Distributed Public-Key Cryptography from Weak Secrets

Michel Abdalla [1], Xavier Boyen [2], Céline Chevalier [1], and David Pointcheval [1]

[1] École Normale Supérieure, CNRS-INRIA, Paris, France
[2] Stanford University, Stanford, California

Abstract. We introduce the notion of distributed password-based public-key cryptography, where a virtual high-entropy private key is implicitly defined as a concatenation of low-entropy passwords held in separate locations. The users can jointly perform private-key operations by exchanging messages over an arbitrary channel, based on their respective passwords, without ever sharing their passwords or reconstituting the key.
Focusing on the case of ElGamal encryption as an example, we start by formally defining ideal functionalities for distributed public-key generation and virtual private-key computation in the UC model. We then construct efficient protocols that securely realize them in either the RO model (for efficiency) or the CRS model (for elegance).
We conclude by showing that our distributed protocols generalize to a broad class of "discrete-log"-based public-key cryptosystems, which notably includes identity-based encryption. This opens the door to a powerful extension of IBE with a virtual PKG made of a group of people, each one memorizing a small portion of the master key.

1 Introduction

Traditional wisdom says that it is impossible to do public-key cryptography from short passwords. This is because any low-entropy private key will quickly succumb to an off-line dictionary attack, made possible by the very publication of the public key, which can thus be used as a non-interactive test function. Since off-line attacks are very effective against weak secrets, it is imperative that the private keys in public-key systems be highly random and complex, but that makes them hopelessly impossible to be remembered by humans.

But, what if, instead of being held as an indivisible entity, the private key were chopped into many little pieces, each one of them independently memorized by a different person in a group of friends or colleagues? The components of the key would be safe in the respective memories of the individual group members, at least as long as it is not used. The only complication is the need to reassemble the full private key from the various components, so that private-key operations can be performed. Naturally, the secret holders should not actually reassemble the key, but instead perform a distributed computation of whichever private-key operation they need, without ever having to meet or even reconstitute the key.

Unusual Requirements. Even if one can perform private-key computations without reassembling the key, there are other, more subtle vulnerabilities.

For starters, we cannot simply assume that the (virtual) private key is simply made of some number of random components (one per user) generated independently and uniformly at random. On the contrary, we must assume that the various components are arbitrary and possibly correlated, and some of them potentially very weak and easily guessable. This is because of our requirement of human-memorability: for the components to be truly memorable, it is imperative that their respective owners choose them in whichever way they please.

A consequence of the above is that it also opens the possibility of *password reuse* by the various users: although this is a bad security practice that should be discouraged, it is also one that is very common and that we should acknowledge and handle the best way we can, rather than pretend that it will not happen.

Additionally, since the various secret holders do not necessarily trust each other, it is necessary that they be able to choose their individual secrets in complete privacy. In fact, any solution to our question must

deal with user corruptions and collusions, and remain as secure as the "sum total" of the key components of the remaining honest users.

Finally, we must have a notion of "group leader", which is the person who will actually "own" the distributed virtual private key. By "own", we mean that only the group leader will be able to use that key, i.e., obtain the results of any private computation based on it, with the help of the other group members. We stress that neither the leader nor anyone else should actually learn the key itself.

An important difference between our requirements and essentially all existing distributed protocols that deal with weak secrets (such as Group Password-based Key Agreement), is that here the secrets are chosen arbitrarily and privately by each user. We neither assume that all the secrets are the same (as in Group PAKE), or that they are all independent (as in Threshold Cryptography). The whole system should thus: (1) not fall apart if some of the passwords become exposed, as long as the combined entropy of the uncompromised passwords remains high; (2) preserve the privacy of all uncompromised passwords at all stages of the process (during the initial computation of the public key and any subsequent utilization of the virtual private key).

The notion of group leader is something necessary for our application. Most password-based protocols seek to achieve a *symmetric* outcome Here, by contrast, the impetus to create a public/private key pair must originate in a particular user, who will become the leader, and who seeks the help of other, semi-trusted individuals to help him or her remember the key. (The leader can return the favor later or share the result of any private computation, outside of the core protocol.) Remark also that whereas it is easy for the leader to share the result of a private computation with the other members, it would be almost impossible to restrict such result to the leader if the computation gave the result to all.

General Approach. The aim of this paper is thus primarily to show how to do asymmetric cryptography from a distributed set of human-memorable secrets. Since public-key cryptography from *single* passwords is irremediably insecure, the best we can hope for is to base it on moderately-sized *distributed collections* of them: Given a regular system (such as signature, encryption, or IBE), we devise a pair of protocols that take independent user passwords as inputs, and, in a distributed manner: 1) generate a publishable public key that corresponds to the set of passwords; 2) do private computations on the virtual private key.

To create a key pair, a group of players led by a designated "group leader" engages in the distributed key generation protocol. The protocol runs over unauthenticated channels, and if all goes well, results in an explicit public key for anyone to see and use. The private key is not explicitly computed and remains implicitly defined by the set of passwords. To use the private key, the same group of players engages in another protocol, using the same passwords as in the key generation protocol. The protocol again runs over unauthenticated channels. If all goes well, the leader, and only the leader, obtains the results of the computation. Again, the private key is not explicitly computed, and the passwords remain private to their respective owners.

Unlike *regular public-key cryptosystems*, the private key is never stored or used all at once; it remains virtual and delocalized, and the private-key operation is done using an interactive protocol. But unlike *threshold cryptography*, where the shares are uniformly randomized and typically as long as the shared secret itself, here the passwords are arbitrary and user-selected. Unlike *password-based encryption*, off-line attacks are thwarted by virtue of the high joint entropy from many distinct user passwords, which must be guessed all at once. On-line attacks against single passwords cannot be prevented, but are very slow as they require an on-line commitment for each guess. Unlike *password-authenticated key exchange protocols*, here the user passwords are not the same or even related to each other: the passwords are truly personal.

Our Results. First, we formalize this class of protocols and their security requirements; for convenience we do so in the UC model [12], which lends itself nicely to the analysis of password-based protocols. Second, we propose a reasonably efficient construction for the ElGamal cryptosystem as a working example [19], which we prove secure both in the RO and CRS models. Third, we conclude by showing that

our construction generalizes easily to a broad class of "discrete-log"-type public-key schemes, and, quite notably, the whole set of schemes derived from the BF and BB identity-based cryptosystems [9, 7].

Even though for simplicity we focus on public-key systems with a special form (those that operate by raising elements of an algebraic group to the power of the private key and/or ephemeral exponents), this structure is general enough to capture many examples of exponentiation-based cryptosystems, and even IBE systems that require a pairing, as we just mentioned.

Remarkably, and of independent interest, this gives us an interesting twist on the notion of IBE, where the "central" key generation authority is replaced by a distributed set of users, each one of them holding a small piece of the master secret in the form of a self-selected easily memorable short password.

Related Work. Although there is no prior work on distributed cryptography from weak secrets *proper*, this notion is of course related to a fairly large body of literature that includes Password-Authenticated Key Exchange (PAKE) and Multi-Party Computation (MPC).

Multi-Party Computation. The first and most famous MPC protocol is due to Yao [31]. Depending on the setup, such protocols allow two participants with secret inputs to compute a public function of their joint inputs, without leaking anything other than the output of the function [25, 24, 6, 16]. MPC protocols typically assume all communications between the players to be authentic: that is, an external mechanism precludes modifications or fake message insertions. The flip side is that such protocols tend to become insecure when the number of dishonest players reaches a certain threshold that allows them to take over the computation and from there recover the other players' inputs [29, 2, 26].

Several works have dealt with the case of MPC over unauthenticated channels [13, 20, 1], by prefacing the multi-party computation proper with some flavor of authentication based on non-malleable commitments or signatures [18]. The work of Barak *et al.* [1] in particular gives general conditions of what can and cannot be achieved in unauthenticated multi-party computations: they show that an adversary is always able to partition the set of players into disjoint "islands" that end up performing independent computations, but nothing else besides dropping messages and/or relaying them faithfully. They show how to transform any (realization of an) UC functionality into a multi-party version of the same that merely lets the adversary split the players into disjoint islands. They also show how to build password-based group key agreement (GPAKE) from this notion, first by creating a random session key for the group by running an MPC protocol without authentication, and then by verifying that all players have the same key using a "string equality" functionality. (By comparison, here, we force the users to commit to their passwords first, and then perform the actual computation based on those commitments.)

Although it is clear that, like so many other things in cryptography, our work can be viewed as a special case of unauthenticated MPC, our contribution lies not in this obvious conceptual step, but in the specification of suitable functionalities for the non-trivial problem of password-based threshold cryptography (and their efficient implementation). In particular, much grief arises from our requirement that each user has its *own* password (which may even be reused in other contexts), instead of a single common password for the whole group as in the applications considered in [1] and elsewhere.

On-line Passwords. The first insight that weak passwords could be used on-line (in a key exchange protocol) with relative impunity was made in [5]. It captured the idea that the success of an adversary in breaking the protocol should be proportional to the number of times this adversary interacts with the server, and only negligibly in its off-line computing capabilities.

In the password-only scenario (without public-key infrastructure), the first protocols with a proof of security appeared contemporaneously in [11] and [3], both in the random-oracle model. A (somewhat inefficient) protocol without any setup assumption was first proposed in [23]. A fairly efficient one in the common random string model was first given in [27] and generalized in [22].

To cope with concurrent sessions, the work of [14] was the first to propose an ideal functionality for PAKE in the UC model, as well as a protocol that securely realizes it. Unlike previous models, one of the major advantages of the UC one is that it makes no assumption on the distribution of the passwords; it

also considers, for instance, some realistic scenarios such as participants running the protocol with different but possibly related passwords.

2 Security Model

The UC Framework. Throughout this paper, we assume basic familiarity with the universal composability (UC) framework [12]. See Appendix A for a short introduction of some UC notions we shall use in this work.

Split Functionalities. Without any strong authentication mechanisms, the adversary $\mathcal{A}$ can always partition the players into disjoint subgroups and execute independent sessions of the protocol with each one, 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 $\mathcal{A}$. The authors of [1] addressed this issue by proposing a new model based on *split functionalities* which guarantees that this attack is the only one available to $\mathcal{A}$.

The split functionality is a generic construction based upon an ideal functionality: Its description can be found on Figure 1. In the initialization stage, the adversary $\mathcal{A}$ adaptively chooses disjoint subsets of the honest parties (with a unique session identifier that is fixed for the duration of the protocol). During the computation, each subset H activates a separate instance of the functionality $\mathcal{F}$. All these functionality instances are independent: The executions of the protocol for each subset H can only be related in the way $\mathcal{A}$ chooses the inputs of the players it controls. The parties $P_i \in H$ provide their own inputs and receive their own outputs, whereas $\mathcal{A}$ plays the role of all the parties $P_j \notin H$.

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 $(\mathsf{Init}, sid, P_i, H, sid_H)$ from $\mathcal{A}$, where H is a set 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 paire (H, sid_H), send $(\mathsf{Init}, sid, sid_H)$ to P_i, and invoke a new functionality $(\mathcal{F}, sid_H)$ denoted as $\mathcal{F}_H$ and with set of 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.
- 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 the sequel, as we describe our two general functionalities $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ and $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$, one has to keep in mind that an attacker controlling the communication channels can always choose to view them as the split functionalities $s\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ and $s\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ implicitly consisting of multiple instances of $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ and $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ for non-overlapping subsets of the original players. Furthermore, one cannot prevent $\mathcal{A}$ from keeping some flows, which will never arrive. This is modelled in our functionalities (Figures 2 and 3) by a bit b, which specifies whether the flow is really sent or not.

The Ideal Functionalities. In the sequel we denote by n the number of users involved in a given execution of the protocol. One of the users plays a particular role and is denoted as the *group leader*, the others are simply denoted as players. Groups can be formed arbitrarily. Each group is *defined* by its leader (who "owns" the group by being the one to receive the result of any private computation) and an arbitrary number of other players in a specific order (who "assist" and "authorize" the leader in his or her use of the group's virtual key).

We stress that the composition and ordering of a group is what defines it and cannot be changed: this ensures that any third-party who uses the group's public key knows exactly how the corresponding private key will be accessed. If another player wants to be the leader, he or she will have to form a new group. (Even though such new group may contain the same set of members with possibly unchanged passwords, the two groups will be distinct and have different incompatible key pairs because of the different ordering).

As in [14], the functionality is not in charge of providing the passwords to the participants. The passwords are chosen by the environment which then hands them to the parties as inputs. This guarantees security even in the case where a honest user executes the protocol with an incorrect password: This models, for instance, the case where a user mistypes its password. It also implies that the security is preserved for all password distributions (not necessarily the uniform one) and in all situations where related passwords are used in different protocols.

Since the functionalities are intended to capture distributed password protocols for (the key generation and private-key operation of) an arbitrary public-key primitive, we will represent all the primitive's algorithms as black box parameters in our definitions. In general, we shall require: a function SecretKeyGen to combine a vector of passwords into a single secret key; a function PublicKeyGen to compute from a password vector a matching public key; a predicate PublicKeyVer to verify such public key against any password vector: this is important for the correctness of the ideal functionalities, but it also simplifies the use of the joint-state UC Theorem since it abstracts away the passwords that then do not need to be considered as part of the joint data; a function PrivateComp to perform the operation of interest using the private key: this could be the decryption function Dec of a public-key encryption scheme, the signing function Sign in a signature scheme, or the identity-based key extraction function Extract in an IBE system.

Both functionalities start with an initialization step, which basically waits for all the users to notify their interest in computing a public key or performing a private computation, as the case may be. Such notification is provided via newSession queries (containing the session identifier sid of the instance of the protocol, the user's identity P_i, the identity of the group Pid, the user's password pw_i, and when computing the private function, a public key pk and input in) sent by the players or by the simulator $\mathcal{S}$ in case of corruptions during the first flow (corresponding to the split functionality). Once all the users (sharing the same sid and Pid) have sent their notification message, the functionality informs the adversary that it is ready to proceed.

In principle, after the initialization stage is over, the eligible users are ready to receive the result. However the functionality waits for $\mathcal{S}$ to send a compute message before proceeding. This allows $\mathcal{S}$ to decide the exact moment when the key should be sent to the users 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 [28]). Also, although in the key generation functionality all users are normally eligible to receive the public key, in the private computation functionality it is important that only the group leader receives the output (though he may choose to reveal it afterwards to others, outside of the protocol, depending on the application).

The Distributed Key Generation Functionality (Figure 2). The aim of this functionality is to provide a public key to the users, computed according to their passwords with respect to the previously mentioned function PublicKeyGen given in parameter, and it ensures that the group leader never receives an incorrect key in the end, whatever does the adversary. The protocol starts with an initialization phase as already described, followed by a key computation phase triggered by an explicit key computation query (so that $\mathcal{S}$ can control its timing.)

After the key is computed, the adversary can choose whether the group leader indeed receives this key. If delivery is denied, then nobody gets the key, and it is as if it was never computed. If delivery is allowed, then the group leader and $\mathcal{S}$ both receive the public key. This behavior captures the fact that the generated public key is intended to be available to all, starting with the opponent. (More to the point, this requirement will also weed out some bogus protocols that could only be secure if the public key remained

The functionality $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ is parametrized by a security parameter k and an efficiently computable function $\mathsf{PublicKeyGen} : (\mathsf{pw}_1, \mathsf{pw}_2, \ldots, \mathsf{pw}_n) \mapsto \mathsf{pk}$ that derives a public key pk from a set of passwords. Denote by *role* either *player* or *leader*. The functionality interacts with an adversary $\mathcal{S}$ and a set of parties $P_1, \ldots, P_n$ via the following queries:

Initialization. Upon receiving a query $(\mathsf{newSession}, sid, Pid, P_i, \mathsf{pw}_i, role)$ from user P_i for the first time, where Pid is a set of at least two distinct identities containing P_i, ignore it if $role = leader$ and if there is already a record of the form $(sid, Pid, *, *, leader)$. Record $(sid, Pid, P_i, \mathsf{pw}_i, role)$ and send $(sid, Pid, P_i, role)$ to $\mathcal{S}$. Ignore any subsequent query $(\mathsf{newSession}, sid, Pid', *, *, *)$ where $Pid' \neq Pid$.

If there are already $|Pid| - 1$ recorded tuples $(sid, Pid, P_j, \mathsf{pw}_j)$ for $P_j \in Pid \setminus \{P_i\}$, and exactly one of them such that $role = leader$, then while recording the $|Pid|$-th tuple, also record $(sid, Pid, \mathsf{ready})$ and send this to $\mathcal{S}$. Otherwise, record $(sid, Pid, \mathsf{error})$ and send $(sid, Pid, \mathsf{error})$ to $\mathcal{S}$.

Key Computation. Upon receiving a message $(\mathsf{compute}, sid, Pid)$ from the adversary $\mathcal{S}$ where there is a recorded tuple $(sid, Pid, \mathsf{ready})$, then compute $\mathsf{pk} = \mathsf{PublicKeyGen}(\mathsf{pw}_1, \ldots, \mathsf{pw}_n)$ and record (sid, Pid, pk).

Leader Key Delivery. Upon receiving a message $(\mathsf{leaderDeliver}, sid, Pid, \mathsf{b})$ from the adversary $\mathcal{S}$ for the first time, where there is a recorded tuple (sid, Pid, pk) and a record $(sid, Pid, P_i, pw_i, leader)$, send (sid, Pid, pk) to P_i and to $\mathcal{S}$ if $\mathsf{b} = 1$, or $(sid, Pid, \mathsf{error})$ otherwise. Record $(sid, Pid, \mathsf{sent})$ and send this to $\mathcal{S}$.

Player Key Delivery. Upon receiving $(\mathsf{playerDeliver}, sid, Pid, \mathsf{b}, P_i)$ from the adversary $\mathcal{S}$ where there are recorded tuples (sid, Pid, pk), $(sid, Pid, P_i, pw_i, player)$ and $(sid, Pid, \mathsf{sent})$, send (sid, Pid, pk) to P_i if $\mathsf{b} = 1$, or $(sid, Pid, \mathsf{error})$ otherwise.

User Corruption. If $\mathcal{S}$ corrupts $P_i \in Pid$ where there is a recorded tuple $(sid, Pid, P_i, \mathsf{pw}_i)$, then reveal pw_i to $\mathcal{S}$. If there also is a recorded tuple (sid, Pid, pk) and if (sid, Pid, pk) has not yet been sent to P_i, send (sid, Pid, pk) to $\mathcal{S}$.

Fig. 2. The Distributed Key Generation Functionality $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$

unavailable to $\mathcal{S}$.) Once they have received the public key, the other players may be allowed to receive it too, according to a schedule chosen by $\mathcal{S}$, and modeled by means of key delivery queries from $\mathcal{S}$. Once $\mathcal{S}$ asks to deliver the key to a player, the key is sent immediately.

Note that given the public key, if the adversary knows sufficiently many passwords that the combined entropy of the remaining passwords is low enough, he will be able to recover these remaining passwords by brute force attack. This is unavoidable and explains the absence of any testPwd query in this functionality. (This has nothing to do with the fact that our system is distributed: off-line attacks are always possible in principle in public-key systems, and become feasible as soon as a sufficient portion of the private key becomes known.)

The Distributed Private Computation Functionality (Figure 3). The aim here is to perform a private computation for the sole benefit of the group leader. The leader is responsible for the correctness of the computation; in addition, it is the only user to receive the end result.

This functionality will thus compute a function of some supplied input *in*, depending on a set of passwords that must define a secret key corresponding to a given public key. More precisely, the functionality will be able to check the compatibility of the passwords with the public key thanks to the verification function PublicKeyVer, and if it is correct it will then compute the secret key sk with the help of the function SecretKeyGen, and from there evaluate $\mathsf{PrivateComp}(\mathsf{sk}, in)$ and give the result to the leader. Note that SecretKeyGen and PublicKeyVer are naturally related to the function PublicKeyGen called by the former functionality. In all generality, unless SecretKeyGen and PublicKeyGen are both assumed to be deterministic, we need the predicate PublicKeyVer in order to verify that a public key is "correct" without necessarily being "equal" (to some canonical public key). Also note that the function SecretKeyGen is not assumed to be injective, lest it unduly restrict the number of users and the total size of their passwords.

PHASES AND QUERIES. During the initialization phase, each user is given as input a password pw_i as outlined earlier, but also an input *in*, and a public key pk. We stress that the security is guaranteed even if the users do not share the same values for *in* and pk, because then the functionality fails directly at the end of the initialization phase. At the end of this step, the adversary is also given knowledge of the common *in* and pk (as these are supposedly public).

After this initialization step is over, but before the actual computation, the adversary $\mathcal{S}$ is given the opportunity to make one or more *simultaneous* password guesses, by issuing a single Password Test query,

$\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ is parametrized by a security parameter k and three functions. PublicKeyVer is a boolean function $\mathsf{PublicKeyVer} : (\mathsf{pw}_1, \mathsf{pw}_2, \ldots, \mathsf{pw}_n, \mathsf{pk}) \mapsto b$, where $b = 1$ if the passwords and the public key are compatible, $b = 0$ otherwise. SecretKeyGen is a function $\mathsf{SecretKeyGen} : (\mathsf{pw}_1, \mathsf{pw}_2, \ldots, \mathsf{pw}_n) \mapsto \mathsf{sk}$, where sk is the secret key obtained from the passwords. Finally, PrivateComp is a private-key function $\mathsf{PrivateComp} : (\mathsf{sk}, c) \mapsto m$, where sk is the secret key, c is the function input (e.g., a ciphertext) and m the private result of the computation (e.g., the decrypted message). Denote by *role* either *player* or *leader*. The functionality interacts with an adversary $\mathcal{S}$ and a set of parties $P_1, \ldots, P_n$ via the following queries:

Initialization. Upon receiving a query $(\mathsf{newSession}, sid, Pid, P_i, \mathsf{pk}, c, \mathsf{pw}_i, role)$ from user P_i for the first time, where Pid is a set of at least two distinct identities containing P_i, ignore it if $role = leader$ and if there is already a record of the form $(sid, Pid, *, *, *, *, leader)$. Record $(sid, Pid, P_i, \mathsf{pk}, c, \mathsf{pw}_i, role)$, mark it fresh, and send $(sid, Pid, P_i, \mathsf{pk}, c, role)$ to $\mathcal{S}$. Ignore any subsequent query $(\mathsf{newSession}, sid, Pid', *, *, *, *, *)$ where $Pid' \neq Pid$.

If there are already $|Pid| - 1$ recorded tuples $(sid, Pid, P_i, \mathsf{pk}, c, \mathsf{pw}_i, role)$, and exactly one of them such that $role = leader$, then after recording the $|Pid|$-th tuple, verify that the values of c and pk are the same for all the users. If the tuples do not fulfill all of these conditions, report $(sid, Pid, \mathsf{error})$ to $\mathcal{S}$ and stop. Otherwise, record $(sid, Pid, \mathsf{pk}, c, \mathsf{ready})$ and send it to $\mathcal{S}$. The group leader is P_j.

Password Test. Upon receiving a first query $(\mathsf{testPwd}, sid, Pid, \{P_{i_1}, \ldots, P_{i_l}\}, \{\mathsf{pw}_{i_1}, \ldots, \mathsf{pw}_{i_l}\})$ from $\mathcal{S}$, if there exist l records $(sid, Pid, P_{i_k}, \mathsf{pk}, c, *, *)$, necessarily still marked fresh, and a record $(sid, Pid, \mathsf{pk}, c, \mathsf{ready})$, then denote by $\mathsf{pw}_{j_{l+1}}, \ldots, \mathsf{pw}_{j_n}$ the passwords of the other users of the group. If $\mathsf{PublicKeyVer}(\mathsf{pw}_1, \ldots, \mathsf{pw}_n, \mathsf{pk}) = 1$, edit the records of $P_{i_1}, \ldots, P_{i_l}$ to be marked compromised and reply to $\mathcal{S}$ with "correct guess". Otherwise, mark the records of the users $P_{i_1}, \ldots, P_{i_l}$ as interrupted and reply to $\mathcal{S}$ with "wrong guess". Ignore all subsequent queries of the form $(\mathsf{testPwd}, sid, Pid, *, *)$.

Private Computation. Upon receiving a message $(\mathsf{compute}, sid, Pid)$ from $\mathcal{S}$ where there is a recorded tuple $(sid, Pid, \mathsf{pk}, c, \mathsf{ready})$, then, if all records are fresh or compromised and $\mathsf{PublicKeyVer}(\mathsf{pw}_1, \ldots, \mathsf{pw}_n, \mathsf{pk}) = 1$, then compute $\mathsf{sk} = \mathsf{SecretKeyGen}(\mathsf{pw}_1, \ldots, \mathsf{pw}_n)$ and $m = \mathsf{PrivateComp}(\mathsf{sk}, c)$, and store (sid, Pid, m); Next, for all $P_i \in Pid$ mark the record $(sid, Pid, P_i, \mathsf{pk}, c, \mathsf{pw}_i, role)$ as complete. In any other case, store $(sid, Pid, \mathsf{error})$. When the computation result is set, report the outcome (either error or complete) to $\mathcal{S}$.

Leader Computation Delivery. Upon receiving $(\mathsf{leaderDeliver}, sid, Pid, \mathsf{b})$ from $\mathcal{S}$, where there is a recorded tuple (sid, Pid, m) such that $m \in \{\textit{well-formed messages}\} \cup \{\mathsf{error}\}$, and there exists a record $(sid, Pid, P_i, \mathsf{pk}, c, \mathsf{pw}_i, leader)$, send (sid, Pid, m) to P_i if b is equal to 1, or send $(sid, Pid, \mathsf{error})$ if b is equal to 0. If the group leader P_i is corrupted or compromised, then send (sid, Pid, m) to $\mathcal{S}$ as well (note that $\mathcal{S}$ gets m automatically if P_j is corrupted).

User Corruption. If $\mathcal{S}$ corrupts $P_i \in Pid$ where there is a recorded tuple $(sid, Pid, P_i, \mathsf{pk}, c, \mathsf{pw}_i, role)$, then reveal pw_i to $\mathcal{S}$. If $role = leader$, if there also is a recorded tuple (sid, Pid, m), and if (sid, Pid, m) has not yet been sent to P_i, then also send (sid, Pid, m) to $\mathcal{S}$.

Fig. 3. The Distributed Private Computation Functionality $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$

to model a "man-in-the-middle" impersonation attack against a subset of users. The query must indicate the subset of user(s) targeted in the attack, and what password(s) $\mathcal{S}$ wishes to test for those user(s). If all passwords are compatible with pk, the affected users are marked as compromised, otherwise they are all marked as interrupted. Unaffected users remain marked as fresh. Observe that it is in the opponent's best interest to target only a single user in the Password Test query to optimize compromising probability.

Once the functionality receives a message of the form (compute, sid, Pid) from $\mathcal{S}$, it proceeds to the computation phase. This is done as follows. If (1) all records are fresh or compromised, and (2) the passwords are compatible with the common public key pk, then the functionality computes the private key sk and then the output *out*. In all other cases, no message is computed.

In any case, after the key generation, the functionality informs the adversary of the result, meaning that $\mathcal{S}$ is told whether a message was actually computed or not. In particular, this means that the adversary also learns whether the users' passwords are compatible with pk or not. At first glance this may seem like a critical information to provide to the adversary. We argue, however, that this is not the case in our setting. Firstly, learning the status of the protocol (that is, whether it succeeded) without having any knowledge of the passwords that went into it is completely pointless, and the only knowledge that the adversary may have about those passwords are the ones it used in the testPwd impersonation query. Hence, as one should expect, from the status of the protocol the only useful thing that the adversary can learn is whether the password guesses it made were *all* good or not (as a single yes/no answer), but nothing else. Secondly, even if the adversary could somehow derive more utility from the protocol status, modeling that status as secret is not sensible because in most real-world scenarios it will be easy to infer from the users' behavior.

At the end, and similarly to the first functionality, the final result can either be released to the group leader, or withheld from it. However, this time, since the final result is a private output, there is no provision to distribute it to the other players. Also, $\mathcal{S}$ only gets the message if the leader either has been previously corrupted or if it is in the compromised state (either the leader has fallen under $\mathcal{S}$'s control, or $\mathcal{S}$ has successfully taken its place in the protocol).

Discussion. We emphasize that in this model only the leader and no other player receives the final result. Although this has the advantage of making the construction simpler, it is also the most useful and the only sensible choice. For starters, this makes our protocol much more resilient to password breaks in on-line impersonation attacks. To see why, suppose that the final output were indeed sent to all users. Then cracking the password of a single user would be all it took to break the system: adding more users would actually decrease the overall on-line security, because with a larger group comes a greater chance that some user will choose a weak password. By contrast, in the actual model, breaking the password of an ordinary user has no dire consequence: the protocol security will simply continue to rest on the passwords that remain. Since compromising ordinary users brings no other direct reward than to expose their passwords, it is just as if broken passwords were removed from the key in future protocol executions, or never contributed to it in the first place.

Of course, cracking the password of the *leader* will compromise the group and grant access to private computations (with the help of the other players, still), but that is only natural since the leader "owns" the group. There is an important distinction between exposure of an ordinary player's password and the leader's password: the leader represents the group with respect to third parties, i.e., when third parties use the group's public key their intention is to communicate with the leader. By contrast, ordinary players are not meant to be trusted and their inclusion to the group is a choice by the leader to help him or her increase the security of the private key — or leave it unchanged if that player turns out to be compromised — but never decrease it.

Revocation. In case of compromise of the leader password, it is possible for the leader to "revoke" the group by instructing the other players to stop participating in that group (e.g., by using the group's resources one last time to sign a revocation certificate using the group's private key). This will prevent any further use of the group's resources, unless of course the adversary manages to crack *all* of the players'

passwords *jointly*. Such revocation mechanism falls outside of the protocol, so we do not model it in the functionalities.

User Corruptions. Our definition of the $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ functionality deals with user corruptions in a way that is quite different to that of other password-based group protocols. E.g., in the group key exchange functionality of [28], if the adversary has obtained the passwords of some participants (via password guesses or user corruptions), it may freely set the resulting session key to any value. Here, our functionalities are much more demanding in two important ways: first, $\mathcal{S}$ is much constrained in the way it can make and test online password guesses; second, $\mathcal{S}$ can never alter the computation in any way once it has started.

PASSWORD TESTS. The first difference is that the testPwd query can only be asked once, early in the protocol, and it does not actually test the password of the users, but rather the compatibility between (1) the guessed passwords of any specified subset of users, (2) the real passwords of the rest of the group (known by the functionality thanks to the newSession queries), and (3) the public key (which at this stage is already guaranteed to be the same in all the users' views). This unusual shape for the testPwd query provides a very high level of security, because (A) at most a single set of password guesses can be tested against any player in any protocol instance, and (B) if $\mathcal{S}$ chooses to test a set of more than one password at once, then to cause a positive response all the guesses must be correct simultaneously (and since this becomes exponentially unlikely, the astute adversary should be content to test sets of one password at a time). After the private computation, all the records, initially fresh, compromised, or interrupted, become either complete or error. No more testPwd query is accepted at this stage, because once the users have completed their task it is too late for $\mathcal{S}$ to impersonate them (though corruption queries can still be made to read their state). Note that one testPwd query is allowed for each instance of $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$, several of which may be invoked by the split functionality $s\mathcal{F}_{\mathsf{pwDistPrivateComp}}$.

ROBUSTNESS. The second difference with the model in [28] is that we do not grant the adversary the right to alter the computation result when corrupting some users or learning some passwords. This in particular means that either the group leader receives something coherent, or he receives an error; he cannot receive something wrong, which makes the protocol robust. Robustness is actually automatic if we make the assumption that the computation function PrivateComp is deterministic; for simplicity, this is the setting of the generic protocol described in detail in this paper. At the end, however, we shall mention some applications that require randomness in the computation. Without going into details, we can keep the protocol robust by having all the parties commit to their random coins in the first round, in the same way as they will also commit to their passwords (see below): this allows us to treat such coins as any regular private input in the model, and hence forbid the adversary from modifying them once the computation has started.

We remark that, although the adversary cannot spoof the computation, the environment does become aware of the completion of the protocol, and hence could distinguish between the ideal and the real worlds if the adversary won more often in one than the other. Such environmental awareness of the final state is of course to be expected in reality, and so it is natural that our model should capture it. (Our implementation will thus have to ensure that the success conditions are the same in both worlds.)

IMPLICIT CORRUPTIONS. Because we have a set of initially unauthenticated players communicating over adversarially controlled channels, it is always possible for the adversary to partition the actual players into isolated islands [1], and act on behalf of the complement of players with respect to each island. We call this an implicit corruption, meaning that the adversary usurps the identity of a regular player (or players) from the very start, before the key generation is even initiated. The adversary then sends the newSession query on behalf of such implicitly corrupted players, who never really *became* corrupted but always *were* the adversary. As mentioned previously, this situation is modeled in the ideal world by the respective split functionalities $s\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ and $s\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ spawning one or more instances of the normal functionalities $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ and $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ over disjoint sets of (actual) players, as illustrated on Figure 1.

3 Protocol Description

The following protocol deals with a particular case of unauthenticated distributed private computation [1], as captured by our functionalities. Informally, assuming s to be a secret key, the aim of the protocol is to compute a value c^s given an element c of the group. This computation can be used to perform distributed BLS signatures [10], ElGamal decryptions [19], linear decryptions [8], and BF or BB1 identity-based key extraction [9, 7].

Here we focus on ElGamal decryptions, relying on the DDH assumption. We emphasize that the protocol as given relies exclusively on DDH, not requiring any additional assumption; and that it can be easily modified to rely on the Decision Linear assumption for compatibility with bilinear groups [8].

Building Blocks. Let $\mathbb{G}$ be a group of prime order p, and g a generator of this group. We furthermore assume to be given an element h in $\mathbb{G}$ as a CRS. We use the following building blocks:

Password Selection. Each user P_i owns a privately selected password pw_i, to act as the i-th share of the secret key sk (see below). For convenience, we write $\mathsf{pw}_i = \mathsf{pw}_{i,1} \dots \mathsf{pw}_{i,\ell} \in \{0, \dots, 2^{L\ell} - 1\}$, i.e., we further divide each password pw_i into ℓ blocks $\mathsf{pw}_{i,j} \in \{0, \dots, 2^L - 1\}$ of L bits each, where $p < 2^{\ell L}$. The segmentation into blocks is a technicality to get efficient extractable commitments for long passwords: in the concrete scheme, for example, we shall use single-bit blocks in order to achieve the most efficient extraction (i.e, $L = 1$ and $\ell = 160$ for a 160-bit prime p). Notice that although we allow full-size passwords of up to $L\ell$ bits (the size of p), users are of course permitted to choose shorter passwords.

Password Combination. The private key sk is defined as the (virtual) combination of all the passwords pw_i. It does not matter how precisely such combination is done, as long as it is reproducible and preserves the joint entropy of the set of passwords (up to $\log_2 p$ bits, since that is the length of sk). For example, if there are n users, all with short passwords $\mathsf{pw}_i^* \in \{0, \dots, \Delta - 1\}$ with $\Delta^n < p$, defining $\mathsf{pw}_i = \Delta^i \mathsf{pw}_i^*$ and taking $\mathsf{sk} = \sum_i \mathsf{pw}_i$ will ensure that there are no "aliasing effects", or mutual cancellation of two or more passwords.

In general, it is preferable that each user independently transforms his or her true password pw_i^* into an effective password pw_i by applying a suitable extractor $\mathsf{pw}_i = \mathsf{H}(i, \mathsf{pw}_i^*, Z_i)$ where Z_i is any relevant public information such as a description of the group and its purpose. We can then safely take $\mathsf{sk} = \sum_i \mathsf{pw}_i$ and be assured that the entropy of sk will closely match the joint entropy of the vector $(\mathsf{pw}_1^*, \dots, \mathsf{pw}_n^*)$ taken together. Such password pre-processing using hashing is very standard but falls outside of the functionalities proper.

Public and Private Keys. We use the (effective) passwords pw_i to define a key pair $(\mathsf{sk}, \mathsf{pk} = g^{\mathsf{sk}})$ for a password-based ElGamal key encapsulation mechanism (KEM). Based on the above, we define $\mathsf{sk} = \mathsf{SecretKeyGen}(\mathsf{pw}_1, \dots, \mathsf{pw}_n) \stackrel{\mathsf{def}}{=} \sum_{i=1}^n \mathsf{pw}_i$ and $\mathsf{pk} = \mathsf{PublicKeyGen}(\mathsf{pw}_1, \dots, \mathsf{pw}_n) \stackrel{\mathsf{def}}{=} g^{\sum \mathsf{pw}_i}$. The public-key verification function is then $\mathsf{PublicKeyVer}(\mathsf{pw}_1, \dots, \mathsf{pw}_n, \mathsf{pk}) \stackrel{\mathsf{def}}{=} \left(\mathsf{pk} \stackrel{?}{=} g^{\sum \mathsf{pw}_i}\right)$.

The ElGamal KEM public-key operation is the encapsulation $\mathsf{Enc} : (\mathsf{pk}, r) \mapsto (c = g^r, m = \mathsf{pk}^r)$, which outputs a random session key m and a ciphertext c. The private-key operation is the decapsulation $\mathsf{Dec} : (\mathsf{sk}, c) \mapsto m = c^{\mathsf{sk}}$, which here is deterministic. Observe that whereas Dec instantiates $\mathsf{PrivateComp}$ in the functionalities, Enc is intended for public third-party usage and never appears in the private protocols.

Entropy Preservation. In order for the low password entropies to combine nicely in the secret key $\mathsf{sk} = \sum_i \mathsf{pw}_i$, the effective pw_i must be properly "decoupled" to avoid mutual cancellations, as just discussed.

We note that, even with the kind of shuffling previously considered, it is quite possible that the actual entropy of sk will be smaller than its maximum value of $\log_2 p$ bits, e.g., if there are not enough non-corrupted users or if their passwords are too small. Nevertheless, there is no known effective attack against discrete logarithm and related problems that can take advantage of any reduced entropy of sk, barring an exhaustive search over the space of possible values. Specifically, regardless of how the passwords are

actually combined, one could easily prove that no generic attack [30] can solve the discrete logarithm or the DDH problem in less than $\sqrt{2^h}$ operations, where h is the min-entropy of the private key sk conditionally on all known passwords.

COMPUTATIONAL ASSUMPTION. Our concrete protocols rely on the Decisional Diffie-Hellman (DDH) assumption, stated here for completeness: Let $\mathbb{G} = \langle g \rangle$ be a multiplicative abelian cyclic group of prime order p. For random $x, y, z \in \mathbb{Z}_p^*$, it is computationally intractable to distinguish (g, g^x, g^y, g^{xy}) from (g, g^x, g^y, g^z).

EXTRACTABLE HOMOMORPHIC COMMITMENTS. The first step of our distributed decryption protocol is for each user to commit to his password (the details are given in the following section). The commitment needs to be extractable, homomorphic, and compatible with the shape of the public key. Generally speaking, one needs a commitment $\mathsf{Commit}(\mathsf{pw}, r)$ that is additively homomorphic on pw and with certain properties on r. In order to simplify the following description of the protocols, we chose to use ElGamal's scheme [19], which is additive on the random value r, and given by: $\mathsf{Commit}_v(\mathsf{pw}, r) = (v^{\mathsf{pw}} h^r, g^r)$. The semantic security relies on the above DDH assumption. Extractability is possible granted the decryption key x, such that $h = g^x$ in the common reference string.

SIMULATION-SOUND NON-INTERACTIVE ZERO-KNOWLEDGE PROOFS. Informally speaking, a zero-knowledge proof system is said to be simulation-sound if it has the property that an adversary cannot give a convincing proof for a false statement, even if it has oracle access to the zero-knowledge simulator. We also require non-malleability, which is to say that a proof of some theorem cannot be turned into a proof of another theorem. De Santis *et al.* proved in [17] the existence of such a scheme, with the additional property of being non-interactive, if we assume the existence of one-way trapdoor permutations. Note that their scheme allows for multiple simulations with a unique common random string (CRS), which is crucial for the multi-session case. If we instantiate all the SSNIZK proofs with those, then our protocols are UC-secure in the CRS model.

However, for sake of efficiency, we can instead instantiate them using Schnorr-like proofs of equality of discrete logarithms [21], which rely on the random-oracle model [4], but are significantly more practical. These SSNIZK are well-known (see details in Appendix C and their proofs in [21]), but along these lines, we use the notation $\mathsf{SSNIZK}(\mathcal{L}(w))$ for a proof that w lies in the language $\mathcal{L}$. More precisely, $\mathsf{CDH}(g, G, h, H)$ will state that (g, G, h, H) lies in the CDH language: there exists a common exponent x such that $G = g^x$ and $H = h^x$.

Intuition. We first describe the distributed decryption algorithm. All the users are provided with a password pw_i, a public key pk, and a ciphertext c. One of them is the leader of the group, denoted by P_1, and the others are $P_2, \ldots, P_n$. For this given ciphertext $c \in \mathbb{G}$, the leader wants to obtain $m = c^{\mathsf{sk}}$. But before computing this value, everybody wants to be sure that all the users are honest, or at least that the combination of the passwords is compatible with the public key.

The protocol starts by verifying that they will be able to decrypt the ciphertext, and thus that they indeed know a representation of the decryption key into shares. Each user sends a commitment C_i of his password. As we see in the proof (see Appendix B), this commitment needs to be extractable so that the simulator is able to recover the passwords used by the adversary: this is a requirement of the UC model, as in [14]. Indeed, the simulator needs to be able to simulate everything without knowing any passwords, he thus recovers the passwords by extracting them from the commitments C_i made by the adversary in this first round, enabling him to adjust his own values before the subsequent commitments, so that all the passwords are compatible with the public key (if they should be in the situation at hand). If we think in terms of ElGamal encryption, the extraction is proportional in the square root of the size of the alphabet, which would be practical for 20-bit passwords but not 160-bit ones (and even if passwords are usually small, we do not want to restrict the size of the passwords). This is the reason why we segmented all the passwords into small blocks: to commit to them block by block. In our concrete description, blocks are of size 1, which will help to make the proof of validity: ElGamal encryption of one bit.

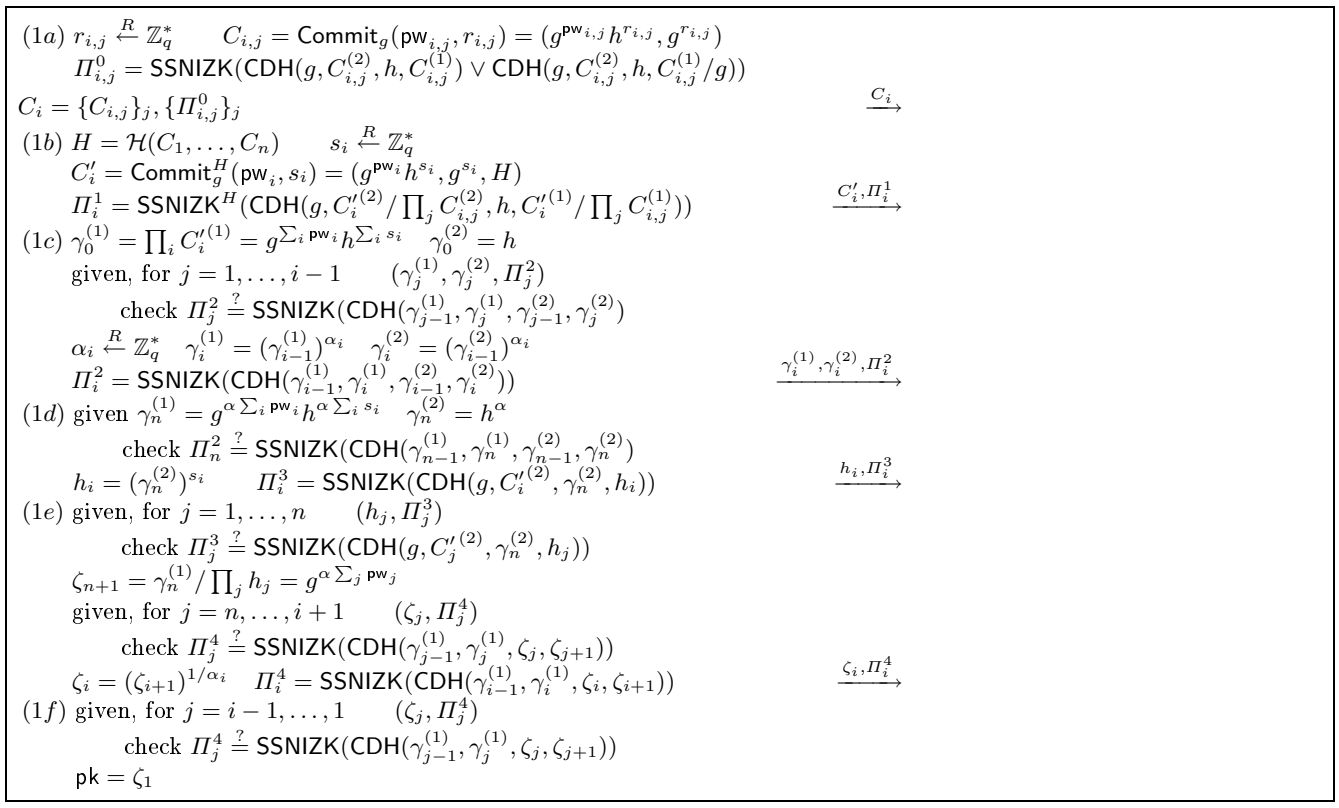

Fig. 4. Individual steps of the distributed key generation protocol

Once this first step is done, the users commit again to their passwords. The new commitments C'_i will be the ones used in the rest of the protocol. They need not be segmented (since we will not extract anything from them), but we ask the users to prove that they are compatible with the former ones. Note that they use the three values $H = \mathcal{H}(C_1, \ldots, C_n)$ (where $\mathcal{H}$ is a collision-resistant hash function), pk, and c, as "labels" of these commitments (see below), to avoid malleability and replay from the previous sessions, granted the SSNIZK proofs that include and thus check these labels.

Next, the users make yet another commitment A_i to their passwords, but this time they do an ElGamal encryption of pw_i in base c instead of in base g (in the above C'_i commitment). That is, each user computes $A_i = (c^{\mathsf{pw}_i} h^{t_i}, g^{t_i})$. The commitment C'_i will be used to check the possibility of the decryption (that it is consistent with $\mathsf{pk} = g^{\mathsf{sk}}$), whereas A_i will be used to actually compute the decryption c^{sk}, hence the two different bases g and c in C'_i and A_i, respectively.

All the users send these last two commitments to everybody, along with a SSNIZK proof that the same password was used each time. These proofs are "labeled" by H, pk, and c, and the verification by the other users will succeed only if their "labels" are identical. This enables all the players to check that everybody shares the same public key pk and the same ciphertext c. It thus avoids situations in which a group leader with an incorrect key obtains a correct decryption message, contrary to the ideal functionality. The protocol will thus fail if H, pk, or c is not the same to everyone, which is the result required by the ideal functionality. Note that the protocol will also fail if the adversary drops or modifies a flow received by a user, even if everything was correct (compatible passwords, same public key, same ciphertext). This situation is modeled in the functionality by the bit b of the key/decryption delivery queries, for when everything goes well but the group leader does not obtain the result.

After these rounds of commitments, a verification step allows for the group leader, but also all the players, to check whether the public key and the passwords are compatible. Note that at this point, everything has become publicly verifiable so that the group leader will not be able to cheat and make the other players believe that everything is correct when it is not. Verification starts from the commitments $C'_i = (C'^{(1)}_i, C'^{(2)}_i)$, and involves two "blinding rings" to raise the two values $\prod_i C'^{(1)}_i$ and $\prod_i C'^{(2)}_i$ to some distributed random exponent $\alpha = \sum_i \alpha_i$. The ratio of the blinded values is taken to cancel the $h^{\sum_i s_i}$, leaving $g^{\alpha \mathsf{sk}}$. A final "unblinding ring" is applied to remove the exponent α and expose g^{sk}. This ends with a decision by the group leader on whether to abort the protocol (when the passwords are incompatible with the public key) or go on to the computation step. We stress that every user is able to check the validity of the group leader's decision: A dishonest execution cannot continue without an honest user becoming aware of it (and aborting it). Note however that an honest execution can also be stopped by a user if the adversary modifies a flow destined to it, as reflected by the bit b in the ideal functionality.

If the group leader decides to go on, the players assist in the computation of c^{sk}, again with the help of two blinding and one unblinding rings, starting from the commitments A_i. Note that if at some point a user fails to send its value to everyone (for instance due to a denial of service attack) or if the adversary modifies a flow (in a man-in-the-middle attack), the protocol will fail. In the ideal world this means that the simulator makes a decryption delivery with a bit b set to zero. Because of the SSNIZK proofs, in these decryption rounds exactly the same sequence of passwords as in the first rounds has to be used by the players. This necessarily implies compatibility with the public key, but may be a stronger condition.

As a side note, observe that all the blinding rings in the verification and the computation steps could be made concurrent instead of sequential, in order to simplify the protocol. Notice however that the final unblinding ring of c^{sk} in the computation step should only be carried out after the public key and the committed passwords are known to be compatible, and the passwords to be the same in both sequences of commitments, *i.e.* after the verification step succeeded.

We show in Appendix B that we can *efficiently* simulate these computations without the knowledge of the pw_i's, so that they do not reveal anything more about the pw_i's than pk already does. More precisely, we show that such computations are indistinguishable to $\mathcal{A}$ under the DDH assumption.

The key generation protocol (computation of $\mathsf{pk} = g^{\mathsf{sk}}$) is a special case of the decryption protocol outlined above (computation of g^{sk}, test that $g^{\mathsf{sk}} = \mathsf{pk}$, computation of $m = c^{\mathsf{sk}}$), only simpler. Indeed, we only need one set of commitments for the last rounds of blinding/unblinding, as we omit all the prior verifications (since there is nothing to verify when the key is first set up).

We now describe more precisely both protocols (see Figures 4 and 5).

Details of the Distributed Key Generation — realizing $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$ (Figure 4).

• First Step of Commitment ($1a$). Each user P_i commits to its share pw_i (divided into ℓ blocks $\mathsf{pw}_{1,1}, \ldots, \mathsf{pw}_{i,\ell}$ of length L —in our description, $L = 1$) of the secret key sk: it computes $C_{i,j} = (C^{(1)}_{i,j}, C^{(2)}_{i,j}) = (g^{\mathsf{pw}_{i,j}} h^{r_{i,j}}, g^{r_{i,j}})$, for $j = 1, \ldots, \ell$, and "publishes" (*i.e.*, tries to send to everybody) $C_i = (C_{i,1}, \ldots, C_{i,\ell})$, with SSNIZK proofs that each commitment indeed commits to an L-bit block.

• Second Step of Commitment ($1b$). Each user P_i computes $H = \mathcal{H}(C_1, \ldots, C_n)$, and commits again to its share pw_i, but this time in a single block and with "label" H: It computes $C'_i = (C'^{(1)}_i, C'^{(2)}_i, C'^{(3)}_i) = (g^{\mathsf{pw}_i} h^{s_i}, g^{s_i}, H)$, an publishes it along with a SSNIZK proof that the passwords committed are the same in the two commitments. The language considered for this proof of membership is

$$L(g,h,H) = \left\{ (\{C_{i,1}, \ldots, C_{i,\ell}\}, C'_i) \;\middle|\; \exists (r_{i,1}, \ldots, r_{i,\ell}, s_i) \text{ such that } \begin{bmatrix} C^{(2)}_{i,j} = g^{r_{i,j}}, \;\; C'^{(2)}_i = g^{s_i}, \\ \prod_{j=1}^{\ell} C^{(1)}_{i,j} / h^{r_{i,j}} = C'^{(1)}_i / h^{s_i}, \\ C'^{(3)}_i = H \end{bmatrix} \right\}$$

Notice that this definition of L implies equality of passwords between the commitments; the passwords are present inside of the $C^{(1)}_{i,j}$ and $C'^{(1)}_i$. See Figure 4, Step ($1b$) for the realization of these SSNIZK proofs.

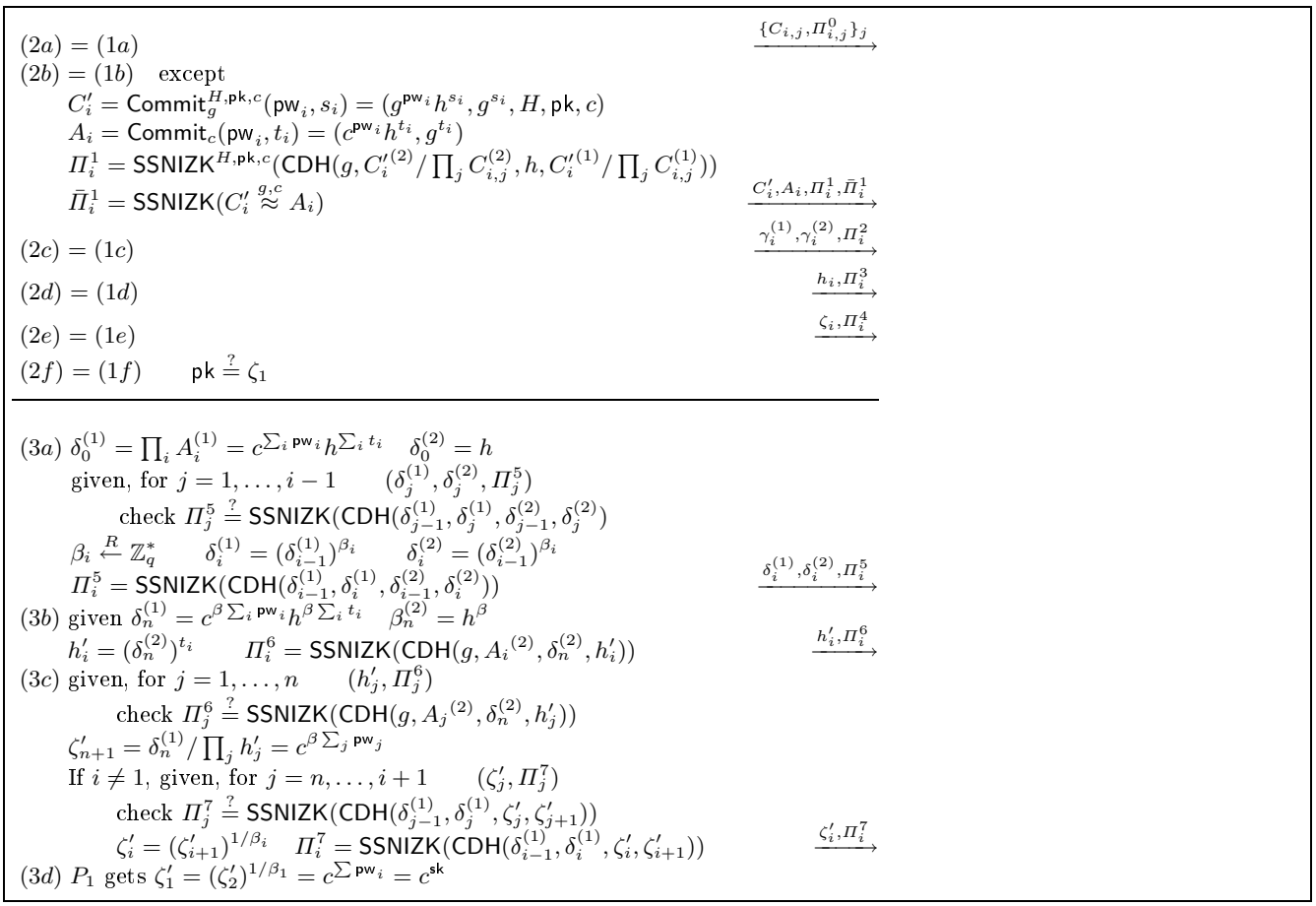

Fig. 5. Individual steps of the distributed decryption protocol

• First Step of Computation (1c). If one of the proofs received by a user is incorrect, it aborts the game. Otherwise, they share the same H and C'_i: they are thus able to compute the same $\gamma_0 = (g^{\sum \mathsf{pw}_i} h^{\sum s_i}, h) = (\gamma^{(1)}_0, \gamma^{(2)}_0)$ by multiplying the first parts of the commitments C'_i to each other.

The group leader P_1 wants to compute $g^{\sum \mathsf{pw}_i} = \mathsf{pk}$. For $i = 1, \ldots, n$, sequentially, P_i chooses a random $\alpha_i \in \mathbb{Z}_n$ and computes $\gamma_i = \gamma^{\alpha_i}_{i-1} = (\gamma^{(1)}_i, \gamma^{(2)}_i)$. It then produces a SSNIZK proof that it used the same α_i in the computations of both $\gamma^{(1)}_i$ and $\gamma^{(2)}_i$, from $\gamma^{(1)}_{i-1}$ and $\gamma^{(2)}_{i-1}$ respectively, and publishes γ_i along with such proof, whose language is the following equality of discrete logarithms — see Figure 4, Step (1c)

$$L_1 = \left\{ (\gamma_i, \gamma_{i-1}) \mid \exists \alpha_i \text{ such that } \gamma^{(1)}_i = \left(\gamma^{(1)}_{i-1}\right)^{\alpha_i} \text{ and } \gamma^{(2)}_i = \left(\gamma^{(2)}_{i-1}\right)^{\alpha_i} \right\}$$

If the proof is not valid, the next user aborts. If all goes well, at the end the users will have performed a "round of blinding" where each user P_i contributed its own random ephemeral exponent α_i.

• Second Step of Computation (1d). Denote $\prod \alpha_i$ by α. When the users receive the last element $\gamma_n = (g^{\alpha \sum \mathsf{pw}_i} h^{\alpha \sum s_i}, h^{\alpha})$, they all compute and publish the value $h_i = h^{\alpha s_i}$, along with a SSNIZK proof that their random value s_i is the same as the one they used in C'_i (as before, the language of this proof is an equality of discrete logarithms). The language of this proof is — see Figure 4, Step (1d)

$$L_2(g, h^{\alpha}) = \left\{ (\gamma_n, C'_i, h_i) \mid \exists s_i \text{ such that } C_i'^{(2)} = g^{s_i} \text{ and } h_i = (h^{\alpha})^{s_i} \right\}$$

• Third Step of Computation ($1e$). At this point, each user is able first to compute $h^{\alpha \sum s_i}$ by multiplying all the h_i together, and then, by division of $\gamma_n^{(1)}$ by that value, obtain $g^{\alpha \sum \mathsf{pw}_i} = \zeta_{n+1}$. Then, for $i = n, \ldots, 2$, sequentially, user P_i computes $\zeta_i = (\zeta_{i+1})^{1/\alpha_i}$, along with a SSNIZK proof that the α_i is the same as before. This is a proof of discrete logarithm equality, whose language is — see Figure 4, Step ($1e$)

$$L_3 = \left\{(\gamma_i, \gamma_{i-1}, \zeta_i, \zeta_{i+1}) \mid \exists \alpha_i \text{ such that } \gamma_i^{(1)} = \left(\gamma_{i-1}^{(1)}\right)^{\alpha_i} \text{ and } \gamma_i^{(2)} = \left(\gamma_{i-1}^{(2)}\right)^{\alpha_i} \text{ and } \zeta_i = (\zeta_{i+1})^{1/\alpha_i}\right\}$$

Each player thus sequentially publishes ζ_i and the proof, allowing the remaining users to proceed. This backward "round of unblinding" removes the blinding α in the reverse order it was applied.

• Last Step of Computation ($1f$). The last player to take part in the unblinding is the group leader P_1, who is thus the first to obtain the final unblinded public key $\zeta_1 = (\zeta_2)^{1/\alpha_1} = g^{\sum \mathsf{pw}_i} = \mathsf{pk}$.

To communicate the key to the others, the group leader publishes ζ_1 and the related SSNIZK proof. All the users can then perform the final unblinding step for themselves and be certain that the resulting key corresponds to the initial password commitments.

Details of the Distributed Decryption — realizing $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ (Figure 5).

• Common Verification of the Public Key ($2a$) – ($2f$). These steps are almost the same as in the former protocol, except for ($2b$) that differs from ($1b$) for the label that not only contains H, but also the public key pk and the ciphertext c: $C'_i = (C_i'^{(1)}, C_i'^{(2)}, C_i'^{(3)}, C_i'^{(4)}, C_i'^{(5)}) = (g^{\mathsf{pw}_i} h^{s_i}, g^{s_i}, H, \mathsf{pk}, c)$. This extended label will make sure that all the players use the same data. We also anticipate the goal of this functionality: the computation of c^{sk}. Each user P_i also commits to pw_i in base c, sending $A_i = (A_i^{(1)}, A_i^{(2)}) = (c^{\mathsf{pw}_i} h^{t_i}, g^{t_i})$, together with a SSNIZK proof that the same password pw_i is committed in both C'_i and A_i, with different bases, g and c respectively.

This proof is a bit more intricate, but it consists of several proofs of equalities of discrete logarithms: we first compute and publish the following elements, for a random $u_i \xleftarrow{R} \mathbb{Z}_q^*$:

$$a_i = (C_i'^{(1)})^{u_i} = g^{u_i \mathsf{pw}_i} h^{u_i s_i} = g^{\pi_i} h^{\sigma_i} \qquad b_i = (C_i'^{(2)})^{u_i} = g^{u_i s_i} = g^{\sigma_i} \qquad G_i = g^{\pi_i}$$
$$c_i = (A_i^{(1)})^{u_i} = c^{u_i \mathsf{pw}_i} h^{u_i t_i} = c^{\pi_i} h^{\tau_i} \qquad d_i = (A_i^{(2)})^{u_i} = g^{u_i t_i} = g^{\tau_i} \qquad \Gamma_i = c^{\pi_i}$$

and also publish SSNIZK proofs that:

- the same u_i is used to compute a_i, b_i, c_i, d_i from $C_i'^{(1)}$, $C_i'^{(2)}$, $A_i^{(1)}$, $A_i^{(2)}$ respectively:
$$\mathsf{CDH}(C_i'^{(1)}, a_i, C_i'^{(2)}, b_i) \quad \wedge \quad \mathsf{CDH}(C_i'^{(1)}, a_i, A_i^{(1)}, c_i) \quad \wedge \quad \mathsf{CDH}(C_i'^{(1)}, a_i, A_i^{(2)}, d_i)$$
- the same π_i is used to compute G_i and Γ_i, from g and c respectively: $\mathsf{CDH}(g, G_i, c, \Gamma_i)$
- this π_i is indeed committed in base g in (a_i, b_i): $\mathsf{CDH}(g, b_i, h, a_i/G_i)$
- this π_i is indeed committed in base c in (c_i, d_i): $\mathsf{CDH}(g, d_i, h, c_i/\Gamma_i)$

If one of the proofs received by a user is incorrect, this user aborts the game. Otherwise, since they share the values H and C'_i, every user is able to compute $\gamma_0 = \left(g^{\sum \mathsf{pw}_i} h^{\sum s_i}, h\right)$ by multiplying the first parts of the commitments C'_i to each other.

The group leader P_1 wants to check whether $g^{\sum \mathsf{pw}_i} = \mathsf{pk}$. This is done by two blinding and unblinding rings and their associated SSNIZK proofs, exactly as in the computation step of the former protocol: at the end, the group leader does publish $\zeta_1 = g^{\sum \mathsf{pw}_i}$ (along with the corresponding proof) and every player is able to check whether the result is correct or not.

• Leader Computation using the Virtual Private Key ($3a$) – ($3d$). We now start the computation of c^{sk} in the same manner as g^{sk}, doing two blinding and unblinding rings (using random t_i instead of s_i and random values β_i different from α_i) and their supporting sequences of SSNIZK. This time, the users use the commitments in base c, namely the A_i, and so in the end the group leader obtains $c^{\sum \mathsf{pw}_i}$ but does not publish it. Since the group leader starts the computation and does not publish the final result $c^{\sum \mathsf{pw}_i}$, it is the only one to learn the message obtained (and even the only one aware that the decryption succeeded, although that information might be difficult to conceal in a real-world application).

Note that in the real-world protocol, a player is compromised if the adversary $\mathcal{A}$ guessed a compatible password in the first flow. Because of the SSNIZK proofs, in this case the adversary is the only one able to send the next flows in an acceptable way.

The proofs of these theorems can be found in Appendix B.

Theorem 1 *Let $\widehat{\mathcal{F}}_{\mathsf{pwDistPublicKeyGen}}$ be the concurrent multi-session extension of $\mathcal{F}_{\mathsf{pwDistPublicKeyGen}}$. The distributed key generation protocol in Figure 4 securely realizes $\widehat{\mathcal{F}}_{\mathsf{pwDistPublicKeyGen}}$ for ElGamal key generation, in the CRS model, in the presence of static adversaries, provided that DDH is infeasible in $\mathbb{G}$, $\mathcal{H}$ is collision-resistant, and SSNIZK proofs for the CDH language exist.*

Theorem 2 *Let $\widehat{\mathcal{F}}_{\mathsf{pwDistPrivateComp}}$ be the concurrent multi-session extension of $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$. The distributed decryption protocol in Figure 5 securely realizes $\widehat{\mathcal{F}}_{\mathsf{pwDistPrivateComp}}$ for ElGamal decryption, in the CRS model, in the presence of static adversaries, provided that DDH is infeasible in $\mathbb{G}$, $\mathcal{H}$ is collision-resistant, and SSNIZK proofs for the CDH language exist.*

As stated above, our protocol is only proven secure against static adversaries. Unlike adaptive ones, static adversaries are only allowed to corrupt protocol participants prior to the beginning of the protocol execution.

4 Discussion and Conclusion

In this work, we have brought together ideas from secret sharing, threshold cryptography, password-based protocols, and multi-party computation, to devise a practical approach to (distributed) password-based public-key cryptography. For a given cryptosystem, the objective was to define, from a set of user-selected weak passwords held in different locations, a virtual private key that is as strong and resistant to attacks as any regular key, and that can be used in a distributed manner without ever requiring its actual reconstitution.

We proposed general definitions of such functionalities in the UC model, carefully justifying all our design choices along the way. In particular, we saw that it is mandatory to require the presence of a "group leader" who directs the private computation process and solely obtains its end result. We then constructed explicit protocols for the simple but instructive case of ElGamal encryption. Specifically, relying on the DDH assumption, we constructed and proved the security of two ElGamal key generation and decryption protocols, whose private key is virtual and implied by a distributed collection of arbitrary passwords.

To conclude, we now argue that the approach outlined in this paper is in fact quite general and has broad applications. It can of course be viewed as a restriction of the Unauthenticated MPC framework of [1]; but this would be missing the point, since as often in the UC model, much (or most) of the work has been done once the functionality definitions have been laid down. The functionalities that we have carefully crafted here should apply essentially without change to most kinds of public-key primitives.

The protocols also generalize easily beyond ElGamal decryption. The same method that let us compute c^{sk} from a distributed $\mathsf{sk} = \langle \mathsf{pw}_1, \ldots, \mathsf{pw}_n \rangle$, can also compute pairs of vectors $(\boldsymbol{c}_i^{\mathsf{sk}}, \boldsymbol{c}_j^{r})$ for a random ephemeral r contributed by all the players — or, precisely, for $r = \sum_i r_i$ where each r_i is initially committed to by each player, in a similar way as they initially commit to their passwords. By the hiding and binding properties of the commitments this guarantees that r is uniform and unpredictable if at least one player draws r_i at random.

Remarkably, this is enough to let us do "password-based distributed IBE", where the private-key generator is decentralized over a set of users, each of them holding only a short private password of their own choosing. $\mathsf{PrivateComp}$ is now a key extraction function that maps user identities id to user decryption keys d_{id}. To get: **"Password-based" Boneh-Franklin (BF) IBE** [9], we need to compute $d_{id} = H(id)^{\mathsf{sk}}$ where $H(id)$ is a public hash of a user's identity. This is analogous to c^{sk}, and thus our protocol works

virtually unchanged. To get: **"Password-based" Boneh-Boyen ($\mathbf{BB_1}$) IBE** [7], here d_{id} is randomized and of the form $(g_0^{\mathsf{sk}}(g_1^{id}g_2)^r, g_3^r)$. This fits the general form of what we can compute by adding ephemerals to our protocol as just discussed.
Note that in some bilinear groups the DDH problem is easy: in those groups, we must replace DDH-based commitments with ones based on a weaker assumption, such as D-Linear [8]; such changes are straightforward.

Acknowledgments

This work was supported in part by the French ANR-07-SESU-008-01 PAMPA Project. The second author thanks ECRYPT and the hospitality of ENS.

References

1. B. Barak, R. Canetti, Y. Lindell, R. Pass, and T. Rabin. Secure computation without authentication. In *CRYPTO 2005*, *LNCS* 3621, pages 361–377. Springer, Aug. 2005.
2. D. Beaver and S. Goldwasser. Multiparty computation with faulty majority. In *30th FOCS*, pages 468–473. IEEE Computer Society Press, Oct. / Nov. 1989.
3. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT 2000*, *LNCS* 1807, pages 139–155. Springer, May 2000.
4. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993.
5. S. M. Bellovin and M. 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.
6. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.
7. D. Boneh and X. Boyen. Efficient selective-ID secure identity based encryption without random oracles. In *EUROCRYPT 2004*, *LNCS* 3027, pages 223–238. Springer, May 2004.
8. D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In *CRYPTO 2004*, *LNCS* 3152, pages 41–55. Springer, Aug. 2004.
9. D. Boneh and M. K. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO 2001*, *LNCS* 2139, pages 213–229. Springer, Aug. 2001.
10. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *ASIACRYPT 2001*, *LNCS* 2248, pages 514–532. Springer, Dec. 2001.
11. V. Boyko, P. D. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *EUROCRYPT 2000*, *LNCS* 1807, pages 156–171. Springer, May 2000.
12. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct. 2001.
13. R. Canetti, S. Halevi, and A. Herzberg. Maintaining authenticated communication in the presence of break-ins. *Journal of Cryptology*, 13(1):61–105, 2000.
14. R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT 2005*, *LNCS* 3494, pages 404–421. Springer, May 2005.
15. R. Canetti and T. Rabin. Universal composition with joint state. In *CRYPTO 2003*, *LNCS* 2729, pages 265–281. Springer, Aug. 2003.
16. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *20th ACM STOC*, pages 11–19. ACM Press, May 1988.
17. A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano, and A. Sahai. Robust non-interactive zero knowledge. In *CRYPTO 2001*, *LNCS* 2139, pages 566–598. Springer, Aug. 2001.
18. D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.
19. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO'84*, *LNCS* 196, pages 10–18. Springer, Aug. 1985.
20. M. Fitzi, D. Gottesman, M. Hirt, T. Holenstein, and A. Smith. Detectable byzantine agreement secure against faulty majorities. In *21st ACM PODC*, pages 118–126. ACM Press, July 2002.
21. P.-A. Fouque and D. Pointcheval. Threshold cryptosystems secure against chosen-ciphertext attacks. In *ASIACRYPT 2001*, *LNCS* 2248, pages 351–368. Springer, Dec. 2001.
22. R. Gennaro and Y. Lindell. A framework for password-based authenticated key exchange. In *EUROCRYPT 2003*, *LNCS* 2656, pages 524–543. Springer, May 2003.

23. O. Goldreich and Y. Lindell. Session-key generation using human passwords only. In *CRYPTO 2001*, *LNCS* 2139, pages 408–432. Springer, Aug. 2001.
24. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *19th ACM STOC*, pages 218–229. ACM Press, May 1987.
25. O. Goldreich, S. Micali, and A. Wigderson. How to prove all NP-statements in zero-knowledge, and a methodology of cryptographic protocol design. In *CRYPTO'86*, *LNCS* 263, pages 171–185. Springer, Aug. 1987.
26. D. Holtby, B. M. Kapron, and V. King. Lower bound for scalable Byzantine agreement. In *25th ACM PODC*, pages 285–291. ACM Press, July 2006.
27. J. Katz, R. Ostrovsky, and M. Yung. Efficient password-authenticated key exchange using human-memorable passwords. In *EUROCRYPT 2001*, *LNCS* 2045, pages 475–494. Springer, May 2001.
28. J. Katz and J. S. Shin. Modeling insider attacks on group key-exchange protocols. In *ACM CCS 05*, pages 180–189. ACM Press, Nov. 2005.
29. T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *21st ACM STOC*, pages 73–85. ACM Press, May 1989.
30. V. Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT'97*, *LNCS* 1233, pages 256–266. Springer, May 1997.
31. A. C. Yao. Protocols for secure computations. In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, Nov. 1982.

A The UC Framework

The aim of the UC security model is to ensure that UC-secure protocols will continue to behave in the ideal way even if executed in arbitrary environments. This model relies on the indistinguishability between two worlds, the ideal world and the real world. In the ideal world, the security is provided by an ideal functionality $\mathcal{F}$.

One can think of it as a trusted party in the context of multi-party computation: this functionality interacts with n users having to compute a function f. These users give their inputs to $\mathcal{F}$, which gives them back their outputs. We stress that there is no communication between the users. $\mathcal{F}$ ensures that the computation is correct and that the users learn nothing more than their own inputs and outputs. Security is then guaranteed since an adversary $\mathcal{A}$ can only learn and thus modify the data of corrupted users.

In order to prove that a protocol Π verifies $\mathcal{F}$, one considers an environment $\mathcal{Z}$ that provides inputs to the users and acts as a distinguisher between the real world (with actual users and a real adversary that can control some of them and also the communication among them) and the ideal world (with dummy users interacting only with the ideal functionality $\mathcal{F}$, and a simulated adversary also interacting with $\mathcal{F}$). We then say that the protocol Π realizes $\mathcal{F}$ if for all polynomial adversary $\mathcal{A}$, there exists a polynomial simulator $\mathcal{S}$ such that no polynomial environment $\mathcal{Z}$ can distinguish between the two worlds (one with $\mathcal{F}$ and $\mathcal{S}$, the other with Π and $\mathcal{A}$) with a significant advantage.

Since there are several copies of a functionality $\mathcal{F}$ running in parallel, each one has a unique session identifier *sid*. All the messages must contain the SID of the copy they are intended for. As in [14], we assume for simplicity that each protocol realizing $\mathcal{F}$ has inputs containing its SID. We also assume that each user starts a session by specifying the SID of $\mathcal{F}$, its identity P_i, its password pw_i, and the identities of the other users *Pid*.

A shortcoming of the UC theorem is that is says nothing about protocols sharing state and randomness (it ensures the security of a single unit only). Here, since we need a common reference string for all instances of the protocol, we need a stronger result, provided by Canetti and Rabin in [15] and called *universal composability with joint state*. Informally, they define the multi-session extension $\widehat{\mathcal{F}}$ of $\mathcal{F}$, which basically runs multiple instances of $\mathcal{F}$, where each of them is identified by a sub-session identifier *ssid*. $\widehat{\mathcal{F}}$ has to be executed with *sid* and *ssid*. When it receives a message m containing *ssid*, it hands m to the copy of $\mathcal{F}$ having the SSID *ssid* (or invokes a new one if such copy does not exist).

For the sake of generality, we shall describe all the functionalities in the context of *adaptive* adversaries, that are allowed to corrupt users whenever they like to. For simplicity, however, we shall only prove the security of our constructions in presence of *static* adversaries, that have to choose which users to corrupt before the beginning of the execution of the protocol, either implicitly in the key generation protocol (when

the adversary starts playing as one of the parties, choosing by himself the password), or explicitly in the decryption protocol (asking a corruption before a new decryption session).

In the UC model, a corruption implies a complete access to the internal memory of the users (which here means the password and the internal state); in addition, the adversary takes the entire control of the corrupted user, and can modify its behavior for the remaining of the protocol.

B Proof of the Security Theorems

In this section we give a sketch of the proof that the protocols described on Figures 4 and 5 respectively realize the functionalities specified on Figures 2 and 3.

The proof of the distributed key generation protocol is similar to that of the distributed decryption given below, with the added simplification that there is no verification step and the difference that all the users receive the result in the end (which corresponds exactly to what happens in the decryption protocol at the end of the verification step, where everyone also receives the result). Thus, we refer to the proof of the second protocol for the workings of the simulation. The additional simplification implied by the adversary receiving the result in the end will be given in remarks.

B.1 Sketch of the Proof.

The objective of the proof is to construct, from a real-world adversary $\mathcal{A}$, an ideal-world simulator $\mathcal{S}$, so that the behavior of $\mathcal{A}$ in the real world and that of $\mathcal{S}$ in the ideal world are indistinguishable to the environment. The ideal functionalities are specified in Figures 2 and 3 and described in Section 2. Since we use the joint state version of the UC theorem, we implicitly consider the multi-session extension of this functionality, and thus replace all sid by $(sid, ssid)$. Note that the passwords of the users depend on the sub-session considered. For sake of simplicity, we denote them by pw_i, but one should implicitly understand $\mathsf{pw}_{i,ssid}$.

In the real game, we know that the protocol cannot continue past the two initial commitment rounds if there is any inconsistency (between the passwords pw_i used in all the commitments, and between the copies of pk and c held by all the users). Any inconsistency will violate the SSNIZK language L, and because the proof system with honest setup is assumed to be sound, it is not feasible for anyone to prove a false statement. Similarly, the two rounds of blinding and unblinding serve to verify the consistency of pk with the pw_i's, and to compute the final output c^{sk}, respectively. To be precise, the security of these rounds follows from our assumptions: cheating in the computation of the blinding/unblinding rounds without getting caught requires a SSNIZK proof forgery; while distinguishing the final decryption c^{sk} from random by anyone other than the group leader is reducible to solving the DDH problem in $\mathbb{G}$.

B.2 Description of the simulator.

This description is based on that of [14]. When initialized with security parameter k, the simulator first chooses a random element $h = g^x \in \mathbb{G}$, and uses the zero-knowledge simulator to obtain its CRS γ. He finally initializes the real-world adversary $\mathcal{A}$, giving him (h, γ) as common reference string.

From this moment on, $\mathcal{S}$ interacts with the environment $\mathcal{Z}$, the functionality $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$ and his subroutine $\mathcal{A}$. For the most part, this interaction is implemented by $\mathcal{S}$ just choosing a dummy password and following the protocol on behalf of all the honest players. In addition, instead of following the honest prover strategy, $\mathcal{S}$ uses the zero-knowledge simulator in all the proofs (which is indistinguishable due to the zero-knowledge property of the proof protocol). If a session aborts or terminates, then $\mathcal{S}$ reports it to $\mathcal{A}$.

Recall that we use the model of split functionalities described in [1] and that the users are partitioned in disjoint sessions according to what they received in the first flow. This means that, in the following,

we can assume that all the players have received the same values of $C_{i,j}$. This is particularly useful when $\mathcal{A}$ controls a set of users since these commitments are extractable. Indeed, note that choosing g and h allows $\mathcal{S}$ to know the discrete logarithm x of h in base g. Since the commitments are ElGamal ciphertexts, knowledge of $\log_g h$ will allow the simulator to decrypt the ciphertexts and then extract the passwords used by $\mathcal{A}$ in the first-round commitments. (The actual extraction requires taking discrete logarithms, but this can be done efficiently using generic methods, because the discrete logarithms to extract are the L-bit password sub-blocks which by design have a very small domain. And the small size is enforced by a SSNIZK .) Once these passwords recovered, $\mathcal{S}$ asks a testPwd query. If it returns correct, it provides the following equation:

$$g^{P_i \sum_{\text{honest}} \mathsf{pw}_i} = \mathsf{pk} \Big/ g^{P_i \sum_{\text{corrupted}} \mathsf{pw}_i} \tag{1}$$

More details on the simulations of the different flows follow. Note that if anything goes wrong or $\mathcal{S}$ receives a message formatted differently from what is expected by the session, then $\mathcal{S}$ aborts that session and notifies $\mathcal{A}$.

Session Initialization. When receiving a message $(ssid, Pid, P_i, \mathsf{pk}, c, role)$ from $\mathcal{F}_{\mathsf{pwDistPrivateComp}}$, the simulator $\mathcal{S}$ starts simulating a new session of the protocol for party P_i, group Pid, session identifier $ssid$, and common reference string $(h = g^x, \gamma)$. We denote this session by $(P_i, ssid)$.

Step (1*a*). $\mathcal{S}$ chooses at random n_h dummy passwords on behalf of each of the n_h honest players, computes the first-round commitments, and sends them along with the corresponding (and simulated) proofs. Since the first-round commitment is computationally hiding under the DDH assumption, this is indistinguishable from a real execution. $\mathcal{S}$ then learns from the functionality whether the users all share the same c and pk or not. In the second case, $\mathcal{S}$ aborts the game.

Step (1*b*). If all users are honest, $\mathcal{S}$ asks a Private Computation query along with a Leader Decryption Delivery query to the functionality, which returns either complete or error. If it returns complete, then $\mathcal{S}$ keeps on using the $n_h - 1$ last dummy passwords, but sets the value g^{pw_1} of the group leader P_1 such that it is compatible with the value $\mathsf{pk} = g^{\mathsf{sk}}$ (without knowing the corresponding password): The computation will be correct. $\mathcal{S}$ then sends the second-round commitments, along with the corresponding (and simulated) proofs. Note that the password used for the group leader will not be compatible with c^{sk} but this does not matter since it will never be disclosed and will not make the protocol fail. Also note that, since C_1 and C_1' will not be compatible, $\mathcal{S}$ will have to prove a false statement for P_1, which is indistinguishable from a real execution since the proofs are simulated.

Remark. Note that for the distributed key generation protocol, things would have been simpler. In this case, the simulator not only receives complete or error, but the exact value of $\mathsf{pk} = g^{\mathsf{sk}}$. He is thus able to modify g^{pw_1} such that the passwords are compatible with the public key. Same thing will apply below.

If the query returns error, then the simulator keeps on using the dummy passwords (there is no need that they should be compatible) and sends the new commitments along with the corresponding (simulated) proofs.

If some users are corrupted, $\mathcal{S}$ extracts the passwords and asks a testPwd query. If it is correct, $\mathcal{S}$ keeps the $n_h - 1$ first dummy passwords and changes the last value $g^{\mathsf{pw}_{i_{n_h}}}$ to be compatible with Equation (1) (without knowing the corresponding password). In the same round, $\mathcal{S}$ must also produce another series of password commitments, this time as ElGamal encryptions of c^{pw_i} and not g^{pw_i}. We now have to consider two cases. First, if the group leader is attacked, $\mathcal{S}$ computes the commitments normally for the first $n_h - 1$ honest players using the simulated passwords. For the last player $P_{i_{n_h}}$, he asks a Private Computation

query along with a Leader Computation Delivery query in order to recover c^{sk}, and computes the missing commitment as an ElGamal encryption of $c^{\mathsf{pw}'_{in_h}}$, which is given by

$$c^{\mathsf{pw}'_{in_h}} = c^{\mathsf{sk}} \Big/ \left(c^{P_i \sum_{\text{honest},\, i \neq in_h} \mathsf{pw}_i} \; c^{P_i \sum_{\text{corrupted}} \mathsf{pw}_i} \right)$$

Otherwise, if the group leader is not attacked, the simulator will not recover c^{sk}. We thus proceed as in the honest case, using an incorrect value c^{pw_1} for the group leader.

Finally, if the testPwd query returns incorrect, since the verification step will fail, $\mathcal{S}$ can keep all the dummy passwords (in base g as in base c), and send these commitments along with the corresponding (simulated) proofs.

The indistinguishability between the simulation of this step and the real execution relies on the non-malleability and the computationally hiding property of the commitment (relying in the DDH assumption). The adversary cannot become aware that the passwords are not the good ones, or that the password for the user P_1 or P_{in_h} changed between the two rounds of commitments.

Following Steps. At this stage, everything is set. If the users are honest and share compatible passwords, then $\mathcal{S}$ continues honestly the protocol, by choosing random values α_i and β_i and executing the four rings as described in the protocol. Recall that the very last step will fail, without any consequence on the final result, and only in the view of the group leader. $\mathcal{S}$ finally asks a Leader Computation Delivery query, setting the bit b to 1 if the execution succeeded and to 0 otherwise (for instance if some flows were non oracle-generated).

If there are some corrupted players, sharing compatible passwords with the rest of the group, $\mathcal{S}$ also continues the game honestly with the four rings. Everything goes well if the group leader is attacked. Otherwise, the last step fails, without any bad consequence since the group leader ends the ring (as before). The bit b is chosen as described above: If something goes wrong and the protocol halts, then $b = 0$, otherwise $b = 1$.

If the players do not share compatible passwords, then the simulator continues honestly the protocol (without knowing the real passwords of the users) until it fails, at the verification step. The simulator then asks a Private Computation query, along with a Leader Computation Delivery query, setting the bit b to 0.

Finally, this simulation is indistinguishable from the ideal world, since the group leader receives a correct message in this simulation if and only if it receives a correct message in the ideal world.

REMARK. Incidentally, what makes the proofs easier in our new paradigm than in key exchange protocols, is that at the end it is not needed to make sure that all the participants obtain the same key in the real world if and only if they do in the ideal world. In our setting, we only need to worry about the key received by a single party: the group leader.

This establishes that, given any adversary $\mathcal{A}$ that attacks the protocol Π in the real world, we can build a simulator $\mathcal{S}$ that interacts with the functionality $\mathcal{F}$ in the ideal world, in such a way that the environment cannot distinguish which world it is in.

B.3 Details of the Proof.

In this proof, we incrementally define a sequence of games starting from the one describing a real execution of the protocol and ending up with game $\boldsymbol{G}_7$ which we prove to be indistinguishable with respect to the ideal experiment.

The objective of the proof is to construct from an adversary $\mathcal{A}$ a simulator $\mathcal{S}$ in the ideal world, so that the behavior of $\mathcal{A}$ in the real world and that of $\mathcal{S}$ in the ideal world are indistinguishable to the environment. $\mathcal{S}$ is incrementally defined in the games, ending up to be completely defined in $\boldsymbol{G}_7$ (though we do not rewrite him entirely in this game since his behavior was described in the previous games). This

final game will then be proven to be indistinguishable to the ideal world, showing that we indeed have constructed an ideal simulator to the real-world adversary $\mathcal{A}$.

In the first games, the simulator has actually access to all the information given to the users by the environment, in particular their passwords. In the last game, we nearly are in the ideal game so that the users do not exist anymore: $\mathcal{S}$ only has access to the information transmitted by his queries to the functionality (not to the passwords, for instance) and he has to simulate the users entirely by himself. Between these two situations, the simulator lives in a world which is not really real, not really ideal.

In order to formally model this situation, we chose to consider three hybrid queries that $\mathcal{S}$ can ask to the functionality all along the games. The CompatiblePwd query checks whether the passwords of the users are compatible with the passwords of the other users and the public key of the group leader. The Computation query gives back the message obtained by decrypting the ciphertext. And the Delivery query gives the result to the group leader–and to the adversary if the former is compromised.

Note that since in the first games, the simulator has access to the users' inputs, he knows their passwords. In such a case a CompatiblePwd query (or a Computation or Delivery query) can be easily implemented by letting the simulator look at the passwords owned by the users. When the users are entirely simulated, without the knowledge of their passwords, $\mathcal{S}$ will replace the queries above with the real testPwd, Decryption Computation and Leader Decryption Delivery queries to the functionality.

We say that a flow is *oracle-generated* if it was sent by an honest user and arrives without any alteration to the user it was meant to. We say it is *non-oracle-generated* otherwise, that is either if it was sent by an honest user and modified by the adversary, or if it was sent by a corrupted user or a user impersonated by the adversary (more generally denoted by *attacked* user, that is, a user whose password is known to the adversary).

Game G_0: Real game.

In this game, we know that the protocol cannot continue past the two initial commitment rounds if there is any inconsistency (between the passwords pw_i used in all the commitments, and between the copies of pk and c held by all the users). Any inconsistency will violate the SSNIZK language L, and because the proof system with honest setup is assumed to be sound, it is not feasible for anyone to prove a false statement.

Similarly, the two rounds of blinding and unblinding serve to verify the consistency of pk with the pw_i's, and to compute the final decryption c^{sk} of c, respectively. To be precise, the security of these rounds follows from our assumptions: cheating in the computation of the blinding/unblinding rounds without getting caught requires a SSNIZK proof forgery; while distinguishing the final decryption c^{sk} from random by anyone other than the group leader is reducible to solving the DDH problem in $\mathbb{G}$.

Game G_1: Simulation of the SSNIZK proofs.

From this game on, we allow the simulator to program (once and for all) the common reference string $(h = g^x, \gamma)$, where γ is a common reference string for the SSNIZK proofs, and h for the commitment (and x the extraction key).

Additionally, this game modifies how the zero-knowledge proofs are performed. Specifically, instead of using the honest-prover strategy, all the proofs in which the prover is an honest user are simulated using the zero-knowledge simulator. (Note that the common reference string γ is also simulated once for all.)

Since the proofs are concurrent zero-knowledge, the environment cannot distinguish between the two games $\boldsymbol{G}_1$ and $\boldsymbol{G}_0$. That is, if an environment could distinguish between these hybrids, one could construct an adversary that breaks the zero-knowledge property of the proof protocol.

Game G_2: Simulation of the first round of commitments.

From this game on, $\mathcal{S}$ simulates the first rounds of commitments in the following way. We suppose that he still knows the passwords of the players. Let ℓ be the number of honest users, i.e. the users $\mathcal{S}$ has to simulate. $\mathcal{S}$ chooses at random ℓ dummy passwords $\widetilde{\mathsf{pw}}_{i_1}, \ldots, \widetilde{\mathsf{pw}}_{i_\ell}$ on behalf of each one of these users. Once all these values are set, $\mathcal{S}$ computes the first-round commitments and send them to everybody.

$\mathcal{S}$ then learns from the functionality whether the users all share the same c and pk or not. In the second case, $\mathcal{S}$ aborts the game. In the first case, $\mathcal{S}$ goes on the execution of the protocol in a honest way, using the real passwords of the users. Note that he will have to prove false statements, which is not a problem since the proofs are simulated since the former game. In the end, if the execution succeeds, he asks a Computation query, and he finally sets the bit b to 1 in the Delivery query. Otherwise, if the execution fails, he also asks a Computation query but sets the bit b to 0 for the delivery.

Since the first-round commitment is hiding, the adversary cannot become aware of the transformation of pw_{i_ℓ} into $\widetilde{\mathsf{pw}}_{i_\ell}$: this game is indistinguishable from $\boldsymbol{G}_1$.

Game $\boldsymbol{G}_3$: Simulation of honest users with compatible passwords.

From this game on, we show how to simulate the users without using their passwords. More precisely, the simulator is still supposed to know the passwords of the users, but little by little we are going to show that he actually never needs them in the simulation. This will ensure in the end that the simulator does not need the knowledge of the passwords in order to perform the simulation honestly.

Note that from this game on, we can suppose that all the c and the pk are identical, since the case of different values has been dealt with in the former game (and the simulator aborts the protocol in this case). The first round of commitments is simulated as in $\boldsymbol{G}_2$. We now face two cases.

First, if there are attacked users among the group, the simulation continues as in the former game, $\mathcal{S}$ being allowed to use the passwords of all the users. (We show in $\boldsymbol{G}_5$ and $\boldsymbol{G}_6$ how to simulate in this case without using the passwords.)

Second, if all users are honest, we show how to continue the simulation without the help of the passwords. Note that if at some point some flows are non-oracle-generated, the protocol will abort: In this case, the simulator will set the bit b to 0 in the Delivery query. This comes from the non-malleability of the SSNIZK proofs. If the adversary has not generated the first commitments, he will not be able to construct valid proofs, with unknown witnesses.

The simulator first asks a Computation query along with a Delivery query to the functionality, which gives him either complete or error. In the second case, $\mathcal{S}$ continues the simulation as in the former game, and we allow him to use the passwords of the users (we show in $\boldsymbol{G}_4$ how to get rid of the use of the passwords in this case).

We now consider the first case and sum up briefly the circumstances which led us here: The users are honest, they have the same c and pk, and their passwords are compatible with the public key. Then, $\mathcal{S}$ keeps on using the passwords $\widetilde{\mathsf{pw}}_2, \ldots, \widetilde{\mathsf{pw}}_n$ for the $n-1$ last users, and he is able to set $g^{\widetilde{\mathsf{pw}}'_1}$ (without knowing $\widetilde{\mathsf{pw}}'_1$) such that all the passwords are compatible with the value g^{sk}, using the equation: $g^{\sum \widetilde{\mathsf{pw}}_i} = g^{\mathsf{sk}}$. But he still uses $c^{\widetilde{\mathsf{pw}}_1}$ for the last commitment, since he does not know c^{sk}. Notice that he will give once again a proof for a false statement. The simulator then continues honestly the game, by choosing random values α_i and β_i and executing the four rings as described in the protocol. Only the very last step will fail, since the last value $c^{\widetilde{\mathsf{pw}}_1}$ is incompatible with the other ones. But this does not matter because this value is never sent or used in the remaining of the protocol. In the Delivery query, the bit b is chosen as described in $\boldsymbol{G}_2$. Finally note that the simulator never needed the knowledge of the passwords of the users.

Due to the non-malleability of the commitments, along with their hiding property, this game is indistinguishable from $\boldsymbol{G}_2$.

Remark. Note that for the distributed key generation protocol, things would have been simpler. In this case, the simulator not only receives complete or error, but the exact value of $\mathsf{pk} = g^{\mathsf{sk}}$. He is thus able to modify $\widetilde{\mathsf{pw}}_1$ such that the passwords are compatible with the public key. Same thing will apply to $\boldsymbol{G}_5$.

Game $\boldsymbol{G}_4$: Simulation of honest users with incompatible passwords.

This game starts exactly as in $\boldsymbol{G}_3$. If there are attacked users, the simulator is granted the right to use the passwords of all the users, and continues as in $\boldsymbol{G}_2$.

If all the users are honest, he continues as in $\boldsymbol{G}_3$, by asking a Computation query. We showed in $\boldsymbol{G}_3$ how to deal with the case where the functionality returns correct (that is, when the passwords are compatible with the public key) without using the passwords of the users. We now consider the other case and show how to treat it. Thus, we suppose that the Computation query returns an error, meaning that the passwords are incompatible with the public key.

The simulator then computes commitments of the values $g^{\widetilde{\mathsf{pw}}_i}$ and $c^{\widetilde{\mathsf{pw}}_i}$ for all the users (there is no need that their passwords should be compatible). He sends them along with the corresponding proofs to the other users. The simulator then continues honestly the game, by choosing random values α_i and β_i and executing the two first rings as described in the protocol. Note that the real passwords of the users are not needed anymore. Since the protocol will fail at the verification step, the simulator will set the bit b to 0 in the Delivery query.

Since the commitments are hiding, this game is indistinguishable from $\boldsymbol{G}_3$.

Game $\boldsymbol{G}_5$: Simulation in case of compatible passwords in presence of an adversary.

This game starts exactly as in $\boldsymbol{G}_2$. The case where all the users are honest has been dealt with in the games $\boldsymbol{G}_3$ and $\boldsymbol{G}_4$. We now consider the case in which the adversary controls a set of users. He can either know their passwords (corrupted users) or not (compromised users). Recall that we denote by ℓ the number of honest users.

Note that choosing g and h allows $\mathcal{S}$ to know the discrete logarithm of h in base g. Since the commitments are ElGamal ciphertexts, knowledge of $\log_g h$ will allow the simulator to decrypt the ciphertexts and then extract the passwords used by $\mathcal{A}$ in the first-round commitments. (The actual extraction requires taking discrete logarithms, but this can be done efficiently using generic methods, because the discrete logarithms to extract are the L-bit password sub-blocks which by design have a very small domain. And the small size is enforced by a SSNIZK.)

After this first round, the simulator thus extracts the passwords used by the adversary in the commitments he sent. Note that the honest users are not supposed to have received the same values from the adversary: We only know that these values are non-oracle-generated, but not necessarily equal. Thus, the simulator chooses at random one of the commitments received from an attacked user to extract and recover its password. One could argue that there is a problem here, but note that the proof given with the second commitment will fail if the label H is not the same for all users (recall that we have assumed a collision-resistant hash function for the computation of H).

Once $\mathcal{S}$ has recovered all the passwords of the attacked users, he asks a CompatiblePwd query. If this query returns incorrect, we continue the simulation as in $\boldsymbol{G}_2$ (we show in $\boldsymbol{G}_6$ how to deal with this case without using the passwords of the users).

We now suppose that the query returns correct. This provides the following equation:

$$g^{\sum_{j=1}^{\ell} \mathsf{pw}_{i_j}} = \mathsf{pk} \Big/ g^{P_i \sum_{\text{attacked}} \mathsf{pw}_i}$$

Recall that the passwords of the honest users were chosen at random, so there is no chance that they should be compatible with the (common) public key. $\mathcal{S}$ thus keeps its $\ell - 1$ first passwords and computes a replacement value $g^{\widetilde{\mathsf{pw}}'_{i_\ell}}$ thanks to the above equation. Remark that he does not know the corresponding password $\widetilde{\mathsf{pw}}'_{i_\ell}$.

In the second round, the simulator must produce commitments that are compatible with the public key. To do so, he makes commitments on the same random passwords as before for the $\ell - 1$ first honest users, and for the last one creates a commitment as an ElGamal encryption of $g^{\widetilde{\mathsf{pw}}'_{i_\ell}}$. He sends them all

out. In the same round, the simulator must also produce another series of password commitments, this time as ElGamal encryptions of c^{pw_i} and not g^{pw_i}.

We now have to consider two cases. First, suppose that the group leader is attacked. The simulator computes the commitments normally for the first $\ell - 1$ honest players using the simulated passwords. For the last player P_{i_ℓ}, he asks a Computation query along with a Delivery query in order to recover c^{sk}, and computes the missing commitment as an ElGamal encryption of $c^{\widetilde{\mathsf{pw}}''_{i_\ell}}$, which is given by

$$c^{\widetilde{\mathsf{pw}}''_\ell} = c^{\mathsf{sk}} \Big/ c^{\sum_{i=1}^{\ell-1} \mathsf{pw}_i} c^{P_i \sum_{\text{attacked}} \mathsf{pw}_i}$$

Otherwise, if the group leader is not attacked, the simulator will not recover c^{sk}. We thus proceed as in $\boldsymbol{G}_3$, using an incorrect value $c^{\mathsf{pw}_{i_j}}$ for the group leader (if i_j is its index among the uncorrupted players).

$\mathcal{S}$ sends out the commitments along with the proofs of consistency. Note that in the second case he will prove a false statement for the group leader.

The simulator then continues the game honestly, by choosing random values α_i and β_i and executing the four rings as described in the protocol. Everything goes well if the group leader is attacked. Otherwise, the last step fails, without any bad consequence on the protocol since the group leader ends the ring (as in $\boldsymbol{G}_3$). The bit b is chosen as described in $\boldsymbol{G}_2$: if something goes wrong and the protocol halts, then $b = 0$, otherwise $b = 1$.

Since the commitments are computationally hiding under the DDH assumption, the adversary cannot become aware that the passwords are not the good ones, or that the password for the user P_{i_ℓ} changed between the two rounds of commitments. This game is indistinguishable from $\boldsymbol{G}_4$.

Game $\boldsymbol{G}_6$: Simulation in case of incompatible passwords in presence of an adversary.

This game starts exactly as in $\boldsymbol{G}_5$. We now suppose that the CompatiblePwd query returns incorrect. Since the verification step will fail, $\mathcal{S}$ can keep all the values $\widetilde{\mathsf{pw}}_{i_1}, \ldots, \widetilde{\mathsf{pw}}_{i_\ell}$ for the second round of commitments (in base g as in base c). He then sends these commitments along with the corresponding proofs.

He then continues honestly the protocol (without knowing the real passwords of the users) until it fails, at the verification step. The simulator then asks a Computation query, and a Delivery query with bit $b = 0$. For the same reasons than in the former game, $\boldsymbol{G}_6$ is indistinguishable from $\boldsymbol{G}_5$.

Game $\boldsymbol{G}_7$: Indistinguishability with the ideal world.

We have shown that $\mathcal{S}$ is able in any case to simulate the whole protocol without using the passwords of the users. Thus, we can now suppose that he does not know these passwords.

The only difference between $\boldsymbol{G}_6$ and $\boldsymbol{G}_7$ is that the CompatiblePwd query is replaced by a testPwd query to the functionality, the Computation by a Decryption query and the Delivery by a Leader Decryption Delivery query. If a session aborts or terminates, $\mathcal{S}$ reports it to $\mathcal{A}$. If a session terminates with a message m, then $\mathcal{S}$ makes a Delivery call to the functionality, specifying a bit $b = 1$. If the protocol fails, he gives a bit $b = 0$.

We now show that this last game $\boldsymbol{G}_7$ is indistinguishable from the ideal-world experiment IWE. More precisely, we have to show that the group leader receives a correct message in $\boldsymbol{G}_7$ if and only if it receives a correct message in the ideal world.

First, if the users share compatible passwords, the same public key, the same ciphertext, and all the flows are oracle-generated until the end of the game, then the group leader will obtain a correct message, both in $\boldsymbol{G}_7$ (from $\boldsymbol{G}_3$) and the ideal world, as there are no testPwd queries and the sessions remain fresh. Second, if they share compatible passwords, the same public key, the same ciphertext, and if there are some impersonation attempts but the adversary plays honestly, then the group leader will also receive a correct message (from $\boldsymbol{G}_5$). Third, if they do not share compatible passwords or if some received flows differ from one user to an other, then the group leader will get an error.

This establishes that, given any adversary $\mathcal{A}$ that attacks the protocol Π in the real world, we can build a simulator $\mathcal{S}$ that interacts with the functionality $\mathcal{F}$ in the ideal world, in such a way that the environment cannot distinguish which world it is in.

C Simulation-Sound Non-Interactive Zero-Knowledge Proofs

We briefly review how we can efficiently build our SSNIZK proofs in the random oracle model [4]. Note that we only need to prove two kinds of languages:

- equality of discrete logarithms, that is, for a tuple $(g, h, G, H) \in \mathcal{G}^4$, one wants to prove that $\mathsf{CDH}(g, G, h, H)$, knowing k such that $G = g^k$ and $H = h^k$;
- ElGamal encryption of 0 or 1, that is an OR-proof of equality of discrete logarithms. Given a tuple $(g, h, G, H) \in \mathcal{G}^4$, one indeed wants to prove that $\mathsf{CDH}(g, G, h, H)$ or $\mathsf{CDH}(g, G, h, H/g)$, knowing k such that $G = g^k$ and $H = h^k$ (for an encryption of 0) or $H = gh^k$ (encryption of 1).

We will also allow a label ℓ to be included in the proof, and in the verification.

C.1 Equality of Discrete Logarithms

Given $(g, h, G = g^k, H = h^k)$,

- one first chooses $r \xleftarrow{R} \mathbb{Z}_q^*$, and computes $G' = g^r$ and $H' = h^r$;
- one then generates the challenge $c = \mathcal{H}(\ell, g, h, G, H, G', H') \in \mathbb{Z}_q$;
- one finally computes $s = r - kc \bmod q$;

The proof consists of the tuple (c, s).

In order to verify the proof, one first computes the expected values for G' and H' respectively: $G'' = g^s G^c$ and $H'' = h^s H^c$; and then checks that $c \stackrel{?}{=} \mathcal{H}(\ell, g, h, G, H, G'', H'')$.

This proof is a SSNIZK proof [21], in the random oracle model.

C.2 ElGamal Encryption of 0 or 1

We now want to combine two of the proofs above, in order to show that one of them is true: given (g, h, G, H), we want to show that there exists k such that $G = g^k$ and either $H = h^k$ or $H = gh^k$. Let us assume that $H = g^b h^k$, for $b \in \{0, 1\}$.

- one first chooses $r_b \xleftarrow{R} \mathbb{Z}_q$ and computes $G'_b = g^{r_b}$ and $H'_b = h^{r_b}$;
- one also chooses $c_{\bar{b}}, s_{\bar{b}} \in \mathbb{Z}_q$, and computes $G'_{\bar{b}} = g^{s_{\bar{b}}} G^{c_{\bar{b}}}$, as well as $H'_{\bar{b}} = h^{s_{\bar{b}}} (H/g^{\bar{b}})^{c_{\bar{b}}}$;
- one then generates the challenge $c = \mathcal{H}(\ell, g, h, G, H, G'_0, H'_0, G'_1, H'_1) \in \mathbb{Z}_q$;
- one computes $c_b = c - c_{\bar{b}} \bmod q$, and $s_b = r_b - kc_b \bmod q$

The proof consists of the tuple (c_0, c_1, s_0, s_1).

In order to verify the proof, one first computes the expected values for G'_0, G'_1, H'_0 and H'_1 respectively: $G''_0 = g^{s_0} G^{c_0}$, $H''_0 = h^{s_0} H^{c_0}$; $G''_1 = g^{s_1} G^{c_1}$, and $H''_1 = h^{s_1} (H/g)^{c_1}$, and then checks that $c_0 + c_1 \stackrel{?}{=} \mathcal{H}(\ell, g, h, G, H, G''_0, H''_0, G''_1, H''_1)$.