Memento: How to Reconstruct your Secrets from a Single Password in a Hostile Environment

Jan Camenisch[1], Anja Lehmann[1], Anna Lysyanskaya[2], and Gregory Neven[1]

[1] IBM Research – Zurich
[2] Brown University

Abstract. Passwords are inherently vulnerable to dictionary attacks, but are quite secure if guessing attempts can be slowed down, for example by an online server. If this server gets compromised, however, the attacker can again perform an off-line attack. The obvious remedy is to distribute the password verification process over multiple servers, so that the password remains secure as long as no more than a threshold of the servers are compromised. By letting these servers additionally host shares of a strong secret that the user can recover upon entering the correct password, the user can perform further cryptographic tasks using this strong secret as a key such as encrypting data in the cloud. Threshold password-authenticated secret sharing (TPASS) protocols provide exactly this functionality. Unfortunately, the two only known schemes by Bagherzandi et al. (CCS 2011) and Camenisch et al. (CCS 2012) leak the password if a user mistakenly executes the protocol with malicious servers. Authenticating to the wrong servers is a common scenario when users are tricked in phishing attacks. We propose the first t-out-of-n TPASS protocol for any $n > t$ that does not suffer from this shortcoming. We prove our protocol secure in the UC framework, which for the particular case of password-based protocols offers important advantages over property-based definitions, e.g., by correctly modeling typos in password attempts.

1 Introduction

You wake up in a motel room. Where are you? How did you get here? You can't remember anything. Or perhaps you can. One word, a password, is engraved in your mind. You go outside and walk into the street. The first person you meet doesn't know you. The second seems to recognize you, or at least pretends to do so. He says he's your friend. He introduces you to other people who claim they are also your friends. They say they can help you reconstruct your memory—if you give the correct password. But why would you trust them? What if they are not your friends? What if they're trying to plant false memories into your brain? What if they're trying to learn your password, so they can retrieve your real memories from your real friends? How can you tell?

The above scenario, inspired by the movie "Memento" in which the main character suffers from short-term memory loss, leads to an interesting cryptographic problem that is also very relevant in practice. Namely, can a user securely recover his secrets from a set of servers, if all the user can or wants to remember is a single password and all of the servers may be adversarial? In particular, can he protect his precious password when

accidentally trying to run the recovery with all-malicious servers? A solution for this problem can act as a natural bridge from human-memorizable passwords to strong keys for cryptographic tasks, all while avoiding offline dictionary attacks on the password. Practical applications include secure password managers (where the shared secret is a list of strongly random website passwords) and encrypting data in the cloud (where the shared secret is the encryption key) based on a single master password.

A single master password may seem a bad idea given that over the past few years, hundreds of millions of passwords have been stolen through server compromises, with major data breaches being reported at popular websites such as LinkedIn, Adobe, Yahoo!, and Twitter. Storing passwords in hashed form offers hardly any protection due to the efficiency of brute-force *offline attacks* using dictionaries. According to NIST [8], sixteen-character human-memorizable passwords have only 30 bits of entropy on average. With current graphical processors testing more than three hundred billion passwords per second [33], security must be considered lost as soon as an offline attack against the password data can be performed. Indeed, more than ninety percent of the 6.5 million password hashes pilfered from LinkedIn were cracked within six days [32]. Dedicated password hashes such as bcrypt [43] or PBKDF2 [37] only give a linear security improvement: a factor x more effort to verify passwords for an honest server makes offline dictionary attacks at a factor x harder.

However, as poorly as passwords stand their ground against offline attacks, they are actually fairly secure against *online* attacks, as long as attempts are slowed down or *throttled* by an honest server, e.g., by blocking accounts, presenting CAPTCHAs, or introducing time delays. The problem is that if a single server can check the correctness of a password, then that server—or any adversary breaking into it—must have access to some information that can be used for an offline attack. The obvious solution is to verify passwords through a distributed protocol involving multiple servers, in such a way that no single server, or no collusion up to a certain threshold, stores or obtains any information that can enable an offline attack.

Scenario. Recall our original goal that we don't just want to authenticate to a set of servers, we also want to store a (strong) secret that the user can later reconstruct from a subset of the servers using a single password, in such a way that the servers don't learn anything about the secret or the password. The secret can be used as a key for any other cryptographic purpose, for example, to encrypt and store a file in the cloud containing strong passwords and other credentials required for websites or online services. Those services thereby do not have to change their standard authentication mechanisms, ensuring a smooth deployment path. A commercial product along these lines called *RSA Distributed Credential Protection* [44] is already available.

When the user sets up his account, he carefully selects a list of names of servers that he will use in the protocol. He may make his selection based on the servers' reputation, perceived trust, or other criteria; the selection is important, because if too many of the selected servers are malicious, his password and secret are already compromised from the beginning. It is also clear that at setup the user must be able to authenticate the servers that he selected. In previous password-based schemes, setup is often assumed to take place out-of-band. Given the importance of the setup phase, we follow Camenisch

et al. [13] by explicitly modeling account setup and assuming that a public-key infrastructure (PKI) is in place to link server names to public keys.

When later the user wants to retrieve his secret, ideally, he should not need anything else than his username and password. In particular, he should not even have to remember the names of the servers he selected at setup. The list may be too long for the user to remember, and he can certainly not be expected to, at every retrieval, spend the same amount of thought on composing the list of names of the servers as during setup. Also, the user may retrieve his secret with a different device than the one that he used to create the account. For example, he may be logging in from his phone instead of his laptop, he may be installing a new device, or he may be borrowing a friend's tablet PC. Of course, we do have to assume that the device on which the user enters his single password is "clean", i.e., is not infected with malware, doesn't have a key-logger attached, etc. We make the minimal requirement that the user has a clean operating system and a clean web browser to work with, containing hardcoded keys of root certification authorities (CAs) and an implementation of our protocol. We explicitly do not want any user-specific state information from the setup phase to be needed on the device at the time of retrieval. Different users may select different server names, so the names of the selected servers cannot be hardcoded in the browser either. The list of servers that is used at retrieval may be different from the list used at setup: the user may forget some servers when authenticating, involve some servers that were not present at setup, mistype server URLs, or even be tricked into running the protocol with a set of all-malicious servers through a sort of phishing attack. Note that a PKI doesn't prevent this: malicious servers also have certified keys. Also note that users cannot rely on the servers to store user-specific state information that they later send back signed to the user, because the servers during retrieval may be malicious and lie about the content or wrongly pretend to have been part of the trusted setup set.

Existing Solutions. Threshold password-authenticated secret sharing (TPASS) schemes are the best fit for our problem: they allow a user to secret-share a secret K among n servers and protect it with a password p, so that the user can later recover K from any subset of $t + 1$ of the servers using p, but so that no coalition smaller than t learns anything about K or can mount an offline attack on p. Unfortunately, the two currently known TPASS protocols by Bagherzandi et al. [3] and Camenisch et al. [13] break down when the user tries to retrieve his secret from a set of all-malicious servers. In the former, the password is exposed to offline attacks, in the latter it is plainly leaked. We outline the attacks on both protocols in our full paper [12]. These attacks are of course quite devastating, as once the password is compromised, the malicious servers can recover the user's secret from the correct servers.

Our Contribution. We provide the first t-out-of-n TPASS protocol for any $n > t$ that does not require trusted, user-specific state information to be carried over from the setup phase. Our protocol requires the user to only remember a username and a password, assuming that a PKI is available; if he misremembers his list of servers and tries to retrieve his secret from corrupt servers, our protocol prevents the servers from learning anything about the password or secret, as well as from planting a different secret into the user's mind than the secret that he stored earlier.

Our construction is inspired by the protocol of Bagherzandi et al. [3] by relying on a homomorphic threshold cryptosystem, but the crucial difference is that in our retrieve protocol, the user never sends out an encryption of his password attempt. Instead, the user derives an encryption of the (randomized) quotient of the password used at setup and the password attempt. The servers then jointly decrypt the quotient and verify whether it yields "1", indicating that both passwords matched. In case the passwords were not the same, all the servers learn is a random value.

The Case for Universal Composability. We prove our protocol is secure in the universal composability (UC) framework [16]. The particular advantages of UC security notions for the special case of password-based protocols have been pointed out before [18, 13]; we recall the main arguments here. First, all property-based security notions for threshold password-based protocols in the literature [40, 47, 38, 3] assume that honest users choose their passwords from known, fixed, independent distributions. In reality, users share, reuse, and leak information related to their passwords outside of the protocol. Second, all known property-based notions allow the adversary to observe or even interact with honest users with their correct passwords, but not on incorrect yet related passwords—which is exactly what happens when a user makes a typo while entering his password. In the UC framework, this is modeled more naturally by letting the environment provide the passwords, so no assumptions need to be made regarding their distributions, dependencies, or leakages. Finally, property-based definitions consider the protocol in isolation, while the composition theorem of the UC framework guarantees secure composition with itself as well as with other protocols. Composition with other protocols is of particular importance in the considered TPASS setting, where a user shares and reconstructs a strong key K with multiple servers, and should be able to securely use that key in a different protocol, for instance to decrypt data kept in the cloud. Modeling secure composition of password-based protocols is particularly delicate given the inherent non-negligible success probability of the adversary when guessing the password. Following previous work [18, 13], our UC notion absorbs the inherent guessing attacks into the ideal functionality itself. A secure protocol guarantees that the real world and ideal world are indistinguishable, thus the composition theorem continues to hold.

Building a UC secure protocol requires many additional tools, such as simulation-sound non-interactive zero-knowledge proofs with online witness extraction (which can be efficiently realized for discrete-logarithm based relations in the random-oracle model) and CCA2-secure encryption. It is all the more surprising that our final protocol is efficient enough for use in practice: It requires only $5n + 15$ and $14t + 24$ exponentiations from the user during setup and retrieval, respectively. Each server has to perform $n + 18$ and $7t + 28$ exponentiations in these respective protocols.

Related Work. In spite of their practical relevance, TPASS protocols only started to appear in the literature very recently. The t-out-of-n TPASS protocol by Bagherzandi et al. [3] was proved secure under a property-based security notion. As mentioned above, it relies on untamperable user memory and breaks down when the user retrieves its secret from all-corrupt servers. Our protocol can be seen as a strengthened version of the Bagherzandi et al. protocol; we refer to Section 4 for a detailed comparison. The 1-out-of-2 TPASS protocol by Camenisch et al. [13] was proved secure in the UC framework,

but, by construction, leaks the password and secret if a user tries to retrieve his secret from all-corrupt servers.

Constructing TPASS protocols from generic multi-party computation (MPC) is possible but yields inefficient protocols. Our strong security requirements require public-key operations to be encoded in the to-be-evaluated circuit, while the state-of-the-art MPC protocols [23, 24, 22] require an expensive joint key-generation step to be performed at each retrieval. We refer to the full paper [12] for details.

The closely related primitive of threshold password-authenticated key exchange (TPAKE) lets the user agree on a fresh session key with each of the servers, but doesn't allow the user to store and recover a secret. Depending on the desired security properties, one can build a TPASS scheme from a TPAKE scheme by using the agreed-upon session keys to transmit the stored secret shares over secure channels [3].

The first TPAKE protocols due to Ford and Kaliski [29] and Jablon [36] were not proved secure. The first provably secure TPAKE protocol, a t-out-of-n protocol, was proposed by MacKenzie et al. [40]. The 1-out-of-2 protocol of Brainard et al. [9, 47] is implemented in EMC's RSA Distributed Credential Protection [44]. Both protocols either leak the password or allow an offline attack when the retrieval is performed with corrupt servers (see the full paper [12]). The t-out-of-n TPAKE protocols by Di Raimondo and Gennaro [26] and the 1-out-of-2 protocol by Katz et al. [38] are proved secure under property-based (i.e., non-UC) notions. These protocols actually remain secure when executed with all-corrupt servers, but are restricted to the cases where $n > 3t$ and $(t, n) = (1, 2)$, respectively.

Boyen [7] presented a protocol related to TPASS, where a user can store a *random* value under a password with a *single* server. While being very efficient, this protocol fails to provide most of the security properties we require, i.e., the server can set up the user with a wrong secret, throttling is not possible, and no UC security is offered.

2 Definition of Security

Recall the goal of a TPASS scheme: at setup, a user secret-shares his data amongst n servers protected by a password p; at retrieval, he can recover his data from a subset of $t + 1$ of these n servers, assuming that at most t of them are corrupt. For the sake of simplicity, we assume that the user's data is a symmetric key K; the user can then always use K to encrypt and authenticate his actual data and store the resulting ciphertext in the cloud.

We want the user to be able to retrieve his data remembering only his username uid and his password, and perhaps the name of one or a couple of his trusted servers. The user has access to the PKI but cannot be assumed to store any additional information, cryptographic or other. In particular, the user does not have to remember the names or public keys of *all* of the servers among which he shared his key. Rather, in a step preceding the retrieval (that we don't model here), he can ask some servers to remind him of his full list of servers. Of course, these servers may lie if they are malicious, tricking the into retrieving his key from servers that weren't part of the original setup. We want to protect the user in this case and prevent the servers from learning the password.

Certain attacks are inherent and cannot be protected against. For example, a corrupt user can always perform an online attack on another user's password p by doing several retrieval attempts. It is therefore crucial that honest servers detect failed retrieval attempts, so that they can apply throttling mechanisms to stop or slow down the attack, such as blocking the user's account or asking the user to solve a CAPTCHA. The throttling mechanism should count retrieval attempts that remain pending for too long as failed attempts, since the adversary can always cut the communication before some of the servers were able to conclude.

A second inherent attack is that if at least $t + 1$ of the n servers at setup are corrupt, then these servers can mount an offline dictionary attack on the user's password p. Given the low entropy in human-memorizable passwords and the efficiency of offline dictionary attacks on modern hardware, one may conservatively assume that in this case the adversary simply learns p and K—which is how we model it here.

A somewhat subtle but equally unavoidable attack is that when an honest user makes a retrieval attempt with a set of all-corrupt servers, the servers can try to plant any key K^* of their choice into the user's output. This attack is unavoidable, because the corrupt servers can always pretend that they participated in a setup protocol for a "planted" password p^* and a "planted" key K^*, and then execute the retrieve protocol with the honest user using the information from this make-believe setup. If the planted password p^* matches the password p' the user is retrieving with, the user will retrieve the planted key K^* instead of his real key. Note that in the process, the adversary learns whether $p^* = p'$, thus he gets a guess at the password p'. This planting attack is even more critical if the user previously set up his account with at least $t + 1$ corrupted servers, because in that case the adversary already knows the real password p, which most likely is equal to the password p' with which the user runs the retrieval.

Finally, in our model, all participants are communicating over an adversarial network, which means that protocol failures are unavoidable: the adversary may block communication between honest servers and the user. As a result, we cannot guarantee that the user always succeeds in retrieving his data. In view of this, we chose to restrict the retrieval protocol to $t + 1$ servers: although this choice causes the retrieve protocol to fail if just one server refuses to (being adversarial), adversarial failures are already unavoidable in our network model. We could still try to guarantee some limited form of robustness (recall that, in the threshold cryptography literature, a protocol is *robust* if it can successfully complete its task despite malicious behavior from a fraction of participants) by requiring that, when $t + 1$ or more honest servers participate and the network does not fail, the user successfully recovers his data. However, while it seems not hard to add robustness to our protocols by applying the usual mechanisms found in the literature, it turns out that modeling robustness would considerably complicate our (already rather involved) ideal functionality.

2.1 Ideal Functionality

Assuming the reader is familiar with the UC framework [16], we now describe the ideal functionality $\mathcal{F}_{TPASS(t,n)}$ of t-out-of-n TPASS. For simplicity, we refer to $\mathcal{F}_{TPASS(t,n)}$ as $\mathcal{F}$ from now on. It interacts with a set of users $\{\mathcal{U}\}$, a set of servers $\{\mathcal{S}_i\}$ and an

adversary $\mathcal{A}$. We consider static corruptions and assume that $\mathcal{F}$ knows which of the servers in $\{\mathcal{S}_i\}$ are corrupt.

The UC framework allows us to focus our analysis on a single protocol instance with a globally unique *session identifier* sid. Security for multiple sessions follows through the composition theorem [16] or, if different sessions are to share state, through the joint-state universal composition (JUC) theorem [19]. Here, we use the username uid as the session identifier sid, and let each setup and retrieve query be assigned a unique *sub-session identifier* $ssid$ and $rsid$ within the single-session functionality for $sid = uid$. When those sub-session identifiers are established through the functionality by Barak et al. [4], they have the form $ssid = (ssid', \mathbf{S})$ and $rsid = (rsid', \mathbf{S}')$, respectively, i.e., they consist of a globally unique string and the identifiers of the servers $\mathbf{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ that agreed on that identifier. We will later motivate these choices; for now, it suffices to know that a session identifier $sid = uid$ corresponds to a single user account, and that the sub-session identifiers $ssid$ and $rsid$ refer to individual setup and retrieve queries for that account.

The functionality $\mathcal{F}$ has two main groups of interfaces, for setup and retrieve. For the sake of readability, we describe the behavior of those interfaces in a somewhat informal way here and provide their formal specification in the full paper [12].

Setup Interfaces: The `SETUP`-related interfaces allow a user $\mathcal{U}$ to instruct $\mathcal{F}$ to store the a key K, protected under a password p, among n servers $\mathbf{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ of the user's choice.

1. A $(\texttt{SETUP}, sid, ssid, p, K)$ message from a user $\mathcal{U}$ initiates the functionality for user name $uid = sid$. The sub-session identifier $ssid$ contains a list of n different server identities $\mathbf{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ among which $\mathcal{U}$ wants to share his key K protected by the password p. If at least $t + 1$ servers in $\mathbf{S}$ are corrupt, $\mathcal{F}$ sends the password and the key to the adversary, otherwise it merely informs $\mathcal{A}$ that a setup sub-session is taking place. $\mathcal{F}$ also creates a record s where it stores $s.ssid, s.p, s.K$ and sets $s.\mathcal{R} \leftarrow \mathcal{U}$.
2. A $(\texttt{JOIN}, sid, ssid, \mathcal{S}_i)$ message from the adversary $\mathcal{A}$ instructs $\mathcal{F}$ to let a server $\mathcal{S}_i$ join the setup. If $\mathcal{S}_i$ is honest, this means that $\mathcal{S}_i$ registers the setup and will not join any further setups for the same username $uid = sid$. The user is informed that $\mathcal{S}_i$ joined the setup.
3. A $(\texttt{STEAL}, sid, ssid, \hat{p}, \hat{K})$ message from $\mathcal{A}$ models a rather benign but unavoidable attack where the adversary "steals" the sub-session $ssid$ by intercepting and replacing the network traffic generated by $\mathcal{U}$, allowing $\mathcal{A}$ to replace the password and the key provided by $\mathcal{U}$ with his own choice $s.p \leftarrow \hat{p}$ and $s.K \leftarrow \hat{K}$. Note that this is not a very powerful attack, since the adversary could achieve essentially the same effect by letting a corrupt user initiate a separate setup session for $\hat{p}, \hat{K}$. Thus, the only difference is that here the adversary uses the $ssid$ generated by an honest user, and not a fresh one. Servers are unaware when such an attack takes place, but the user $\mathcal{U}$ cannot be made to believe that an honest server $\mathcal{S}_i$ has accepted his inputs. This is modeled by setting the recipient of server confirmations $s.\mathcal{R} \leftarrow \mathcal{A}$.

Retrieve Interfaces: The RETRIEVE-related interfaces allow $\mathcal{U}'$ to retrieve the key from $t+1$ servers $\mathbf{S}'$ if $\mathbf{S}' \subseteq \mathbf{S}$ and $\mathcal{U}'$ furnishes the correct password; it also models the planting attack described earlier.

4. A $(\texttt{RETRIEVE}, sid, rsid, p')$ message from a user $\mathcal{U}'$ instructs $\mathcal{F}$ to initiate a retrieval for username $uid = sid$ with password p' from the set $\mathbf{S}' = \mathcal{S}_1, \ldots, \mathcal{S}_{t+1}$ of $t+1$ servers included in the sub-session identifier $rsid$. $\mathcal{F}$ then creates a retrieve record r, where it stores $r.rsid, r.p'$, sets $r.\mathcal{R} \leftarrow \mathcal{U}'$, and initially sets $r.ssid \leftarrow \perp$ and $r.K \leftarrow \perp$. If there was a setup sub-session $ssid$ that all honest servers in $\mathbf{S}'$ have joined and where all servers in $\mathbf{S}'$ also occur in $\mathbf{S}$, then $\mathcal{F}$ links this retrieve to $ssid$ by setting $r.ssid \leftarrow ssid$. $\mathcal{F}$ notifies the adversary and (with an adversarially determined delay) the honest servers in $\mathbf{S}'$ that a new retrieval is taking place. Note that the password attempt p' is *not* leaked to the adversary, even if all servers in $\mathbf{S}'$ are corrupt.
5. A $(\texttt{PLANT}, sid, rsid, p^*, K^*)$ message from the adversary $\mathcal{A}$ allows him to perform the *planting attack* described earlier. Namely, if all $t+1$ servers in the retrieval are corrupt, $\mathcal{A}$ can submit a password p^* and a key K^* to be planted. The functionality $\mathcal{F}$ tells $\mathcal{A}$ whether p^* matches the password attempt p'. If so, $\mathcal{F}$ also sets the key $r.K$ that will eventually be returned in this session to the to-be-planted key K^* provided by the adversary. Note that the adversary can perform only one planting attack per retrieval. So even if all $t+1$ servers are corrupt, the adversary only obtains a single guess for the retrieval password p'.
6. A $(\texttt{STEAL}, sid, rsid, \hat{p})$ message from $\mathcal{A}$ allows the adversary to "steal" the sub-session identifier $rsid$, replacing the original password attempt $r.p'$ with $\hat{p}$ of his choice. Servers do not notice this attack taking place, but the originating user will conclude that the protocol failed, or not receive any output at all. This is modeled again by setting $r.\mathcal{R} \leftarrow \mathcal{A}$.
7. A $(\texttt{PROCEED}, sid, rsid, a)$ message with $a \in \{\texttt{allow}, \texttt{deny}\}$ coming from an honest server $\mathcal{S}_i$ (after having been notified that a retrieval is taking place) indicates its (un)willingness to participate in the retrieval. This models the opportunity for an external throttling mechanism to refuse this retrieval attempt. Only when *all* honest servers have agreed to participate, the retrieval continues and the adversary learns whether the passwords matched (i.e., whether $r.p' = s.p$ with s being the setup record for $ssid$). If they matched, $\mathcal{F}$ also sets the key to be returned $r.K$ to the key shared during setup $s.K$.
8. A $(\texttt{DELIVER}, sid, rsid, \mathcal{P}, a)$ message from $\mathcal{A}$ where $a = \texttt{allow}$ instructs $\mathcal{F}$ to output the final result of this retrieval to party $\mathcal{P}$, which can either be an honest server $\mathcal{S}_i$ or the user specified in $r.\mathcal{R}$. If $\mathcal{P} = r.\mathcal{R}$, the user will obtain the value $r.K$, where the result will signal a successful retrieval only if $r.K \neq \perp$, i.e., a key was assigned after the passwords matched. When $\mathcal{P} = \mathcal{S}_i$, the server will receive either a success or failure notification, indicating whether the passwords matched. Note that, in both cases, $\mathcal{A}$ can still turn a successful result into a failed one by passing $a = \texttt{deny}$ as input. This is because in the real world, the adversary can always make a party believe that a protocol ended unsuccessfully by simply dropping or invalidating correct messages. However, the inverse is not possible, i.e., the adversary can not make a mismatch of the passwords look like a match.

Session Identifiers. Our choice of (sub-)session identifiers merits some further explanation. In the UC framework, all machine instances participating in a protocol execution, including ideal functionalities, share a globally unique session identifier sid. Obviously, our SETUP and RETRIEVE interfaces must be called with the same sid to provide the expected functionality, because otherwise the instance cannot keep state between setup and retrieval. However, we insisted that a user can only be expected to remember a username and a password between setup and retrieve, but no further information such as public keys or random nonces. The sid therefore consists only of the username uid and thus cannot be used to uniquely identify different setup or retrieval sub-sessions for this username. To allow the functionality to refer to multiple simultaneous setup and retrieve sub-sessions, the participants of each sub-session establish a unique sub-session identifier $ssid$ or $rsid$ using the standard techniques mentioned earlier [4]. Therein, a unique identifier is created by simply concatenating the identities of the communicating parties and random nonces sent by all parties.

3 Preliminaries

In this section we introduce the building blocks for our protocols. These are three kinds of public-key encryption schemes, a signature scheme, and zero-knowledge proof protocols. We require two of the encryption schemes to be compatible, i.e., the message space to be the same algebraic group. To this end we make use of a probabilistic polynomial-time algorithm GGen that on input the security parameter 1^τ outputs the description of a multiplicative cyclic group $\mathbb{G}$, its prime order q, and a generator g, and require the key generation algorithms of the compatible encryption schemes to take $\mathbb{G}$ as input instead of the security parameter.

CPA-Secure Public-Key Encryption Scheme. Such a scheme consists of three algorithms $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$. The key generation algorithm KGen on input $(\mathbb{G}, q, g)$ outputs a key pair (epk, esk). The encryption algorithm Enc, on input a public key epk and a message $m \in \mathbb{G}$, outputs a ciphertext C, i.e., $C \leftarrow \mathsf{Enc}_{epk}(m)$. The decryption algorithm Dec, on input the secret key esk and a ciphertext C, outputs a message $m \leftarrow \mathsf{Dec}_{esk}(C)$. We require this scheme to satisfy the standard CPA-security properties, with key generation defined as $\mathsf{KGen}(\mathsf{GGen}(1^\tau))$.

CCA2-Secure Labeled Public-Key Encryption Scheme. Any standard CCA2-secure scheme $(\mathsf{KGen2}, \mathsf{Enc2}, \mathsf{Dec2})$ that supports labels [14] is sufficient fir our protocols. Therein, $(epk, esk) \leftarrow \mathsf{KGen2}(1^\tau)$ denotes the key generation algorithm. The encryption algorithm takes as input the public key epk, a message m, a label $l \in \{0,1\}^*$ and outputs a ciphertext $C \leftarrow \mathsf{Enc2}_{epk}(m, l)$. The decryption $\mathsf{Dec2}_{esk}(C, l)$ of C will either output a message m or a failure symbol $\perp$. The label l can be seen as context information which is non-malleably attached to a ciphertext C and restricts the decryption of C to that context, i.e., decryption with a label different from the one used for encryption will fail.

Semantically Secure (t, n)-Threshold Homomorphic Cryptosystem. Such a scheme consists of five algorithms $(\mathsf{TKGen}, \mathsf{TEnc}, \mathsf{PDec}, \mathsf{VfDec}, \mathsf{TDec})$. The key generation

algorithm TKGen, on input $(\mathbb{G}, q, g, t, n)$, outputs a public key tpk and n partial key pairs $(tpk_1, tsk_1), \ldots, (tpk_n, tsk_n)$.

The encryption algorithm TEnc, on input a public key tpk and a message $m \in \mathbb{G}$, outputs a ciphertext C. The partial decryption algorithm PDec, on input (tsk_i, C), outputs a decryption share d_i and a proof π_{d_i}. The decryption share verification algorithm VfDec, on input $(tpk_i, C, d_i, \pi_{d_i})$, verifies that d_i is correct w.r.t. C and tpk_i. The threshold decryption algorithm TDec, on input C and $k \geq t+1$ decryption shares $d_{i_1}, \ldots, d_{i_k}$, outputs a plaintext m or $\perp$.

Our protocol will require that the threshold scheme has an appropriate *homomorphic property*, namely that there is an efficient operation $\odot$ on ciphertexts such that, if $C_1 \in \mathsf{TEnc}_{tpk}(m_1)$ and $C_2 \in \mathsf{TEnc}_{tpk}(m_2)$, then $C_1 \odot C_2 \in \mathsf{TEnc}_{tpk}(m_1 \cdot m_2)$. We will also use exponents to denote the repeated application of $\odot$, e.g., C_1^2 to denote $C_1 \odot C_1$.

Further, the scheme needs to be *sound* and *semantically secure*. In a nutshell, the former means that for a certain set of public keys $tpk, tpk_1, \ldots, tpk_n$ a ciphertext C can be opened only in an unambiguous way. The latter property of semantic security can be seen as an adaptation of the normal semantic security definition to the threshold context, where the adversary can now have up to t of the partial secret keys. In our full paper [12], we provide a detailed description of those properties, which are an adaption of the definitions by Cramer, Damgård, and Nielsen [20] for semantically secure threshold homomorphic encryption. The full paper further contains a construction based on the ElGamal cryptosystem that achieves our security notion.

Existentially Unforgeable Signature Scheme. By $(\mathsf{SKGen}, \mathsf{Sign}, \mathsf{Vf})$ we denote such schemes, with $(spk, ssk) \leftarrow \mathsf{SKGen}(1^\tau)$ being the key generation algorithm. For signing of a message $m \in \{0,1\}^*$, we write $\sigma \leftarrow \mathsf{Sign}_{ssk}(m)$, and for verification we write $b \leftarrow \mathsf{Vf}_{spk}(m, \sigma)$, where the output b will be either 1 or 0, indicating success or failure.

Simulation-Sound Zero-Knowledge Proof System. We require a non-interactive zero-knowledge (NIZK) proof system to prove relations among different ciphertexts. We use an informal notation for this proof system, e.g., $\pi \leftarrow \mathrm{NIZK}\{(m) : C_1 = \mathsf{TEnc}_{tpk}(m) \wedge C_2 = \mathsf{Enc}_{epk}(m)\}$ $(ctxt)$ denotes the generation of a non-interactive zero-knowledge proof that is bound to a certain context *ctxt* and proves that C_1 and C_2 are both proper encryptions of the same message m under the public key tpk and epk for the encryption scheme TEnc and Enc, respectively. We require the proof system to be simulation-sound [45] and zero-knowledge. In the full paper [12], we give concrete realizations of the NIZK proofs that we require in our protocols assuming specific instantiations of the encryption schemes.

4 Our TPASS Protocol

The core of our construction bears a lot in common with that of Bagherzandi et al. [3], which however does rely on trusted user storage and is not proven to be UC secure. We first summarize the idea of their construction and then explain the changes and extensions we made to remove the trusted storage assumption and achieve UC security according to our TPASS functionality.

The high-level idea of Bagherzandi et al. [3] is depicted in Figure 1 and works as follows: In the setup protocol, the user generates keys for a threshold encryption

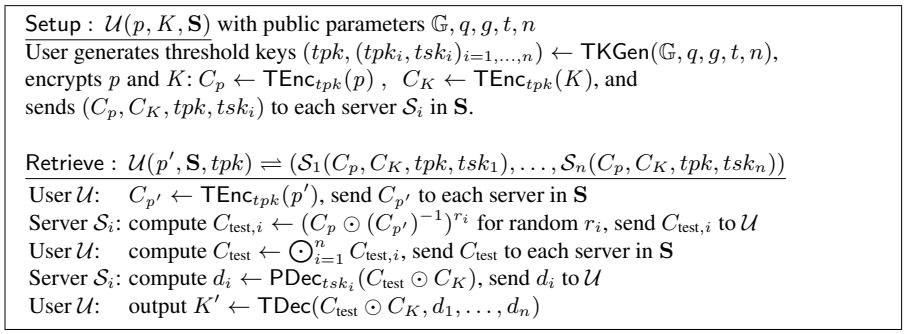

Fig. 1. Construction outline of the Bagherzandi et al. protocol. For the sake of simplicity, we slightly deviate from the notation introduced in Section 3 and omit the additional output of π_{d_i} of PDec.

scheme, encrypts both the password p and the key K using the generated public key, and sends these encryptions and generated decryption key shares to all n servers in $\mathbf{S}$. In addition to his username and password, the user here needs to remember the main public key tpk of the threshold scheme and the servers he ran the setup with. In the retrieve protocol, the user encrypts his password attempt p' under tpk and sends the ciphertext to all the servers in $\mathbf{S}$. The servers now compute an encryption of the password quotient p/p' and combine it with the encryption of the key K. With their help, the user decrypts this combined encryption. If $p = p'$, this will decrypt to 1 the original key K, otherwise it will decrypt to a random value.

It is easy to see that the user *must* correctly remember tpk and the exact set of servers, as he sends out an encryption of his password attempt p' under tpk. If tpk can be tampered with and changed so that the adversary knows the decryption key, then the adversary can decrypt p'. (Bagherzandi et al. [3] actually encrypt $g^{p'}$, so that the malicious servers must still perform an offline attack to obtain p' itself. However, given the typical low entropy of passwords, the password p' can be considered as leaked.)

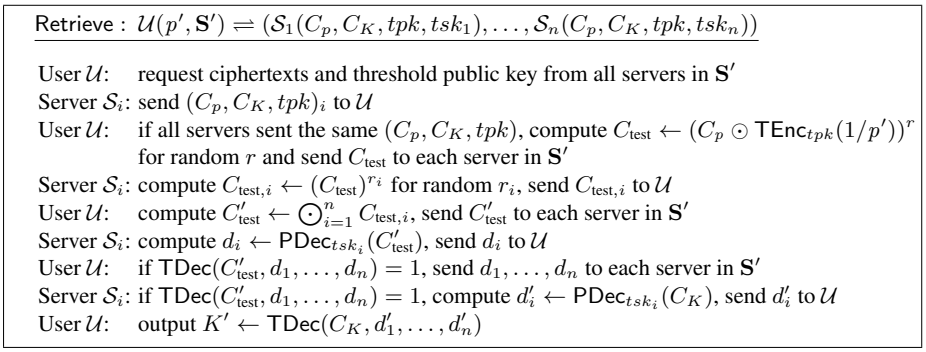

Fig. 2. Construction outline of our retrieval protocol (setup idea as in Figure 1).

Removing the Trusted User-Storage Requirement. Roughly, we change the retrieval protocol such that the user never sends out an encryption of his password attempt p', but instead sends an encryption of the randomized quotient p/p'. Thus, if the user mistakenly talks to adversarial servers instead of his true friends, these servers can try a guess at p', but will not be able to learn anything more. Our retrieval protocol begins with the user requesting the servers in $\mathbf{S}'$ (which may or may not be a subset of $\mathbf{S}$) to send him the ciphertexts and threshold public key he allegedly used in setup. If all servers respond with the same information, the user takes the received encryption of p and uses the homomorphism to generate a randomized encryption of p/p'. The servers then jointly decrypt this ciphertext. If it decrypts to 1, i.e., the two passwords match, then the servers send the user their decryption shares for the ciphertext encrypting the key K. By separating the password check and the decryption of K, the user can actually double-check whether his password was correct and whether he reconstructed his real key K.

Making the Protocol UC-Secure. The second main difference of our protocol is its UC security, which requires further mechanisms and steps added to the construction outlined in Figure 2. We briefly summarize the additional changes, the detailed description of our protocol follows later. First, in the security proof we need to extract p, p', and K from the protocol messages. This is achieved through a common reference string (CRS) that contains the public key PK of a semantically secure encryption scheme and the parameters for a non-interactive zero-knowledge (NIZK) proof system. Values that need be extractable are encrypted under PK and NIZK proofs are added to ensure that the correct value is encrypted. Further, all $t + 1$ servers explicitly express their consent with previous steps by signing all messages. The user collects, verifies, and forwards these signatures, so that all servers can verify the consent of all other servers. Some of these ideas were discussed by Bagherzandi et al., but only for a specific instantiation of ElGamal encryption and without aiming for full-blown UC security. Our protocol, on the other hand, is based on generic building blocks and securely implements the UC functionality presented in Section 2.

How to Remember the Servers. For the retrieve protocol, we assume that the input of the user contains $t+1$ server names. In practice, however, the user might not remember these names. This is an orthogonal issue and there are a number of ways to deal with it. For instance, if the user remembers a single server name, he can contact that server and ask to be reminded of the names of all n servers. The user can then decide with which $t + 1$ of these servers to run the retrieve protocol. The user could even query more than one server and see whether they agree on the full server list. Again, the crucial point is that the security of our protocol does not rely on remembering the $t + 1$ server names correctly, as the security of the password p' is not harmed, even when the user runs the retrieve protocol with $t + 1$ malicious servers.

A Note on Robustness. As discussed in Section 2, the restriction to run the retrieve protocol with exactly $t + 1$ servers rather stems from the complexity that robustness would add to our ideal functionality, than from an actual protocol limitation. With asynchronous communication channels, one can achieve only a very limited form of robustness where the protocol succeeds if there are enough honest players *and* the adversary,

who controls the network, lets the honest players communicate. Conceptually, one could add such limited robustness by running the retrieve protocol with all n servers and in each step continue the protocol only with the first k servers that sent valid response, where $t + 1 \leq k \leq n$. Bagherzandi et al. [3] handle robustness similarly by running the protocol with all n servers, mark servers that cause the protocol to fail as corrupt, and restart the protocol with at least $t + 1$ servers that appear to be good. To obtain better robustness guarantees, one must impose stronger requirements on the network such as assuming synchronous and broadcast channels, as is often done in the threshold cryptography literature [1, 2, 20]. With synchronous channels, protocols can achieve a more meaningful version of robustness, where it is ensured that inputs of all honest parties will be included in the computation and termination of the protocol is guaranteed when sufficient honest parties are present [39]. However, in practice, networks are rarely synchronous, and it is known that the properties guaranteed in a synchronous world cannot simultaneously be ensured in an asynchronous environment [21, 6]. Thus, given the practical setting of our protocol, we prefer the more realistic assumptions over modeling stronger (but unrealistic) robustness properties.

4.1 Detailed Description of our TPASS Protocol

In our protocol description, when we say that a party sends a message m as part of the setup or retrieve protocol, the party actually sends a message $(\texttt{SETUP}, sid, ssid, i, m)$ or $(\texttt{RETRIEVE}, sid, rsid, i, m)$, respectively, where i is a sequence number corresponding to the step number in the respective part of the protocol. Each party will only accept the first message that it receives for a specific (sub-)session identifier and sequence number. All subsequent messages from the same party for the same step of the protocol will be ignored.

Each party locally maintains state information throughout the different steps of one protocol execution; servers $\mathcal{S}_i$ additionally maintain a persistent state variable $st_i[sid]$ associated with the username $sid = uid$ that is common to all executions. Before starting a new execution of the setup or retrieve protocol, we assume that the parties use standard techniques [16, 4] to agree on a fresh and unique sub-session identifier $ssid'$ and $rsid'$, respectively, that is given as an input to the protocol. Each party then only accepts messages that include a previously established sub-session identifier, messages with unknown identifiers will be ignored. We also assume that the sub-session identifiers $ssid$ and $rsid$ explicitly contain the identities of the communicating servers $\mathbf{S}$ and $\mathbf{S}'$, respectively. Using the techniques described in [4], the sub-session identifier would actually also contain the identifier of the user. However, as we do not assume that users have persistent public keys, we could not verify whether a certain user indeed belongs to a claimed identifier, and thus we discard that part of the output.

Setup Protocol. We assume that the system parameters contain a group $\mathbb{G} = \langle g \rangle$ of order q that is a τ-bit prime, and that the password p and the key K can be mapped into $\mathbb{G}$. In the following we assume that p and K are indeed elements of $\mathbb{G}$. We further assume that each server $\mathcal{S}_i$ has a public key (epk_i, spk_i), where epk_i is a public encryption key for the CCA2-secure encryption scheme generated by $\mathsf{KGen2}$ and spk_i is

a signature verification key generated by SKGen. We also assume a public-key infrastructure where servers can register their public keys, modeled by the ideal functionality $\mathcal{F}_{CA}$ by Canetti [17]. Moreover, we require a common reference string, retrievable via functionality $\mathcal{F}_{CRS}$, containing a public key $PK \in \mathbb{G}$ of the CPA-secure public-key encryption scheme, distributed as if generated through KGen, but to which no party knows the corresponding secret key.

The user $\mathcal{U}$, on input $(\texttt{SETUP}, sid, ssid, p, K)$ with $ssid = (ssid', \mathbf{S})$, runs the following protocol with all servers in $\mathbf{S}$. Whenever a check fails for a party (either the user or one of the servers), the party aborts the protocol without any output.

Step S1. The user $\mathcal{U}$ sets up secret key shares and note:

(a) Query functionality $\mathcal{F}_{CRS}$ to obtain PK and, for each $\mathcal{S}_i$ occurring in $\mathbf{S}$, query $\mathcal{F}_{CA}$ to obtain $\mathcal{S}_i$'s public keys (epk_i, spk_i).

(b) Run $(tpk, tpk_1, \ldots, tpk_n, tsk_1, \ldots, tsk_n) \leftarrow \mathsf{TKGen}(\mathbb{G}, q, g, t, n)$ and encrypt the password p and the key K under both tpk and PK, i.e., compute

$$C_p \leftarrow \mathsf{TEnc}_{tpk}(p)\ ,\ C_K \leftarrow \mathsf{TEnc}_{tpk}(K)\ ,\ \widetilde{C}_p \leftarrow \mathsf{Enc}_{PK}(p)\ ,\ \widetilde{C}_K \leftarrow \mathsf{Enc}_{PK}(K).$$

(c) Generate a non-interactive zero-knowledge proof π_0 that the ciphertexts encrypt the same password and key, bound to $\texttt{ctxt} = (sid, ssid, tpk, \mathbf{tpk}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K)$, where $\mathbf{tpk} = (tpk_1, \ldots, tpk_n)$:

$$\pi_0 \leftarrow \mathrm{NIZK}\{(p, K) :\ C_p = \mathsf{TEnc}_{tpk}(p)\ \wedge\ C_K = \mathsf{TEnc}_{tpk}(K)\ \wedge\ \widetilde{C}_p = \mathsf{Enc}_{PK}(p)\ \wedge\ \widetilde{C}_K = \mathsf{Enc}_{PK}(K)\ \}\ (\texttt{ctxt})\ .$$

(d) Set $note = (ssid, tpk, \mathbf{tpk}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K, \pi_0)$.

(e) Compute $C_{S,i} \leftarrow \mathsf{Enc2}_{epk_i}(tsk_i, (sid, note))$ and send a message $(note, C_{S,i})$ to server $\mathcal{S}_i$ for $i = 1, \ldots, n$.

Step S2. Each server $\mathcal{S}_i$ checks & confirms user message:

(a) Receive $(note, C_{S,i})$ with $note = (ssid, tpk, \mathbf{tpk}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K, \pi_0)$. Check that the variable $st_i[sid]$ has not been initiated yet. Check that the note is valid, i.e., that the proof π_0 is correct and that the sets $\mathbf{tpk}$ and $\mathbf{S}$ have the same cardinality (recall that $\mathbf{S}$ is included in $ssid$). Further, check that $\mathsf{Dec2}_{esk_i}(C_{S,i}, (sid, note))$ decrypts to a valid threshold decryption key tsk_i w.r.t. the received public keys.

(b) Sign sid and note as $\sigma_{1,i} \leftarrow \mathsf{Sign}_{ssk_i}(sid, note)$ and send the signature $\sigma_{1,i}$ to $\mathcal{U}$.

Step S3. The user $\mathcal{U}$ verifies & forwards server signatures:

(a) When valid signatures $(\sigma_{1,1}, \ldots, \sigma_{1,n})$ are received from all servers $\mathcal{S}_i$ in $\mathbf{S}$, forward them to all servers in $\mathbf{S}$.

Step S4. Each server $\mathcal{S}_i$ verifies & confirms server consent:

(a) Upon receiving a message $(\sigma_{1,1}, \ldots, \sigma_{1,n})$ from $\mathcal{U}$, check that all signatures $\sigma_{1,i}$ for $i = 1, \ldots, n$ are valid w.r.t. the local $note$.

(b) Store necessary information in the state $st_i[sid] \leftarrow (note, tsk_i)$.

(c) Compute $\sigma_{2,i} \leftarrow \mathsf{Sign}_{ssk_i}((sid, note), \texttt{success})$, send $\sigma_{2,i}$ to $\mathcal{U}$, and output the tuple $(\texttt{SETUP}, sid, ssid)$.

Step S5. The user $\mathcal{U}$ outputs the servers' acknowledgments:

(a) Whenever receiving a valid signature $\sigma_{2,i}$ from a server $\mathcal{S}_i$ in $\mathbf{S}$, output $(\texttt{SETUP}, sid, ssid, \mathcal{S}_i)$.

Retrieval Protocol. The user $\mathcal{U}'$ on input $(\texttt{RETRIEVE}, sid, rsid, p')$ where $rsid = (rsid', \mathbf{S}')$ runs the following retrieval protocol with the list of $t+1$ servers specified in $\mathbf{S}'$. Whenever a check fails for a party, the party sends a message $(\texttt{RETRIEVE}, sid, rsid, \texttt{fail})$ to all other parties and aborts with output $(\texttt{DELIVER2S}, sid, rsid, \texttt{fail})$ if the party is a server, or with output $(\texttt{DELIVER2U}, sid, rsid, \perp)$ if it is the user. Further, whenever a party receives a message $(\texttt{RETRIEVE}, sid, rsid, \texttt{fail})$, it aborts with the same respective outputs.

Step R1. The user $\mathcal{U}'$ creates ephemeral encryption key & requests notes:

(a) Query $\mathcal{F}_{CRS}$ to obtain PK and, for each $\mathcal{S}_i$ in $\mathbf{S}'$, query $\mathcal{F}_{CA}$ to obtain $\mathcal{S}_i$'s public keys (epk_i, spk_i).
(b) Generate a key pair $(epk_U, esk_U) \leftarrow \mathsf{KGen2}(1^\tau)$ for the CCA2-secure encryption scheme that will be used to securely obtain the shares of the key K from the servers.
(c) Encrypt the password attempt p' under the CRS as $\widetilde{C}_{p'} \leftarrow \mathsf{Enc}_{PK}(p')$.
(d) Request the note from each server by sending $(epk_U, \widetilde{C}_{p'})$ to each server $\mathcal{S}_i \in \mathbf{S}'$.

Step R2. Each server $\mathcal{S}_i$ retrieves & sends signed note:

(a) Upon receiving a retrieve request $(epk_U, \widetilde{C}_{p'})$, check if a record $st_i[sid] = (note, tsk_i)$ exists. Parse $note = (ssid, tpk, \mathbf{tpk}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K, \pi_0)$ and check that all servers in $\mathbf{S}'$ also occur in $\mathbf{S}$. (Recall, that sid and $rsid$ are contained in the header of the message, $\mathbf{S}'$ is included in $rsid$ and $\mathbf{S}$ in $ssid$.)
(b) Query $\mathcal{F}_{CA}$ to obtain the public keys (epk_j, spk_j) of all the other servers $\mathcal{S}_j$ in $\mathbf{S}'$.
(c) Compute $\sigma_{4,i} \leftarrow \mathsf{Sign}_{ssk_i}(sid, rsid, epk_U, \widetilde{C}_{p'}, note)$ and send $(note, \sigma_{4,i})$ back to the user.

Step R3. The user $\mathcal{U}'$ verifies & distributes signatures:

(a) Upon receiving the first message $(note_i, \sigma_{4,i})$ from a server $\mathcal{S}_i \in \mathbf{S}'$, verify the validity of $\sigma_{4,i}$ w.r.t. the previously sent values, and parse $note_i$ as $(ssid, tpk, \mathbf{tpk}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K, \pi_0)$. Check that all servers in $\mathbf{S}'$ occur in $\mathbf{S}$, that the lists $\mathbf{tpk}$ and $\mathbf{S}$ are of equal length, and that the proof π_0 is valid w.r.t. sid. If all checks succeed, set $note \leftarrow note_i$.
(b) Upon receiving any subsequent message $(note_j, \sigma_{4,j})$ from $\mathcal{S}_j$ in $\mathbf{S}'$, check that $\sigma_{4,j}$ is valid for the same note the first server had sent, i.e., verify $note_j = note$. Proceed only after $(note_j, \sigma_{4,j})$ messages from all servers $\mathcal{S}_j \in \mathbf{S}'$ have been received and processed.
(c) Send $(\sigma_{4,j})_{\mathcal{S}_j \in \mathbf{S}'}$ to all servers in $\mathbf{S}'$.

Step R4. Each server $\mathcal{S}_i$ proceeds or halt:

(a) Upon receiving a message $(\sigma_{4,j})_{\mathcal{S}_j \in \mathbf{S}'}$ from the user, verify the validity of every signature $\sigma_{4,j}$ w.r.t. to the locally stored $note$. Output $(\texttt{RETRIEVE}, sid, rsid)$ to the environment.
(b) Upon input $(\texttt{PROCEED}, sid, rsid, a)$ from the environment, check that $a = \texttt{allow}$, otherwise abort the protocol.
(c) Compute a signature $\sigma_{5,i} \leftarrow \mathsf{Sign}_{ssk_i}(rsid, \texttt{allow})$ and send $\sigma_{5,i}$ to $\mathcal{U}'$.

Step R5. The user $\mathcal{U}'$ computes the encrypted password quotient:

(a) Upon receiving a message $\sigma_{5,i}$ from a server $\mathcal{S}_i$ in $\mathbf{S}'$, check that $\sigma_{5,i}$ is a valid signature on $(rsid, \texttt{allow})$. Proceed only after a valid signature $\sigma_{5,i}$ has been received from all servers $\mathcal{S}_i \in \mathbf{S}'$.

(b) Use the homomorphic encryption scheme to encrypt p' and entangle it with the ciphertext C_p from $note$, which supposedly encrypts the password p. That is, select a random $r \leftarrow_{\text{R}} \mathbb{Z}_q$ and compute $C_{\text{test}} \leftarrow (C_p \odot \mathsf{TEnc}_{tpk}(1/p'))^r$.

(c) Generate a proof that C_{test} and $\widetilde{C}_{p'}$ are based on the same password attempt p'. To prevent man-in-the-middle attacks, the proof is bound to $\texttt{ctxt} = (sid, rsid, note, epk_U, C_{\text{test}}, \widetilde{C}_{p'})$ which in particular includes the values epk_U, C_{test}, and $\widetilde{C}_{p'}$ provided by the user so far:

$$\pi_1 \leftarrow \text{NIZK}\{(p', r) : C_{\text{test}} = (C_p \odot \mathsf{TEnc}_{tpk}(1/p'))^r \ \wedge \ \widetilde{C}_{p'} = \mathsf{Enc}_{PK}(p')\}(\texttt{ctxt})$$

(d) Send a message $(C_{\text{test}}, \pi_1, (\sigma_{5,j})_{\mathcal{S}_j \in \mathbf{S}'})$ to all servers in $\mathbf{S}'$.

Step R6. Each server $\mathcal{S}_i$ re-randomizes the quotient encryption:

(a) Upon receiving a message $(C_{\text{test}}, \pi_1, (\sigma_{5,j})_{\mathcal{S}_j \in \mathbf{S}'})$, verify the proof π_1 and validate all signatures $\sigma_{5,j}$.

(b) Choose $r_i \leftarrow_{\text{R}} \mathbb{Z}_q$, compute the re-randomized ciphertext $C'_{\text{test},i} \leftarrow (C_{\text{test}})^{r_i}$ and the proof of correctness $\pi_{2,i} \leftarrow \text{NIZK}\{(r_i) : C'_{\text{test},i} = (C_{\text{test}})^{r_i}\}$. Sign the ciphertext together with the session information as $\sigma_{6,i} \leftarrow \mathsf{Sign}_{ssk_i}(sid, rsid, C_{\text{test}}, \widetilde{C}_{p'}, C'_{\text{test},i})$. Send the message $(C'_{\text{test},i}, \pi_{2,i}, \sigma_{6,i})$ to $\mathcal{U}'$.

Step R7. The user $\mathcal{U}'$ verifies & distributes the re-randomized quotient encryptions:

(a) Upon receiving $(C'_{\text{test},j}, \pi_{2,j}, \sigma_{6,j})$ from all servers $\mathcal{S}_j$ in $\mathbf{S}'$, where the proof $\pi_{2,i}$ and the signature $\sigma_{6,i}$ are valid w.r.t. the previously sent C_{test}, send $(C'_{\text{test},j}, \pi_{2,j}, \sigma_{6,j})_{\mathcal{S}_j \in \mathbf{S}'}$ to all servers in $\mathbf{S}'$.

Step R8. Each server $\mathcal{S}_i$ computes combined quotient encryption & sends its decryption share:

(a) Upon receiving $t+1$ tuples $(C'_{\text{test},j}, \pi_{2,j}, \sigma_{6,j})_{\mathcal{S}_j \in \mathbf{S}'}$ from the user, verify all proofs $\pi_{2,j}$ and all signatures $\sigma_{6,j}$.

(b) Derive $C'_{\text{test}} \leftarrow \bigodot_{S_j \in \mathbf{S}'} C'_{\text{test},j}$ and compute the verifiable decryption share of C'_{test} as $(d_i, \pi_{d_i}) \leftarrow \mathsf{PDec}_{tsk_i}(C'_{\text{test}})$.

(c) Sign the share as $\sigma_{7,i} \leftarrow \mathsf{Sign}_{ssk_i}(rsid, C'_{\text{test}}, d_i)$ and send $(d_i, \pi_{d_i}, \sigma_{7,i})$ to $\mathcal{U}'$.

Step R9. The user $\mathcal{U}'$ checks if $p = p'$ & distributes shares:

(a) When receiving a tuple $(d_i, \pi_{d_i}, \sigma_{7,i})$ from a server $\mathcal{S}_i$ in $\mathbf{S}'$, verify that the signature $\sigma_{7,i}$ and the proof π_{d_i} for the decryption share are valid w.r.t. the locally computed $C'_{\text{test}} \leftarrow \bigodot_{\mathcal{S}_j \in \mathbf{S}'} C'_{\text{test},j}$.

(b) After having received correct decryption shares from all $t+1$ servers in $\mathbf{S}'$, check whether the passwords match by verifying that $\mathsf{TDec}(C'_{\text{test}}, \{d_j\}_{\mathcal{S}_j \in \mathbf{S}'}) = 1$.

(c) Send all decryption shares, proofs, and signatures, $(d_j, \pi_{d_j}, \sigma_{7,j})_{\mathcal{S}_j \in \mathbf{S}'}$ to all servers $\mathcal{S}_i$ in $\mathbf{S}'$.

Step R10. Each servers $\mathcal{S}_i$ checks if $p = p'$ & sends decryption share for K:

(a) Upon receiving $t+1$ tuples $(d_j, \pi_{d_j}, \sigma_{7,j})_{\mathcal{S}_j \in \mathbf{S}'}$, verify that all proofs π_{d_j} and signatures $\sigma_{7,j}$ are valid w.r.t. the locally computed C'_{test}.
(b) Check whether $\mathsf{TDec}(C'_{\text{test}}, \{d_j\}_{\mathcal{S}_j \in \mathbf{S}'}) = 1$.
(c) Compute the decryption share for the key K as $(d'_i, \pi_{d'_i}) \leftarrow \mathsf{PDec}_{tsk_i}(C_K)$.
(d) Compute the ciphertext $C_{R,i} \leftarrow \mathsf{Enc2}_{epk_U}((d'_i, \pi_{d'_i}), (epk_U, spk_i))$ using the user's public key and the own signature public key as label, generate $\sigma_{8,i} \leftarrow \mathsf{Sign}_{ssk_i}(rsid, C_{R,i})$, and send $(C_{R,i}, \sigma_{8,i})$ to the user. Output $(\texttt{DELIVER2S}, sid, rsid, \texttt{success})$.

Step R11. The user $\mathcal{U}'$ reconstructs K:

(a) Upon receiving a pair $(C_{R,i}, \sigma_{8,i})$ from a server $\mathcal{S}_i$ in $\mathbf{S}'$, check that $\sigma_{8,i}$ is valid and, if so, decrypt $C_{R,i}$ to $(d'_i, \pi_{d'_i}) \leftarrow \mathsf{Dec2}_{esk_U}(C_{R,i}, (epk_U, spk_i))$. Verify the validity of d'_i by verifying the proof $\pi_{d'_i}$ w.r.t. C_K taken from $note$.
(b) Once all $t+1$ valid shares have been received, restore the key $K' \leftarrow \mathsf{TDec}(C_K, \{d'_j\}_{\mathcal{S}_j \in \mathbf{S}'})$ and output $(\texttt{DELIVER2U}, sid, rsid, K')$.

4.2 Security and Efficiency

We now provide the results of our security analysis. The proof of Theorem 1 is given in the full paper [12].

Theorem 1. *If* $(\mathsf{TKGen}, \mathsf{TEnc}, \mathsf{PDec}, \mathsf{VfDec}, \mathsf{TDec})$ *is a semantically secure* (t, n)*-threshold homomorphic cryptosystem,* $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *is a CPA-secure encryption scheme,* $(\mathsf{KGen2}, \mathsf{Enc2}, \mathsf{Dec2})$ *is a CCA2-secure labeled encryption scheme, the signature scheme* $(\mathsf{SKGen}, \mathsf{Sign}, \mathsf{Vf})$ *is existentially unforgeable, and a simulation-sound concurrent zero-knowledge proof system is deployed, then our* Setup *and* Retrieve *protocols described in Section 4 securely realize $\mathcal{F}$ in the $\mathcal{F}_{CA}$ and $\mathcal{F}_{CRS}$-hybrid model.*

When instantiated with the ElGamal based encryption scheme for $(\mathsf{TKGen}, \mathsf{TEnc}, \mathsf{PDec}, \mathsf{VfDec}, \mathsf{TDec})$ and $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ (as described in the full version [12]), with the ElGamal encryption scheme with Fujisaki-Okamoto padding [27, 30] for $(\mathsf{KGen2}, \mathsf{Enc2}, \mathsf{Dec2})$, with Schnorr signatures [46, 42] for $(\mathsf{SKGen}, \mathsf{Sign}, \mathsf{Vf})$, and with the Σ-protocols described in the full paper [12], then by the UC composition theorem and the security of the underlying building blocks we have the following corollary:

Corollary 1. *The* Setup *and* Retrieve *protocols described in Section 4 and instantiated as described above, securely realize $\mathcal{F}$ under the DDH-assumption for the group generated by* GGen *in the random-oracle and the $\mathcal{F}_{CA}, \mathcal{F}_{CRS}$-hybrid model.*

Efficiency Analysis: With the primitives instantiated as for Corollary 1, the user has to do $5n + 15$ exponentiations in $\mathbb{G}$ for the Setup protocol and $14t + 24$ exponentiations in the Retrieve protocol. The respective figures for each server are $n + 18$ and $7t + 28$. Counting hash values as half a group element, setup requires four rounds of communication with $n(2.5n + 18.5)$ total transmitted group elements, while retrieval takes ten rounds with $(t+1)(36.5 + 2.5n + 10.5(t+1))$ elements.

Acknowledgements

This research was supported by the European Community's Seventh Framework Programme through grant PERCY (agreement no. 321310). Anna Lysyanskaya is supported by NSF awards 0964379 and 1012060 and by IBM and Google faculty awards.

References

1. M. Abe, S. Fehr. Adaptively Secure Feldman VSS and Applications to Universally-Composable Threshold Cryptography. *CRYPTO 2004*.
2. J. Almansa, I. Damgård, J. B. Nielsen. Simplified Threshold RSA with Adaptive and Proactive Security. *EUROCRYPT 2006*.
3. A. Bagherzandi, S. Jarecki, N. Saxena, Y. Lu. Password-protected secret sharing. *ACM CCS 2011*.
4. B. Barak, Y. Lindell, T. Rabin. Protocol initialization for the framework of universal composability. Cryptology ePrint Archive, Report 2004/006, 2004.
5. M. Bellare, P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. *ACM CCS 1993*.
6. M. Ben-Or, R. Canetti, O. Goldreich. Asynchronous secure computation. *STOC 1993*.
7. X. Boyen. Hidden credential retrieval from a reusable password. *ASIACCS 2009*
8. W. E. Burr, D. F. Dodson, E. M. Newton, R. A. Perlner, W. T. Polk, S. Gupta, E. A. Nabbus. Electronic authentication guideline. *NIST Special Publication 800-63-1, 2011*.
9. J. Brainard, A. Juels, B. S. Kaliski Jr., M. Szydlo. A new two-server approach for authentication with short secrets. *USENIX 2003*.
10. J. Camenisch, A. Kiayias, M. Yung. On the portability of generalized Schnorr proofs. *EUROCRYPT 2009*.
11. J. Camenisch, S. Krenn, V. Shoup. A framework for practical universally composable zero-knowledge protocols. *ASIACRYPT 2011*.
12. J. Camenisch, A. Lehmann, A Lysyanskaya, G. Neven. Memento: how to reconstruct your secrets from a single password in a hostile environment. Cryptology ePrint Archive, Report 2014/429, 2014.
13. J. Camenisch, A. Lysyanskaya, G. Neven. Practical yet universally composable two-server password-authenticated secret sharing. *ACM CCS 2012*.
14. J. Camenisch, V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. *CRYPTO 2003*.
15. J. Camenisch, M. Stadler. Efficient group signature schemes for large groups (extended abstract). *CRYPTO 1997*.
16. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. *FOCS 2001*.
17. R. Canetti. Universally composable signature, certification, and authentication. *17th Computer Security Foundations Workshop*. IEEE Computer Society, 2004.
18. R. Canetti, S. Halevi, J. Katz, Y. Lindell, P. D. MacKenzie. Universally composable password-based key exchange. *EUROCRYPT 2005*.
19. R. Canetti, T. Rabin. Universal composition with joint state. *CRYPTO 2003*.
20. R. Cramer, I. Damgård, J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. *EUROCRYPT 2001*.
21. B. Chor, L. Moscovici. Solvability in asynchronous environments. *FOCS 1989*.
22. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, N. Smart. Practical covertly secure MPC for dishonest majority—or: Breaking the SPDZ limits. *ESORICS 2013*.

23. I. Damgård, J. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. *CRYPTO 2013*.
24. I. Damgård, V. Pastro, N. Smart, S. Zakarias. Multiparty computation from somewhat homomorphic encryption. *CRYPTO 2012*.
25. Y. Desmedt, Y. Frankel. Threshold cryptosystems. *CRYPTO 1989*.
26. M. Di Raimondo, R. Gennaro. Provably secure threshold password-authenticated key exchange. *EUROCRYPT 2003*.
27. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *CRYPTO 1984*.
28. A. Fiat, A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. *CRYPTO 1986*.
29. W. Ford, B. S. Kaliski Jr. Server-assisted generation of a strong secret from a password. *WETICE 2000*, IEEE Computer Society, 2000.
30. E. Fujisaki, T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *CRYPTO 1999*.
31. J. A. Garay, P. D. MacKenzie, K. Yang. Strengthening zero-knowledge protocols using signatures. *EUROCRYPT 2003*.
32. D. Goodin. Why passwords have never been weaker—and crackers have never been stronger. Ars Technica, 2012.
33. J. Gosney. Password Cracking HPC. *Passwords^12 Conference*, 2012.
34. C. Herley, P. C. van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 2012.
35. C. Herley, P. C. van Oorschot, A. S. Patrick. Passwords: If we're so smart, why are we still using them? (panel). *Financial Cryptography 2009*.
36. D. P. Jablon. Password authentication using multiple servers. *CT-RSA 2001*.
37. B. Kaliski. PKCS #5: Password-Based Cryptography Specification. *IETF RFC 2898*, 2000.
38. J. Katz, P. D. MacKenzie, G. Taban, V. D. Gligor. Two-server password-only authenticated key exchange. *ACNS 2005*.
39. J. Katz, U. Maurer, B. Tackmann, V. Zikas. Universally Composable Synchronous Computation. *TCC 2013*.
40. P. D. MacKenzie, T. Shrimpton, M. Jakobsson. Threshold password-authenticated key exchange. *CRYPTO 2002*.
41. P. D. MacKenzie, K. Yang. On simulation-sound trapdoor commitments. *EUROCRYPT 2004*.
42. D. Pointcheval, J. Stern. Security proofs for signature schemes. *EUROCRYPT 1996*.
43. N. Provos, D. Mazières. A Future-Adaptable Password Scheme. *USENIX 1999*.
44. RSA, The Security Division of EMC. New RSA innovation helps thwart "smash-and-grab" credential theft. Press release, 2012.
45. A. Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. *FOCS 1999*.
46. C.-P. Schnorr. Efficient signature generation by smart cards. *J. Cryptol.*, 4(3):161–174, 1991.
47. M. Szydlo, B. S. Kaliski Jr. Proofs for two-server password authentication. *CT-RSA 2005*.