# Optimal Distributed Password Verification

Jan Camenisch
IBM Research – Zurich
jca@zurich.ibm.com

Anja Lehmann
IBM Research – Zurich
anj@zurich.ibm.com

Gregory Neven
IBM Research – Zurich
nev@zurich.ibm.com

## ABSTRACT

We present a highly efficient cryptographic protocol to protect user passwords against server compromise by distributing the capability to verify passwords over multiple servers. Password verification is a single-round protocol and requires from each server only one exponentiation in a prime-order group. In spite of its simplicity, our scheme boasts security against dynamic and transient corruptions, meaning that servers can be corrupted at any time and can recover from corruption by going through a non-interactive key refresh procedure. The users' passwords remain secure against offline dictionary attacks as long as not all servers are corrupted within the same time period between refreshes. The only currently known scheme to achieve such strong security guarantees incurs the considerable cost of several hundred exponentiations per server. We prove our scheme secure in the universal composability model, which is well-known to offer important benefits for password-based primitives, under the gap one-more Diffie-Hellman assumption in the random-oracle model. Server initialization and refresh must take place in a trusted execution environment. Initialization additionally requires a secure message to each server, but the refresh procedure is non-interactive. We show that these requirements are easily met in practice by providing an example deployment architecture.

## Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Cryptographic control; D.4.6 [**Security and Protection**]: Access controls; D.4.6 [**Security and Protection**]: Authentication

## Keywords

Password verification, proactive security, UC security.

## 1. INTRODUCTION

In spite of all their shortcomings in terms of security and usability, passwords are still the predominant method of online user authentication. One of the main threats currently posed to password security is server compromise. More than one billion personal data records were reported stolen in 2014 alone [16]; most of these records included user passwords. With more personal and financial data moving into the cloud, a further increase in breaches targeting usernames and passwords is expected for 2015 [14].

Even when properly salted and hashed, the low entropy in human-memorizable passwords is no match for the brute force of modern hardware: already in 2012, a rig of 25 GPUs could test up to 350 billion guesses per second in an offline dictionary attack. More complicated password hash functions [20, 24] can provide some relief, but at a linear rate at best: the computational effort to verify passwords for an honest server increases by the same factor as for the attacker—while the latter is probably better equipped with dedicated password-cracking hardware.

The problem of offline dictionary attacks when a server is compromised is inherent whenever that single server can test the correctness of passwords. A natural solution, first proposed by Ford and Kaliski [15], is therefore to split up the capability to verify passwords over two or more servers, so that security is preserved as long as less than a threshold of them are hacked. This has been the central idea behind several threshold password-authenticated key exchange (TPAKE) [17, 22, 2, 12, 25, 21] and threshold password-authenticated secret sharing (TPASS) [3, 10, 9, 18, 6] protocols as well as behind the RSA product Distributed Credential Protection (DCP) [13].

Resistance against server compromise is one thing, but knowing how to recover from it is another. Without secure recovery, all one can do in case of a detected breach is to re-initialize all servers and request all users to reset their passwords—which is probably exactly what one wanted to avoid by deploying the scheme. In cryptographic literature, recovery from compromise is known as *proactive security* or security against *transient corruptions*. Of the aforementioned threshold password-authenticated protocols, only Camenisch et al. [6] describe a recovery procedure and prove their protocol secure against transient corruptions. Di Raimondo and Gennaro [12] mention the possibility to refresh shares and the RSA DCP product description [13] mentions a re-randomization feature, but neither provides details or a security proof. Proactive security in the protocol of Camenisch et al. [6] unfortunately comes at a considerable cost: "a few hundred exponentiations" per server may be within practical reach for occasional data retrieval, but not for high-volume password verification.


*CCS'15,* October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813722 .

### *Our contribution.*

We present two simple and extremely efficient proactively secure *distributed password verification* protocols, allowing a login server $\mathcal{LS}$ and a number of back-end servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ to jointly determine the correctness of a user's password, while ruling out offline dictionary attacks unless *all* servers are corrupted during the *same* time period. A corrupt $\mathcal{LS}$ only sees the passwords of user accounts that are created or logged into during the corruption. No passwords, password hashes, or any other offline-attackable information is leaked for accounts that are inactive during the corruption. We think this is a reasonable compromise for not requiring user-side software, as it provides adequate protection against "smash-and-grab" attacks and short-term corruptions.

Login, i.e., password verification, is a single-round protocol requiring just one exponentiation in a prime-order group on each server (two for $\mathcal{LS}$), which is essentially optimal unless schemes without public-key operations can be found. The recovery and key refresh procedure is non-interactive and only involves a couple of additions and pseudo-random function evaluations per server, making it more than efficient enough to perform it preventively on a regular basis instead of just after a detected breach. Our first construction works in any prime-order group, including elliptic curves, and involves a three-round account creation (password setup) protocol with three exponentiations per server (six for $\mathcal{LS}$). Our second construction is based on elliptic curves with bilinear maps and also offers single-round account creation with one exponentiation per back-end server and one exponentiation and one pairing computation for $\mathcal{LS}$. Both our protocols assume that the key refresh procedure has access to a special backup tape that is not connected during normal operation. In practice, this can be achieved by using smart cards or by making use of properties of modern cloud platforms, as we will explain.

Given their extreme efficiency, it is all the more surprising that we managed to prove our constructions secure under a very strong universally composable (UC) [5] notion with transient corruptions. Parties can be dynamically corrupted at any point in the protocol, even between communication rounds. Transiently corrupted parties leak their full state, but not the content of their backup tape, to the adversary and remain corrupted until the next key refresh. Permanently corrupted parties additionally leak the backup tape and cannot be recovered.

As was argued before [21, 10, 9, 6], universal composability offers important advantages over traditional game-based definitions in the particular case of password-based protocols. Namely, UC notions leave the choice of passwords to the environment, so that arbitrary distributions and dependencies between passwords are correctly modeled. This is crucial to guarantee security in real-life settings where users make typos when entering their passwords, share passwords, or use the same password for different accounts—none of which are covered by currently known game-based notions. Also, it is very unclear whether protocols can be securely composed with the non-negligible attack probabilities that game-based definitions tend to employ. We prove our constructions secure in the random-oracle model under the (gap) one-more Diffie-Hellman assumption that was previously used to prove security for blind signature [4], oblivious transfer [11], TPASS protocols [18], and set intersection protocols [19].

We achieved this rare combination of strong security and high efficiency by careful proof techniques in the random-oracle model, as well as through some of compromises in security that are very reasonable for practical use, but save on cryptographic machinery in the protocol design. First, we assume that the initialization of all servers takes place in a trusted environment where all servers are honest. During initialization, we assume that $\mathcal{LS}$ can transmit one secure message to each back-end server $\mathcal{S}_i$. This secure initialization is not hard to achieve in practice, as we explain in Section 6. Server refresh, i.e., whereby a server can recover from a transient corruption, does not require any interaction with other servers.

Second, the back-end servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ do not learn which user is logging in or whether the password was correct. This definitely limits their ability to throttle failed login attempts, but since $\mathcal{LS}$ can apply clever throttle algorithms based on user id and login results, the natural throttling of back-end servers just by requiring network communication should suffice to fend off attacks. Finally, we do not cover robustness: an adversary can make $\mathcal{LS}$ "err on the safe side" and conclude that the password was false while in fact it was correct—but not the other way around. This could be fixed by adding the same zero-knowledge or pairing verification as during account registration. This would have a major impact on efficiency, however, so we prefer to accept this rather benign attack in the model.

As a technical contribution, our scheme employs a novel technique to obtain proactive security that may be of independent interest. In a nutshell, we start off from a basic scheme that is secure under dynamic but non-transient corruptions. The basic scheme is secure under the gap one-more Diffie-Hellman assumption, but the security proof requires guessing one server at the beginning of the game that will not get corrupted during the game. This guessing induces a tightness loss in the reduction equal to the number of servers. While that loss could still be tolerated, things get worse when moving this scheme into a proactive setting. Here one would have to guess an uncorrupted server at the beginning of each epoch, so that the tightness loss blows up exponentially in the number of epochs. An easy but unsatisfying solution could be to restrict the scheme to a logarithmic number of epochs, or to only model semi-static corruptions where the adversary has to announce all servers that it wants to corrupt at the beginning of each epoch. Instead, we modify the scheme to apply random-oracle-generated blinding factors to all protocol messages, so that protocol messages do not commit servers to their keys, without ruining the overall functioning of the protocol. In the simulation, we can therefore choose a server's keys only at the moment that it is corrupted and carefully program the random oracle to ensure consistency of previous protocol messages, without having to guess anything upfront.

### *Related work.*

Our constructions are closely related to the prime-order-group and bilinear-map instantiations of TPASS by Jarecki et al.'s [18] (which they call "PPSS"). In their construction, each server has a key for a verifiable oblivious pseudo-random function (V-OPRF). For each server, the user encrypts a share of his secret under a key that is the evaluation of the VOPRF of that server on his password. The scheme supports thresholds as well as robustness thanks to the ver-

ifiability of the V-OPRF. In principle, our protocol could be seen as a variant where all servers jointly evaluate a single, distributed V-OPRF, rather than a separate one each, and where servers can update their key shares for the V-OPRF. This is not a straightforward change, however, and doesn't work for any V-OPRF in general. Moreover, whereas their protocol requires $\mathcal{LS}$ to perform $t$ V-OPRF verifications (i.e., zero-knowledge proofs or pairings) during login, our protocol doesn't need any at all, which has a tremendous impact on efficiency. Even during account creation, our protocol only involves a single verification. Finally, we prove our protocol secure in the UC framework, as opposed to their game-based model, which offers important security improvements as mentioned earlier.

## 2. PRELIMINARIES

Let $\kappa \in \mathbb{N}$ be a security parameter. A polynomial-time algorithm $\mathcal{A}$ is an algorithm that takes $\kappa$ as an implicit input and that has running time bounded by a polynomial in $\kappa$. A function $\nu(\kappa)$ is said to be negligible if for every polynomial $p(\kappa)$ there exists a $\kappa' \in \mathbb{N}$ s.t. $\nu(\kappa) < 1/p(\kappa)$ for all $\kappa > \kappa'$. For concrete security, one could typically use $\kappa = 128$.

### *Gap One-More Diffie-Hellman.*

Let $\mathbb{G}$ be a multiplicative group of prime order $q > 2^{2\kappa}$ with generator $g$. The *gap one-more Diffie-Hellman assumption* for $\mathbb{G}$ says that no polynomial-time adversary $\mathcal{A}$ has a non-negligible advantage of winning the following game. On input $(g, X)$ where $X \leftarrow g^x$ for $x \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, the adversary is given access to the following oracles:

- A target oracle $\mathsf{T}$ that returns a random target point $t \leftarrow_{\mathrm{R}} \mathbb{G}$ each time it is called.
- A computational Diffie-Hellman oracle $\mathsf{CDH}$ that, on input a group element $h \in \mathbb{G}$, returns $h^x$.
- A decisional Diffie-Hellman oracle $\mathsf{DDH}$ that, on input group elements $h, z$, returns 1 if $z = h^x$ and returns 0 otherwise.

Eventually, $\mathcal{A}$ outputs a list of tuples $((t_1, z_1), \ldots, (t_n, z_n))$. It wins the game if $t_1, \ldots, t_n$ are different target points generated by $\mathsf{T}$, $z_i = t_i^x$ for all $i = 1, \ldots, n$, and $\mathcal{A}$ made less than $n$ queries to its $\mathsf{CDH}$ oracle. The adversary's advantage $\mathbf{Adv}^{\mathrm{gomcdh}}_{\mathcal{A},\mathbb{G}}(\kappa)$ is defined as the probability that $\mathcal{A}$ wins the game.

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_{\mathrm{t}}$ be multiplicative groups of prime order $q > 2^{2\kappa}$ with generators $g_1, g_2, g_{\mathrm{t}}$, respectively, and with an efficiently computable pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_{\mathrm{t}}$ that is a non-trivial bilinear map, i.e., for all $a \in \mathbb{G}_1$, $b \in \mathbb{G}_2$, and $x, y \in \mathbb{Z}_q$, $e(a^x, b^y) = e(a, b)^{xy}$, and $e(g_1, g_2) = g_{\mathrm{t}}$.

The *one-more Diffie-Hellman assumption* for $(\mathbb{G}_1, \mathbb{G}_2)$ is defined analogously to the game above, but now $\mathcal{A}$ is given $(g_1, g_2, X = g_2^x)$ as input and the $\mathsf{T}$ and $\mathsf{CDH}$ oracles generate, respectively raise to the $x$, elements of $\mathbb{G}_1$. There is no $\mathsf{DDH}$ oracle, but depending on the type of curve, DDH may be easy via the pairing function.

The one-more DH [4, 11, 18] and the gap one-more DH [18] were used to prove the security of protocols, as well as non-adaptive variants [19]. Cheon [8] presented an attack on the (gap) one-more Diffie-Hellman assumptions that reduces the complexity of recovering $x$ from $O(\sqrt{q})$ to $O(\sqrt{q/d})$ if $d|p-1$ and $g^{x^d}$ is given to the adversary. That is, the security is reduced by a factor $O(\sqrt{d})$, so it is prudent to prevent this attack by increasing the group order with $\log d$ bits.

1. Upon input $(\mathtt{SEND}, sid, \mathcal{S}, \mathcal{R}, m)$ from $\mathcal{S}$, send $(\mathtt{SENT}, sid, \mathcal{S}, \mathcal{R}, |m|))$ to $\mathcal{A}$, generate a private delayed output $(\mathtt{SENT}, sid, \mathcal{S}, m)$ to $\mathcal{R}$ and halt.
2. Upon receiving $(\mathtt{CORRUPT}, sid, \mathcal{P})$ from $\mathcal{A}$, where $\mathcal{P} \in \{\mathcal{S}, \mathcal{R}\}$, disclose $m$ to $\mathcal{A}$. Next, if the adversary provides a value $m'$, and $\mathcal{P} = \mathcal{S}$, and no output has been yet written to $\mathcal{R}$, then output $(\mathtt{SENT}, sid, \mathcal{S}, m')$ to $\mathcal{R}$ and halt.

**Figure 1: The functionality $\mathcal{F}_{\mathrm{smt}}$.**

### *Combinatorial Secret Sharing.*

A straightforward way to create $n$-out-of-$n$ secret shares of the unity element in a group $\mathbb{G}$ among parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ is to choose shares $s_2, \ldots, s_n \leftarrow_{\mathrm{R}} \mathbb{G}$ and set $s_1 \leftarrow 1/\prod_{i=2}^n s_i$. Each party $\mathcal{P}_i$ is given secret share $s_i$; they are correct shares of one because $\prod_{i=1}^n s_i = 1$. An alternative way to compute the same shares is by choosing $s_{\{i,j\}} \leftarrow_{\mathrm{R}} \mathbb{G}$ for all $1 \le i < j \le n$ and handing $(s_{\{i,j\}})_{j=1, j\neq i}^n$ to $\mathcal{P}_i$ for $i = 1, \ldots, n$. Note that each share $s_{\{i,j\}}$ is known to parties $\mathcal{P}_i$ and $\mathcal{P}_j$. Party $\mathcal{P}_i$ computes its share of unity $s_i \leftarrow \prod_{j=1, j\neq i}^n s_{\{i,j\}}^{\Delta_{i,j}}$, where $\Delta_{i,j} = 1$ if $i < j$ or $\Delta_{i,j} = -1$ otherwise. One can easily see that $\prod_{i=1}^n s_i = \prod_{i=1}^n \prod_{j=1, j\neq i}^n s_{\{i,j\}}^{\Delta_{i,j}} = \prod_{i=1}^n \prod_{j=i+1}^n s_{\{i,j\}} \cdot s_{\{i,j\}}^{-1} = 1$. This construction is particularly interesting because it offers a practical way to non-interactively generate arbitrarily many shares of unity by letting $s_{\{i,j\}}$ be generated pseudorandomly from a seed that is known to parties $\mathcal{P}_i$ and $\mathcal{P}_j$ only.

### *Secure Message Transmission.*

The ideal functionality for secure message transmission $\mathcal{F}_{\mathrm{smt}}$ depicted in Figure 1 allows a sender $\mathcal{S}$ to send a private and integrity-protected message to a receiver $\mathcal{R}$. It is the special case of Canetti's [5] functionality for leakage function $l(m) = |m|$

### *Pseudo-Random Generators.*

A pseudo-random generator (PRG) is a function $\mathsf{PRG} : \mathcal{D} \to \mathcal{R}$ where no polynomial-time adversary can distinguish the output of $\mathsf{PRG}$ on a random input from a truly random string. The advantage $\mathbf{Adv}^{\mathrm{pr}}_{\mathcal{A},\mathsf{PRG}}(\kappa)$ of an adversary $\mathcal{A}$ is defined as $\big|\Pr[1 = \mathcal{A}(y) \ : \ x \leftarrow_{\mathrm{R}} \mathcal{D}, y \leftarrow \mathsf{PRG}(x)] - \Pr[1 = \mathcal{A}(y) \ : \ y \leftarrow_{\mathrm{R}} \mathcal{R}]\big|$.

### *Message Authentication Codes.*

A message authentication code (MAC) is a function $\mathsf{MAC} : \mathcal{K} \times \{0,1\}^* \to \mathcal{T}$ that on input a key $\mu$ and a message $m \in \{0,1\}^*$ returns a tag $\tau$. We say that $\mathsf{MAC}$ is unforgeable against chosen-message attack if all polynomial-time adversaries $\mathcal{A}$ have negligible advantage $\mathbf{Adv}^{\mathrm{ufcma}}_{\mathcal{A},\mathsf{MAC}}(\kappa)$ defined as $\Pr[\tau = \mathsf{MAC}(\mu, m) \wedge m \notin Q \ : \ \mu \leftarrow_{\mathrm{R}} \mathcal{K} \ ; \ (m, \tau) \leftarrow_{\mathrm{R}} \mathcal{A}^{\mathsf{MAC}(\mu,\cdot)}]$, where $Q$ is the set of messages that $\mathcal{A}$ submitted to its $\mathsf{MAC}(\mu, \cdot)$ oracle.

## 3. SECURITY DEFINITION

In this section we now formally define our distributed password verification scheme by describing its ideal functionality in the universal composability (UC) framework [5]. Roughly, a protocol is said to securely realize an ideal functionality $\mathcal{F}$ if an environment $\mathcal{E}$ cannot distinguish whether it is interacting with the real protocol $\pi$ and a real adversary $\mathcal{A}$, or with $\mathcal{F}$ and a simulator $\mathcal{SIM}$. We denote the

probability that $\mathcal{E}$ outputs 1 in both worlds as $\mathbf{Real}^{\pi}_{\mathcal{E},\mathcal{A}}(\kappa)$ and $\mathbf{Ideal}^{\mathcal{F}}_{\mathcal{E},\mathcal{SIM}}(\kappa)$, respectively.

First, let's briefly recall the goal of our distributed password verification system, before we present our ideal functionality. In our system, a login server $\mathcal{LS}$ is the main access point where users provide their username *uid* and password *pwd*. Once a user has created an account for such a username-password combination with the $\mathcal{LS}$, he can subsequently login by providing the correct username and password again. Thus, the login server must be able to verify whether a password attempt $pwd'$ matches the stored password *pwd* or not. Our goal is to provide that functionality without introducing a single point of failure that, when corrupted, leaks all passwords to the adversary or allows offline attacks against them. Therefore, $\mathcal{LS}$ is assisted by $n$ servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ running in the back-end. Those servers have to actively contribute to allow the verification of a password verification and thus can refuse whenever they notice suspicious activity that might be aimed at an online password guessing attack. Note that password changes are not explicitly modeled; these can always be implemented by performing a login under the old password followed by an account creation with the new password (if necessary for a new username, e.g., containing an increased index).

To model a realistic setting, we consider active and adaptive corruptions, allowing the adversary to take control of any initially honest party at any time. We distinguish between transient and permanent corruptions. Transiently corrupted parties do not leak the contents of their backup tape and can recover from an attack by going through a refresh procedure. In a permanent corruption, the backup tape is leaked to the adversary, and there is no way to recover, meaning that the server is corrupted for all future epochs. As long as the adversary does not corrupt *all* servers $\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n$ in the *same* epoch, our distributed password verification scheme protects the stored passwords, meaning that the adversary neither learns the passwords nor is able to perform offline attacks on them.

## 3.1 Ideal Functionality

The detailed description of our ideal functionality $\mathcal{F}_{\text{dpv}}$ is given in Figure 2. When describing our functionality, we use the following writing conventions to reduce repetitive notation:

- The functionality ignores all inputs other than `INIT` until the instance is active. Once the instance is active, it ignores further calls to `INIT`.
- For all interfaces (except `INIT`), the ideal functionality only considers the first input for each *ssid* and for each originating party $\mathcal{P}$. Subsequent inputs to the same interface for the same *ssid* coming from the same party $\mathcal{P}$ are ignored.
- At each invocation, the functionality checks that $sid = (\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n, sid')$ for server identities $\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n$, and $sid' \in \{0,1\}^*$. Also, whenever we say that the functionality receives input from or provides output to $\mathcal{LS}$ or $\mathcal{S}_i$, we mean $\mathcal{LS}$ or $\mathcal{S}_i$ as specified in the *sid*.
- When we say that the functionality "looks up a record", we implicitly understand that if the record is not found, $\mathcal{F}$. ignores the input and returns control to the environment.
- We assume that the session identifier *sid* and sub-session identifiers *ssid* given as input to our functionality are globally unique, and that honest parties drop any inputs with (sub)session identifiers that are not locally unique.

We now also describe the behavior of the main interfaces in a somewhat informal manner to clarify the security properties that our functionality provides.

### *Account Creation.*

The creation of a new account for username *uid* and password *pwd* is initiated by the login server $\mathcal{LS}$ and requires the active approval of all $n$ back-end servers $\mathcal{S}_1, \ldots \mathcal{S}_n$ (if $\mathcal{LS}$ is honest). Several account creation (and login) sessions can be run in parallel; a unique sub-session identifier *ssid* is input to all create and login related interfaces and identifies the respective sub-session.

**2:** The `CREATE` interface allows the login server to trigger the creation of a new user record (`setup`, *ssid*, *uid*, *pwd*, *proceed*, *finished*). The two flags, *proceed* and *finished*, reflect the status of the record and are both initially set to 0.

**3:** The `PROCEED` interface can be invoked by the back-end servers $\mathcal{S}_i$ to signal their willingness to continue an account creation (or login) session, identified by the given *ssid*. Only if *all* $n$ servers have given the ok to proceed, the `setup` (or `login`) account associated with *ssid* gets activated for finalization, which is modeled by setting *proceed* $\leftarrow 1$.

Awaiting explicit approval of all servers gives each server the opportunity to throttle or block a session if they detect some suspicious behaviour, which is crucial to prevent offline attacks against the password.

If the login server is corrupt, an activated account creation (or login) session also increases the global *guesses* counter, giving the adversary one more password guess (via the interface `PWDGUESS` ).

**4:** The `CREATEOK` interface can be invoked by the adversary to allow completion of the `setup` account for *ssid*, which is realized by setting *finished* $\leftarrow 1$. However, if the login server is honest, the adversary can only complete records for those *ssid*'s to which all servers have already agreed to proceed. This restriction does not hold for a corrupt login server though, as in the real world, the corrupt $\mathcal{LS}$ could always create as many (bogus) user records as he wants. Whenever the $\mathcal{LS}$ gets honest again, the `login` will most likely fail for such bogus records though. This is modeled accordingly in our `RESULT` interface where the adversary can always make the verification fail for such forged accounts.

### *Login.*

To verify whether a provided username-password combination $uid, pwd'$ is correct, the login server $\mathcal{LS}$ can initiate a login request. Then, if all servers agree to proceed (using the 3.`PROCEED` interface), the adversary can instruct the ideal functionality to inform the $\mathcal{LS}$ whether the provided password attempt $pwd'$ matches the setup password *pwd* stored for *uid*. Again, each login sub-session is identified via a unique $ssid'$.

**5:** The `LOGIN` interface is invoked by the $\mathcal{LS}$ on input $ssid'$, $uid, pwd'$ and triggers the creation of a new login record (`login`, $ssid'$, $uid, pwd'$, $proceed'$) with *proceed* $\leftarrow 0$.

**6:** The `RESULT` interface allows the adversary to instruct $\mathcal{F}_{\text{dpv}}$ to release the result of the password verification to the

1. `Initialization.`
On input (`INIT`, *sid*) from login server $\mathcal{LS}$:

- Record this instance as active, set *guesses* $\leftarrow 0$ and create a record (`corrupt`, $\mathbf{TC}, \mathbf{PC}$) with $\mathbf{TC}, \mathbf{PC} \leftarrow \emptyset$.
- Send (`INIT`, *sid*) to $\mathcal{A}$.

---

2. `Account Creation Request.`
On input (`CREATE`, *sid*, *ssid*, *uid*, *pwd*) from login server $\mathcal{LS}$:

- If $\mathcal{LS}$ is honest, and a `setup` record for *uid* exists, then ignore this input.
- Create a new record (`setup`, *ssid*, *uid*, *pwd*, *proceed*, *finished*) with *proceed* $\leftarrow 0$ and *finished* $\leftarrow 0$.
- Send (`CREATE`, *sid*, *ssid*, *uid*) to $\mathcal{A}$.

3. `Server Proceed` *(used in account creation and login).*
On input (`PROCEED`, *sid*, *ssid*) from a server $\mathcal{S}_i$:

- Look up `setup` or `login` record for *ssid*.
- If `PROCEED` messages from *all* $n$ servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ have been received for *ssid*, update the `login` or `setup` record for *ssid* by setting *proceed* $\leftarrow 1$, and if $\mathcal{LS}$ is corrupt, set *guesses* $\leftarrow$ *guesses* $+ 1$.
- Send (`PROCEED`, *sid*, *ssid*, $\mathcal{S}_i$) to $\mathcal{A}$.

4. `Creation Result.`
On input (`CREATEOK`, *sid*, *ssid*) from $\mathcal{A}$:

- Look up setup record (`setup`, *ssid*, *uid*, *pwd*, *proceed*, *finished*) for *ssid*.
- If the $\mathcal{LS}$ is honest, only proceed if *proceed* $= 1$.
- Update the record by setting *finished* $\leftarrow 1$ and output (`CREATEOK`, *sid*, *ssid*) to $\mathcal{LS}$.

---

5. `Login Request.`
On input (`LOGIN`, *sid*, *ssid*$'$, *uid*, *pwd*$'$) from $\mathcal{LS}$:

- Create a new record (`login`, *ssid*$'$, *uid*, *pwd*$'$, *proceed*$'$) with *proceed*$'$ $\leftarrow 0$. and send (`LOGIN`, *sid*, *ssid*$'$, *uid*) to $\mathcal{A}$.

6. `Login Result.`
On input (`RESULT`, *sid*, *ssid*$'$, *fail*) from adversary $\mathcal{A}$:

- Look up login record (`login`, *ssid*$'$, *uid*, *pwd*$'$, *proceed*$'$) for *ssid*$'$ and the corresponding setup record (`setup`, *ssid*, *uid*, *pwd*, *proceed*, *finished*) for *uid*. Ignore this input if *proceed*$'$ $= 0$ or *finished* $= 0$.
- If *pwd* $\neq$ *pwd*$'$, or if *fail* $= 1$ and at least one server from $\mathcal{S}_1, \ldots, \mathcal{S}_n$ is corrupt or *proceed* $= 0$, then set *pwdok* $\leftarrow 0$. Else, set *pwdok* $\leftarrow 1$.
- If $\mathcal{LS}$ is corrupt, set *guesses* $\leftarrow$ *guesses* $- 1$.
- Delete the login record for *ssid*$'$ and send a delayed output (`RESULT`, *sid*, *ssid*$'$, *pwdok*) to $\mathcal{LS}$.

---

7. `SSID Timeout.`
On input (`TIMEOUT`, *sid*, *ssid*) from $\mathcal{LS}$:

- If a `login` record for *ssid* exists, delete the record.
- If a `setup` record (`setup`, *ssid*, *uid*, *pwd*, *proceed*, *finished*) for *ssid* and with *finished* $= 0$ exists, then delete the record.

---

8. `Server Corruption.`
On input (`CORRUPT`, *sid*, $\mathcal{S}$, *mode*) from $\mathcal{A}$, where $\mathcal{S} \in \{\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n\}$ and *mode* $\in \{$`trans`, `perm`$\}$:

- Look up record (`corrupt`, $\mathbf{TC}, \mathbf{PC}$).
- If *mode* $=$ `trans`, update the record with $\mathbf{TC} \leftarrow \mathbf{TC} \cup \{\mathcal{S}\}$.
- If *mode* $=$ `perm`, update the record with $\mathbf{PC} \leftarrow \mathbf{PC} \cup \{\mathcal{S}\}$.
- If $\mathbf{TC} \cup \mathbf{PC} = \{\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n\}$ then set *guesses* $\leftarrow \infty$.
- If $\mathcal{S} = \mathcal{LS}$, then assemble $\mathbf{L} \leftarrow \{(ssid_i, uid_i, pwd_i)\}$ for all ongoing sessions, i.e., extract the passwords from all setup records (`setup`, $ssid_i, uid_i, pwd_i, proceed_i, finished_i$) with $finished_i = 0$ and all stored login records (`login`, $ssid_i, uid, pwd_i, proceed_i$). If $\mathcal{S} \neq \mathcal{LS}$, set $\mathbf{L} \leftarrow \emptyset$.
- Send (`CORRUPT`, *sid*, $\mathbf{L}$) to $\mathcal{A}$.

9. `Server Refresh.`
On input (`REFRESH`, *sid*) from $\mathcal{LS}$:

- Look up the corruption record (`corrupt`, $\mathbf{TC}, \mathbf{PC}$) and update the record to (`corrupt`, $\emptyset, \mathbf{PC}$).
- Delete all `setup` records with *finished* $= 0$ and all `login` records.
- Send (`REFRESH`, *sid*, $\mathcal{S}$) to $\mathcal{A}$.

10. `Password Guessing.`
On input of (`PWDGUESS`, *sid*, *uid*, *pwd*$^*$) from adversary $\mathcal{A}$:

- Look up the setup record (`setup`, *ssid*, *uid*, *pwd*, *proceed*, *finished*) with *finished* $= 1$.
- If *guesses* $= 0$ set *pwdok* $\leftarrow \perp$.
- Else, set *guesses* $\leftarrow$ *guesses* $- 1$ and, if *pwd*$^*$ $=$ *pwd*, set *pwdok* $\leftarrow 1$ , otherwise set *pwdok* $\leftarrow 0$.
- Send (`PWDGUESS`, *sid*, *uid*, *pwdok*) to $\mathcal{A}$.

**Figure 2: Ideal Functionality $\mathcal{F}_{\text{dpv}}$ with $sid = (\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n, sid')$.**

login server $\mathcal{LS}$. The adversary can do so only for those login sessions for which *all* servers already gave the ok to proceed, i.e., the `login` record for *ssid*$'$ contains *proceed*$'$ $= 1$ (set via the 3.`PROCEED` interface). Note that here the check whether *proceed*$'$ $= 1$ is also required for a corrupt $\mathcal{LS}$, as otherwise a corrupt login server could offline attack the user passwords.

If all servers agreed to proceed, the ideal functionality then looks up the corresponding setup record (`setup`, *ssid*, *uid*, *pwd*, *proceed*, 1) for *uid* and sets the verification result to *pwdok* $\leftarrow 1$ if the password match, i.e., *pwd* $=$ *pwd*$'$, and *pwdok* $\leftarrow 0$ otherwise. If at least one back-end server $\mathcal{S}_i \in \{S_1, \ldots, \mathcal{S}_n\}$ is corrupt, or the account was created by a corrupt $\mathcal{LS}$, then $\mathcal{A}$ can enforce a negative result *pwdok* $\leftarrow 0$, by passing *fail* $= 1$ as extra input. However, the adversary can only turn a successful result into a failed one, but not vice versa, i.e., he cannot make a mismatch of the passwords look like a match.

Further, if the login result is delivered to a corrupt $\mathcal{LS}$, then the global *guesses* counter is decreased. Recall that *guesses* gets increased in the `PROCEED` interface when $\mathcal{LS}$ is corrupt and all servers want to proceed with *ssid*$'$. Thus, for login, the adversary can basically choose whether it wants to use that "guess" to complete the login request, or to perform a password guess at an arbitrary user account via the `PWDGUESS` interface. Note that for the latter, the $\mathcal{LS}$ can already be honest again (if a refresh took place), i.e., that the adversary can keep the password guess for a later time.

Finally, when a login session is completed, the corresponding `login` record is deleted. This is important for corruption, because an adversary who corrupts the $\mathcal{LS}$ learns the passwords of all ongoing (or interrupted) setup and login sessions.

### *Time Out.*

**7:** The `TIMEOUT` interface allows the login server to terminate ongoing account creation or login sessions. The ideal functionality then deletes the login or setup record for the specified *ssid*. For setup accounts this is only possible for incomplete records, i.e., where *finished* $= 0$. This models the desired ability of a real world $\mathcal{LS}$ to abandon sessions when it hasn't received all server responses in an appropriate time, e.g., if a server refuses to proceed, or the response got intercepted by the adversary.

*(Un)Corruption & Password Guessing.*

Our functionality supports adaptive and transient as well as permanent corruptions. The environment can, at any time, decide to corrupt any initially honest server $\mathcal{LS}$ or $\mathcal{S}_i$ and specify the corruption type. In a *transient corruption*, the party remains corrupted until the next refresh of that server. Parties that are *permanently corrupted* cannot be recovered and remain corrupted until the end of the game. As long as not all parties are corrupted at the same time (regardless of whether they are transiently or permanently corrupted), the adversary has only very limited power for attacking the stored passwords, which is modeled by the password guessing interface. Note that we do not follow the standard UC corruption model which, upon corruption of a party, gives all past in- and outputs to the adversary. This is clearly not desirable in the given context of protecting bulk user passwords that are processed by the login server. Thus, we aim at stronger security guarantees, despite adaptive corruptions, which is modeled by the following interfaces.

**8:** The `CORRUPT` interface allows the adversary to transiently ($mode = \texttt{trans}$) or permanently ($mode = \texttt{perm}$) corrupt any party $\mathcal{S} \in \{\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n\}$. If $\mathcal{S} = \mathcal{LS}$, i.e., the adversary decided to corrupt the login server, it learns the passwords of all ongoing setup and login sessions. When all parties are corrupted at the same time, the adversary is still not given the stored passwords. Instead, the functionality sets $guesses \leftarrow \infty$, which gives the adversary unlimited access to the `PWDGUESS` interface described below.

**9:** By invoking the `REFRESH` interface, all transiently corrupted servers become honest again. From then on, inputs and outputs of non-permanently-corrupted servers go to the environment, instead of to the adversary (until a server is corrupted again). Once the adversary has corrupted all parties at the same time, however, the unlimited capabilities for offline attacks remain. Further, the functionality deletes all records of incomplete setup or login sessions.

**10:** The `PWDGUESS` interface is the only possibility of the adversary to attack the stored user passwords. The access to this interface is limited by the *guesses* counter. As long as not all parties got corrupted at the same time, *guesses* is only increased when a corrupt login server started a new setup or login session, and all servers agreed to proceed. For each such session, the adversary gets one more guess against a password for a *uid* of his choice.

# 4. OUR FIRST CONSTRUCTION

The basic idea of the protocol is so simple that it can actually be explained in a couple of lines. Each server $\mathcal{S}_i \in \{\mathcal{LS} = \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n\}$ has its own secret exponent $K_i$. The "password hash" stored by $\mathcal{LS}$ for user *uid* and password *pwd* is $\mathrm{G}(uid, pwd, \mathrm{H}(uid, pwd)^K)$, where $K = \sum_{i=0}^{n} K_i \bmod q$ and G and H are hash functions. To compute this value, $\mathcal{LS}$ chooses a random nonce $N \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and sends $u \leftarrow \mathrm{H}(uid, pwd)^N$ to $\mathcal{S}_i$, who responds with $v_i \leftarrow u^{K_i}$ so that $\mathcal{LS}$ can compute $v \leftarrow \prod_{i=0}^{n} v_i^{1/N} = \mathrm{H}(uid, pwd)^K$. This computation is performed at account creation and again at each login to check that the recomputed value matches the stored hash. To refresh their keys, all servers add a pseudorandomly and non-interactively generated share of zero to their $K_i$ so that the individual keys are independent of those in the previous epoch, but their sum $K = \sum_{i=0}^{n} K_i \bmod q$ remains constant.

There are two problems that slightly complicate the protocol, however. First, to obtain proactive security for arbitrarily many epochs, it is crucial that previous protocol messages do not commit a server $\mathcal{S}_i$ to its secret key $K_i$. Non-committing encryption [7] doesn't help, because the adversary could corrupt $\mathcal{LS}$ and decrypt the elements $v_i$ that commit $\mathcal{S}_i$ to its key $K_i$. Instead, we apply a clever combination of blinding factors to each protocol message that preserve the overall result of the protocol, but that avoid honest servers from having to commit to their keys.

Second, a corrupt server $\mathcal{S}_i$ may misbehave and use a different exponent $K_i' \neq K_i$ during its computation of $v_i$. This isn't much of a problem if it happens during login: at most, it could make an honest $\mathcal{LS}$ erroneously conclude that a correct password was incorrect, but our functionality explicitly tolerates such "true negatives". A server using a different exponent during account creation is more problematic, however. While there doesn't seem to be an obvious attack, the reduction to the gap one-more Diffie-Hellman problem ceases to go through. Normally, the reduction works by inserting CDH target points as responses to $\mathrm{H}(\cdot)$ queries and observing the adversary's $\mathrm{G}(\cdot)$ queries for CDH solutions $\mathrm{H}(uid, pwd)^K$. When $\mathcal{LS}$ stores a password hash $\mathrm{G}(uid, pwd, \mathrm{H}(uid, pwd)^{K'})$ for $K' \neq K$, however, the reduction can no longer extract $\mathrm{H}(uid, pwd)^K$ when the adversary guesses the password.

To prevent this, $\mathcal{LS}$ must verify at account creation that the obtained value $v$ is indeed $\mathrm{H}(uid, pwd)^K$. In our second construction, the $\mathcal{LS}$ can do so using a pairing computation. In our first protocol, we let the servers engage in a distributed interactive zero-knowledge protocol allowing $\mathcal{LS}$ to check that the "overall" exponent $K$ was correct, but without committing servers to their individual exponents $K_i$.

## 4.1 Scheme

Let $\mathbb{G}$ be a multiplicative group of prime order $q > 2^{2\kappa}$ with generator $g$. Let $\mathrm{H} : \{0,1\}^* \times \{0,1\}^* \to \mathbb{G}$, $\mathrm{G} : \{0,1\}^* \times \{0,1\}^* \times \mathbb{G} \to \{0,1\}^{2\kappa}$, $\mathrm{C} : \mathbb{Z}_q \to \{0,1\}^{2\kappa}$, $\mathrm{B}_0 : \{0,1\}^\kappa \times \mathbb{N} \to \mathbb{G}$, $\mathrm{B}_1 : \{0,1\}^\kappa \times \mathbb{N} \to \mathbb{G}$, $\mathrm{B}_2 : \{0,1\}^\kappa \times \mathbb{N} \to \mathbb{G}$, and $\mathrm{B}_3 : \{0,1\}^\kappa \times \mathbb{N} \to \mathbb{Z}_q$ be hash functions modeled as random oracles. Let $\mathsf{PRG} : \{0,1\}^\kappa \to \{0,1\}^\kappa \times \mathbb{Z}_q \times \{0,1\}^\kappa \times \{0,1\}^\kappa$ be a pseudo-random generator and $\mathsf{MAC} : \{0,1\}^\kappa \times \{0,1\}^* \to \mathcal{T}$ be a message authentication code.

Initialization takes place in a secure environment where all parties are uncorrupted and can communicate securely over a secure message transmission functionality $\mathcal{F}_{\mathrm{smt}}$. During initialization and refresh, each party additionally has read/write access to a backup tape *backup*. As the backup tape is not used during account creation and login, it is easier to protect by disconnecting it during regular operation. The difference between a transient and a permanent corruption in the real world is that, in a transient corruption, the adversary is given control of that party and its current state information, but not its backup tape. In a permanent corruption, the adversary is additionally given the content of the backup tape. Rather than assuming that parties revert to a fixed default state when recovering from corruption, as done in previous works [1], we assume that a party refreshes by starting from a clean copy of its code and deriving its new state information from its backup tape and its last state be-

fore refresh (which may have been tampered with by the adversary).

Once initialization is finished, the servers $\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n$ communicate over an untrusted network, where messages can be arbitrarily observed, modified, and delayed by the adversary, but all messages are integrity-protected with a MAC. Our protocol provides $\mathcal{LS}$ with a shared MAC key $\mu_{\{0,i\}}$ with each server $\mathcal{S}_i$, $i = 1, \ldots, n$. Whenever the description below says that "$\mathcal{LS}$ sends $m$ to $\mathcal{S}_i$", it actually means that $\mathcal{LS}$ computes $\tau \leftarrow \mathsf{MAC}(\mu_{\{0,i\}}, (m, \mathcal{LS}))$ and sends $(m, \tau)$ to $\mathcal{S}_i$. Whenever it says that "$\mathcal{S}_i$ receives $m$ from $\mathcal{LS}$", it actually means that $\mathcal{S}_i$ receives $(m, \tau)$ and checks that $\tau = \mathsf{MAC}(\mu_{\{0,i\}}, (m, \mathcal{LS}))$, ignoring the message $m$ if that is not the case. The communication in the other direction from server $\mathcal{S}_i$ back to $\mathcal{LS}$ is protected in the same way with the same MAC key $\mu_{\{0,i\}}$.

In the protocol below, the state information of each server $\mathcal{S}_i \in \{\mathcal{LS} = \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n\}$ contains a list of blinding seeds $s_{\{i,j\}}$ for $j = 1, \ldots, n, j \neq i$ that are used to generate random shares of the unity element in $\mathbb{G}$ or of zero in $\mathbb{Z}_q$ using the combinatorial secret sharing scheme recalled in the preliminaries. All servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ maintain an associative array *USED* to keep track of previously used subsession identifiers *ssid*. In each account creation or login session, the servers derive fresh shares $\beta_{i,0}, \ldots, \beta_{i,3}$ of unity or zero using the random oracles $\mathrm{B}_0, \ldots, \mathrm{B}_3$ applied to $s_{\{i,j\}}$ and *ssid*, and use these shares as blinding factors for their protocol messages so that $\prod_{i=0}^{n} \beta_{i,k} = 1$ for $k = 0, 1, 2$ and $\sum_{i=0}^{n} \beta_{i,3} = 0 \bmod q$. More precisely, $\mathcal{S}_i$'s blinding factors are computed as $\beta_{i,k} \leftarrow \prod_{j=0, j\neq i}^{n} \mathrm{B}_k(s_{\{i,j\}}, ssid)^{\Delta_{i,j}}$ for $k = 0, 1, 2$ and as $\beta_{i,3} \leftarrow \sum_{j=0, j\neq i}^{n} \Delta_{i,j} \mathrm{B}_3(s_{\{i,j\}}, ssid) \bmod q$, where $\Delta_{i,j} = 1$ if $i < j$ and $\Delta_{i,j} = -1$ otherwise.

**Initialization.** During initialization, all servers are uncorrupted and can communicate through the secure message transmission functionality $\mathcal{F}_{\mathrm{smt}}$.

1. $\underline{\mathcal{LS}}$: The $\mathcal{LS}$ generates and distributes master keys $mk_{\{i,j\}}$ for all servers in the system. It also generates a secret key $K$ for a joint public key $L$. It uses the master keys to compute the initial key share $K_0$ of $K$ for $\mathcal{LS}$, as well as its initial blinding seeds $s_{\{0,j\}}$. The key share and blinding seeds are kept in the initial state of $\mathcal{LS}$, the master keys $mk_{\{0,j\}}$ are written to the backup tape.
   - On input $(\texttt{INIT}, sid)$, check that $sid = (\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n, sid')$ for his own identity $\mathcal{LS}$ and server identities $\mathcal{S}_1, \ldots, \mathcal{S}_n$.
   - For all $0 \leq i < j \leq n$, choose a master key $mk_{\{i,j\}} \leftarrow_{\mathrm{R}} \{0,1\}^{\kappa}$. The master key $mk_{\{i,j\}}$ will be known only to servers $\mathcal{S}_i$ and $\mathcal{S}_j$, so that each pair of servers $\{i, j\} \subseteq \{0, \ldots, n\}$ will have a common master key that is unknown to the other servers.
   - For $i = 1, \ldots, n$, securely send $(mk_{\{i,j\}})_{j=0,j\neq i}^{n}$ to server $\mathcal{S}_i$ by providing input $(\texttt{SEND}, (\mathcal{LS}, \mathcal{S}_i, sid), \mathcal{LS}, \mathcal{S}_i, (mk_{\{i,j\}})_{j=0,j\neq i}^{n})$ to $\mathcal{F}_{\mathrm{smt}}$ for $i = 1, \ldots, n$.
   - For all $j = 1, \ldots, n$, compute $(mk'_{\{0,j\}}, \delta_{\{0,j\}}, s_{\{0,j\}}, \mu_{\{0,j\}}) \leftarrow \mathsf{PRG}(mk_{\{0,j\}})$.
   - Choose $K \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and set $L \leftarrow g^K$. Compute $K_0 \leftarrow K + \sum_{j=1}^{n} \delta_{\{0,j\}} \bmod q$.
   - Initialize $BUSY$, $USED_0$, and the password hash table $PH$ as empty associative arrays and store $st_0 = (K_0, (s_{\{0,j\}})_{j=1}^{n}, (\mu_{\{0,j\}})_{j=1}^{n}, L, PH, BUSY, USED_0)$ as initial state and store $backup_0 \leftarrow (K_0, (mk'_{\{0,j\}})_{j=1}^{n}, L, PH)$ on the backup tape.

| $\underline{\mathcal{LS}}$: | | $\underline{\mathcal{S}_i}$: |
|---|---|---|
| $N \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ | | |
| $u \leftarrow \mathrm{H}(uid, pwd)^N$ | | |
| $c \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, $ch \leftarrow \mathrm{C}(c)$ | $\xrightarrow{u, ch}$ | |
| $v_0 \leftarrow u^{K_0} \cdot \beta_{0,0}$ | | $v_i \leftarrow u^{K_i} \cdot \beta_{i,0}$ |
| $r_0 \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ | | $r_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ |
| $R_{1,0} \leftarrow g^{r_0} \cdot \beta_{0,1}$ | | $R_{1,i} \leftarrow g^{r_i} \cdot \beta_{i,1}$ |
| $R_{2,0} \leftarrow u^{r_0} \cdot \beta_{0,2}$ | $\xleftarrow{v_i, R_{1,i}, R_{2,i}}$ | $R_{2,i} \leftarrow u^{r_i} \cdot \beta_{i,2}$ |
| $s_0 \leftarrow K_0 c + r_0$ | $\xrightarrow{c}$ | Check $\mathrm{C}(c) = ch$ |
| $+ \beta_{0,3} \bmod q$ | $\xleftarrow{s_i}$ | $s_i \leftarrow K_i c + r_i + \beta_{i,3} \bmod q$ |
| $v \leftarrow \prod_{i=0}^{n} v_i^{1/N}$ | | |
| $R_1 \leftarrow \prod_{i=0}^{n} R_{1,i}$, $R_2 \leftarrow \prod_{i=0}^{n} R_{2,i}$ | | |
| $s \leftarrow \sum_{i=0}^{n} s_i \bmod q$ | | |
| If $g^s = L^c R_1$ and $u^s = v^{Nc} R_2$ then $PH[uid] \leftarrow \mathrm{G}(uid, pwd, v)$ | | |

**Figure 4: The account creation protocol. All communication between $\mathcal{LS}$ and $\mathcal{S}_i$ is integrity-protected with a MAC key $\mu_{\{0,i\}}$. See the text for more information on the blinding factors $\beta_{i,k}$.**

2. $\underline{\mathcal{S}_i}$: Each server stores the received master keys $mk_{\{i,j\}}$ in backup memory and derives the initial state for $\mathcal{S}_i$.
   - Upon input $(\texttt{SENT}, (\mathcal{LS}, \mathcal{S}_i, sid), \mathcal{LS}, (mk_{\{i,j\}})_{j=0,j\neq i}^{n})$ from $\mathcal{F}_{\mathrm{smt}}$, for all $j = 0, \ldots, n$, $j \neq i$, compute $(mk'_{\{i,j\}}, \delta_{\{i,j\}}, s_{\{i,j\}}, \mu_{\{i,j\}}) \leftarrow \mathsf{PRG}(mk_{\{i,j\}})$.
   - Compute the initial key share as $K_i \leftarrow \sum_{j=0,j\neq i}^{n} \Delta_{i,j} \delta_{\{i,j\}} \bmod q$, where $\Delta_{i,j}$ is as defined above.
   - Initialize $USED_i$ as an empty associative array and store $st_i \leftarrow (K_i, (s_{\{i,j\}})_{j=0,j\neq i}^{n}, \mu_{\{0,i\}}, USED_i)$ as initial state and store $backup_i \leftarrow (K_i, (mk'_{\{i,j\}})_{j=0,j\neq i}^{n})$ on the backup tape.

**Account creation.** To create an account for user *uid* with password *pwd*, the $\mathcal{LS}$ runs the following protocol with all $n$ servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$:

1. $\underline{\mathcal{LS}}$: The $\mathcal{LS}$ sends a blinded password hash and a challenge hash to all servers.
   - On input $(\texttt{CREATE}, sid, ssid, uid, pwd)$, check whether $PH[uid]$, $BUSY[uid]$ or $USED_0[ssid]$ is already defined. If so, abort. Else, set and store $BUSY[uid] \leftarrow 1$ and $USED_0[ssid] \leftarrow 1$. (Note that we already assumed that servers check that *ssid* is locally unique, but since it is crucial for the security of our protocol, we make these checks explicit here.)
   - Generate a random nonce $N \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and a random challenge $c \leftarrow_{\mathrm{R}} \mathbb{Z}_q$. Compute $u \leftarrow \mathrm{H}(uid, pwd)^N$ and $ch \leftarrow \mathrm{C}(c)$.
   - Send $(ssid, u, ch)$ to all servers for $i = 1, \ldots, n$.
   - Store $(uid, pwd, N, u, c)$ associated with *ssid*.
2. $\underline{\mathcal{S}_i}$: Each server sends a blinded response using its secret key share and the blinded first move of a zero-knowledge proof.
   - On input $(\texttt{PROCEED}, sid, ssid)$ from the environment and after having received $(ssid, u, ch)$ from $\mathcal{LS}$, check that $USED_i[ssid]$ is undefined. If not, abort.
   - Compute $v_i \leftarrow u^{K_i} \cdot \prod_{j=0,j\neq i}^{n} \mathrm{B}_0(s_{\{i,j\}}, ssid)^{\Delta_{i,j}}$ and set and store $USED_i[ssid] \leftarrow 1$.
   - Compute $R_{1,i} \leftarrow g^{r_i} \cdot \prod_{j=0,j\neq i}^{n} \mathrm{B}_1(s_{\{i,j\}}, ssid)^{\Delta_{i,j}}$ and $R_{2,i} \leftarrow u^{r_i} \cdot \prod_{\substack{j=0 \\ j\neq i}}^{n} \mathrm{B}_2(s_{\{i,j\}}, ssid)^{\Delta_{i,j}}$ where $r_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$.

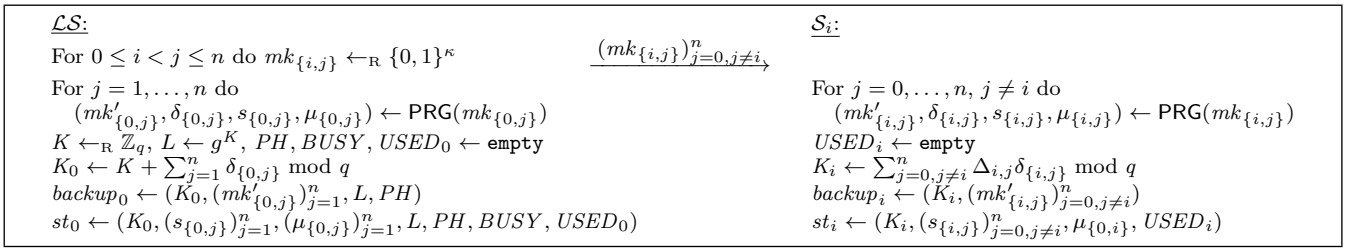


**Figure 3: The initialization protocol. All communication takes place via $\mathcal{F}_{\text{smt}}$.**

- Respond by sending $(ssid, v_i, R_{1,i}, R_{2,i})$ to $\mathcal{LS}$.
- Store $(r_i, ch)$ associated with $ssid$.

3. $\underline{\mathcal{LS}}$: The $\mathcal{LS}$ sends the challenge for the zero-knowledge proof.
   - After having received $(ssid, v_i, R_{1,i}, R_{2,i})$ from servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$, retrieve $(uid, pwd, N, u, c)$ associated with $ssid$. Abort if it doesn't exist.
   - Update the information stored with $ssid$ to $(uid, pwd, N, u, c, (v_i, R_{1,i}, R_{2,i})_{i=1}^n)$.
   - Send $(ssid, c)$ to all servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$.
4. $\underline{\mathcal{S}_i}$: Each server checks the challenge hash from the previous round and sends the blinded last move of a zero-knowledge proof.
   - When receiving $(ssid, c)$ from $\mathcal{LS}$, retrieve $(r_i, ch)$ associated with $ssid$. Abort if it doesn't exist.
   - If $\mathrm{C}(c) \neq ch$, abort.
   - Compute $s_i \leftarrow K_i c + r_i + \sum_{j=0,j\neq i}^n \Delta_{i,j} \mathrm{B}_3(s_{\{i,j\}}, ssid) \bmod q$.
   - Respond by sending $(ssid, s_i)$ to $\mathcal{LS}$. Remove all information associated to $ssid$.
5. $\underline{\mathcal{LS}}$: The $\mathcal{LS}$ verifies aggregated server responses through the zero-knowledge proof and computes final password hash.
   - After having received $(ssid, s_i)$ from all servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$, retrieve $(uid, pwd, N, u, c, (v_i, R_{1,i}, R_{2,i})_{i=1}^n)$ stored for $ssid$. Abort if it doesn't exist.
   - Compute $v_0 \leftarrow u^{K_0} \cdot \prod_{j=1}^n \mathrm{B}_0(s_{\{0,j\}}, ssid)$. Choose $r_0 \leftarrow_R \mathbb{Z}_q$, compute $R_{1,0} \leftarrow g^{r_0} \cdot \prod_{j=1}^n \mathrm{B}_1(s_{\{0,j\}}, ssid)$ and $R_{2,0} \leftarrow u^{r_0} \cdot \prod_{j=1}^n \mathrm{B}_2(s_{\{0,j\}}, ssid)$. Also compute $s_0 \leftarrow K_0 c + r_0 + \sum_{j=1}^n \Delta_{i,j} \mathrm{B}_3(s_{\{0,j\}}, ssid) \bmod q$.
   - Compute $v \leftarrow \prod_{i=0}^n v_i^{1/N}$, $R_1 \leftarrow \prod_{i=0}^n R_{1,i}$, $R_2 \leftarrow \prod_{i=0}^n R_{2,i}$, and $s \leftarrow \sum_{i=0}^n s_i \bmod q$.
   - Verify that $g^s = L^c R_1$ and $u^s = v^{Nc} R_2$; if not, set $BUSY[uid]$ to undefined in the state information and abort.
   - Store $PH[uid] \leftarrow \mathrm{G}(uid, pwd, v)$ as the password hash for $uid$ and output (CREATEOK, $sid$, $ssid$).
   - Remove all information associated to $ssid$.

**Login request.** The login protocol is a simplified version of account creation, without zero-knowledge proof.

1. $\underline{\mathcal{LS}}$: The $\mathcal{LS}$ sends a blinded password hash to all servers.
   - Upon input (LOGIN, $sid$, $ssid$, $uid$, $pwd'$), first check that $PH[uid]$ is defined and $USED_0[ssid]$ is not defined. Abort otherwise.
   - Set and store $USED_0[ssid] \leftarrow 1$.
   - Generate a random nonce $N \leftarrow_R \mathbb{Z}_q$ and compute $u \leftarrow \mathrm{H}(uid, pwd')^N$. Send $(ssid, u)$ to all servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$.

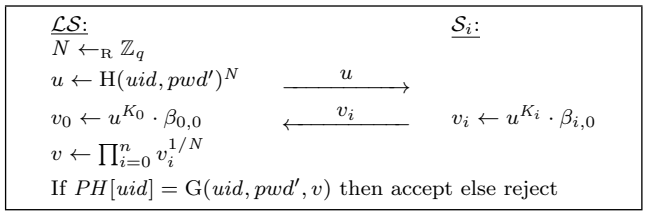


**Figure 5: The login protocol. All communication between $\mathcal{LS}$ and $\mathcal{S}_i$ is integrity-protected with a MAC key $\mu_{\{0,i\}}$. See the text for more information on the blinding factors $\beta_{i,k}$.**

   - Store $(uid, pwd', N, u)$ associated with $ssid$.
2. $\underline{\mathcal{S}_i}$: Each server sends a blinded response using its secret key share.
   - On input (PROCEED, $sid$, $ssid$) from the environment and after receiving $(ssid, u)$ from $\mathcal{LS}$, first check whether $USED_i[ssid] = 1$. If so, abort.
   - Compute $v_i \leftarrow u^{K_i} \cdot \prod_{j=0,j\neq i}^n \mathrm{B}_0(s_{\{i,j\}}, ssid)^{\Delta_{i,j}}$ and set and store $USED_i[ssid] \leftarrow 1$.
   - Respond by sending $(ssid, v_i)$ to $\mathcal{LS}$.
3. $\underline{\mathcal{LS}}$: The $\mathcal{LS}$ verifies the re-computed final password hash against the stored password.
   - After having received $(ssid, v_i)$ from all servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$, retrieve $(uid, pwd', N, u)$ associated to $ssid$. Abort if it doesn't exist.
   - Compute $v_0 \leftarrow u^{K_0} \cdot \prod_{j=1}^n \mathrm{B}_0(s_{\{i,j\}}, ssid)^{\Delta_{i,j}})$ and $v \leftarrow \prod_{i=0}^n v_i^{1/N}$.
   - If $PH[uid] = \mathrm{G}(uid, pwd', v)$, then set $pwdok \leftarrow 1$, else $pwdok \leftarrow 0$.
   - Output (RESULT, $sid$, $ssid$, $pwdok$) and delete the stored tuple $(uid, pwd', N, u)$ for $ssid$.

**Timeout.** The $\mathcal{LS}$ interrupts a creation or login protocol.

$\underline{\mathcal{LS}}$:

- Upon input (TIMEOUT, $sid$, $ssid$), retrieve record ($uid$, $pwd$, ...) for $ssid$.
- If $ssid$ is an unfinished account creation, set $BUSY[uid]$ to undefined and delete all information stored for $ssid$.
- If $ssid$ is an unfinished login, then delete all information stored for $ssid$.

**Refresh.** Refresh is a non-interactive process where each server has access to its backup memory. We assume that all servers synchronize to refresh simultaneously, e.g., by performing refreshes at regular time intervals, or by agreeing on the timing through out-of-band communication.

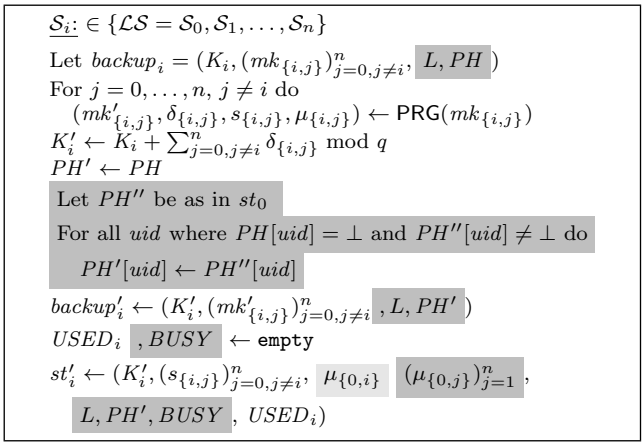


**Figure 6: The refresh protocol. Items in dark gray apply to $\mathcal{S}_i = \mathcal{S}_0 = \mathcal{LS}$ only, items in light gray apply to $\mathcal{S}_i \neq \mathcal{S}_0$ only.**

1. $\mathcal{S}_i \in \{\mathcal{LS} = \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n\}$: Based on the backup $backup_i$ and the current state $st_i$, $\mathcal{S}_i$ computes its new state $st'_i$.
   - If $\mathcal{S}_i = \mathcal{LS}$, on input (REFRESH, $sid$) recover the backup tape $backup_0 = (K_0, (mk_{\{0,j\}})_{j=1}^n, L, PH)$ and obtain the password hashes $PH''$ from $st_0$.
   - If $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$, recover the backup $backup_i = (K_i, (mk'_{\{i,j\}})_{j=0,j\neq i}^n)$.
   - For all $j = 0, \ldots, n$, $j \neq i$ compute $(mk'_{\{i,j\}}, \delta_{\{i,j\}}, s_{\{i,j\}}, \mu_{\{i,j\}}) \leftarrow \mathsf{PRG}(mk_{\{i,j\}})$ and compute the new key share $K'_i \leftarrow K_i + \sum_{j=0,j\neq i}^n \delta_{\{i,j\}} \bmod q$.
   - If $\mathcal{S}_i = \mathcal{LS}$, first set $PH' \leftarrow PH$. For all $uid$ that were newly created during the past epoch, set $PH'[uid] \leftarrow PH''[uid]$. Store $backup'_0 \leftarrow (K'_0, (mk'_{\{0,j\}})_{j=1}^n, L, PH')$ and set the new state $st'_0 \leftarrow (K'_0, (s_{\{0,j\}})_{j=1}^n, (\mu_{\{0,j\}})_{j=1}^n, L, PH', BUSY, USED_0)$.
   - If $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$, store the new backup $backup'_i \leftarrow (K'_i, (mk'_{\{i,j\}})_{j=0,j\neq i}^n)$ and set the new state to $st'_i \leftarrow (K'_i, (s_{\{i,j\}})_{j=0,j\neq i}^n, \mu_{\{0,i\}}, USED_i)$.

## 4.2 Security

The security properties of our first construction are summarized in the following theorem.

**Theorem 4.1.** *If the gap one-more Diffie-Hellman assumption holds in $\mathbb{G}$, PRG is a secure pseudo-random generator, and MAC is an unforgeable MAC, then the protocol $\pi$ of Section 4 securely implements the functionality $\mathcal{F}$ in the $(\mathcal{F}_{\text{smt}}, \mathcal{F}_{\text{ro}})$-hybrid model. For any polynomial-time algorithms $\mathcal{E}, \mathcal{A}$, there exist polynomial-time algorithms $\mathcal{SIM}$ and $\mathcal{B}, \mathcal{B}_1, \mathcal{B}_2$ such that*

$$\left|\mathbf{Real}^{\pi}_{\mathcal{E},\mathcal{A}}(\kappa) - \mathbf{Ideal}^{\mathcal{F}}_{\mathcal{E},\mathcal{SIM}}\right| \leq n_e n^2 \cdot \mathbf{Adv}^{\text{pr}}_{\mathcal{B}_1,\mathsf{PRG}}(\kappa)$$
$$+ \mathbf{Adv}^{\text{gomcdh}}_{\mathcal{B},\mathbb{G}}(\kappa) + n_e n \cdot \mathbf{Adv}^{\text{ufcma}}_{\mathcal{B}_2,\mathsf{MAC}}(\kappa)$$
$$+ \frac{7(q_{\text{ro}} + nq_c + q_l)^2}{2^{2\kappa}} + \frac{2n^2 n_e (q_{\text{ro}} + n^2 n_e)}{2^{\kappa}} .,$$

*where $n$, $n_e$, $q_{\text{ro}}$, $q_c$, $q_l$ are the number of back-end servers, epochs, random-oracle queries, account creation sessions, and login sessions, respectively.*

As mentioned earlier, the Cheon attack [8] on the (gap) one-more Diffie-Hellman assumption potentially reduces security with a factor $O(\sqrt{d})$ if the adversary is given $g^{x^d}$. For our construction, we have that $d \leq q_c + q_l$, so it would be advisable to use a group order $q$ that is $\log(q_c + q_l)$ bits longer than usual to compensate for the attack.

Due to space limitations, we only sketch the simulator $\mathcal{SIM}$ for the above theorem and the reduction from the gap one-more Diffie-Hellman problem.

### 4.2.1 Simulator

The simulator interacts as adversary with the functionality $\mathcal{F}$ and internally runs simulated versions "$\mathcal{LS}$", "$\mathcal{S}_1$", ..., "$\mathcal{S}_n$" of all honest servers against the real-world adversary $\mathcal{A}$, who also plays the role of all corrupt servers.

#### *Initialization.*

The initialization procedure takes place in a trusted environment and hence is completely under control of the simulator $\mathcal{SIM}$. It generates the initial keys so that it knows $K = \sum_{i=0}^n K_i \bmod q$ and sets $L \leftarrow g^K$. Rather than generating blinding seeds $s_{\{i,j\}}$ and MAC keys $\mu_i$ through the pseudo-random generator PRG, $\mathcal{SIM}$ chooses them truly at random. Values are assigned consistently across machines, though, in the sense that if different machines $\mathcal{S}_i, \mathcal{S}_j$ use the same master key $mk_{\{i,j\}}$ to derive a value, then the same random value will be assigned in both cases. The simulation is aborted whenever an honest login server "$\mathcal{LS}$" receives a network message for which the MAC tag verifies correctly under $\mu_i$ but that was never sent by "$\mathcal{S}_i$" and vice versa.

#### *Random Oracles.*

$\mathcal{SIM}$ simulates random oracles $\mathrm{B}_0, \ldots, \mathrm{B}_3, \mathrm{C}$ by returning random values from the appropriate ranges, storing the values in tables for consistency. It responds to random-oracle queries $\mathrm{H}(uid, pwd)$ so that it knows the discrete logarithm of all responses, i.e., choosing $\mathrm{HTL}[uid, pwd] \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and returning $\mathrm{HT}[uid, pwd] \leftarrow g^{\mathrm{HTL}[uid,pwd]}$. Random-oracle queries $\mathrm{G}(uid, pwd, v)$ are answered with the help of the PWDGUESS interface; we provide details in a moment. The simulator aborts when a collision is detected between outputs of C, H, or G.

#### *Account creation.*

The simulator executes simulated versions of all honest back-end servers "$\mathcal{S}_1$", ..., "$\mathcal{S}_n$" by following the real protocol after receiving (PROCEED, $sid$, $ssid$, $\mathcal{S}_i$) from $\mathcal{F}$. It can do so because it knows all of the relevant secrets $K_1, \ldots, K_n$ and $s_{\{i,j\}}$. If $\mathcal{LS}$ is corrupt and $\mathcal{A}$ delivers $(ssid, u)$ to an honest server "$\mathcal{S}_i$" for a new $ssid$, then $\mathcal{SIM}$ sends (CREATE, $sid, ssid, uid = \perp, pwd = \perp$) to $\mathcal{F}$ on behalf of $\mathcal{LS}$ and (PROCEED, $sid$, $ssid$) on behalf of all corrupt servers $\mathcal{S}_i \in \mathbf{C}$.

To simulate the honest "$\mathcal{LS}$", however, it must perform an honest-looking protocol without knowing the actual password. When $\mathcal{SIM}$ receives (CREATE, $sid$, $ssid$, $uid$) from $\mathcal{F}$, $\mathcal{SIM}$ uses $u \leftarrow g^N$ in the first round. If at the end of the protocol $g^s L^{-c} = R_1$ and $u^s \prod_{i=0}^n v_i^{-c} = R_2$ but $\prod_{i=0}^n v_i \neq u^K$, then $\mathcal{SIM}$ aborts. If $g^s L^{-c} = R_1$, $u^s \prod_{i=0}^n v_i^{-c} = R_2$, and $\prod_{i=0}^n v_i = u^K$, then it it assigns a random value as password hash $PH[uid] \leftarrow_{\mathrm{R}} \{0,1\}^{2\kappa}$ and sends (CREATEOK, $ssid$, $sid$, $uid$) to $\mathcal{F}$.

To make sure that the password hash looks correct when $\mathcal{LS}$ gets corrupted, it answers $\mathcal{A}$'s queries $\mathrm{G}(uid, pwd, v)$ as follows. If $v \neq \mathrm{H}(uid, pwd)^K$, then it simply returns a ran-

dom value. If $v = \mathrm{H}(uid, pwd)^K$, then $\mathcal{SIM}$ decreases a counter *guesses* that mirrors the counter maintained by $\mathcal{F}$, i.e., it is initially zero, is increased each time the last honest server in a subsession *ssid* receives $(\texttt{PROCEED}, sid, ssid, \mathcal{S}_i)$, and is set to infinity when all servers get corrupted in the same epoch. If $guesses < 0$, then $\mathcal{SIM}$ aborts the simulation and gives up; we will later show how this event gives rise to solving the gap one-more Diffie-Hellman problem. If $guesses \geq 0$, it sends $(\texttt{PWDGUESS}, sid, uid, pwd)$ to $\mathcal{F}$ to obtain a response $(\texttt{PWDGUESS}, sid, uid, pwdok)$. If $pwdok = 1$, then it returns $PH[uid]$ as hash output, else it returns a random value.

### *Login.*

Login protocols for a corrupt $\mathcal{LS}$ are simulated similarly as account creations above: $\mathcal{SIM}$ sends $(\texttt{LOGIN}, sid, ssid, uid = \bot, pwd = \bot)$ to $\mathcal{F}$ whenever the first honest server "$\mathcal{S}_i$" receives a message for a protocol *ssid*, and otherwise runs the honest code of $\mathcal{S}_i$.

Login protocols with an honest $\mathcal{LS}$ are simulated differently depending on whether the account for *uid* was created when $\mathcal{LS}$ was honest or corrupt. In the first case, the value $PH[uid]$ may not be assigned to any output $\mathrm{G}(uid, pwd, v)$ yet, but we are sure that at the time of account creation, the corrupt servers (if any) behaved "honestly overall", in the sense that they did not affect the computation of the overall exponent $K$ in $\prod_{i=0}^{n} v_i = u^K$, because the zero-knowledge proof was verified by the honest "$\mathcal{LS}$". Since there is no such proof during login, real-world corrupted servers can use a different overall exponent $K'$, causing $\mathcal{LS}$ to conclude that the password was false even though it was correct. The simulator forces the same outcome in the ideal world by setting the *fail* flag in the `RESULT` interface. Namely, it lets "$\mathcal{LS}$" use $u \leftarrow g^N$ and, after having received all values $v_i$, checks whether $\prod_{i=0}^{n} v_i = u^K$. If so, it sets $fail \leftarrow 0$, otherwise it sets $fail \leftarrow 1$.

In the second case, the password hash $PH[uid]$ was stored by a corrupt $\mathcal{LS}$. If there is no registered output $\mathrm{G}(uid, pwd, v) = PH[uid]$, then for a successful login to take place, $\mathcal{A}$ must "predict" an output of G, which can happen only with negligible probability. In this case, "$\mathcal{LS}$" runs the protocol using $u \leftarrow g^N$ but always sets $fail \leftarrow 1$ at the end. If there is one (and only one, as $\mathcal{SIM}$ aborts on collisions) output $\mathrm{G}(uid, pwd, v) = PH[uid]$, then we still cannot be sure that $v = \mathrm{H}(uid, pwd)^K$. The corrupt $\mathcal{LS}$ could for example have stored $PH[uid] = \mathrm{G}(uid, pwd, v) = \mathrm{H}(uid, pwd)^{K'}$ for $K' \neq K$, and during login, corrupt servers $\mathcal{S}_i$ could bias the overall exponent to $K'$ again, causing the honest $\mathcal{LS}$ to recompute $v' = v$ and conclude that login succeeded. For any other overall exponent, however, login must fail, even if the correct password was used. The simulator therefore lets "$\mathcal{LS}$" perform the honest protocol with the correct password *pwd*, which it knows from the entry in GT, and checks whether the recomputed value is equal to $v$. If not, it sets $fail \leftarrow 1$, otherwise it sets $fail \leftarrow 0$.

### *Corruption.*

When $\mathcal{A}$ transiently corrupts a back-end server $\mathcal{S}_i$, $\mathcal{SIM}$ can hand over the full state of $\mathcal{S}_i$ as it knows all the secret keys and subsession states. When it corrupts $\mathcal{LS}$, $\mathcal{SIM}$ knows the long-term state $st_0 = (K'_0, (s_{\{0,j\}})_{j=1}^n, (\mu_{\{0,j\}})_{j=1}^n, L, PH', BUSY, USED_0)$, but does not necessarily know the state of ongoing subsessions that contain the password *pwd* and the nonce $N$ such that $u = \mathrm{H}(uid, pwd)^N$. It obtains the actual passwords for all ongoing protocols $(\texttt{CORRUPT}, sid, \mathbf{L})$ from $\mathcal{F}$. It can then compute simulated nonces $N'$ for the correct password using the discrete logarithms of $\mathrm{H}(uid, pwd)$ stored in HTL.

When $\mathcal{A}$ permanently corrupts a server $\mathcal{S}_i \in \{\mathcal{LS} = \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n\}$, it additionally chooses master keys $mk_{\{i,j\}} \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$ for $j = 0, \ldots, n$, $j \neq i$, to simulate the contents of the backup tape $backup_i$.

### *Refresh.*

When the environment instructs all (non-permanently-corrupted) servers to refresh, the simulator $\mathcal{SIM}$ computes $(mk_{\{i,j\}}, \delta_{\{i,j\}}, s_{\{i,j\}}, \mu_{\{i,j\}}) \leftarrow \mathsf{PRG}(mk_{\{i,j\}})$ for all servers $i = 0, \ldots, n$ and all permanently corrupted servers $\mathcal{S}_j \in \mathbf{PC}$, where $mk_{\{i,j\}}$ are the values given to $\mathcal{A}$ as part of the backup tape when $\mathcal{S}_j$ was permanently corrupted. For all other servers $\mathcal{S}_j \notin \mathbf{PC}$, $\mathcal{SIM}$ chooses random values for $\delta_{\{i,j\}}, s_{\{i,j\}}, \mu_{\{i,j\}}$. It otherwise computes the new state of $\mathcal{S}_i$ as in the real protocol.

For all new entries *uid* that were added to the final state $PH'$ of a corrupt $\mathcal{LS}$ but were not yet defined at the beginning of the epoch, $\mathcal{SIM}$ checks whether there exists an output $\mathrm{G}(uid, pwd, v) = PH'[uid]$, setting $pwd \leftarrow \bot$ if not. The simulator registers a new account for each such *uid* by sending $(\texttt{CREATE}, sid, ssid, uid, pwd)$ and $(\texttt{CREATEOK}, sid, ssid)$ to $\mathcal{F}$ for a fresh *ssid*.

## 4.2.2 Reduction from Gap One-More DH

Suppose we are given an adversary $\mathcal{A}$ and an environment $\mathcal{E}$ that cause the event $guesses < 0$ to occur, where *guesses* is initially zero, is decreased at each random-oracle query $\mathrm{G}(uid, pwd, v)$ with $v = \mathrm{H}(uid, pwd)^K$, is increased for each protocol session *ssid* where all honest servers participate, and is set to infinity when all servers are corrupted in the same epoch. We show how such $\mathcal{E}, \mathcal{A}$ give rise to a solver $\mathcal{B}$ for the gap one-more Diffie-Hellman problem.

Algorithm $\mathcal{B}$ is given input $(g, X)$ and has access to oracles $\mathsf{T}$, $\mathsf{CDH}$, and $\mathsf{DDH}$. It sets $L \leftarrow X$, thereby implicitly setting $K = \sum_{i=0}^{n} K_i = x$, and answers random-oracle queries $\mathrm{H}(uid, pwd)$ with target points generated by its $\mathsf{T}$ oracle. It only fixes values of the individual $K_i$ and blinding seeds $s_{\{i,j\}}$ at the moment that $\mathcal{S}_i$ gets corrupted, however, avoiding that $\mathcal{B}$ has to guess a server that will remain uncorrupted in the next epoch. Note that $\mathcal{B}$ never needs to simulate values for $K_i$ for *all* servers within the same epoch, because then the event $guesses < 0$ cannot occur.

### *Account creation.*

When $\mathcal{E}$ instructs an honest $\mathcal{LS}$ to create an account *uid* with password *pwd*, $\mathcal{B}$ first $\mathcal{LS}$ honestly perform step 1 of the real protocol, but it lets all honest servers $\mathcal{S}_i$ choose random values for $v_i, R_{i,1}, R_{i,2}, s_i$. These are correctly distributed because, if at least one of $\mathcal{S}_1, \ldots, \mathcal{S}_n$ is honest, then at least one of the blinding factors $\mathrm{B}_k(s_{\{i,j\}}, ssid)$ remains unknown to $\mathcal{A}$, and if all $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are corrupt, then $v_0, R_{0,1}, R_{0,2}$ remains internal to the honest $\mathcal{LS}$ anyway. Only when $\mathcal{S}_i$ later gets corrupted will we program the random oracles $\mathrm{B}_k$ so that these responses make sense. The $\mathcal{LS}$ cannot verify the zero-knowledge proof as usual, but, because it previously assigned values to the secrets $K_i$ and $s_{\{i,j\}}$ of corrupt servers $\mathcal{S}_i \in \mathbf{C} = \mathbf{PC} \cup \mathbf{TC}$, it can check whether they behaved "honestly overall", meaning, in a way that would have made

the zero-knowledge proof work out if the honest $\mathcal{S}_i$ would have responded correctly. If so, then $\mathcal{LS}$ accepts the creation but stores a random string in $PH[uid]$.

When $\mathcal{A}$ later makes a query $\mathrm{G}(uid, pwd', v')$ with $v' = \mathrm{H}(uid, pwd')^x$, which $\mathcal{B}$ can test using its $\mathsf{DDH}$ oracle, then $\mathcal{B}$ decreases *guesses* and adds $(\mathrm{H}(uid, pwd'), v')$ to a set *Sol* of CDH solutions. If $pwd'$ is the password $pwd$ used at creation for $uid$, then $\mathcal{B}$ responds with $PH[uid]$, otherwise it returns a random string.

If a corrupt $\mathcal{LS}$ initiates an account creation, then the honest servers $\mathcal{S}_i \in \overline{\mathbf{C}} = \{\mathcal{LS}, \mathcal{S}_1, \ldots, \mathcal{S}_n\} \setminus \mathbf{C}$ must behave "honestly overall" to ensure that $\mathcal{LS}$ computes the correct value $v = \mathrm{H}(uid, pwd)^K$ and a correct zero-knowledge proof if it chooses to follow the protocol honestly. They do so by returning random values $v_i$, except for the last honest server to respond $\mathcal{S}_l$, where $\mathcal{B}$ increases *guesses* and uses one query to its $\mathsf{CDH}$ oracle to compute a response $v_l$ so that

$$\begin{aligned}\prod_{\mathcal{S}_i \in \overline{\mathbf{C}}} v_i &= \prod_{\mathcal{S}_i \in \overline{\mathbf{C}}} u_i^{\kappa_i} \prod_{\mathcal{S}_j=1, j\neq i}^{n} \mathrm{B}(s_{\{i,j\}}, ssid)^{\Delta_{i,j}} \\ &= \prod_{\mathcal{S}_i \in \overline{\mathbf{C}}} u_i^{\kappa_i} \prod_{\mathcal{S}_j \in \mathbf{C}} \mathrm{B}(s_{\{i,j\}}, ssid)^{\Delta_{i,j}}\end{aligned}$$

for some random exponents $\kappa_i \in \mathbb{Z}_q$ so that $\sum_{\mathcal{S}_i \in \overline{\mathbf{C}}} \kappa_i + \sum_{\mathcal{S}_i \in \mathbf{C}} K_i = x \bmod q$, where $u_i$ is the value for $u$ that $\mathcal{S}_i$ received in subsession *ssid*. It simulates the zero-knowledge proof for honest $\mathcal{S}_i$ in a similar way, choosing random values for $R_{1,i}, R_{2,i}, s_i$ except for the last server, where $\mathcal{B}$ uses a simulated zero-knowledge proof using the challenge $c_i$ that it can look up from a response $\mathrm{C}(c_i) = ch_i$.

### *Login.*

When $\mathcal{E}$ instructs the honest $\mathcal{LS}$ to perform a login with password $pwd'$ for account $uid$ that was created by an honest $\mathcal{LS}$ with password $pwd$, $\mathcal{B}$ lets $\mathcal{LS}$ run the honest protocol with $uid, pwd'$, but lets honest $\mathcal{S}_i$ return random values $v_i$. At the end, $\mathcal{LS}$ checks whether the corrupt servers behaved honestly overall as defined earlier. If so, and $pwd' = pwd$, then $\mathcal{LS}$ outputs $pwdok = 1$, else it outputs $pwdok = 0$. The $\mathcal{LS}$ proceeds similarly for accounts $uid$ created by a corrupt $\mathcal{LS}$ if there exists no output $\mathrm{G}(uid, pwd, v) = PH[uid]$, or if such output exists but $pwd \neq pwd'$. At the end of the protocol, however, it always outputs $pwdok = 0$.

For an account created by a corrupt $\mathcal{LS}$ with an existing entry $\mathrm{G}(uid, pwd, v) = PH[uid]$ with $pwd = pwd'$, things are slightly more complicated because, as explained for the simulator above, we cannot be sure that $v = \mathrm{H}(uid, pwd)^K$, yet login may still succeed if corrupt servers $\mathcal{S}_i \in \mathbf{C}$ apply the same bias to the overall exponent during login as during account creation. The $\mathcal{LS}$ detects whether a real protocol would have reconstructed $v' = v$ by checking whether

$$\begin{aligned}v &= \prod_{i=0}^{n} v_i^{1/N} = \left(\prod_{\mathcal{S}_i \in \mathbf{C}} v_i \cdot \prod_{\mathcal{S}_i \in \overline{\mathbf{C}}} v_i\right)^{\frac{1}{N}} \\ &= \left(\prod_{\mathcal{S}_i \in \mathbf{C}} v_i \cdot u^{x - \sum_{\mathcal{S}_i \in \mathbf{C}} K_i} \prod_{\mathcal{S}_i \in \overline{\mathbf{C}}} \prod_{\mathcal{S}_j \in \mathbf{C}} \mathrm{B}(s_{\{i,j\}}, ssid)^{\Delta_{i,j}}\right)^{\frac{1}{N}}\end{aligned}$$

which $\mathcal{B}$ can test using its $\mathsf{DDH}$ oracle. If so, then $\mathcal{LS}$ outputs $pwdok = 1$, otherwise it outputs $pwdok = 0$.

Login protocols with a corrupt $\mathcal{LS}$ are simulated similarly as account creation, but without the zero-knowledge proof. Note that here too, $\mathcal{B}$ will make one $\mathsf{CDH}$ oracle query to compute the last honest server's response for each *ssid*.

### *Corruption and refresh.*

If $\mathcal{A}$ corrupts all servers during the same epoch, *guesses* gets set to infinity, so $\mathcal{B}$ can abort without affecting its success probability. When $\mathcal{A}$ transiently corrupts $\mathcal{S}_i$, $\mathcal{B}$ chooses a random key $K_i$ and random blinding seeds $(s_{\{i,j\}})_{j=0,j\neq i}^{n}$, and programs the entries $\mathrm{B}_k(s_{\{i,j\}}, ssid)$ of all previous subsessions *ssid* so that the previous responses make sense, i.e., so that $v_i = u_i^{K_i} \cdot \prod_{j=0,j\neq i}^{n} \mathrm{B}_0(s_{\{i,j\}}, ssid)$. As $\mathcal{A}$ cannot corrupt all servers, there is at least one seed $s_{\{i,j\}}$ that is unknown to $\mathcal{A}$, so that $\mathcal{B}$ can program the entries $\mathrm{B}_0(s_{\{i,j\}}, ssid)$ to satisfy the above equation. It proceeds similarly for the values $R_{1,i}, R_{2,i}, s_i$ in account creation protocols. For ongoing account creation protocols, $\mathcal{B}$ additionally chooses $r_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and programs $\mathrm{B}_1, \mathrm{B}_2$ so that $g^{s'_i} = g^{c_i K_i} \cdot R_{1,i} \prod_{j=0,j\neq i}^{n} \mathrm{B}_1(s_{\{i,j\}}, ssid)^{-\Delta_{i,j}}$ and $u_i^{s'_i} = v_i^{c_i} \cdot R_{2,i} \prod_{j=0,j\neq i}^{n} \mathrm{B}_2(s_{\{i,j\}}, ssid)^{-\Delta_{i,j}}$, where $s'_i = s_i - \sum_{j=0,j\neq i}^{n} \Delta_{i,j} \mathrm{B}_3(s_{\{i,j\}}, ssid) \bmod q$, so that it can hand $r_i$ to $\mathcal{A}$ as part of the state of $\mathcal{S}_i$.

When $\mathcal{A}$ permanently corrupts $\mathcal{S}_i$, then $\mathcal{B}$ additionally chooses random master keys $mk_{\{i,j\}}$ for all $j = 0, \ldots, n$, $j \neq i$, to simulate the backup tape of $\mathcal{S}_i$. When a non-permanently-corrupted server $\mathcal{S}_i$ is refreshed, $\mathcal{B}$ takes back control of $\mathcal{S}_i$ and forgets all previously chosen values for $K_i$ and $s_{\{i,j\}}$.

### *CDH solutions.*

When the event *guesses* $< 0$ occurs, $\mathcal{B}$ just added one more CDH solution to *Sol* than the number of times that it invoked its $\mathsf{CDH}$ oracle. Indeed, $\mathcal{B}$ only invokes the $\mathsf{CDH}$ oracle only once for each account creation or login protocol with a corrupt $\mathcal{LS}$ where all honest servers participate. The counter *guesses* is increased immediately before $\mathcal{B}$ invokes its $\mathsf{CDH}$ oracle and is only decreased when a valid CDH solution is detected in a $\mathrm{G}(\cdot)$ query. Therefore, $\mathcal{B}$ wins its game by returning *Sol*.

## 5. CONSTRUCTION WITH PAIRINGS

We now present an even more efficient scheme based on pairings. It is almost identical to the discrete-logarithm scheme, except that the interactive zero-knowledge proof is replaced by a pairing computation by $\mathcal{LS}$.

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{t}$ be multiplicative groups of prime order $q$ with generators $g_1, g_2, g_\mathrm{t}$, respectively, and an efficiently computable pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_\mathrm{t}$. Let $\mathrm{H} : \{0,1\}^* \times \{0,1\}^* \to \mathbb{G}_1$, $\mathrm{G} : \{0,1\}^* \times \{0,1\}^* \times \mathbb{G}_1 \to \{0,1\}^{2\kappa}$, and $\mathrm{B}_0 : \{0,1\}^\kappa \times \mathbb{N} \to \mathbb{G}_1$ be hash functions modeled as random oracles.

Initialization, login, timeout, and refresh are identical to the discrete-logarithm scheme, except that $L \leftarrow g_2^K$ and that group operations during login take place in $\mathbb{G}_1$. Account creation is considerably simpler, as the two-round zero-knowledge protocol is now replaced with a pairing computation, as depicted in Figure 7.

**Account creation.** To create an account for user $uid$ with password $pwd$, the $\mathcal{LS}$ runs the following protocol with all $n$ servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$:

1. $\underline{\mathcal{LS}}$: The $\mathcal{LS}$ sends a blinded password hash to all servers.

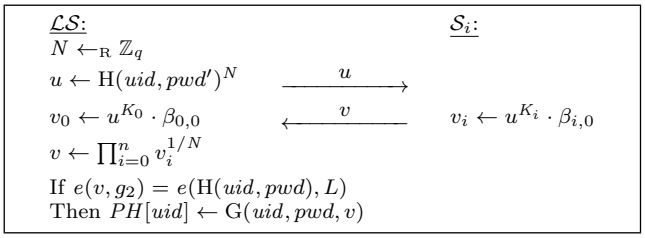


**Figure 7: The account creation protocol for the pairing-based scheme.**

- On input (CREATE, $sid$, $ssid$, $uid$, $pwd$), check if $PH[uid]$, $BUSY[uid]$ or $USED_0[ssid]$ is already defined. If so, abort.
- Set $BUSY[uid] \leftarrow 1$ and $USED_0[ssid] \leftarrow 1$.
- Compute $u \leftarrow \mathrm{H}(uid, pwd)^N$ and send $(ssid, epoch_0, u)$ to all servers $\mathcal{S}_i$ for $i = 1, \ldots, n$.
- Store $(uid, pwd, N, u)$ associated with $ssid$.

2. $\underline{\mathcal{S}_i}$: Each server sends a blinded response using its secret key share.
   - On input (PROCEED, $sid$, $ssid$) from the environment, and after having received $(ssid, epoch_0, u)$ from $\mathcal{LS}$, check whether $USED_i[ssid] = 1$ or $epoch_0 \neq epoch_i$. If so, abort.
   - Compute $v_i \leftarrow u^{K_i} \cdot \prod_{j=0, j\neq i}^{n} \mathrm{B}(s_{\{i,j\}}, ssid)^{\Delta_{i,j}}$ and set $USED[ssid] \leftarrow 1$.
   - Respond by sending $(ssid, v_i)$ to $\mathcal{LS}$.
3. $\underline{\mathcal{LS}}$: The $\mathcal{LS}$ verifies the server contributions and computes final password hash.
   - After having received $(ssid, v_i)$ from $\mathcal{S}_i$ for all $i = 1, \ldots, n$, retrieve $(uid, pwd, N, u)$ stored for $ssid$. Abort if it doesn't exist.
   - Compute $v_0 \leftarrow u^{K_0} \cdot \prod_{j=1}^{n} \mathrm{B}(s_{\{0,j\}}, ssid)^{\Delta_{0,j}}$ and $v \leftarrow \prod_{i=0}^{n} v_i^{1/N}$.
   - Verify that $e(v, g_2) = e(\mathrm{H}(uid, pwd), L)$; if not, set $BUSY[uid]$ to undefined and abort.
   - Store $PH[uid] \leftarrow \mathrm{G}(uid, pwd, v)$ as the password hash for $uid$ and output (CREATEOK, $sid$, $ssid$).
   - Remove all information associated to $ssid$.

**Theorem 5.1.** *If the one-more Diffie-Hellman assumption holds in $(\mathbb{G}_1, \mathbb{G}_2)$, then the protocol $\pi$ in Section 5 securely realizes the functionality $\mathcal{F}$ in the $(\mathcal{F}_{\mathrm{smt}}, \mathcal{F}_{\mathrm{ro}})$-hybrid model.*

# 6. DEPLOYMENT OF OUR SCHEME

As discussed, our scheme requires the initialization to be run in a trusted execution environment and, to warrant the difference between transient and permanent corruptions, requires the backup tape to be better protected from attacks than normal state information.

The initialization could be run on a single trusted machine who then distributes the keys to the other servers, e.g., by smart cards which then can also act as backup tapes. A better alternative seems to make use of cloud platforms which will make it also easier to recover from corruption by starting a fresh virtual machine. We discuss this in the following.

The features of modern cloud computing platforms such as Openstack [23] can be nicely exploited to realize proactive security for protocols. Such platforms offer strong separation between the virtual machines that are exposed to the Internet, and are thus subject to attacks, and the cloud management interfaces that run in a protected, de-militarized zone. New virtual machines can be created on the fly from images, machines can be shut down, and the routing of traffic to machines be dynamically configured. The platforms also virtualize the storage for the virtual machines, i.e., they offer different kinds of abstraction of hard-disks such as file system, block store, object store, etc. The management of all of this is typically a manual process via a web interface in the de-militarized zone, but can easily be automated with scripts, which is how it should be done for our protocol.

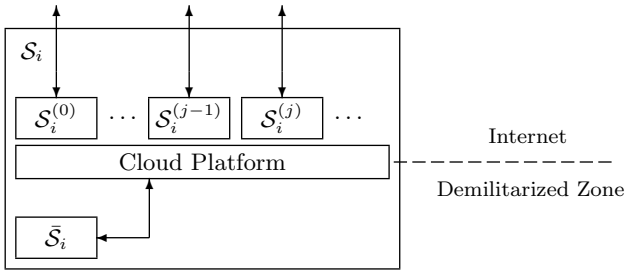


**Figure 8: The different components of server $\mathcal{S}_i$.**

The main idea to implement our scheme in this setting is that each server is realized with its own cloud platform (cf. Figure 8). Thus, each server $\mathcal{S}_i$ (and analogously $\mathcal{LS}$) consists of a cloud platform, a number of virtual machines $\mathcal{S}_i^{(0)}, \ldots, \mathcal{S}_i^{(j-1)}, \mathcal{S}_i^{(j)}, \ldots$ that are run on the cloud platform on a (physical) machine $\bar{\mathcal{S}}_i$. The cloud platform is usually a single physical machine or a cluster of them. The virtual machines are exposed to the internet while the cloud platform and $\bar{\mathcal{S}}_i$ are run in the de-militarized zone, i.e., in a protected environment.

For each epoch $j$, a fresh *virtual* machine $\mathcal{S}_i^{(j)}$ is started on the cloud platform. These virtual machines run the *account creation* and the *login* protocols and access their states from the virtual storage provided by the cloud platform.

The machine $\bar{\mathcal{S}}_i$ controls the cloud platform, maintains the images for the virtual machines $\mathcal{S}_i^{(j)}$, and prepares the state (storage) that is given to each $\mathcal{S}_i^{(j)}$ in order for them to run the account creation and the login request protocols. Indeed, $\bar{\mathcal{S}}_i$ needs to be connected only to the cloud software platform and in practice such connections are typically physically isolated. To prepare the state for the $\mathcal{S}_i^{(j)}$'s, the machine $\bar{\mathcal{S}}_i$ runs the *initialization* protocol, which requires $\mathcal{LS}$ to securely send a message to each of the $\mathcal{S}_i$'s. As this is a one-time event that will be part of setting up the overall system, this communication can for instance be realized by writing the messages to a physical medium such as a USB drive and distribute it by courier. In fact, the master keys could even be written on paper and entered manually, as each server in our protocol receives only $n \cdot \kappa$ bits, amounting to about $18n$ alphanumeric (7-bit) characters for practical scenarios with $\kappa = 128$. The master keys for $\mathcal{S}_i$ are stored in backup memory that is available to $\bar{\mathcal{S}}_i$ but not to any of the instances $\mathcal{S}_i^{(j)}$. During *refresh*, $\bar{\mathcal{S}}_i$ derives the initial state for $\mathcal{S}_i^{(j)}$ for the next epoch from the master keys and updates the master keys in the backup memory.

# 7. IMPLEMENTATION

To demonstrate the practical feasibility and test the performance of our protocols, we created a prototype imple-

**Table 1: Performance figures of our first protocol over the NIST P-256 elliptic curve.**

| | # dedicated cores | | | | throughput | delay (ms) | |
|---|---|---|---|---|---|---|---|
| $n$ | $\mathcal{LS}$ | $\mathcal{S}_1$ | $\mathcal{S}_2$ | $\mathcal{S}_3$ | (logins/s) | mean | 99% |
| 1 | 2 | 2 | | | 40 | 94 | 155 |
| 1 | 4 | 4 | | | 80 | 71 | 111 |
| 1 | 8 | 8 | | | 157 | 53 | 79 |
| 1 | 16 | 8 | | | 214 | 90 | 153 |
| 1 | 16 | 16 | | | 310 | 50 | 86 |
| 2 | 16 | 16 | 16 | | 293 | 59 | 94 |
| 3 | 16 | 16 | 16 | 16 | 285 | 53 | 85 |

mentation in Java. We implemented our first construction (without pairings) over the NIST P-256 elliptic curve using SHA-256 as a hash function. All elliptic-curve operations are performed using the Bouncy Castle cryptographic library. We expect that performance can be considerably improved by using other libraries or implementation languages.

We tested our implementation on a commercial cloud infrastructure for different numbers of dedicated 2.9 GHz computing cores for each server. Selected performance numbers for login protocols, the most relevant operation, are summarized in Table 1. Roughly, our prototype implementation handles about 20 logins per second and per server core dedicated to the $\mathcal{LS}$. The mean computation and communication delay incurred from the moment that $\mathcal{LS}$ receives the request until it reaches a decision is always below 100 milliseconds, with a 99 percentile well below 200 ms, small enough to not be noticeable to the user.

Since the $\mathcal{LS}$ performs two exponentiations in each login protocol, versus only one for each $\mathcal{S}_i$, each $\mathcal{S}_i$ takes slightly more than half of the computational resources of the $\mathcal{LS}$. It would therefore make sense to assign more computational power to the $\mathcal{LS}$ than to each $\mathcal{S}_i$. Because all servers $\mathcal{S}_i$ operate in parallel, increasing the number of servers $n$ has only a minor impact on the throughput and delays.

## Acknowledgements

This work was supported by the European Commission's Seventh Framework Programme under the PERCY grant (agreement #321310) and the FutureID project (agreement #318424). We are very grateful to Daniel Kovacs and Franz-Stefan Preiss for their work on the prototype implementation and performance testing, as well as for their valuable feedback. We would also like to thank Marc Bütikofer, Robin Künzler, Christoph Lucas, and Adrian Schneider for their feedback and implementing our protocol at Ergon.

# 8. REFERENCES

[1] J. F. Almansa, I. Damgård, J. B. Nielsen. Simplified threshold RSA with adaptive and proactive security. *EUROCRYPT 2006.*

[2] J. Brainard, A. Juels, B. S. Kaliski Jr., Michael Szydlo. A new two-server approach for authentication with short secrets. *USENIX Security Symposium 2003.*

[3] A. Bagherzandi, S. Jarecki, N. Saxena, Y. Lu. Password-protected secret sharing. *ACM CCS 2011.*

[4] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. *PKC 2003.*

[5] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. *FOCS 2001.* Full version on Cryptology ePrint Archive, Report 2000/067, 2000.

[6] J. Camenisch, R. R. Enderlein, G. Neven. Two-server password-authenticated secret sharing UC-secure against transient corruptions. *PKC 2015.*

[7] R. Canetti, U. Feige, O. Goldreich, M. Naor. Adaptively secure multi-party computation. *28th ACM STOC 1996.*

[8] J. H. Cheon. Security analysis of the strong Diffie-Hellman problem. *EUROCRYPT 2006.*

[9] J. Camenisch, A. Lehmann, A. Lysyanskaya, G. Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. *CRYPTO 2014, Part II.*

[10] J. Camenisch, A. Lysyanskaya, G. Neven. Practical yet universally composable two-server password-authenticated secret sharing. *ACM CCS 2012.*

[11] C.-K. Chu, W.-G. Tzeng. Efficient k-out-of-n oblivious transfer schemes with adaptive and non-adaptive queries. *PKC 2005.*

[12] M. Di Raimondo, R. Gennaro. Provably secure threshold password-authenticated key exchange. *EUROCRYPT 2003.*

[13] EMC Corporation. RSA distributed credential protection. `http://www.emc.com/security/rsa-distributed-credential-protection.htm`.

[14] Experian. 2015 Second annual data breach industry forecast, 2015.

[15] W. Ford, B. S. Kaliski Jr. Server-assisted generation of a strong secret from a password. *WETICE 2000.*

[16] Gemalto. 2014 Year of mega breaches & identity theft: Findings from the 2014 breach level index, 2015.

[17] D. P. Jablon. Password authentication using multiple servers. *CT-RSA 2001.*

[18] S. Jarecki, A. Kiayias, H. Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. *ASIACRYPT 2014, Part II.*

[19] S. Jarecki, X. Liu. Fast secure computation of set intersection. *SCN 2010.*

[20] B. Kaliski. PKCS #5: Password-based cryptography specification. IETF RFC 2898, 2000.

[21] J. Katz, P. D. MacKenzie, G. Taban, V. D. Gligor. Two-server password-only authenticated key exchange. *ACNS 05.*

[22] P. D. MacKenzie, T. Shrimpton, M. Jakobsson. Threshold password-authenticated key exchange. *CRYPTO 2002.*

[23] Openstack website. `www.openstack.org`.

[24] N. Provos, D. Mazières. A future-adaptable password scheme. *USENIX Annual Technical Conference, FREENIX Track*, 1999.

[25] M. Szydlo, B. S. Kaliski Jr. Proofs for two-server password authentication. *CT-RSA 2005.*