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

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

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

Anna Lysyanskaya
Brown University
anna@cs.brown.edu

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

**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 offline 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, e.g., encrypting data in the cloud. Threshold password-authenticated secret sharing (TPASS) protocols provide exactly this functionality, but 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 in 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 [9], sixteen-character human-memorizable passwords only have 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: $n$ times more effort to verify passwords for an honest server makes offline dictionary attacks at most $n$ times harder.

However, as poorly as passwords stand their ground against offline attacks, they are actually fairly secure against *online* attacks as long as attempts can be *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 exploited in 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 purposes, 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 work, setup is often assumed to take place out-of-band. Given the importance of the setup phase, we follow Camenisch et al. [13] in that we explicitly model account setup and assume 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, with hardcoded keys of root certification authorities (CAs) and an implementation of our protocol. Indeed, we do not want to assume any user-specific state information from the setup phase to be available 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 that 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 sign and send back 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 the above 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 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 Appendix B. 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; 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, if only because users derive passwords from real-world phenomena. (Property-based notions for dependent distributions are possible in principle [5], but it's not clear whether the results carry over to this stronger setting.) 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 such secure composition of password-based protocols is particularly delicate given the inherent non-negligible success probability of the adversary by 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 on-line 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 in a PKI setting. As mentioned above, it relies on untamperable user memory and breaks down when the user retrieves its secret from all-corrupt servers (see Appendix B). 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 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 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 Appendix C 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 key to transmit the stored secret share over a secure channel [3].

The first TPAKE protocols due to Ford and Kaliski [29] and Jablon [36] were not proved secure. The first provably secure TPAKE protocol was proposed by MacKenzie et al. [40], a $t$-out-of-$n$ protocol in a PKI setting. The 1-out-of-2 protocol of Brainard et al. [10] was proved secure by Szydlo and Kaliski [47] and 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 Appendix B). 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 in a hybrid password-only/PKI setting, where the user does not know any public keys, but the servers and an intermediate gateway do have a PKI. These protocols actually remain secure when executed with all-corrupt servers, but are restricted to the cases that $n > 3t$ and $(t, n) = (1, 2)$.

Boyen [8] 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 over $n$ 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 the original $n$ servers 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 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 the servers that he thinks he remembers to remind him of his full list of servers. Of course, these servers may lie if they are malicious, so the user may be tricked into retrieving his key from servers that weren't part of the original setup, some or even all of which may be malicious. 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 the 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 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 off-line 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-belief 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 $p^* = p'$, giving him one free 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 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 fact, we chose to also restrict the retrieval protocol to $t+1$ servers: although this choice causes the retrieve protocol to fail if just one server refuses to participate, failures were 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 TPASS that is parametrized with $(t, n)$. 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 $\{\mathcal{S}_i\}$ 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 $sid = uid$. When those identifiers are established through the functionality described in [4], they will consist of a globally unique string and the identifiers of the parties 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 below, and provide the formal specification of our functionality in Appendix A.

**Setup Functionality:** The SETUP-related interfaces allow a user $\mathcal{U}$ to instruct $\mathcal{F}$ to store the user's key $K$, protected with his 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 $uid = sid$. (See below for a discussion on session identifiers.) 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 password $p$. If at least $t+1$ servers in $\mathbf{S}$ are corrupt, $\mathcal{F}$ sends the password and 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. With a $(\texttt{JOIN}, sid, ssid, \mathcal{S}_i)$ message, 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 this username $uid = sid$. The user is informed that $\mathcal{S}_i$ joined the setup.
3. The $(\texttt{STEAL}, sid, ssid, \hat{p}, \hat{K})$ message 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 key provided by $\mathcal{U}$ with his own choice $s.p \leftarrow \hat{p}, s.K \leftarrow \hat{K}$. 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 $s.\mathcal{R}$ to the adversary $\mathcal{A}$.

**Retrieve Functionality:** 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 plant attack described above.

4. A $(\texttt{RETRIEVE}, sid, rsid, p')$ message is provided by a user $\mathcal{U}'$ to $\mathcal{F}$ to initiate a retrieval for username $uid = sid$ with password $p'$ from the set of $t+1$ servers $\mathbf{S}' = \mathcal{S}_1, \ldots, \mathcal{S}_{t+1}$ 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 is a setup 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) also 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 allows the adversary $\mathcal{A}$ to perform the *planting attack* described above. Namely, if all $t+1$ servers in the retrieval are corrupt, $\mathcal{A}$ can submit a planted password $p^*$ and key $K^*$. The functionality tells $\mathcal{A}$ whether $p^*$ matched the password attempt $p'$. If so, $\mathcal{F}$ also sets the key $r.K$ that will eventually be returned in this session to the 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 again 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. After having been notified that a retrieval is taking place, each honest server $\mathcal{S}_i$ can indicate its (un)willingness to participate in the retrieval using a $(\texttt{PROCEED}, sid, rsid, a)$ message specifying $a \in \{\texttt{allow}, \texttt{deny}\}$. 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. With a $(\texttt{DELIVER}, sid, rsid, \mathcal{P}, a)$ message, the adversary can instruct $\mathcal{F}$ to output the final result of this retrieval to an honest server $\mathcal{S}_i$ or to the user $r.\mathcal{U}'$ (indicated by input $\mathcal{P}$). 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. The servers will receive either a success or failure notication, 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 an extra 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 consist 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 random nonces sent by all parties.

*Relation to 2PASS.* One can verify that a protocol that securely implements $\mathcal{F}_{TPASS(1,2)}$, i.e., the above functionality for the 1-out-of-2 case, is also a secure implementation of the 2PASS ideal functionality due to Camenisch et al. [13]. The converse does not hold, since a 2PASS protocol may leak the password attempt $p'$ during a retrieve with two corrupt servers.

*Insecurity of Protocols with Trusted User-Memory.* In Appendix B we show why existing password-based secret sharing protocols [47, 13, 40, 3] are not secure according to the notion just presented. In a nutshell, the problem is that those protocols rely on trusted-memory on the user side, i.e., the user has to remember the servers he run the setup protocol with. If this memory can be tampered with, or the user simply mistypes a server name in the retrieve phase, then a majority of malicious servers can fully recover the user password $p'$ used in the retrieval. However, our functionality guarantees that even if all servers in the retrieval are malicious, the adversary will only get a single guess $p^*$ for which he learns whether or not $p' = p^*$, but he will not learn the password $p'$ itself.

# 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. Thus we make use of a probabilistic polynomial-time algorithm $\mathsf{GGen}$ that on input the security parameter $1^\tau$ outputs the description of a cyclic group $\mathbb{G}$, its prime order $q$, and a generator $g$, and require the key generation algorithms of the encryption scheme to take $\mathbb{G}$ as input instead of the security parameter.

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

**Semantically secure $(t, n)$-threshold homomorphic cryptosystem:** It consists of five algorithms ($\mathsf{TKGen}$, $\mathsf{TEnc}$, $\mathsf{PDec}$, $\mathsf{VfDec}$, $\mathsf{TDec}$). The key generation algorithm $\mathsf{TKGen}$ 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 $\mathsf{TEnc}$ on input $(tpk, m)$ for a message $m \in \mathbb{G}$ outputs a ciphertext $C$. The partial decryption algorithm $\mathsf{PDec}$ on input $(tsk_i, C)$ outputs a decryption share $d_i$ and a proof $\pi_{d_i}$. The decryption share verification algorithm $\mathsf{VfDec}$ 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 $\mathsf{TDec}$ 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 adaption of the normal semantic security definition to the threshold context, where the adversary can now have up to $t$ of the partial secret keys. More precisely, it consists of two properties:

*(1) Indistinguishability:* This notion is similar to standard indistinguishability and requires that an adversary knowing most $t$ partial secret keys cannot link ciphertexts and messages. That is, for any subset $\{i_1, \ldots, i_t\} \subset \{1, \ldots, n\}$, for any PPT adversary that on input $(tpk, \{tpk_i\})$ and $t$ secret shares $(tsk_{i_1}, \ldots, tsk_{i_t})$, selects a pair of messages $m_0$ and $m_1$, the ciphertexts $C_0 \leftarrow \mathsf{TEnc}_{tpk}(m_0)$ and $C_1 \leftarrow \mathsf{TEnc}_{tpk}(m_1)$ are indistinguishable.

*(2) Simulatability:* Roughly, this property guarantees that there is a simulator that on input $t$ partial secret keys, a ciphertext $C$ and message $m$ can produce decryption shares $(\hat{d}_j, \pi_{\hat{d}_j})$ that convincingly pretend that $C$ is an encryption of $m$. More precisely, there exists an efficient simulator $S$ that on input the public values $(tpk, \{tpk_i\})$, $t$ secret shares $(tsk_{i_1}, \ldots, tsk_{i_t})$, a ciphertext $C$, and a message $m$, outputs $(\hat{d}_j, \pi_{\hat{d}_j})$ for all $j \notin \{i_1, \ldots, i_t\}$ such that they are indistinguishable from the correct values $(d_j, \pi_{d_j})$ that would have been obtained if $C$ was indeed an encryption of $m$ and was correctly decrypted with the $tsk_j$ secret key.

Those definitions are an adaption of the definitions by Cramer, Damgård and Nielsen [20] for semantically secure threshold homomorphic encryption. We refer to Appendix D for a more detailed description of the changed properties and a construction based on the ElGamal cryptosystem that achieves our security notion.

**CCA2-secure labeled public-key encryption scheme:** We can use any standard CCA2-secure scheme ($\mathsf{KGen2}$, $\mathsf{Enc2}, \mathsf{Dec2}$) that supports labels [14]. 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 $\perp$. Roughly, the label 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 different label from the one used for encryption will fail. The *CCA2-security for labeled encryption* is defined similar to standard CCA2-encryption, with the difference that the adversary can send tuples $(m_0, l)$ and $(m_1, l)$ to the challenge oracle which returns the encryption $C_b \leftarrow \mathsf{Enc2}_{epk}(m_b, l)$. The adversary is allowed to query its decryption oracle on tuples $(C_i, l_i) \neq (C_b, l)$, i.e., $C_i$ can even be the challenge ciphertext as long it comes with a different label.

**Existentially unforgeable signature scheme:** Denoted as $(\mathsf{SKGen}, \mathsf{Sign}, \mathsf{Vf})$, with $(spk, ssk) \leftarrow \mathsf{SKGen}(1^\tau)$ being the key generation algorithm. For the signing of a message $m \in \{0,1\}^*$ we write $\sigma \leftarrow \mathsf{Sign}_{ssk}(m)$, and with $b \leftarrow \mathsf{Vf}_{spk}(m, \sigma)$ we denote the public verification algorithm that outputs 1 or 0 to indicate success or failure.

**Simulation-sound zero-knowledge proof system:** We further need a non-interactive zero-knowledge (NIZK) proof system to prove certain relations among different ciphertexts. We use a somewhat informal notation for this proof system, e.g., we use $\pi \leftarrow \mathrm{NIZK}\{(m) : C_1 = \mathsf{TEnc}_{tpk}(m) \wedge C_2 = \mathsf{Enc}_{epk}(m)\}$ $(ctxt)$ to denote 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 $\mathsf{TEnc}$ and $\mathsf{Enc}$, respectively. We require the proof system to be simulation-sound [45] and zero-knowledge. The latter roughly says that there must exist a simulator that can generate simulated proofs which are indistinguishable from real proofs from the view of the adversary. The simulation-soundness is a strengthened version of normal soundness and guarantees that an adversary, even after having seen simulated proofs of false statements of his choice, cannot produce a valid proof of a false statement himself.

In Appendix E we give concrete realizations of the NIZK proofs that we require in our protocols assuming specific instantiations of the encryption schemes described.

# 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. Thus, 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.

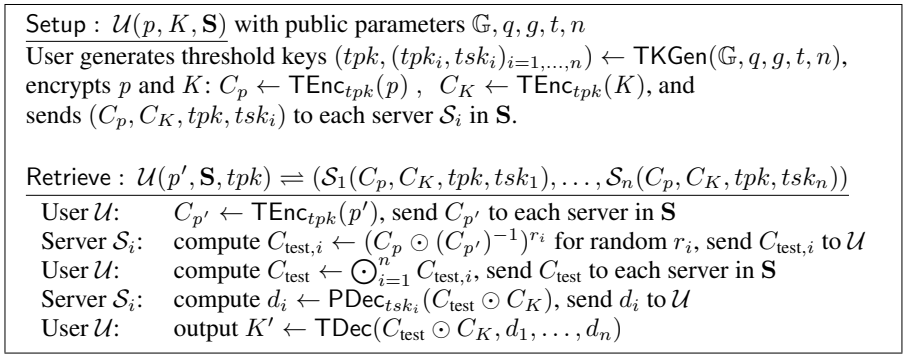


Figure 1: Construction outline of the Bagherzandi et al. protocol. For the sake of simplicity, we slightly deviate from the notation introduced in Section 3 and omit the additional output of $\pi_{d_i}$ of $\mathsf{PDec}$.

The high-level idea of Bagherzandi et al. [3] is depicted in Figure 1 and works as follows: In the setup protocol, the user encrypts both the password $p$ and the key $K$ using a threshold encryption scheme and sends these encryptions and secret key shares to all $n$ servers in $\mathbf{S}$. In addition to its username and password, the user has to remember the main public key $tpk$ of the threshold scheme and the servers he run 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 the encrypted password quotient $p/p'$ and jointly decrypt the combined ciphertext of the quotient and the key $K$. If $p = p'$, the quotient will decrypt to 1 and thus the decryption shares will yield the original key $K$.

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 efficiency of offline attacks and the low entropy of password, the password $p'$ can be considered as leaked.)

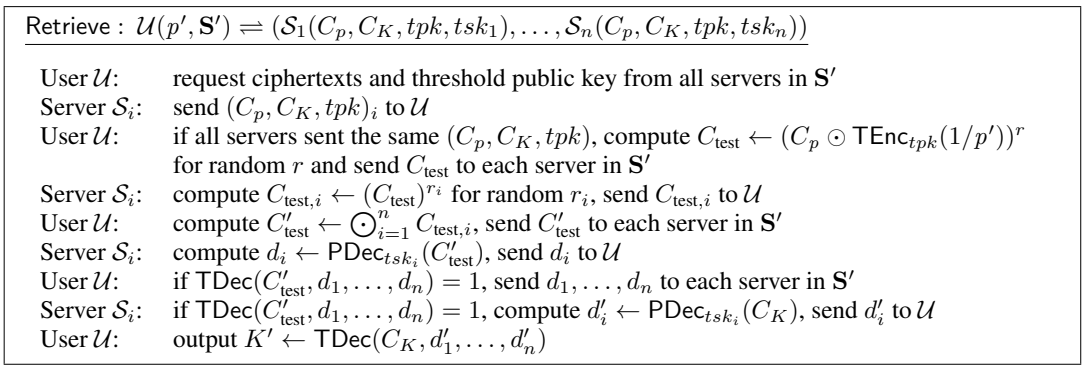


Figure 2: Construction outline of our retrieval protocol (setup idea as in Figure 1).

*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, the adversary gets one free guess at $p'$, but does not 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 is in the following section. First, for simulatability in the security proof, we need to be able to extract $p$, $p'$, and $K$ from the protocols. 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. Extractable values are encrypted under $PK$ with NIZK proofs 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, 7]. Thus, given the practical setting of our protocol, we prefer the more realistic assumptions over modeling stronger (but unrealistic) robustness properties.

## 4.1 Detailed Descriptions

In the following, 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 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 id 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. Finally, we implicitly assume that whenever a checks fails, the checking party will abort the protocol.

### 4.1.1 The Setup Protocol

We assume that the system parameters contain a group $\mathbb{G} = \langle g \rangle$ of prime order $q = \Theta(\tau)$ 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 $\mathsf{SKGen}$. 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 $\mathsf{KGen}$, 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 protocol participant (either the user or one of the servers), the participant aborts the protocol without output.

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

(a) Query $\mathcal{F}_{CRS}$ to obtain $PK$ and, for each $\mathcal{S}_i$ occurring in $\mathbf{S}$, query $\mathcal{F}_{CA}$ with $(\texttt{RETRIEVE}, sid, \mathcal{S}_i)$ 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 $tpk$ and $PK$, i.e., compute

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

(c) Generate a non-interactive zero-knowledge proof $\pi_0$ that the ciphertexts encrypt the same password and key, bound to $\texttt{context} = (sid, ssid, tpk, \mathbf{tpk}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K)$ where $\mathbf{tpk} = (tpk_1, \ldots, tpk_n)$:

$$\begin{aligned} \pi_0 \leftarrow \mathrm{NIZK}\{(p, K) : \quad & C_p = \mathsf{TEnc}_{tpk}(p) \ \wedge \quad C_K = \mathsf{TEnc}_{tpk}(K) \\ \wedge \quad & \widetilde{C}_p = \mathsf{Enc}_{PK}(p) \ \wedge \quad \widetilde{C}_K = \mathsf{Enc}_{PK}(K) \ \}\ (\ \texttt{context}\ ) \end{aligned}$$

(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 $\pi_0$ 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 the $sid$ 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 $\sigma_{2,i} \leftarrow \mathsf{Sign}_{ssk_i}((sid, note), \texttt{success})$ and send $\sigma_{2,i}$ to $\mathcal{U}$. Output $(\texttt{SETUP}, sid, ssid, \mathbf{S})$.

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

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

### 4.1.2 The Retrieve Protocol

The user $\mathcal{U}'$ on input $(\texttt{RETRIEVE}, sid, rsid, p')$ where $rsid = (rsid', \mathbf{S}')$ runs the following retrieve protocol with the list of $t+1$ servers specified in $\mathbf{S}'$. Whenever a check fails for a protocol participant, the participant 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 participant is a server, or with output $(\texttt{DELIVER2U}, sid, rsid, \perp)$ if it is the user. Further, whenever a participant 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 $\mathcal{F}_{CA}$ with $(\texttt{RETRIEVE}, sid, \mathcal{S}_i)$ 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 data 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 $\mathcal{F}_{CA}$ with $(\texttt{RETRIEVE}, sid, \mathcal{S}_j)$ for all $\mathcal{S}_j$ in $\mathbf{S}'$ to obtain the public keys $(epk_j, spk_j)$ of the other servers.

(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 $\pi_0$ 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 sent, i.e., $note_j = note$. Only proceed 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 local $note$. 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 for $\texttt{allow}$. Only proceed 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 $r \leftarrow_R \mathbb{Z}_q$ and compute $C_{\text{test}} \leftarrow (C_p \odot \mathsf{TEnc}_{tpk}(1/p'))^r$.

(c) Generate a proof that $C_{\text{test}}$ and $\widetilde{C}_{p'}$ are based on the same password attempt $p'$. To prevent man-in-the-middle attacks, the proof is also bound to $sid, rsid, note$, and the values provided by the user so far, i.e., $epk_U, C_{\text{test}}, \widetilde{C}_{p'}$: $\pi_1 \leftarrow \text{NIZK}\{(p', r) : C_{\text{test}} = (C_p \odot \mathsf{TEnc}_{tpk}(1/p'))^r \wedge \widetilde{C}_{p'} = \mathsf{Enc}_{PK}(p')\}(sid, rsid, note, epk_U, C_{\text{test}}, \widetilde{C}_{p'})$.

(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 $\pi_1$, and validate all signatures $\sigma_{5,j}$.

(b) Choose $r_i \leftarrow_R \mathbb{Z}_q$, 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 $C_{\text{test}}$, 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 the $t+1$ 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 $\sigma_{6,j}$. Compute $C'_{\text{test}} \leftarrow \bigodot_{S_j \in \mathbf{S}'} C'_{\text{test},j}$.

(b) Compute the decryption share of $C'_{\text{test}}$ as $(d_i, \pi_{d_i}) \leftarrow \mathsf{PDec}_{tsk_i}(C'_{\text{test}})$ and sign the share as $\sigma_{7,i} \leftarrow \mathsf{Sign}_{ssk_i}(rsid, C'_{\text{test}}, d_i)$.

(c) 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 $\pi_{d_i}$ 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 the $t+1$ tuples $(d_j, \pi_{d_j}, \sigma_{7,j})_{\mathcal{S}_j \in \mathbf{S}'}$, verify that all proofs $\pi_{d_j}$ and signatures $\sigma_{7,j}$ are valid w.r.t. the locally computed $C'_{\text{test}}$.

(b) Check whether $\mathsf{TDec}(C'_{\text{test}}, \{d_j\}_{\mathcal{S}_j \in \mathbf{S}'}) = 1$.

(c) Compute the verifiable decryption share for the data key 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 its 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 data key as $K' \leftarrow \mathsf{TDec}(C_K, \{d'_j\}_{\mathcal{S}_j \in \mathbf{S}'})$ 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 4.1 is given in Appendix F.

**Theorem 4.1** *If* $(\mathsf{TKGen}, \mathsf{TEnc}, \mathsf{PDec}, \mathsf{VfDec}, \mathsf{TDec})$ *is a* $(t, n)$*-semantically secure 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 encryption scheme [27] for $(\mathsf{TKGen}, \mathsf{TEnc}, \mathsf{PDec}, \mathsf{VfDec}, \mathsf{TDec})$ (as described in Section D) and $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$, ElGamal with Fujisaki-Okamoto padding [30] for $(\mathsf{KGen2}, \mathsf{Enc2}, \mathsf{Dec2})$, Schnorr signatures [46, 42] for $(\mathsf{SKGen}, \mathsf{Sign}, \mathsf{Vf})$, and the $\Sigma$-protocols of Section E, 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 $7t + 28$. (This includes also the procedure to establish the sub-session identifier $ssid, rsid$, where each party simply sends a random nonce and the concatenation of all inputs is used as unique identifier [4].)

# 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, T. Ristenpart, S. Tessaro. Multi-instance security and its application to password-based cryptography. *CRYPTO 2012*.

[6] M. Bellare, P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. *ACM CCS 1993*.

[7] M. Ben-Or, R. Canetti, O. Goldreich. Asynchronous secure computation. *STOC 1993*.

[8] X. Boyen. Hidden credential retrieval from a reusable password. *ASIACCS 2009*

[9] 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*.

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

[11] J. Camenisch, A. Kiayias, M. Yung. On the portability of generalized Schnorr proofs. *EUROCRYPT 2009*.

[12] J. Camenisch, S. Krenn, V. Shoup. A framework for practical universally composable zero-knowledge protocols. *ASIACRYPT 2011*.

[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. *9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (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*.

# A Ideal Functionality

We now give the full description of our ideal functionality as described in Section 2. In Figures 3 and 4, depicting the Setup and Retrieve related interfaces respectively, we make the following assumptions to save on notation which would only distract from the main ideas:
(i) For each combination of message type (e.g., SETUP, JOIN, etc.), party ($\mathcal{U}, \mathcal{U}'$, $\mathcal{S}_i$ or $\mathcal{A}$) and sub-session identifier ($ssid, rsid$), the ideal functionality considers only the first incoming message. Subsequent messages in the same sub-session for the same interface and party will be ignored.
(ii) For all incoming messages except SETUP and RETRIEVE messages, the functionality always checks if for a message with sub-session identifier $ssid$ (respectively, $rsid$) a record $s$ exists with $s.ssid$ (respectively, a record $r$ with $r.rsid$). If so, it retrieves that record and uses the information stored therein for the handling of the message. If no such record exists, $\mathcal{F}$ ignores the message.
(iii) When receiving messages that include the identity of an *honest* server $\mathcal{S}_i$ or is coming from an honest server $\mathcal{S}_i$ for a sub-session $ssid$ or $rsid$, $\mathcal{F}$ only continues if $\mathcal{S}_i$ is supposed to participate in that sub-session, i.e., if $ssid = (ssid', \mathbf{S})$ then it must hold that $\mathcal{S}_i \in \mathbf{S}$, and similarly, if $rsid = (rsid', \mathbf{S}')$ then $\mathcal{S}_i$ must appear in $\mathbf{S}'$.

1. Upon input $(\texttt{SETUP}, sid, ssid, p, K)$ from a user $\mathcal{U}$, where the setup sub-session identifier $ssid = (ssid', \mathbf{S})$ contains a list of $n$ different server identities $\mathbf{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$:
   - If $\mathcal{F}$ does not have a `setup` record associated with $ssid$ yet, it creates such a record $s$ in which it stores $\mathcal{U}$, $p$ and $K$, and sets the response recipient to $s.\mathcal{R} \leftarrow \mathcal{U}$.
   - If at least $t+1$ of the servers in $\mathbf{S}$ are corrupt, $\mathcal{F}$ sends $(\texttt{SETUP}, sid, ssid, \mathcal{U}, p, K)$ to $\mathcal{A}$; otherwise, it sends $(\texttt{SETUP}, sid, ssid, \mathcal{U})$ to $\mathcal{A}$.
2. Upon receiving $(\texttt{JOIN}, sid, ssid, \mathcal{S}_i)$ from the adversary $\mathcal{A}$ for a server identity $\mathcal{S}_i$:
   - If $\mathcal{S}_i$ is honest and has not joined another setup sub-session for the same username $uid = sid$, then $\mathcal{F}$ outputs $(\texttt{SETUP}, sid, ssid)$ to $\mathcal{S}_i$ and sends a public delayed output $(\texttt{SETUP}, sid, ssid, \mathcal{S}_i)$ to $s.\mathcal{R}$.
   - If $\mathcal{S}_i$ is corrupt, $\mathcal{F}$ outputs $(\texttt{SETUP}, sid, ssid, \mathcal{S}_i)$ to $s.\mathcal{U}$.
3. Upon receiving $(\texttt{STEAL}, sid, ssid, \hat{p}, \hat{K})$ from the adversary $\mathcal{A}$:
   - If no honest server has joined the setup session yet, $\mathcal{F}$ replaces the original input for $ssid$ by setting $s.p \leftarrow \hat{p}, s.K \leftarrow \hat{K}$ and $s.\mathcal{R} \leftarrow \mathcal{A}$.

Figure 3: The Setup-related Interfaces of our Single-Session Functionality $\mathcal{F}$ with $sid = uid$

# B Tampering Attacks on Trusted-Memory Protocols

We now briefly discuss why existing password-based secret sharing protocols [47, 13, 40, 3] are not secure according to the notion just presented. For the protocols by Szydlo-Kaliski [47] and Camenisch et al. [13] it is easy to see that the password is plainly revealed to the adversary, when the retrieval is performed with malicious servers. To learn the password $p'$ in the protocols by Bagherzandi et al. [3] and MacKenzie et al. [40] we can mount the following attacks:

In the TPAKE protocol of MacKenzie et al. [40], the client stores a public key $y = g^x$ in its trusted memory, of which the corresponding secret key $x$ is shared among the servers, as well as an ElGamal ciphertext $E_C = (y^\alpha g^{p^{-1}}, g^\alpha)$, where $p$ is the user's password. During the authentication protocol with password attempt $p'$, the user sends a ciphertext $B = (y^\beta, g^\beta) \times (E_C)^{p'} \times (g^{-1}, 1)$ to all servers. If the adversary replaces the values $y$ and $E_C$ in the client's memory with $y' = g^{x'}$ and $E_C = (g, 1)$ for a known $x'$, respectively, then we have that $B = (g^{x'\beta + p' - 1}, g^\beta)$, from which the adversary can compute $g^{p'}$ and perform an offline attack on $p'$.

Similarly, the TPASS protocol of Bagherzandi et al. [3] lets the client store a public key $y$ in trusted memory, and during authentication sends an ElGamal ciphertext $(c_{p'}, d_{p'}) = (g^r, y^r h^{p'})$ to all servers. If an adversary replaces $y$ with $y' = g^{x'}$ for a known $x'$, then it can use $(c_{p'}, d_{p'})$ to mount an offline attack on $p'$.

4. Upon input $(\texttt{RETRIEVE}, sid, rsid, p')$ from a user $\mathcal{U}'$ where the sub-session identifier $rsid = (rsid', \mathbf{S}')$ contains a list of $t+1$ different server identities $\mathbf{S}'$:
   - If $\mathcal{F}$ does not have a `retrieve` record associated with $rsid$ yet, it creates such a record $r$ in which it stores $p'$ and initially sets $r.K \leftarrow \perp$, $r.\mathcal{R} \leftarrow \mathcal{U}'$, and $r.ssid \leftarrow \perp$.
   - If there is at least one honest server in $\mathbf{S}'$, $\mathcal{F}$ will link the retrieval to the correct setup record. That is, $\mathcal{F}$ checks if a setup session $ssid = (ssid', \mathbf{S})$ exists with $\mathbf{S}' \subseteq \mathbf{S}$ and that all honest servers in $\mathbf{S}'$ have joined. If so, $\mathcal{F}$ links this retrieval to that setup by setting $r.ssid \leftarrow ssid$.
   - $\mathcal{F}$ sends $(\texttt{RETRIEVE}, sid, rsid, \mathcal{U}')$ to $\mathcal{A}$ and if $r.ssid \neq \perp$ it further sends a public delayed output $(\texttt{RETRIEVE}, sid, rsid)$ to each honest server in $\mathbf{S}'$.
5. Upon receiving $(\texttt{PLANT}, sid, rsid, p^*, K^*)$ from the adversary $\mathcal{A}$:
   - If all $t+1$ servers in $\mathbf{S}'$ are corrupt, then $\mathcal{F}$ sends $(\texttt{NOTIFY}, sid, rsid, c)$ to $\mathcal{A}$, where $c \leftarrow \texttt{correct}$ if $r.p' = p^*$ and $c \leftarrow \texttt{wrong}$ otherwise. If $c = \texttt{correct}$, $\mathcal{F}$ sets $r.K \leftarrow K^*$.
6. Upon receiving $(\texttt{STEAL}, sid, rsid, \hat{p})$ from the adversary $\mathcal{A}$:
   - If no message $(\texttt{RETRIEVE}, sid, rsid)$ was delivered to any honest server yet, $\mathcal{F}$ sends the public delayed output $(\texttt{DELIVER2U}, sid, rsid, \perp)$ to $\mathcal{U}'$ and sets $r.p' \leftarrow \hat{p}$ and $r.\mathcal{R} \leftarrow \mathcal{A}$.
7. Upon input $(\texttt{PROCEED}, sid, rsid, a)$ from an honest server $\mathcal{S}_i$ where $a \in \{\texttt{allow}, \texttt{deny}\}$:
   - $\mathcal{F}$ notifies the adversary by sending $(\texttt{PROCEED}, sid, rsid, \mathcal{S}_i, a)$ to $\mathcal{A}$.
   - If *all* honest servers in $\mathbf{S}'$ have agreed to proceed, $\mathcal{F}$ obtains the setup record $s$ for $r.ssid$ and sends $(\texttt{NOTIFY}, sid, rsid, c)$ to $\mathcal{A}$ where $c \leftarrow \texttt{correct}$ if $r.p' = s.p$ and $c \leftarrow \texttt{wrong}$ otherwise. If $c = \texttt{correct}$, $\mathcal{F}$ sets $r.K \leftarrow s.K$.
8. Upon input $(\texttt{DELIVER}, sid, rsid, \mathcal{P}, a)$ from the adversary, where $\mathcal{P}$ is either an honest server $\mathcal{S}_i$, a user $\mathcal{U}'$, or the adversary $\mathcal{A}$, and where $a \in \{\texttt{allow}, \texttt{deny}\}$:
   - If $a = \texttt{allow}$ and $r.K \neq \perp$, then $\mathcal{F}$ outputs $(\texttt{DELIVER2S}, sid, rsid, \texttt{success})$ to $\mathcal{P}$ if $\mathcal{P} = \mathcal{S}_i$, or outputs $(\texttt{DELIVER2U}, sid, rsid, r.K)$ to $r.\mathcal{R}$ if $\mathcal{P} = r.\mathcal{R}$.
   - Otherwise, $\mathcal{F}$ outputs $(\texttt{DELIVER2S}, sid, rsid, \texttt{fail})$ to $\mathcal{P}$ if $\mathcal{P} = \mathcal{S}_i$ or outputs $(\texttt{DELIVER2U}, sid, rsid, \perp)$ to $r.\mathcal{R}$ if $\mathcal{P} = r.\mathcal{R}$.

Figure 4: The Retrieve−related Interfaces of our Single-Session Functionality $\mathcal{F}$ with $sid = uid$

# C On Generic Constructions from Multi-Party Computation

Constructing a TPASS scheme a from multi-party computation (MPC) protocol is actually not quite as simple as one might think. One can easily design an arithmetic circuit that tests password correctness: for example, $t = (p - p') \cdot r$, where $r$ is a jointly generated random value, is zero if $p = p'$ and is random otherwise. Evaluating this circuit in a way that meets our stringent security and memory restrictions is not so straightforward, however. First, the ideal functionality for arithmetic multi-party computation [23] and the state-of-the-art protocols [23, 24, 22] are all for $n$ participants up to $n-1$ of whom can be corrupted, i.e., the threshold $t$ is fixed to $n-1$. They also require that *all* protocol participants be online during the circuit evaluation. The user machine $\mathcal{U}$ that performs setup and would provide the input $p$ to the circuit is not necessarily online during retrieval, however. Alternatively, one could let $\mathcal{U}$ secret-share $p$ to all servers and, at retrieval, let each server provide his share as input and let the user $\mathcal{U}'$ provide $p'$. To satisfy our security notion, the secret shares distributed by $\mathcal{U}$ at setup must be extractable by the simulator, however, for example by encrypting them under a public key in a CRS and giving all ciphertexts to all servers. Moreover, the circuit must use these ciphertexts to check that the shares used as input by the servers are the same as those given to them during setup, so the circuit must encode public-key operations. Finally, the state-of-the-art protocols mentioned above all require an initialization step during which all participants jointly generate a shared encryption key pair. Such protocols are very inefficient, e.g., even if we compromise on security and tolerate a covert adversary with $2^{-20}$ cheating probability, it takes 99.4 seconds on a 2.80 GHz Intel Core i7 for the minimum of three participants (two servers and the user) [22]. Since the user $\mathcal{U}'$ cannot trust the key material that is given to him by the servers, and since the password attempt $p'$ will be encrypted under these keys, this step must be performed anew at *each* retrieval with the participation of $\mathcal{U}'$, making the protocol hopelessly inefficient.

# D Threshold cryptosystem

Below we provide the details of our definition for semantically secure threshold homomorphic encryption, which is an adaption of the definition of Cramer, Damgård and Nielsen [20]. The differences between our definition and that of CDN are as follows: (1) to simplify exposition, our definition is less general when it comes to access structures, since we only concern ourselves with the *threshold* access structure; (2) not only do we require that there is a secure protocol for threshold decryption, but we also require that it has a certain structure: each server has a public share of the public key that corresponds to its private share of the secret key, and outputs a share of the decryption and a proof that it computed it correctly corresponding to its public share; (3) the homomorphic properties we need are only a subset of those that CDN need.

Let $(\mathsf{TKGen}, \mathsf{TEnc}, \mathsf{PDec}, \mathsf{VfDec}, \mathsf{TDec})$ be a theshold encryption scheme with algorithms as defined in Section 3. These algorithms constitute a semantically secure threshold homomorphic cryptosystem if the following properties hold:

**Correctness:** For any message $m$, and key tuple $(tpk, \{(tpk_i, tsk_i)\})$ generated through $\mathsf{TKGen}$, for all $C \in \mathsf{TEnc}_{tpk}(m)$, any $1 \leq i \leq n$, any $(d_i, \pi_{d_i}) \in \mathsf{PDec}_{tsk_i}(C)$, $\mathsf{VfDec}_{tpk_i}(C, d_i, \pi_{d_i})$ accepts; moreover, for any $\{i_1, \ldots, i_{t+1}\} \subset [n]$, $\mathsf{TDec}(C, d_{i_1}, \ldots, d_{i_{t+1}})$ outputs $m$.

**Soundness/robustness:** For all PPT adversaries, for all $(tpk, \{tpk_i\})$ generated through $\mathsf{TKGen}$, for all $C \in \mathcal{C}$, for all $i$ there exists a unique $d_i$ such that the adversary can compute $\pi_{d_i}$ that will be accepted by $\mathsf{VfDec}$ with non-negligible probability (over the choice of the system parameters/random oracle); moreover, for these unique $(d_1, \ldots, d_n)$, for all subsets $\{i_1, \ldots, i_{t+1}\} \subset [n]$, $\mathsf{TDec}(C, d_{i_1}, \ldots, d_{i_{t+1}})$ outputs the same value.

**Threshold semantic security:** The cryptosystem is semantically secure even when the adversary has $t$ of the partial secret keys; formally, it consists of two properties: (1) indistinguishability: for any subset $\{i_1, \ldots, i_t\} \subset [n]$, for any PPT adversary, that, on input $(tpk, \{tpk_i\})$, and $t$ secret shares $(tsk_{i_1}, \ldots, tsk_{i_t})$, selects a pair of messages $m_0$ and $m_1$, the ciphertexts $C_0$ and $C_1$ are indistinguishable, where $C_b \leftarrow \mathsf{TEnc}_{tpk}(m_b)$. (2) simulatability: there exists an efficient simulator $S$ that, on input the public values $(tpk, \{tpk_i\})$, $t$ secret shares $(tsk_{i_1}, \ldots, tsk_{i_t})$, a ciphertext $C$ and a message $m$ outputs $(d_j, \pi_{d_j})$ for all $j \notin \{i_1, \ldots, i_t\}$ such that, if $m$ is the correct decryption of $C$ then the simulator's output is indistinguishable from the correct values $(d_j, \pi_{d_j})$.

**Homomorphism:** The message space is the multiplicative group $\mathbb{G}$. Moreover, 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 often omit the operation and just talk about multiplying messages and ciphertexts.)

**Lemma D.1** *Let* $(\mathsf{TKGen}, \mathsf{TEnc}, \mathsf{PDec}, \mathsf{VfDec}, \mathsf{TDec})$ *be a semantically secure threshold homomorphic cryptosystem, and let* $S$ *be its simulator. Then for all* $n$*,* $t$*, stateful PPT adversaries* $\mathcal{A}$*, for all subsets* $I = \{i_1, \ldots, i_t\} \subset [n]$*,* $E^{\mathcal{A}}_{0,I} \equiv E^{\mathcal{A}}_{1,I}$ *where the experiment* $E^{\mathcal{A}}_{b,I}(1^\tau)$ *is defined as follows:*

- *The encryption key* $tpk$ *and the partial keys pairs* $((tpk_1, tsk_1, ) \ldots, (tpk_n, tsk_n))$ *are generated by* $\mathsf{TKGen}$*.*
- *The adversary* $\mathcal{A}$ *is given all the public keys, as well as the secret keys for subset* $I$*; the adversary* $\mathcal{A}$ *selects a message* $m \in M$*.*
- *Let* $m_0 = m$*,* $m_1 = 1$ *(i.e.,* $m_1$ *is the identity element of the message space* $M$*). Generate ciphertext* $C \leftarrow \mathsf{TEnc}_{tpk}(m_b)$*.*
- *The simulator* $S((tpk, \{tpk_i\}), (tsk_{i_1}, \ldots, tsk_{i_t}), C, m)$ *computes* $(d_j, \pi_{d_j})$ *(i.e. pretends that* $C$ *decrypts to* $m$*).*
- $\mathcal{A}$ *receives the values* $(d_j, \pi_{d_j})$ *and outputs its view.*

**Proof.** Suppose that a PPT adversary $\mathcal{A}$ exists such that the two views are distinguishable by a distinguisher $D$. Then consider a PPT adversary $\mathcal{B}$ that breaks semantic security as follows: on input $(tpk, \{tpk_i\})$, and $t$ secret shares $(tsk_{i_1}, \ldots, tsk_{i_t})$, $\mathcal{B}$ runs $\mathcal{A}$ and gets a message $m$. $\mathcal{B}$ sets $m_0 = m$, $m_1 = 1$ and gives $(m_0, m_1)$ to its challenger. It then receives the challenge ciphertext $C$, and runs the simulator $S((tpk, \{tpk_i\}), (tsk_{i_1}, \ldots, tsk_{i_t}), C, m)$ to compute $(d_j, \pi_{d_j})$. It sends $(d_j, \pi_{d_j})$ to $\mathcal{A}$ and, once $\mathcal{A}$ outputs its view, feeds this view to $D$, and outputs whatever $D$ outputs. Note that all the inputs to $\mathcal{A}$ are distributed as in $E^{\mathcal{A}}_{0,I}$ if $C$ is an encryption of $m_0$, and as in $E^{\mathcal{A}}_{1,I}$ if $C$ is an encryption of $m_1$; therefore, the lemma follows. ■

## D.1 Construction

The following construction is based on the ElGamal cryptosystem [27]. It has appeared in the literature before [25], but not with the same definition, so we reproduce it here for completeness.

**Message space and key generation:** The message space is a group $\mathbb{G} = \langle g \rangle$ of prime order $q = \Theta(\tau)$ in which the decisional Diffie-Hellman problem is hard. The key generation algorithm $\mathsf{TKGen}$ takes $(\mathbb{G}, g, q, t, n)$ as input. It picks $t+1$ coefficients $(a_0, \ldots, a_t)$ at random from $\mathbb{Z}_q$; let $p(x) = \sum_{k=0}^{t} a_k x^k \bmod q$ be the polynomial defined by these coefficients. Then $tsk_i = p(i)$, $tpk_i = g^{tsk_i}$, and $tpk = g^{a_0} = g^{p(0)} = tpk_0$.

**Encryption** This is the standard ElGamal encryption with public key $tpk$: to encrypt a message $m \in \mathbb{G}$, pick a random $r \in \mathbb{Z}_q$, and output $C = \mathsf{TEnc}_{tpk}(m) = (g^r, tpk^r m)$.

**Partial decryption:** $\mathsf{PDec}$ on input $C = (U, V)$ and partial secret key $tsk_j$ outputs the partial decryption value $d_j = U^{tsk_j}$ and a robust NIZK proof $\pi_{d_j}$ that $d_j$ is correct (the NIZK is done in the RO model, using the Fiat-Shamir heuristic applied to the Schnorr-Pedersen $\Sigma$-protocol for proving knowledge and equality of discrete logarithms, see Section E.4).

**Verification of partial decryption:** The algorithm $\mathsf{VfDec}$ on input $(tpk_j, C, \pi_{d_j}, d_j)$ verifies the proof $\pi_{d_j}$ w.r.t. $tpk_j, C$ and $d_j$.

**Decryption:** The algorithm $\mathsf{TDec}$ takes as input the ciphertext $C = (U, V)$ and $t+1$ partial decryptions corresponding to some subset $I = \{i_1, \ldots, i_{t+1}\}$ of the servers, $d_{i_k} = U^{tsk_{i_k}} = U^{p(i_k)}$. Note that, by standard polynomial interpolation techniques, if $I = \{i_0, i_1, \ldots, i_t\} \subseteq [n]$ is of size $t+1$, then $tsk_j = p(j)$ can be expressed as a linear function $f_{j,I}$ of $p(i_0), p(i_1), \ldots, p(i_t)$: $f_{j,I} = \sum_{k=0}^{t} u_k p(i_k) \bmod q$. We can use *polynomial interpolation in the exponent* to compute $U^{p(0)}$ as follows: express the value $p(0)$ as a linear combination of $p(i_1), \ldots, p(i_{t+1})$, say $p(0) = f_{0,I} = \sum_{k=1}^{t+1} u_k p(i_{t+1})$. Then let $W = \prod_{k=1}^{t+1} d_{i_k}^{u_k}$, and output $m = V/W$.

**Theorem D.2** *The construction above constitutes a semantically secure threshold homomorphic cryptosystem under the decisional Diffie-Hellman assumption in the random-oracle model.*

**Proof.** We give a proof sketch. To verify correctness, observe that $W = \prod_{k=1}^{t+1} d_{i_k}^{u_k} = \prod_{k=1}^{t+1} (U^{p(i_k)})^{u_k} = U^{\sum_{k=1}^{t+1} p(i_k) u_k = U^{p(0)}} = g^{rp(0)} = (g^{p(0)})^r = tpk^r$. Soundness/robustness follows because $t+1$ points uniquely define a polynomial of degree $t$.

The indistinguishability part of threshold semantic security follows by a reduction from the security of the standard ElGamal cryptosystem, as follows: let $\mathcal{A}$ be an adversary that breaks semantic security for subset $I$. Let us describe our reduction $\mathcal{B}$: on input the standard ElGamal public key $h = g^x$ (where $x$ is the secret key), $\mathcal{B}$ picks $tsk_{i_1}, \ldots, tsk_{i_k}$ at random from $\mathbb{Z}_q$, and sets $tpk = tpk_0 = h$, and $tpk_{i_k} = g^{tsk_{i_k}}$ for $1 \le k \le t$. For $j \notin I$, $\mathcal{B}$ computes $tpk_j$ by using polynomial interpolation $f_{j,I \cup \{0\}}$ in the exponent. Now $\mathcal{B}$ can run $\mathcal{A}$ and obtain the two messages $m_0$ and $m_1$; it forwards them to its own challenger and obtains a challenge ciphertext $C$ which it proceeds to forward to $\mathcal{A}$. It then outputs whatever $\mathcal{A}$ outputs. Note that, whenever $C$ is an encryption of $m_b$, $\mathcal{A}$ receives the view that is distributed as if its challenger picked $m_b$; therefore, $\mathcal{B}$ is correct whenever $\mathcal{A}$ is, and indistinguishability follows.

For the simulatability part of threshold semantic security, consider the following construction for the simulator: on input $(tpk, tpk_1, \ldots, tpk_n)$ and the values $(tsk_{i_1}, \ldots, tsk_{i_t})$, a ciphertext $C = (U, V)$ and a message $m$, the simulator computes $U^{p(0)} = W = V/m$, partial decryptions $U^{p(i_k)} = d_{i_k} = U^{tsk_{i_k}}$, and for $j \notin I$, solves for the values $U^{p(j)} = d_j$ using polynomial interpolation $f_{j,I \cup \{0\}}$ in the exponent. It computes the proofs $\pi_j$ using the zero-knowledge simulator. Note that, if the ZK simulator is perfect (which is true for these types of RO-based simulators), then the values output by the simulator is distributed identically to what honest servers output.

Finally, our threshold version of the ElGamal cryptosystem is homomorphic, just as the regular ElGamal cryptosystem is: if $C_1 = (U_1, V_1) = (g^{r_1}, tpk^{r_1} m_1) \in \mathsf{TEnc}_{tpk}(m_1)$ and $C_2 = (U_2, V_2) = (g^{r_2}, tpk^{r_2} m_2) \in \mathsf{TEnc}_{tpk}(m_2)$, then $C_1 \odot C_2 = (U_1 U_2, V_1 V_2) = (g^{r_1 + r_2}, tpk^{r_1 + r_2} m_1 m_2) \in \mathsf{TEnc}_{tpk}(m_1 \cdot m_2)$. ■

# E Concrete Instantiations of Proof Protocols

We now describe how to instantiate the different proofs used in our protocol. For a concrete instantiation of the threshold encryption scheme we refer to Appendix D.

Assume that the ElGamal encryption scheme is used for Enc and the threshold ElGamal encryption scheme is used for TEnc. Also assume the system parameters contain a group $\mathbb{G} = \langle g \rangle$ of prime order $q = \Theta(\tau)$ that is used by both these encryption schemes, i.e., let $\mathcal{F}_{\text{CRS}}$ return an element $PK \in \mathbb{G}$ and let $(\mathbb{G}, g, q)$ be the input TKGen so that the values $tpk, tpk_i$ output by TKGen are elements of $G$. In this discrete-logarithm-based setting, the various NIZK's used in our protocol can be instantiated using so-called generalized Schnorr protocols [46, 11].

When referring to such proof protocols, we use the following notation [15, 11]. For instance, $SPK\{(a, b, c) : y = g^a h^b \wedge \tilde{y} = g^a h^c\}(m)$ denotes a "*Signature based on a zero-knowledge Proof of Knowledge of integers $a$, $b$, $c$ such that $y = g^a h^b$ and $\tilde{y} = g^a h^c$holds*," where $y, g, h$, and $\tilde{y}$ are elements of $\mathbb{G}$ and where $m$ is included into the hash that is used to make the proof of knowledge protocol non-interactive (Fiat-Shamir transformation). The convention is that the letters in the parenthesis $(a, b, c)$ denote quantities of which knowledge is being proven, while all other values are known to the verifier.

Given a protocol in this notation, it is straightforward to derive an actual protocol implementing the proof. Indeed, the computational complexities of the proof protocol can be easily derived from this notation: basically for each term $y = g^a h^b$, the prover and the verifier have to perform an equivalent computation, and to transmit one group element and one response value for each exponent. We refer to, e.g., Camenisch, Kiayias, and Yung [11] for details on this.

The most efficient way to make these protocol concurrent zero-knowledge and simulation-sound is by the Fiat-Shamir transformation [28]. In this case, we will have to resort to the random-oracle model [6] for the security proof. To make the resulting non-interactive proofs simulation-sound, it suffices to let the prover include context information as an argument to the random oracle in the Fiat-Shamir transformation, such as the system parameters, $uid$, $rsid$, and the protocol step in which the statement is being proven, and a collision-resistant hash of the communication transcript that the prover and verifier have engaged in so far, so that the proof is resistant to a man-in-the-middle attack. In particular, notice that all the statements we require the parties to prove to each other are proofs of membership (i.e., that some computation was done correctly) and *not* proofs of knowledge. Therefore, it is not necessary that the prover can be re-wound to extract the witnesses.

We note, however, that there are alternative methods one could employ instead to make $\Sigma$-protocols non-interactive that do not rely on the random oracle model (e.g., [41, 31, 12]). Unfortunately, these methods come with some performance penalty.

We now provide the concrete instantiations of all NIZK that our scheme requires.

## E.1 Realization of NIZK $\pi_0$

The first proof

$$\pi_0 \leftarrow \text{NIZK}\{(p, K) : C_p = \mathsf{TEnc}_{tpk}(p) \wedge C_K = \mathsf{TEnc}_{tpk}(K) \wedge \widetilde{C}_p = \mathsf{Enc}_{PK}(p) \wedge \widetilde{C}_K = \mathsf{Enc}_{PK}(K)\}(\text{ context }) \quad \text{with context = } (uid, ssid, tpk, \mathbf{tpk}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K).$$

is used in Step S1(c) of the Setup protocol, where the user needs to prove that the same $p$ and $K$ are encrypted under both $PK$ and $tpk$. Let $C_p = \mathsf{TEnc}_{tpk}(p) = (c_{11}, c_{12}) \leftarrow (g^{r_1}, tpk^{r_1} p)$, $C_K = \mathsf{TEnc}_{tpk}(K) = (c_{21}, c_{22}) \leftarrow (g^{r_2}, tpk^{r_2} K)$, $\widetilde{C}_p = \mathsf{Enc}_{PK}(p) = (\tilde{c}_{11}, \tilde{c}_{12}) \leftarrow (g^{r_3}, PK^{r_3} p)$, and $\widetilde{C}_K = \mathsf{Enc}_{PK}(K) = (\tilde{c}_{21}, \tilde{c}_{22}) \leftarrow (g^{r_4}, PK^{r_4} K)$. Then $\pi_0$ can be realized as follows

$$\pi_0 \leftarrow \text{SPK}\{(r_1, r_2, r_3, r_4) : c_{11} = g^{r_1} \wedge c_{21} = g^{r_2} \wedge \tilde{c}_{11} = g^{r_3} \wedge \tilde{c}_{21} = g^{r_4} \wedge c_{12}/\tilde{c}_{12} = tpk^{r_1} PK^{-r_3} \wedge c_{22}/\tilde{c}_{22} = tpk^{r_2} PK^{-r_4}\}(\text{ context }).$$

Considering the decryption algorithms, it is not hard to see that this protocol indeed proves that both $C_p$ and $\widetilde{C}_p$ will decrypt to the same message as $c_{12}/tpk^{r_1} = \tilde{c}_{12}/PK^{r_3}$ holds (and similarly for $C_K$ and $\widetilde{C}_K$).

Note: the efficiency of the protocol can be improved by using the same randomness $r$ for all four ciphertexts and only have the same element $g^r$ in all ciphertexts.

### E.2 Realization of NIZK $\pi_1$

The next proof is used in Step R5(c) of the retrieve protocol:

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

Here the user proves that $C_{\text{test}}$ and $\widetilde{C}_{p'}$ are based on the same password attempt $p'$. More precisely, the user proves that the plaintext underlying $\widetilde{C}_{p'}$ is also used in the computation of $C_{\text{test}}$ from $C_p$. Let $C_p = (c_{11}, c_{12})$ and $C_{\text{test}} = (c'_{11}, c'_{12}) = (C_p \odot \mathsf{TEnc}_{tpk}(1/p'))^r \leftarrow ((c_{11}/g^{r'_1})^r, (c_{12}/(tpk^{r'_1}p'))^r)$ with $(r, r'_1) \leftarrow_{\text{R}} \mathbb{Z}_q^2$. Recall that $\widetilde{C}_{p'} = \mathsf{Enc}_{PK}(p') = (\tilde{c}'_{11}, \tilde{c}'_{12}) = (g^{\tilde{r}_1}, PK^{\tilde{r}_1}p')$. Then this proof can be implemented as

$$\pi_1 \leftarrow \text{SPK}\{(r, \tilde{r}_1) : \tilde{c}'_{11} = g^{\tilde{r}_1} \ \wedge \ c_{11} = {c'_{11}}^{1/r} g^{-r'_1} \wedge \\ c_{12}\tilde{c}'_{12} = {c'_{12}}^{1/r} tpk^{-r'_1} PK^{\tilde{r}_1}\}(sid, rsid, note, epk_U, C_{\text{test}}, \widetilde{C}_{p'}).$$

This proof is somewhat more involved. By definition $\tilde{c}'_{12}/PK^{\tilde{r}_1}$ is the decryption of $\widetilde{C}_{p'}$. Rewriting the last equation in the proof we get ${c'_{12}}^{1/r} = c_{12} tpk^{r'_1}(\tilde{c}'_{12}/PK^{\tilde{r}_1})$ and, together with ${c'_{11}}^{1/r} = c_{11} g^{r'_1}$, the statement we wanted to show follows.

### E.3 Realization of NIZK $\pi_{2,i}$

The NIZK $\pi_{2,i} \leftarrow \text{NIZK}\{(r_i) \ : \ C'_{\text{test},i} = (C_{\text{test}})^{r_i}\}$ is used in Step R6(b) where each server $\mathcal{S}_i$ proves that the new ciphertext $C'_{\text{test},i}$ is a correct randomization of the original ciphertext $C_{\text{test}}$. While at an abstract level, this proof looks like a proof of knowledge it is in fact a proof of correctness as we shall see from the instantiation below. Let $C_{\text{test}} = (c'_{11}, c'_{12})$ and $C'_{\text{test},i} = (c'_{11,i}, c'_{12,i}) \leftarrow ({c'_{11}}^{r_i}, {c'_{12}}^{r_i})$ and thus $\pi_{2,i} \leftarrow \text{SPK}\{(r_i) \ : \ c'_{11,i} = {c'_{11}}^{r_i} \ \wedge \ c'_{12,i} = {c'_{12}}^{r_i})\}$. This proof is rather straightforward – it just shows that the two ciphertext components of $C'_{\text{test},i}$ we obtained from the two components of $C_{\text{test}}$ are raised to the same value.

### E.4 Realization of NIZK's $\pi_{d_i}$ and $\pi_{d'_i}$

The last NIZK's $\pi_{d_i}$ and $\pi_{d'_i}$ appear in Step R8 and R9, respectively, where each server has to prove that the decryption share sent was computed correctly. This proof depends on the threshold encryption scheme used and in our case is as follows $\pi_{d_i} \leftarrow \text{SPK}\{(tsk_i) : d_i = {c''_{11}}^{tsk_i} \ \wedge \ tpk_i = g^{tsk_i}\}$, where $C'_{\text{test}} = (c''_{11}, c''_{12})$. It proves that the server $\mathcal{S}_i$ indeed computed $d_i$ by raising $c''_{11}$ to its secret key. The proof $\pi_{d'_i}$ can be done analogously using $C_K$ instead of $C'_{\text{test}}$.

## F Proof of Theorem 4.1

We now prove that our protocol presented in Section 4 indeed securely implements our ideal functionality described in Section 2.

**Setup Assumptions** Our protocol relies on two underlying ideal functionalities. The first is $\mathcal{F}_{CRS}$, which models the availability of a public common reference string. The second is the certification functionality $\mathcal{F}_{CA}$ by Canetti [17], which models the existence of some form of a public-key infrastructure, i.e., servers can register their public keys and the user can look up these public keys on input a server identifier. We refer to [17] for the detailed description of the $\mathcal{F}_{CA}$ functionality. As our protocol relies on those functionalities, we have designed it in a hybrid world where parties can make calls to $\mathcal{F}_{CRS}$ and $\mathcal{F}_{CA}$.

### F.1 Sequence of Games

Our proof consist of a sequence of games that a challenger runs with the real-world adversary. Therein, we stepwise change parts of the simulation, mainly replacing protocol messages either by "dummy" messages that do not depend on $p$, $K$ and $p'$ anymore but that are indistinguishable from the real ones, or we derive them based only on knowing

whether or not the passwords $p$ and $p'$ match. In our final game we then make the transition to let the challenger run internally the ideal functionality $\mathcal{F}$ and simulate all messages based merely on the information he can obtain from $\mathcal{F}$.

We will now describe each game $i$ and why we have $\text{GAME}_i \approx \text{GAME}_{i-1}$ for each transition, meaning that the environment can not distinguish between $\text{GAME}_i$ and $\text{GAME}_{i-1}$. In our series of games, the challenger plays the role of all honest parties, obtaining their inputs from and passing their outputs to the environment.

**GAME 1** The challenger simply executes the real protocol for all honest players, thereby giving the environment the same view as in the real world.

**GAME 2** In this game we abort whenever the challenger sees a valid signature $\sigma_i$ under the public key $spk_i$ of an honest server $\mathcal{S}_i$ on a message that was never signed by $\mathcal{S}_i$. Clearly, we have $\text{GAME}_1 \approx \text{GAME}_2$ by the existential unforgeability of the signature scheme $(\mathsf{SKGen}, \mathsf{Sign}, \mathsf{Vf})$.

In particular, this means that whenever an honest server proceeds beyond Step (S.4) in a setup protocol execution or beyond Step (R.4) in a retrieve execution, then all other honest servers in $\mathbf{S}$ or $\mathbf{S}'$ agree on the same $note$ and, in the case of a retrieve, also on the same $rsid$, $epk_U$, $\widetilde{C}'_p$ and $\mathbf{S}'$. Since $note$ contains ciphertexts $C_p, \widetilde{C}_p, C_K, \widetilde{C}_K$ which encrypt the password $p$ and data key $K$, this in particular means that they agree on the same $p$ and $K$. Moreover, when an honest server proceeds beyond Step (R.8) in a retrieve protocol, then they also agree on the same ciphertext $C'_{\text{test}}$ that encrypts the randomized password quotient $p \backslash p'$.

**GAME 3** Here we replace all non-interactive zero-knowledge proofs, when provided by honest parties, by simulations. Any environment distinguishing this game from the previous one breaks the zero-knowledge property of the proof system.

**GAME 4** We now substitute the public key in the CRS by $(PK, SK) \leftarrow \mathsf{KGen}(1^k)$ generated by the challenger, i.e., he knows the corresponding secret key. Whenever an honest server then receives a ciphertext $\widetilde{C}_p$ or $\widetilde{C}_K$ in setup or a ciphertext $\widetilde{C}_{p'}$ in retrieve from a corrupt user, the challenger decrypts these ciphertexts using $SK$. He then stores the recovered $p$, $K$, and $p'$ in a local record $s$ using $s.sid$ and $s.ssid$ as identifiers. Further, whenever completing the setup protocol for an honest user and less than $t+1$ corrupt servers, the challenger also stores all the created threshold key pairs as $s.\mathbf{TKeys} \leftarrow (tpk_1, tsk_1), \ldots, (tpk_n, tsk_n)$. This game hop is purely conceptional and hence has no influence on the adversary's view of the game.

**GAME 5** We replace all honestly generated ciphertexts $\widetilde{C}_p$, $\widetilde{C}_K$ or $\widetilde{C}_{p'}$ under the CRS public key by "dummy" ciphertext, i.e., by encrypting 1. Thus from now on, all those ciphertexts, when coming from an honest party, are independent of $p, K$ and $p'$. Clearly, we have $\text{GAME}_5 \approx \text{GAME}_4$ by the semantic security of the CPA-encryption scheme $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$. (We do not require CCA-security here, because in the game above we decrypted the adversarial provided ciphertexts only for internal book-keeping, i.e., in the reduction we would not be able to maintain such a record, but that remains invisible to the adversary.)

Note that the simulated zero-knowledge proofs $\pi_0$ and $\pi_1$ now actually prove false statements, as the encryptions under the threshold public key and the CRS public key are not consistent anymore. The simulation-soundness of the proof system guarantees that an adversary, even after having seen such simulated proofs of false statements, cannot produce a valid proof of a false statement himself.

**GAME 6** In this game we change how the encrypted password quotient is computed in a retrieval done by an honest user. Namely, we exploit the challenger's knowledge of whether or not the passwords match to compute ciphertexts containing the correct information but without using $C_p$. That is, if $p = p'$ he replaces $C_{\text{test}}$, for the honest user, and $C'_{\text{test},i}$ for an honest server, by random encryptions of "1": $\mathsf{TEnc}_{tpk}(1)$. Likewise, if $p \neq p'$, the challenger replaces $C_{\text{test}}$ and $C'_{\text{test},i}$ by encryptions of random values, indicating that the passwords did not match. Recall that for the latter case, the user and all servers contribute (secret) randomness to the finally derived ciphertext $C'_{\text{test}}$, i.e., if the password did not match and at least one honest party participates, the adversary can not predict the outcome of the (joint) decryption of $C'_{\text{test}}$. Thus, the replacement by random chosen values for the case $p \neq p'$ is legitimate.

Note that the challenger always knows whether $p = p'$ before such a ciphertext must be generated. Namely, the challenger knows $p'$ as the retrieval is done by an honest user. The original password $p$ is obtained from the local record $s$ for $s.sid$ and $s.ssid$ being the values contained in the $note$ file to which all servers have given their consent in Step (R.4a) of the retrieval. The password $p$ stored in $s.p$ either stems from having created the corresponding account as honest user, or by having decrypted $\widetilde{C}_p$ when the account was created by a corrupt user. Thus, if the account was created by a corrupt user, the challenger has to rely on the fact that $\widetilde{C}_p$ indeed encrypts the same $p$ as $C_p$. Since $\mathcal{A}$ has to prove that fact in $\pi_0$, he can provide inconsistent ciphertexts with negligible probability only.

Recall that we replaced the ciphertext $\widetilde{C}_p$ in the previous game by a dummy ciphertext and provided a "false" proof $\pi_0$ whenever a setup was done by an honest user. Thus, if the adversary, when running a setup for a dishonest user, manages to re-use such a ciphertext pair of a real $C_p$ and faked $\widetilde{C}_p$, he would be able to distinguish the current game hop. However, this will happen with negligibly probability only, as the (faked) proof of knowledge $\pi_0$ of the supposedly same $p$ is bound to $(sid, ssid, tpk, \mathbf{tpk}, \mathbf{S}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K)$. That is, if an adversary wants to transfer (a derivation of) $C_p, \widetilde{C}_p$ to a different setup context, he must forge the corresponding proof $\pi_0$.

Overall, we have $\text{GAME}_5 \approx \text{GAME}_6$ based on the simulation-soundness of the proof system.

**GAME 7** We now substitute the encryptions $C_{S,i}$ of the threshold secret keys by "dummy" ciphertexts whenever they are sent by an honest user to an honest server in a setup with less than $t+1$ corrupt servers. That is, in Step (S.1e) those ciphertexts are replaced by encryptions of "1": $\mathsf{Enc2}_{epk_i}(1, (sid, note))$. An honest server receiving such a ciphertext sent by the honest user, continues with the protocol without decrypting $C_{S,i}$. In the retrieval for such a record, the honest servers simply uses its threshold secret key stored by the challenger in $s.\mathbf{TKeys}$.

However, if an honest server received a pair $(note, C_{S,i})$ that does not stem from the honest user, it decrypts the ciphertext and checks if it contains a valid threshold secret key. Such an even can occur if (i) an honest server receives a *different* ciphertext $C_{S,i}$ than the honest user had sent, but the sub-session $ssid$ got *not* stolen, i.e., the $note$ is the same, or (ii) the sub-session $ssid$ gets stolen, i.e., an honest server receives a ciphertext $C_{S,i}$ with a *different* $note$ that was originally sent by the honest user. If the decryption leads to a valid threshold secret key $tsk_i$, the honest servers keeps it in its local record, and in case of event (ii) the challenger also sets $s.\mathbf{TKeys} = \emptyset$. In case the decryption or the verification of $tsk_i$ fails, the honest server will abort normally and thereby forces the entire setup to fail (as it requires the consent of *all* servers).

Note, that due to the CCA2-security of the encryption scheme and the fact that it is labeled with the $note$ of a session, which in turn includes the session identifier $(ssid', \mathbf{S})$ with the set of servers $\mathbf{S}$, the adversary will not be able to re-use a dummy ciphertext $C_{S,i}$ in a different context.

More precisely, any adversary $\mathcal{A}$ that with non-negligible probability distinguishes between $\text{GAME}_6$ and $\text{GAME}_7$, allows to construct an adversary $\mathcal{B}$ breaking the semantic security of the labeled CCA2-secure encryption scheme $(\mathsf{KGen2}, \mathsf{Enc2}, \mathsf{Dec2})$ (using a series of hybrids, replacing the ciphertexts one by one). Thereby, whenever $\mathcal{A}$ sends a "fresh" combination of ciphertext $C_{S,i}$ and label $note$ that needs to be decrypted by the challenger, $\mathcal{B}$ forwards both to its decryption oracle. As it does so only for ciphertext/label pairs that are different to what he had sent before, all those oracle calls are legitimate.

**GAME 8** In this game we abort if the adversary manages to replace a dummy ciphertext $C_{S,i}$ by a ciphertext that actually decrypts to a valid threshold secret key. That is, we abort whenever a "successful" event (i) as described in the game above occurs. Therein, the honest user sends a dummy ciphertext $C_{S,i}$ instead of an encryption of $tsk_i$ to each honest server. If an honest server now receives a different ciphertext $C'_{S,i}$ that decrypts to a valid threshold secret key (w.r.t. the public keys generated by the honest user), the challenger halts.

If an adversary $\mathcal{A}$ can distinguish between $\text{GAME}_8$ and $\text{GAME}_7$ with non-negligible probability, we derive an adversary $\mathcal{B}$ against the semantic security of the threshold encryption scheme. To this end, $\mathcal{B}$ upon receiving $(tpk, \{tpk_i\})$ and $t$ secret shares $(tsk_{i_1}, \ldots tsk_{i_t})$ as input, embeds those challenge keys in one of the setup sessions initiated by an honest user. Thus, it guesses a session $i$ in which he sends the known challenge secret shares $(tsk_{i_1}, \ldots, tsk_{i_t})$ to the corrupt servers and dummy ciphertexts to the honest servers. If adversary $\mathcal{B}$ in the role of an honest server then receives a ciphertext $C_{S,i}$ which is not the one he had sent out and that decrypts to a valid threshold secret key $tsk_{i_{t+1}}$, he can use that $t+1$ share to immediately break the semantic security of the threshold scheme. Namely, he simply chooses two random plaintexts $m_0, m_1$ for the challenge oracle and then decrypts the returned challenge ciphertext with his knowledge of the $t+1$ secret key shares. Note though, that we have to abort the game after the $i$-th setup session even if no ciphertext got replaced, as $\mathcal{B}$ does not know the secret keys of the honest servers, and thus could not correctly simulate a retrieval for that account.

**GAME 9** We now change the way the challenger performs the threshold encryption and decryption steps whenever dealing with an account that was created by an honest user with at most $t$ corrupt servers. At setup, the challenger (as honest user) replaces the threshold encryptions $C_p$ and $C_K$ of $p$ and $K$, respectively, by encryptions of 1, i.e $C_p = \mathsf{TEnc}_{tpk}(1), C_K = \mathsf{TEnc}_{tpk}(1)$. As the adversary knows at most $t$ secret keys of the threshold scheme, he can not distinguish between a correct and dummy ciphertext under $tpk$ due to the semantic security of the threshold encryption scheme.

When the challenger subsequently participates in a retrieval for such an account where the adversary pretends to be the former honest user, the challenger also replaces all decryption shares that are computed by honest servers. This is done using the threshold simulator as described in Section 3 and exploiting the knowledge of the threshold secret keys of the dishonest servers (those are stored for honestly generated setups since GAME$_4$) and the knowledge of the ciphertexts and their corresponding plaintexts. That is, for shares $(d_i, \pi_{d_i})$, the challenger uses the received $C'_{\text{test}}$ and its knowledge of whether or not $p = p'$. Recall that $p$ is known to the challenger as the setup was done by an honest user and $p'$ was extracted from $\widetilde{C}_{p'}$ sent by the adversary in the first step of the retrieval. Further, the challenger retrieves all threshold keys from its local record $s.\mathbf{TKeys}$ and invokes the threshold simulator on input $(tpk, \{tpk_i\})_{S_i \in \mathbf{S}}$, the at most $t$ secret shares $(tsk_j)$ of the corrupt servers that participated in the setup, the ciphertext $C'_{\text{test}}$ computed in Step (R.8a) and a message $m$ where $m \leftarrow 1$ if $p = p'$ and $m \leftarrow R$ for a random $R$ if $p \neq p'$. The threshold simulator then outputs verifiable decryption shares $(d_j, \pi_{d_j})$ for all honest servers $\mathcal{S}_j$, which $\mathcal{S}_j$ then sends instead of the real values in Step (R.8c).

Conditioned on the fact that $\widetilde{C}_{p'}$ contains the same $p'$ that was used to derive $C_{\text{test}}$, and all $C'_{\text{test},i}$ provided by the adversary are correct, the semantic security of the threshold encryption scheme guarantees that the view of the adversary has not significantly changed. More precisely, according to the threshold semantic security it follows that the adversary holding at most $t$ secret keys[1] can distinguish with negligible probability only, if the shares $(d_j, \pi_{d_j})$ are simulated or the correct shares that would have been computed if $C'_{\text{test}}$ is indeed the encryption of $m$. The condition that this simulation is based on "correct" ciphertexts holds due to the security of the proof system, as the adversary has to prove the consistency between $C_{\text{test}}$ and $\widetilde{C}_{p'}$ in $\pi_1$, and the correctness of $C'_{\text{test},j}$ in $\pi_{2,i}$.

Similarly, the shares $(d'_i, \pi_{d'_i})$ for the data key $K$ are simulated using $C_K$ and $K$, both are known to the challenger from the honest setup.

Overall, we have GAME$_8 \approx$ GAME$_9$ conditioned on the semantic security of the threshold encryption scheme and the simulation-soundness of the proof system.

**GAME 10** We now abort the game whenever the challenger recognizes some inconsistency between his knowledge about $p, K$ or $p'$ and the information provided by the adversary. More precisely, he halts if he receives decryption shares $d_i$ such that the joint decryption of the password quotient leads to 1 but $p \neq p'$ (we do not have to consider the opposite case, as the protocol then fails anyway), or he receives decryption shares $d'_i$ leading to a key $K' \neq K$. An adversary will notice the change only when he was able to provoke such inconsistent decryptions in the previous game. That could happen if (1) adversarial provided ciphertexts $C_p, C_K, C'_{\text{test}}$ were not consistent with $\widetilde{C}_p, \widetilde{C}_K, \widetilde{C}_{p'}$, or (2) the adversary was able to produce valid decryption shares that opened a ciphertext of the threshold scheme to a different value that was originally encrypted.

However, case (2) implies breaking the soundness of the threshold encryption scheme and case (1) requires the adversary to forge the corresponding proofs that must always be provided in combination with the ciphertexts. Note that the adversary can also not re-use dummy ciphertexts and faked proofs that the challenger sends in other sessions, as those proofs are always bound to the context the ciphertexts are supposed to be used in. In case of $\pi_0$ the proof was bound to $(sid, ssid, tpk, \mathbf{tpk}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K)$ and for $\pi_1$ the context was $(rsid, note, epk_U, C_{\text{test}}, \widetilde{C}_{p'})$. Thus, whenever the adversary wants to re-use a randomized version of such dummy ciphertexts (which is not excluded, as we only have CPA security for those), or use the same ciphertext in a different sub-session, or in combination with a different public key $epk_U$, the context changes which prevents the adversary to re-use the (faked) proof seen by the challenger. Instead, he needs to come up with a proof of a false statement himself, which will succeed only with negligible probability due to the simulation-soundness of the zero-knowledge proof system.

Thus we have GAME$_9 \approx$ GAME$_{10}$ based on the soundness of the threshold encryption scheme and the simulation-soundness of the proof system.

[1] Note that the definition of threshold semantic security considers an adversary holding *exactly* $t$ secret keys, whereas the adversary here gets only *at most* $t$ keys. However, security for that case follows trivially.

**GAME 11** In this game we also modify the ciphertexts $C_{R,i}$ that are supposed to securely transport the shares of the data key $K$ in the retrieval. Again, those ciphertexts are replaced by "dummy" ciphertexts $\mathsf{Enc2}_{epk_U}(1, (epk_U, spk_i))$ whenever sent by an honest server to an honest user. Using the public signing key of the sending server $spk_i$ as well as the ephemeral public key $epk_U$ of the user as label thereby ensures that the adversary can not re-use such a "dummy" ciphertext.

If an adversary can distinguish between $\text{GAME}_{10}$ and $\text{GAME}_{11}$ with non-negligible probability, we can turn him into an adversary breaking the CCA2-security of the encryption scheme $(\mathsf{KGen2}, \mathsf{Enc2}, \mathsf{Dec2})$. We need CCA2-security here, as the challenger will still receive correct ciphertexts $C_{R,j}$ from the corrupt servers that he has to decrypt. Including the public signing key of the sending server as label ensures that even when the adversary provides a ciphertext $C_{R,j}$ for a corrupt server that equals a previously send dummy ciphertext from one of the honest servers, it results in a legitimate query to the decryption oracle. Likewise, using the ephemeral public key $epk_U$ of the user as label ensures that a dummy ciphertext from one session can not be maliciously reused in another retrieval session started by the honest user.

**GAME 12** In our final game we now make the transition from letting the challenger run the "real" protocol (w.r.t. $\text{GAME}_{11}$) to letting him internally run the ideal functionality $\mathcal{F}$ and simulate all messages based merely on the information he can obtain from $\mathcal{F}$. For that, it is crucial to observe that in the series of games we have replaced several messages by dummy messages, that are indistinguishable from the real ones but do not depend on $p, K$ and $p'$ anymore. However, others we have derived based on the knowledge of whether or not $p = p'$, and some are still depending on the real values of $p, K$ and $p'$. Thus, for those messages it remains to be shown that we can obtain the necessary information in time from $\mathcal{F}$ as well. The description of how to build such a simulator depending on the different combinations of honest and corrupts parties is given in following part.

## F.2 The Simulator

We finally have to show that there exist a simulator $\mathcal{SIM}$ such that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$ that controls a fixed subset of the parties, the view of the environment in the real world, when running the protocol (according to $\text{GAME}_{11}$) with the adversary, is indistinguishable from its view in the ideal world where it interacts with the ideal functionality and the simulator (which corresponds to $\text{GAME}_{12}$).

To do so, we describe the simulator for the different cases, i.e., different combinations of corrupt parties. Thereby, the simulator will play the role of all honest parties in the real protocol, which is denoted by "$\mathcal{S}_i$" for a simulated server, or "$\mathcal{U}$" for a simulated user. The simulator will store all records created by an honest party, according to the real protocol. We denote by $\mathbf{S}_c \subseteq \mathbf{S}$ the set of corrupt servers, and with $\mathbf{S}_h \subseteq \mathbf{S}$ the set of all honest servers.

### F.2.1 Setup - Simulation

#### F.2.1.1 Honest User

The setup process is triggered when the environment invokes the ideal functionality $\mathcal{F}$, on behalf of an honest user $\mathcal{U}$, on some input $(\texttt{SETUP}, sid, ssid, p, K)$ with $ssid = (ssid', \mathbf{S})$. The ideal functionality then signals the initiated setup towards the simulator $\mathcal{SIM}$, where the information it reveals to $\mathcal{SIM}$ depends on the number of corrupt servers in $\mathbf{S}$. Thus, we split our simulation in two cases, where in the first case less than $t+1$ servers in $\mathbf{S}$ are corrupt, and in the second case the threshold of $t+1$ corrupt servers is reached.

**Less than $t+1$ Servers are Corrupt.** The simulator $\mathcal{SIM}$ receives $(\texttt{SETUP}, sid, ssid)$ and then runs the real setup protocol for a simulated "$\mathcal{U}$" on input $sid, ssid$ and $p = K = \perp$ where it also plays the role of all honest servers "$\mathcal{S}_i$" $\in \mathbf{S}_h$. Thus, instead of computing $C_p, C_K, \widetilde{C}_p, \widetilde{C}_K, \pi_0, (C_{S,i})_{\mathcal{S}_i \in \mathbf{S}_h}$, as in the real protocol, "$\mathcal{U}$" encrypts only ones and simulates the corresponding proofs. See $\text{GAME}_3$, $\text{GAME}_5$ and $\text{GAME}_7$. The simulator also creates a record $s$ with $s.sid \leftarrow sid, s.ssid \leftarrow ssid, s.p \leftarrow \perp, s.K \leftarrow \perp$ and stores all the threshold keys created by "$\mathcal{U}$" as $s.\mathbf{TKeys} \leftarrow \{(tpk_1, tsk_1), \ldots, (tpk_n, tsk_n)\}$. The rest of the protocol is then done correctly, i.e., "$\mathcal{U}$" and all "$\mathcal{S}_i$" $\in \mathbf{S}_h$ follow the steps as described in Section 4, except that the honest servers do not decrypt the received ciphertext $C_{S,i}$ unless it is different than the one "$\mathcal{U}$" had sent out before. Thereby, the simulator also aborts when an honest server receives a replaced ciphertext $C_{S,i}$ which decrypts to a valid threshold key (see $\text{GAME}_8$).

When the first honest server "$\mathcal{S}_i$" outputs $(\texttt{SETUP}, sid, ssid, \mathcal{S}_i)$, indicating that all servers gave their consent to a common $note$ file, $\mathcal{SIM}$ checks if "$\mathcal{S}_i$" received a $note$ that is different than the one originally sent by "$\mathcal{U}$". If

so, the simulator recognizes the session as being "stolen" by the adversary, decrypts $\widetilde{C}_p, \widetilde{C}_K$ contained in $note$ to obtain $\hat{p}, \hat{K}$ and sends $(\texttt{STEAL}, sid, ssid, \hat{p}, \hat{K})$ to $\mathcal{F}$. It also sets $s.p \leftarrow \hat{p}, s.K \leftarrow \hat{K}$ and $s.\mathbf{TKeys} \leftarrow \emptyset$. The rest of the simulation then continues as in the case of a setup done by a dishonest user. Thus, the description below only considers sessions where the adversary did not steal the sub-session identifier$ssid$.

The simulator subsequently sends $(\texttt{JOIN}, sid, ssid, \mathcal{S}_i)$ to $\mathcal{F}$. This message triggers the output $(\texttt{SETUP}, sid, ssid)$ for $\mathcal{S}_i$ and the public delayed output $(\texttt{SETUP}, sid, ssid, \mathcal{S}_i)$ for $\mathcal{U}$. The latter is delivered to $\mathcal{U}$ only if "$\mathcal{U}$" in the real word outputs $(\texttt{SETUP}, sid, ssid, \mathcal{S}_i)$.

For every further honest server "$\mathcal{S}_j$" $\in \mathbf{S}_h$ that outputs $(\texttt{SETUP}, sid, ssid)$ in the real world, $\mathcal{SIM}$ directly sends $(\texttt{SETUP}, sid, ssid, \mathcal{S}_j)$ to $\mathcal{F}$. Again, the output from $\mathcal{F}$ to $\mathcal{U}$ is only delivered if "$\mathcal{U}$" outputs $(\texttt{SETUP}, sid, ssid, \mathcal{S}_j)$.

When "$\mathcal{U}$" outputs $(\texttt{SETUP}, sid, ssid, \mathcal{S}_k)$ for a corrupt server $\mathcal{S}_k \in \mathbf{S}_c$, the simulator sends $(\texttt{JOIN}, sid, ssid, \mathcal{S}_k)$ to $\mathcal{F}$.

At the end of a successfully completed setup, the simulator maintains a record where $p = K = \perp$ and it stores all the secret threshold keys of the honest and corrupt servers. Further, the ciphertexts $C_p, C_K, \widetilde{C}_p, \widetilde{C}_K$ contained in the $note$ that is stored by all servers, are encryptions of ones with a corresponding faked proof of correctness.

**At least $t+1$ Servers are Corrupt.** The simulator $\mathcal{SIM}$ receives $(\texttt{SETUP}, sid, ssid, p, K)$ and starts the simulation of "$\mathcal{U}$" with the same input and creates a record $s.sid \leftarrow sid, s.ssid \leftarrow ssid, s.p \leftarrow p, s.K \leftarrow K, s.\mathbf{TKeys} = \emptyset$. It also simulates all honest servers "$\mathcal{S}_i$" $\in \mathbf{S}_h$. Due to the knowledge of $p, K$, the simulation in this case simply follows the normal protocol instructions. Whenever an honest party outputs a message, the simulator mimics the output in ideal world as well.

When the setup session identifier $ssid$ gets stolen, i.e., an honest server "$\mathcal{S}_i$" outputs $(\texttt{SETUP}, sid, ssid)$ but received a different note file than was sent by "$\mathcal{U}$", $\mathcal{SIM}$ decrypts $\widetilde{C}_p, \widetilde{C}_K$ to $\hat{p}, \hat{K}$ and sends $(\texttt{STEAL}, sid, ssid, \hat{p}, \hat{K})$ to $\mathcal{F}$. It also sets $s.p \leftarrow \hat{p}, s.K \leftarrow \hat{K}$ and continues the simulation as in the case of a dishonest user.

At the end of a successfully completed setup, the simulator maintains a record where $p, K$ are the same values as stored in the ideal functionality, and the ciphertexts $C_p, C_K, \widetilde{C}_p, \widetilde{C}_K$ contained in the $note$ file are proper encryptions of $p, K$.

#### F.2.1.2 Dishonest User (and at least one Honest Server)

If the user is dishonest, we do not have to condition our simulation on the number of corrupt servers in $\mathbf{S}$, as the simulator can extract $p, K$ from the adversary (and the knowledge of $p, K$ was the main difference between the two cases above). In this simulation, $\mathcal{SIM}$ plays the role of the honest servers $\mathcal{S}_i \in \mathbf{S}_h \neq \emptyset$ and waits for a setup request sent by the corrupt user in the real world. An honest server "$\mathcal{S}_i$" upon receiving the message $(\texttt{SETUP}, sid, ssid, 1, note, C_{S,i})$ then starts with the protocol as specified. The simulator also extracts $p, K$ from the note by decrypting $\widetilde{C}_p, \widetilde{C}_K$ and creates a setup record with $s.sid \leftarrow sid, s.ssid \leftarrow ssid, s.p \leftarrow p, s.K \leftarrow K, s.\mathbf{TKeys} \leftarrow \emptyset$. $\mathcal{SIM}$ further sends $(\texttt{SETUP}, sid, ssid, p, K)$ to $\mathcal{F}$, thereby "simulating" an honest user in the ideal world, ensuring that a record with the correct values $p, K$ will be created. Whenever an honest server "$\mathcal{S}_i$" outputs $(\texttt{SETUP}, sid, ssid)$ in the real world, the simulator sends $(\texttt{JOIN}, sid, ssid, \mathcal{S}_i)$ to $\mathcal{F}$. Note that the real world adversary could sent different notes (and thus possibly different $p, K$) to the honest servers, however, then the setup would fail for all of them.

At the end of a successfully completed setup, the simulator maintains a record where $p, K$ are the same values as stored in the ideal functionality, and the ciphertexts $C_p, C_K, \widetilde{C}_p, \widetilde{C}_K$ contained in the $note$ file are proper encryptions of $p, K$.

### F.2.2 Retrieval - Simulation

The retrieval protocol is run among a user $\mathcal{U}'$ and $t+1$ servers, denoted as $\mathbf{S}'$. Whenever a simulated honest user "$\mathcal{U}'$" in the real world outputs $(\texttt{DELIVER2U}, sid, rsid, \perp)$, the simulator sends $(\texttt{DELIVER}, sid, rsid, \mathcal{U}', \texttt{deny})$ to $\mathcal{F}$. Likewise, if an honest server "$\mathcal{S}_i$" played by the simulator outputs $(\texttt{DELIVER2S}, sid, rsid, \texttt{fail})$, $\mathcal{SIM}$ sends $(\texttt{DELIVER}, sid, rsid, \mathcal{S}_i, \texttt{deny})$ to $\mathcal{F}$.

#### F.2.2.1 Honest User

When the retrieve is initiated by an honest user, the environment will invoke the ideal functionality on some input $(\texttt{RETRIEVE}, sid, rsid, p')$ which in turn sends $(\texttt{RETRIEVE}, sid, rsid, \mathcal{U}')$ to $\mathcal{SIM}$. Recall that $rsid$ contains the set

of servers $\mathbf{S}'$ the user wishes to tun the retrieval with. The rest of simulation then depends if either all $t+1$ servers in $\mathbf{S}'$ are corrupt, or at least one honest server participates. We start the description with the latter case.

**Less than $t+1$ Servers are Corrupt.** The simulator upon receiving $(\texttt{RETRIEVE}, sid, rsid, \mathcal{U}')$ from $\mathcal{F}$ starts the real-world retrieval protocol on behalf of the honest user "$\mathcal{U}'$" for input $(\texttt{RETRIEVE}, sid, rsid, \perp)$. That is, using $p' = \perp$ instead of the real password attempt, which is unknown to the simulator. Thus, in Step (R.1c) the honest user "$\mathcal{U}'$" sends $\widetilde{C}_{p'}$ as being an encryption of ones, according to $\text{GAME}_5$. $\mathcal{SIM}$ also simulates the protocol for all honest servers whenever a server receives a retrieve request $(\texttt{RETRIEVE}, sid, rsid, 1, epk_U, \widetilde{C}_{p'})$.

If an honest server "$\mathcal{S}_i$" outputs $(\texttt{NOTIFY}, sid, rsid)$ in Step (R.4a), $\mathcal{SIM}$ checks whether the received ciphertext $\widetilde{C}_{p'}$ is different than the one "$\mathcal{U}'$" had sent. If this is the case, i.e., the retrieve session got stolen, $\mathcal{SIM}$ obtains $\hat{p}$ by decrypting $\widetilde{C}_{p'}$ and sends $(\texttt{STEAL}, sid, rsid, \hat{p})$ to $\mathcal{F}$[2]. The simulation of a stolen retrieve session then continues as the simulation for a dishonest user. Thus, the rest of the simulation described here only considers intact sessions.

An honest server, upon successfully passing Step (R.4a) then asks the environment for permission to proceed, which is done accordingly in the ideal world by delivering the message $(\texttt{NOTIFY}, sid, rsid)$ to $\mathcal{S}_i$. This message was released as a public delayed output already in the first interface of the ideal functionality. The same is repeated for any subsequent honest server outputting $(\texttt{NOTIFY}, sid, rsid)$.

When $\mathcal{SIM}$ in turn receives a message $(\texttt{PROCEED}, sid, rsid, \mathcal{S}_i, a)$ from $\mathcal{F}$ it sends $(\texttt{PROCEED}, sid, rsid, a)$ to "$\mathcal{S}_i$" and proceeds with the protocol in case $a = \texttt{allow}$.

At some point, namely when the environment has allowed *all* honest servers to proceed, the simulator will finally obtain the message $(\texttt{NOTIFY}, sid, rsid, c)$, with $c$ indicating whether $p' \neq p$ or not, where $p$ is the original password stored by the ideal functionality. Depending on whether the passwords match or not, "$\mathcal{U}'$" now has to prepare the encrypted password quotient $C_{\text{test}}$ such that it either encrypts 1 in case of $c = \texttt{correct}$ or a random value otherwise (see $\text{GAME}_6$). Thus, it simply encrypts the targeted result as $C_{\text{test}}$ and fakes the corresponding proof $\pi_1$. The simulation of the honest user and servers then continues as in the normal protocol. Note that consequently all decryption shares released by the honest servers are correct and not simulated. This is important as the adversary, despite being present with only $|\mathbf{S}_c| < t+1$ servers in the retrieve, actually knows $|\mathbf{S}_c| \geq |\mathbf{S}'_c|$ valid threshold secret keys from the corresponding setup (plus even the secret keys of the honest servers, in case the account was created by a dishonest user). Thus, the adversary must be able to take any subset of the shares $d_i$ and obtain a decrypted value that is consistent with the rest of the protocol. This is ensured by our procedure as described above.

If in Step (R.9b) the threshold decryption of $C'_{\text{test}}$ did not result in 1, all honest parties will subsequently abort the protocol which is reflected by the simulator in the ideal-world as well. The simulation will also abort if the decryption leads to 1 but $c = \texttt{wrong}$. This is justified in $\text{GAME}_{10}$. Thus, the rest of the simulation is now conditioned on the fact that $p' = p$. In Step (R.10d) of the protocol, an honest server has to compute its threshold share $d'_i$ of the data key $K$ and send it encrypted under $epk_U$ to "$\mathcal{U}'$". As the simulator might not know $K$ (namely if the setup was done by an honest user and less than $t+1$ corrupt servers), it let "$\mathcal{S}_i$" simply compute $C_{R,i}$ as encryption of ones and sends it signed to "$\mathcal{U}'$". This is a legitimate simulation, because we only consider non-stolen retrieval sessions here, and thus, the adversary does not know $esk_U$. See also $\text{GAME}_3$ and $\text{GAME}_{10}$. Whenever an honest party "$\mathcal{S}_i$" then outputs $(\texttt{DELIVER2S}, sid, rsid, \texttt{success})$, $\mathcal{SIM}$ also sends $(\texttt{DELIVER}, sid, rsid, \mathcal{S}_i, \texttt{allow})$ to $\mathcal{F}$.

If the user "$\mathcal{U}'$" receives "valid" shares (meaning that the associated proofs were correct) from all corrupt servers, he is supposed to reconstruct and output his data key $K$. Thus, when "$\mathcal{U}'$" ends with output $(\texttt{DELIVER2U}, sid, rsid, K')$ (where $K'$ is a random key), $\mathcal{SIM}$ sends $(\texttt{DELIVER}, sid, rsid, \mathcal{U}', \texttt{allow})$ to $\mathcal{F}$ which will lead to the output $(\texttt{DELIVER2U}, sid, rsid, K')$ to the environment, with $K'$ being the correct key as stored by the ideal functionality. The latter is guaranteed, since we considered intact sessions (i.e., $rsid$ got not stolen for another session), we have at most $t$ corrupt servers in the retrieval and the simulator only reached that point in case it learned that $p' = p$.

**All $t+1$ Servers are Corrupt.** In this case, the simulator will only play the role of the honest user in the real word. Thus, upon receiving $(\texttt{RETRIEVE}, sid, rsid', \mathcal{U}')$ from $\mathcal{F}$, $\mathcal{SIM}$ starts the real-world retrieval protocol on behalf of the honest user "$\mathcal{U}'$" for input $(\texttt{RETRIEVE}, sid, rsid, \perp)$. That is, again using $p' = \perp$ instead of the real password attempt (which is unknown to the simulator), and thus simulating $\widetilde{C}_{p'}$ by encrypting only ones (see $\text{GAME}_5$).

[2]Note that the adversary could potentially send different ciphertexts $\widetilde{C}_{p'}$ to all honest servers. That would also result in different steal calls to the ideal functionality (with different $\hat{p}$) which, except for the first call, would be ignored by $\mathcal{F}$. However, such an "attack" would result in a failed retrieval in Steps (R.4c) / (R.5a) of the protocol (and thus also in the ideal world) anyway.

If "$\mathcal{U}'$" passes Step (R.3b), i.e., he received the same $note = (ssid, tpk, \mathbf{tpk}, C_p, C_K, \widetilde{C}_p, \widetilde{C}_K, \pi_0)$ correctly signed by all servers in $\mathbf{S}'$, he extracts $p^*, K^*$ by decrypting $\widetilde{C}_p, \widetilde{C}_K$ from the $note$ file. Note that those ciphertext will (whp) not be dummy ciphertext the simulator has produced for another session, as those dummy ciphertexts (i.e., encryptions of "1") were only used by accounts that were setup by an honest user and *less* than $t+1$ corrupt servers. Since they must come with a proof of correctness which is bound to the setup server set $\mathbf{S}$, it is ensured that $\mathcal{A}$ cannot reuse some dummy ciphertexts in a different context, e.g., adding more corrupt servers to $\mathbf{S}$.

The simulator then sends $(\texttt{PLANT}, sid, rsid, p^*, K^*)$ to $\mathcal{F}$, obtaining $(\texttt{NOTIFY}, sid, rsid, c)$ in return where $c = \texttt{correct}$ if $p' = p^*$ and $c = \texttt{wrong}$ otherwise. This information is then used by the simulator to set the ciphertext $C_{\text{test}}$ correctly. Namely, if $c = \texttt{correct}$ the simulator computes $C_{\text{test}}$ in Step (R.5c) as an encryption of ones, and of a random value $R$ otherwise. The proof $\pi_1$ is simulated w.r.t. the ciphertext $\widetilde{C}_{p'}$ that "$\mathcal{U}'$" has already sent.

When "$\mathcal{U}'$" in Step (R.9) receives decryption shares of all servers, which result in the joint decryption into 1 and we had $c = \texttt{correct}$ the simulator proceeds with the normal protocol. Otherwise "$\mathcal{U}'$" aborts the protocol and $\mathcal{SIM}$ sends $(\texttt{DELIVER}, sid, rsid, \mathcal{U}', \texttt{deny})$ to $\mathcal{F}$. Thus, we will abort also when the adversary sends decryption shares which are not consistent with the information that "$\mathcal{U}'$" had encrypted in $C_{\text{test}}$, as justified in $\text{GAME}_9$.

When "$\mathcal{U}'$" eventually ends with output $(\texttt{DELIVER2U}, sid, rsid, K)$ where $K = K^*$, $\mathcal{SIM}$ sends $(\texttt{DELIVER}, sid, rsid, , \mathcal{U}', \texttt{allow})$ to $\mathcal{F}$, and aborts otherwise.

#### F.2.2.2 Dishonest User (and at least one Honest Server)

In case the user $\mathcal{U}'$ doing the retrieval is corrupt, we only have to consider the case that at least one server is honest, as otherwise all the communication will be internal to the adversary.

The simulator plays the part of all honest servers in $\mathbf{S}'$, and will start the simulation when an honest server "$\mathcal{S}_i$" receives a message $(\texttt{RETRIEVE}, sid, rsid, 1, epk_U, \widetilde{C}_{p'})$. $\mathcal{SIM}$ then extracts $p'$ from $\widetilde{C}_{p'}$ and invokes $\mathcal{F}$ on input $(\texttt{RETRIEVE}, sid, rsid, p')$, thus, starting the retrieve on behalf of $\mathcal{U}'$ in the ideal world. Whenever an honest server "$\mathcal{S}_i$" successfully passes Step (R.4a) of the protocol with output $(\texttt{RETRIEVE}, sid, rsid)$, $\mathcal{SIM}$ delivers the message $(\texttt{RETRIEVE}, sid, rsid)$ to $\mathcal{S}_i$ in the ideal world as well. Hereby, $\text{GAME}_2$ again ensures that all honest servers agree on the same $note, epk_U$ and $\widetilde{C}_{p'}$, or the protocol will have already failed at that point.

The simulator then obtains its setup record record $s$ for $s.sid = sid$ and $s.ssid = ssid$ where the latter is taken from the received note file. The rest of the simulation will now branch into two cases, depending on how the setup record $s$ was created[3]. If $s.p = s.K = \perp$, which can only happen if the account was created by an honest user with less than $t+1$ corrupt servers, the simulator continues with CASE 1 below. CASE 2 then covers the accounts that were created (i) by an honest user and at least $t+1$ corrupt servers; or (ii) by a dishonest user (either directly or via $ssid$-stealing). Recall that in those settings the simulator has stored the correct values $p, K$ in its record $s$ and, more importantly, also the $note$ file stored by all servers contains encryptions $C_p, C_K$ of the real values $p$ and $K$. Thus, whenever we are in CASE 2 the simulation can proceed according to the normal protocol instructions, with the simulator simply reflecting all outputs obtained in the real world equivalently in the ideal world as well. Whereas in CASE 1, the simulator did not know $p, K$ yet, and thus also the ciphertexts $C_p, C_K$ stored in the note file by all servers merely contain encryptions of ones. On the other hand, the simulator therein knows the threshold secret keys of all corrupt servers, which he can exploit to tweak the decryption shares of the honest servers the way he needs it:

**CASE 1 – $s.p = s.K = \perp$.** The simulator upon receiving a message $(\texttt{PROCEED}, sid, rsid, \mathcal{S}_i, a)$ from the environment, forwards the same message to "$\mathcal{S}_i$". If $a = \texttt{allow}$, the honest server proceeds with the normal protocol and waits until it receives a valid input $(C_{\text{test}}, \pi_1, (\sigma_{5,j})_{\mathcal{S}_j \in \mathbf{S}'})$ from the dishonest $\mathcal{U}'$. Note that the tuple includes signatures of all honest servers, and thus this input can only arrive if the environment allowed all honest servers to proceed, which in turn will have triggered the delivery of $(\texttt{NOTIFY}, sid, rsid, c)$ from $\mathcal{F}$ to $\mathcal{SIM}$. Therein, $c \leftarrow \{\texttt{correct}, \texttt{wrong}\}$ indicates whether the password attempt $p'$ matches $p$ as maintained by the ideal functionality or not. This information will later be used by $\mathcal{SIM}$ to provide consistent decryption shares of the supposed encrypted password quotient. Before that, the honest servers when reaching Step (R.6b) derive $C'_{\text{test},i}$ correctly by re-randomizing the received ciphertext. Thus, the ciphertext $C'_{\text{test}}$ is also merely an encryption of "1", which however the adversary is not able to recognize as we now tweak the decryption shares to make $C'_{\text{test}}$ look like the "correct" ciphertext:

[3]The third case of running the retrieve for an account that was never successfully created will result – due to the presence of at least one honest server – in the termination of the protocol in Step (R.4a), thus nothing further needs to be simulated there.

When subsequently the first server reaches Step (R.8b), and thus must release its decryption share of $C'_{\text{test}}$, $\mathcal{SIM}$ uses the simulation property of the threshold scheme (as described in $\text{GAME}_7$) to derive the shares and correctness proof for all honest servers, such that they are consistent with $c$. More precisely, the simulator computes $(d_i, \pi_{d_i})$ for all honest $\mathcal{S}_i \in \mathbf{S}'_h$ such that $\mathsf{TDec}(C'_{\text{test}}, (d_j)_{\mathcal{S}_j \in \mathbf{S}'}) = 1$ if $c = \texttt{correct}$ and $\mathsf{TDec}(C'_{\text{test}}, (d_j)_{\mathcal{S}_j \in \mathbf{S}'}) = R$ for a random $R$ if $c = \texttt{wrong}$. Note that the threshold secret keys of the at most $t$ corrupt parties (which are needed for that trick) are stored by the simulator as $s.\mathbf{TKeys}$ in the record $s$, as the account was set up by an honest user (=simulator). For any honest server "$\mathcal{S}_i$" reaching Step (R.8b), $\mathcal{SIM}$ uses the $(d_i, \pi_{d_i})$ values from the threshold simulation.

If an honest servers "$\mathcal{S}_i$" obtains decryption shares in Step (R.10b) that indicate that $p' = p$, and $c$ was indeed $\texttt{correct}$, the simulation sends $(\texttt{DELIVER}, sid, rsid, \mathcal{S}_i, \texttt{allow})$ to $\mathcal{F}$ and $(\texttt{DELIVER}, sid, rsid, \mathcal{S}_i, \texttt{deny})$ otherwise (i.e., $\mathcal{SIM}$ also aborts the protocol if the joint decryption leads to 1 but $c = \texttt{wrong}$, as described in $\text{GAME}_9$). Now, the simulator has to produce decryption shares that will allow $\mathcal{U}'$ to open $C_K$ to the correct value. However, as $C_K$ was just a dummy encryption we need again the threshold simulator to tweak the decryption share accordingly. To learn the required plaintext message $K$ to which the shares for $C_K$ are supposed to decrypt, the simulator sends $(\texttt{DELIVER}, sid, rsid, \mathcal{U}', \texttt{allow})$ to $\mathcal{F}$. The ideal functionality will in turn respond with the message $(\texttt{DELIVER}, sid, rsid, K)$ where $K$ will be the correct data key, since we only reached that step when $\mathcal{F}$ has indicated that the passwords matched.

The simulator now invokes the threshold simulation on input the at most $t$ secret keys of all corrupt servers that participated in the setup and are kept in $\mathcal{SIM}$'s record, the ciphertext $C_K$ and the message $K$. The simulator outputs verifiable decryption share tuples $(d'_i, \pi_{d'_i})$ for all honest servers which are sent instead of the "real" values derived from the dummy ciphertext.

**CASE 2 –** $s.p = p, s.K = K$**.** The simulation of the second case is much simpler, as the simulator already knew $p, K$ in the setup of the account that is now used in the retrieval. Consequently, the ciphertexts $C_p, C_K$ that are included in the $note$ file and that were accepted by the honest servers, are proper encryptions of $p, K$. Thus, the honest servers – for which the environment allowed to proceed – simply follow the normal protocol instructions and whenever outputting a message, the simulator mimics the behaviour towards the ideal functionality. Note, that this means that we actually base our simulation on the real ciphertext $C_{\text{test}}$ as provided by the adversary, but have invoked the ideal functionality $p'$ extracted from $\widetilde{C}_{p'}$. Thus, we will end up in an inconsistent simulation if the adversary managed to provide inconsistent ciphertexts $C_{\text{test}}$ and $\widetilde{C}_{p'}$, i.e., that were not based on the same password attempt $p'$. However, as this requires the adversary to forge the proof $\pi_1$, this can only occur with negligible advantage due to the soundness of the proof system.

We also let $\mathcal{SIM}$ abort the protocol as soon as it detects such an inconsistency, i.e., if in Step (R.10b) an honest server receives decryption shares, such that the joint decryption results in 1 (indicating that the password matches), but $p \neq p'$. This is a legitimate action according to $\text{GAME}_9$.

Thus, we have shown how to construct a simulator for all combinations of honest and corrupt parties that, conditioned on the simulation-soundness of the proof-system, provides a view that is indistinguishable to the one described in $\text{GAME}_{11}$, which concludes our proof. ■