the password of the player concerned and forwards the corruption query as well as the password to $\mathcal{S}_{\mathsf{PAKE}}$, who deals with this corruption, handing to $\mathcal{S}$ enough information to provide $\mathcal{A}$ with consistent data. Note that $\mathcal{S}_{\mathsf{PAKE}}$ has to deal with two corruptions, since each GPAKE player plays into two different 2PAKE executions. $\mathcal{S}$ then asks two PAKEDelivery queries for this player, learning K_i^L and K_i^R. He sets $K_{i+1}^L = K_i^R$ and everything continues honestly as before. Note that the non-malleability of the commitments ensures that the distribution of the X_i^L's and X_i^R's is independent. We then again face two cases.

If a corruption occurs before $(2b)$, $\mathcal{S}$ recovers the password of the player, and provides the adversary with the values K_i and K_i^L chosen at random (these values are indistinguishable). In this case (or if a corruption had already occurred before), then the simulator extracts all the values X_j^L's and X_j^R's from the adversary's commitments. He asks a query GPAKEDelivery for a corrupted player and learns the session key sk_0 (since there is at least one honest player, the adversary cannot set the key). As we assume at least one player, say P_{i_0}, is honest, $\mathcal{S}$ can change its value K_{i_0}, using $g_{i_0}(\{K_i, i \neq i_0\}, \mathsf{sk}_0 \| \mathsf{sk}_1)$ for a random sk_1, so that the random extraction leads to sk_0 as session key and sk_1 as MAC key. Note that the probability that the players obtain the same sk_1 without having the same sk_0 is negligible. He then equivocates the commitments sent, in order that they correspond to the new values $X_{i_0}^L$ and $X_{i_0}^R$ he has to open to in the next flow. Everything else is computed honestly. This way, if a player gets corrupted afterwards (even P_{i_0}), $\mathcal{S}$ will be able to give consistent data to the adversary. Finally, in round (4), $\mathcal{S}$ sets b to no for the players receiving a non-oracle-generated flow in round (3), and to yes for the others.

Otherwise, if no corruption occurs before $(2b)$, $\mathcal{S}$ has no means to learn the key: He thus asks all the GPAKEDelivery queries for the players (giving him no information) and executes everything honestly. A corruption is dealt with as before, except that the data given to the adversary will not lead to the key chosen by the functionality, but this corresponds exactly to the corruption query of the functionality: Since the key has been sent to the player (thanks to the GPAKEDelivery query), the adversary does not learn it. Finally, $\boldsymbol{G}_5$ and $\boldsymbol{G}_4$ are indistinguishable.

Game $\boldsymbol{G}_6$: In this game, we consider the case where some corruptions occurred before the 2PAKE protocols. Such a case is easily dealt with as the case where corruptions occur during the 2PAKE protocols, as we showed in the previous game. Thus, this game is indistinguishable from $\boldsymbol{G}_5$.

Game $\boldsymbol{G}_7$: This game is almost the same as the previous one, except that we formalize the behavior of the simulator by introducing the queries to the functionality, in place of the PAKESamePwd, PAKEDelivery, GPAKESamePwd and GPAKEDelivery-queries. More precisely, we only replace the hybrid queries with their ideal equivalents. Informally, $\mathcal{S}$ behaves not according to the messages

sent, but to the messages received (probably modified by the adversary). In round 1, $\mathcal{S}$ (using $\mathcal{S}_{\mathsf{PAKE}}$) sends random values (honestly committed to) on behalf of each non-corrupted player. For round 2 and round 3, see games $\boldsymbol{G}_5$ and $\boldsymbol{G}_6$. In round 4, $\mathcal{S}$ sets b to no for the players receiving a non-oracle generated flow in round 3, and to yes for the others. If a session aborts or terminates, then $\mathcal{S}$ reports it to $\mathcal{A}$. We now show that $\boldsymbol{G}_7$ is indistinguishable from the ideal game. Say that the players have matching sessions if they share the same ssid' (which implies that they share the same VK_i and due to the use of the split functionality).

It is clear that players sharing compatible password will obtain a random key, both in $\boldsymbol{G}_7$ (from $\boldsymbol{G}_5$) and IWE (except for players receiving non-oracle generated flows, modeled by the bit b). This key will be not chosen by the adversary unless all players are corrupted (since there will always be an honest player, whose K_i is unpredictable). Finally, if the players do not share compatible passwords, they will receive an error, thanks to the mutual authentication of $\mathsf{2PAKE}$. Now, we need to show that two players will receive the same key in $\boldsymbol{G}_7$ if and only if it happens in IWE.

This is clearly the case for players with matching session (with or without the same password). This follows from $\boldsymbol{G}_5$ in the real world, and from the $\mathsf{NewSession}$ queries mentioning the same group of players in the ideal world. Finally, consider the case of players with no matching sessions. It is clear that in $\boldsymbol{G}_7$ the session keys of those players will be independent because they are not set in any of the games. In IWE, the only way that they receive matching keys is that the functionality receives two $\mathsf{NewSession}$ queries with the same ssid' and a group where all players share the same passwords.

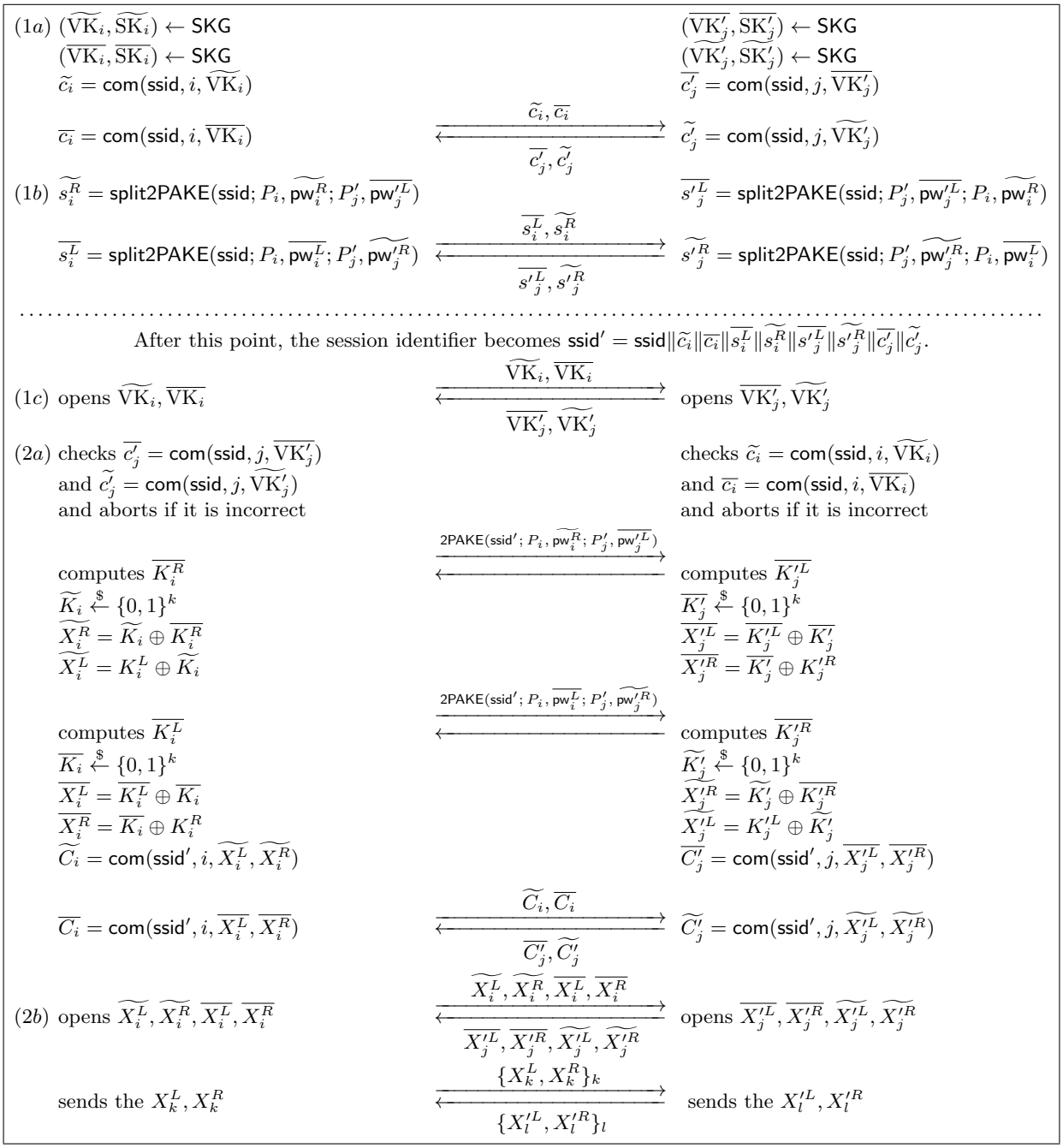

Fig. 7. Description of the protocol for merging the group $P_1, \ldots, P_n$, represented by P_i with password pw, and the group $P'_1, \ldots, P'_m$, represented by P'_j with password pw' (first part)

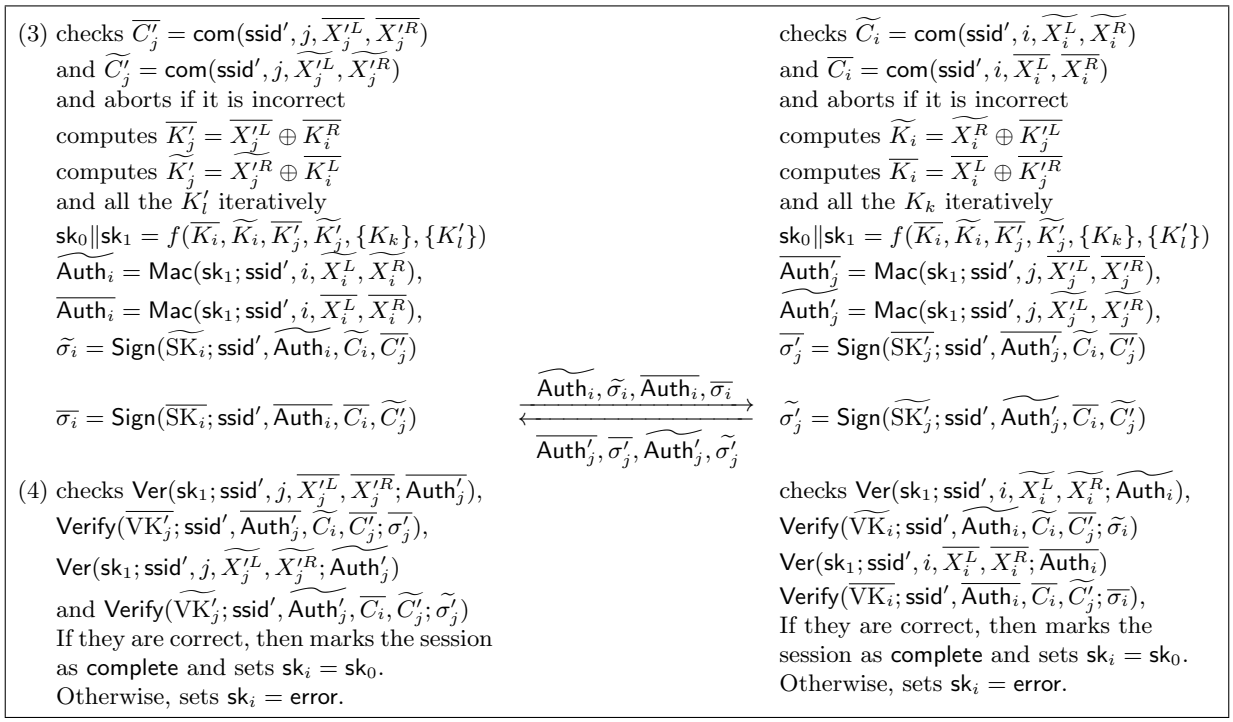

Fig. 8. Description of the protocol for merging the group $P_1, \ldots, P_n$, represented by P_i with password pw, and the group $P'_1, \ldots, P'_m$, represented by P'_j with password pw' (second part)