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 [14], 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 [18], the authors of [20] show how their new model (based on the ideal functionality $\mathcal{F}_{\mathsf{pwKE}}$) relates to previous PAKE models, such as [12] or [8]. 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 [12]. Other works in the UC framework include [24] and [26], where the authors study static corruptions without random oracles as well.

* Work done while being at Télécom ParisTech, Paris, France.

A. Kiayias (Ed.): CT-RSA 2011, LNCS 6558, pp. 142–160, 2011.

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.*, [9,15,5]) and those whose number of communication rounds depends on the number of users involved in the protocol execution (*e.g.*, [16]). 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 [20]. Like [5], our protocol achieves a strong 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.* [10], 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 [17] 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 [25,5], but they either relied on idealized models [5] or were not applicable to the password-based scenario [25].

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 [18,20] for details. The model considered in this paper is the UC framework with joint state proposed by Canetti and Rabin [21] (the CRS will be in the joint state).

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

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 $s\mathcal{F}$

case in which they interact with each other from the case where they interact with the adversary. The authors of [10] 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 [20] 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 [27,23], 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 [7], 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 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 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 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

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 one and there is 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 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 $(\mathsf{sid}, \mathsf{error})$ 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 $\mathcal{F}_{\mathsf{PAKE}}^{MA}$

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 [28]: 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 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 [20].

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 [20], 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.

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

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

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 [28]). 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 [28]). 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 [28], 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.

4 Scheme

Intuition. The main idea of our protocol is to apply the Burmester-Desmedt technique [17] 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 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 $\mathcal{F}_{\mathsf{GPAKE}}$

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

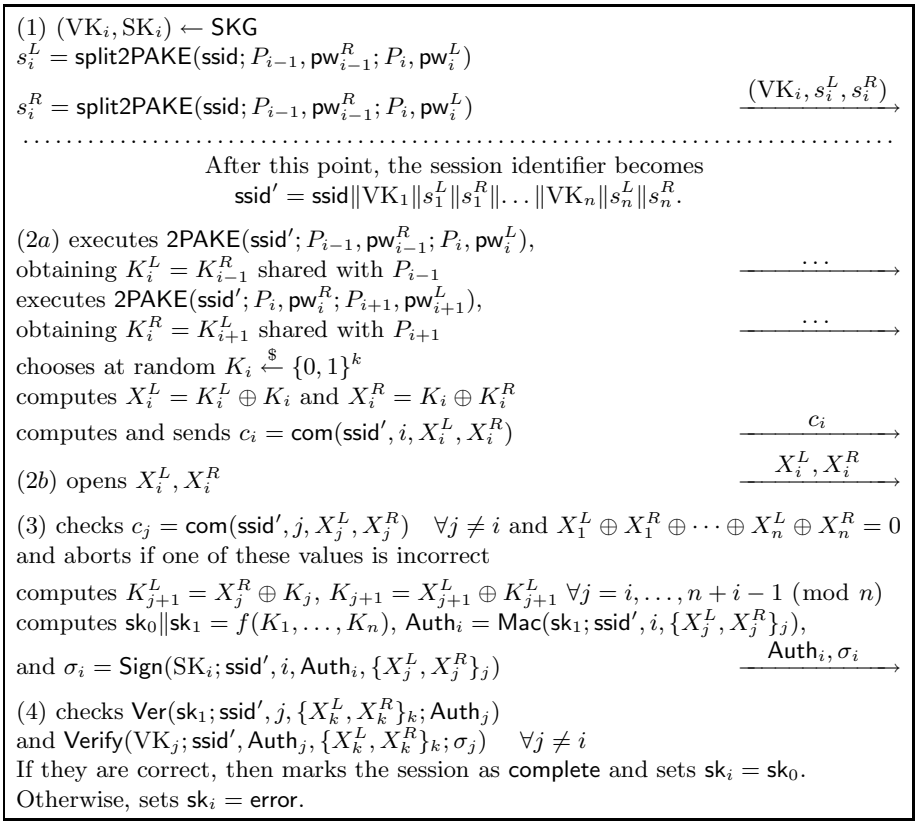

Fig. 4. Description of the protocol for player P_i, with index i and passwords pw_i^L and pw_i^R

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.

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 [11].

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 [22], 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

$$\begin{array}{ll} (\alpha_1, \ldots, \alpha_n, f(\alpha_1, \ldots, \alpha_n; h_1, \ldots, h_n)), & (\alpha_1, \ldots, \alpha_n) \stackrel{\$}{\leftarrow} \mathbb{F}^{*n}, (h_1, \ldots, h_n) \leftarrow \mathcal{D}_i \\ (\alpha_1, \ldots, \alpha_n, U), & (\alpha_1, \ldots, \alpha_n) \stackrel{\$}{\leftarrow} \mathbb{F}^{*n}, U \stackrel{\$}{\leftarrow} \mathbb{F} \end{array}$$

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 [29], 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 [19,1,7]. 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:

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

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 P_i doesn't match with the expected value, it aborts, setting the key $\mathsf{sk}_i = \mathsf{error}$. In particular, every player 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 the full version [6]).

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 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, and that 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 5, 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).

For lack of space, we refer the interested reader to the full version [6] for the precise description of the protocol. 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 5, 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

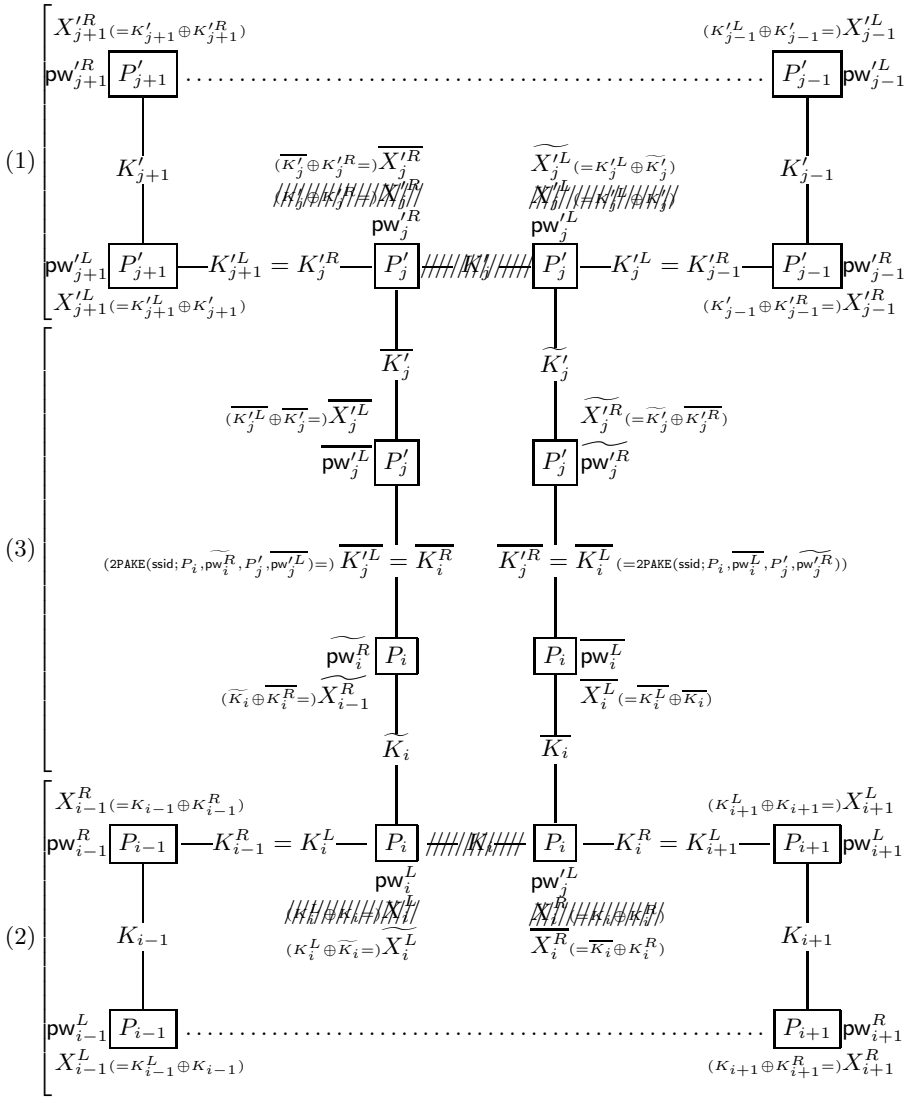

Fig. 5. Merging two Groups: (1) represents the former group $(P'_1, P'_2, \ldots, P'_m)$; (2) represents the former group $(P_1, P_2, \ldots, P_n)$; (3) is the proper merge protocol, between the inviter P_i and the invited P'_j

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 5). 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 5, letting the player P_{i-1} with a new value X^R_{i-1} equal to $X^R_{i-1} \oplus \widetilde{X^L_i}$ and the player P'_j with a new value $\overline{X'^L_j}$ equal to $\overline{X'^L_j} \oplus \widetilde{X^R_i}$. The new 2PAKE-value shared between them is $\widetilde{K_i}$. The same thing happens on the right part of (3) in Figure 5: P'_j vanishes, letting the player P'_{j-1} with the new value X'^R_{j-1} equal to $X'^R_{j-1} \oplus \widetilde{X'^L_j}$ and P_i with the new value $\overline{X^L_i}$ equal to $\widetilde{X'^R_j} \oplus \overline{X^L_i}$. The new 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 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. First, the left part of (3) in Figure 5 without P_i:

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

with $\widetilde{K_i}, \overline{K'_j} \xleftarrow{\$} \{0,1\}^k$, $\overline{X'^R_j} = \overline{K'_j} \oplus K'^R_j$ and $X'^L_{j+1} = K'^L_{j+1} \oplus K'_{j+1}$. Then, the right part of (3) in Figure 5 without P'_j (with $\widetilde{K'_j}, \overline{K_i} \xleftarrow{\$} \{0,1\}^k$, $\overline{X^R_i} = \overline{K_i} \oplus K^R_i$ and $X^L_{i+1} = K^L_{i+1} \oplus K_{i+1}$):

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

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. These changes, presented in the full version [6] for lack of space, are two-fold: First, to obtain the contributiveness, we impose to each player of both groups to participate in the later phases of the protocol, issuing a new fresh value K_k or K'_ℓ; Second, in order to achieve forward secrecy, we change the way in which we compute the local key values (all the K's used by a user) by using the initial ones as the seed or state of a forward-secure stateful pseudorandom generator [13] and then use this state to generate the actual K's values, as well as the next state.

6 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 in the full version [6], 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 PAMPA Project.

References

1. Abdalla, M., Bohli, J.-M., González Vasco, M.I., Steinwandt, R.: (Password) authenticated key establishment: From 2-party to group. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 499–514. Springer, Heidelberg (2007)
2. Abdalla, M., Boyen, X., Chevalier, C., Pointcheval, D.: Distributed public-key cryptography from weak secrets. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 139–159. Springer, Heidelberg (2009)
3. Abdalla, M., Boyen, X., Chevalier, C., Pointcheval, D.: Strong cryptography from weak secrets: Building efficient pke and ibe from distributed passwords in bilinear groups. In: Bernstein, D.J., Lange, T. (eds.) AFRICACRYPT 2010. LNCS, vol. 6055, pp. 297–315. Springer, Heidelberg (2010)
4. Abdalla, M., Catalano, D., Chevalier, C., Pointcheval, D.: Efficient two-party password-based key exchange protocols in the UC framework. In: Malkin, T.G. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 335–351. Springer, Heidelberg (2008)
5. Abdalla, M., Catalano, D., Chevalier, C., Pointcheval, D.: Password-authenticated group key agreement with adaptive security and contributiveness. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 254–271. Springer, Heidelberg (2009)
6. Abdalla, M., Chevalier, C., Granboulan, L., Pointcheval, D.: Contributory Password-Authenticated Group Key Exchange with Join Capability. In: Kiayias, A. (ed.) CT-RSA 2011. LNCS, vol. 6558, pp. 142–160. Springer, Heidelberg (2011); Full version available from the web page of the authors
7. Abdalla, M., Chevalier, C., Pointcheval, D.: Smooth projective hashing for conditionally extractable commitments. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 671–689. Springer, Heidelberg (2009)
8. Abdalla, M., Fouque, P.-A., Pointcheval, D.: Password-based authenticated key exchange in the three-party setting. In: Vaudenay, S. (ed.) PKC 2005. LNCS, vol. 3386, pp. 65–84. Springer, Heidelberg (2005)
9. Abdalla, M., Pointcheval, D.: A scalable password-based group key exchange protocol in the standard model. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 332–347. Springer, Heidelberg (2006)
10. Barak, B., Canetti, R., Lindell, Y., Pass, R., Rabin, T.: Secure computation without authentication. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 361–377. Springer, Heidelberg (2005)

11. Bellare, M., Kilian, J., Rogaway, P.: The security of the cipher block chaining message authentication code. Journal of Computer and System Sciences 61(3), 362–399 (2000)
12. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 139–155. Springer, Heidelberg (2000)
13. Bellare, M., Yee, B.S.: Forward-security in private-key cryptography. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 1–18. Springer, Heidelberg (2003)
14. Bellovin, S.M., Merritt, M.: Encrypted key exchange: Password-based protocols secure against dictionary attacks. In: 1992 IEEE Symposium on Security and Privacy, May 1992, pp. 72–84. IEEE Computer Society Press, Los Alamitos (1992)
15. Bohli, J.-M., Vasco, M.I.G., Steinwandt, R.: Password-authenticated constant-round group key establishment with a common reference string. Cryptology ePrint Archive, Report 2006/214 (2006)
16. Bresson, E., Chevassut, O., Pointcheval, D.: Dynamic group Diffie-Hellman key exchange under standard assumptions. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 321–336. Springer, Heidelberg (2002)
17. Burmester, M., Desmedt, Y.: A secure and scalable group key exchange system. Information Processing Letters 94(3), 137–143 (2005)
18. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS, October 2001, pp. 136–145. IEEE Computer Society Press, Los Alamitos (2001)
19. Canetti, R., Fischlin, M.: Universally composable commitments. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 19–40. Springer, Heidelberg (2001)
20. Canetti, R., Halevi, S., Katz, J., Lindell, Y., MacKenzie, P.D.: Universally composable password-based key exchange. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 404–421. Springer, Heidelberg (2005)
21. Canetti, R., Rabin, T.: Universal composition with joint state. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 265–281. Springer, Heidelberg (2003)
22. Carter, L., Wegman, M.N.: Universal classes of hash functions. J. Comput. Syst. Sci. 18(2), 143–154 (1979)
23. Gennaro, R., Lindell, Y.: A framework for password-based authenticated key exchange. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 524–543. Springer, Heidelberg (2003)
24. Gentry, C., MacKenzie, P., Ramzan, Z.: A method for making password-based key exchange resilient to server compromise. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 142–159. Springer, Heidelberg (2006)
25. Choudary Gorantla, M., Boyd, C., Nieto, J.M.G.: Universally composable contributory group key exchange. In: ASIACCS 2009, March 2009, pp. 146–156. ACM Press, New York (2009)
26. Groce, A., Katz, J.: A new framework for efficient password-based authenticated key exchange. In: ACM CCS 2010. Springer, Heidelberg (2010)
27. Katz, J., Ostrovsky, R., Yung, M.: Efficient password-authenticated key exchange using human-memorable passwords. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 475–494. Springer, Heidelberg (2001)
28. Katz, J., Shin, J.S.: Modeling insider attacks on group key-exchange protocols. In: ACM CCS 2005, November 2005, pp. 180–189. ACM Press, New York (2005)
29. Shoup, V.: A Computational Introduction to Number Theory and Algebra. Cambridge University Press, Cambridge (2005)