Practical Yet Universally Composable Two-Server Password-Authenticated Secret Sharing*

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

Anna Lysyanskaya
Brown University
anna@cs.brown.edu

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

Abstract

Password-authenticated secret sharing (PASS) schemes, first introduced by Bagherzandi et al. at CCS 2011, allow users to distribute data among several servers so that the data can be recovered using a single human-memorizable password, but no single server (or collusion of servers up to a certain size) can mount an off-line dictionary attack on the password or learn anything about the data. We propose a new, universally composable (UC) security definition for the two-server case (2PASS) in the public-key setting that addresses a number of relevant limitations of the previous, non-UC definition. For example, our definition makes no prior assumptions on the distribution of passwords, preserves security when honest users mistype their passwords, and guarantees secure composition with other protocols in spite of the unavoidable non-negligible success rate of online dictionary attacks. We further present a concrete 2PASS protocol and prove that it meets our definition. Given the strong security guarantees, our protocol is surprisingly efficient: in its most efficient instantiation under the DDH assumption in the random-oracle model, it requires fewer than twenty elliptic-curve exponentiations on the user's device. We achieve our results by careful protocol design and by exclusively focusing on the two-server public-key setting.

1 Introduction

Personal computing has long moved beyond the "one computer on every desk and in every home" to a world where most users own a plethora of devices, each of which is capable of general computation but is better suited for a specific task or environment. However, keeping personal data synchronized across laptops, mobile phones, tablets, portable media players, and other devices is not straightforward. Since most of these have some way of connecting to the Internet, the most obvious solution is to synchronize data "over the cloud". Indeed, many services doing exactly this are commercially available today.

Data synchronization over the cloud poses severe security and privacy threats, however, as the users' whole digital lives are at risk when the cloud host turns out to be malicious or is compromised by an attack. A first solution could be to encrypt the data under a key that is stored on the user's devices but is unknown to the cloud host. This approach has security as well as usability problems: If one of the devices gets lost or stolen, the owner's data is again at risk. Moreover, securely (and most often, manually) entering strong cryptographic keys on devices is tedious and error-prone.

A much better approach is to protect the data under a secret that is associated with the human user such as a human-memorizable password or biometric data. Passwords are still the most prevalent and easily deployable alternative. Although passwords are inherently vulnerable to dictionary attacks, an important distinction must be made between *online* and *offline* dictionary attacks. The former type of attacks, where an attacker simply repeatedly tries to login to an online server, are easily prevented by blocking the account, presenting CAPTCHAs, or enforcing time delays after a number of failed login attempts. Offline attacks, however, allow the adversary to test passwords independently and are therefore more dangerous. With sixteen-character passwords having an estimated 30 bits of entropy [BDN$^+$11] and modern GPUs able to test billions of passwords per second, security should be considered lost as soon as an offline attack can be performed. Therefore, to offer any relevant security, protocols need to be designed such that the correctness of a password can only be tested by interacting with an online server that can refuse cooperation after too many failed attempts.

*An extended abstract of this paper appeared in Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 525–536, ACM Press, October 2012. This is the full version.

One possibility to safely store password-protected data in the cloud is to use a password-authenticated key exchange (PAKE) protocol to establish a secure and authenticated channel with the server and to send and retrieve the data over this channel. There is a considerable amount of literature on single-server PAKE protocols that protect against offline dictionary attacks [BM92, GLNS93, HK99, BPR00, KOY09, CHK$^+$05].

It is easy to see, however, that no single-server scheme can protect against offline dictionary attacks by a malicious or compromised server. A better approach is to secret-share [Sha79] the data as well as the information needed to verify the password across *multiple* servers, and to design the authentication protocol so that no single server (or collusion of servers up to a certain size) learns anything that allows it to perform an offline dictionary attack. This is what a password-authenticated secret sharing (PASS) scheme does.

One way to obtain a PASS scheme is by combining a multi-server PAKE protocol with a secret-sharing scheme so that the user first establishes secure channels with each of the servers using her (single) password, and then submits and retrieves the shares over these channels. Ford and Kaliski [FK00] were the first to propose a multi-server PAKE protocol in a setting where the user remembers her password as well as the public keys of n servers, of which $n-1$ can be compromised. Jablon [Jab01] proposed a similar protocol in the password-only setting, i.e., where the user cannot remember public keys. Brainard et al. [BJKS03] proposed a dedicated two-server protocol in the public-key setting. None of these protocols had formal security notions or proofs, however.

The first provably secure multi-server PAKE protocol, by MacKenzie et al. [MSJ02], is a t-out-of-n protocol supporting $t < n$ malicious servers in the public-key setting. Szydlo and Kaliski [SK05] provided security proofs for slight variants of the two-server protocol by Brainard et al. [BJKS03] mentioned earlier. Di Raimondo and Gennaro [DG03] proposed the first provably secure solution in the password-only model, which was at the same time the first solution not relying on random oracles [BR93] in the security proof. Their protocol tolerates the compromise of $t < n/3$ out of n servers, which means that it cannot be used for two servers—probably the most relevant setting in practice. This gap was filled by Katz et al. [KMTG05], who presented a dedicated two-server PAKE protocol for the password-only setting, also without random oracles.

All the solutions mentioned so far are multi-server PAKE protocols. However, PASS is a simpler primitive than PAKE and so one can hope to obtain more efficient and easier to analyze PASS protocols from scratch, rather than from PAKE protocols. Indeed, Bagherzandi et al. [BJSL11] recently introduced the first direct PASS scheme, supporting coalitions of any $t < n$ out of n servers.

Properly defining security of password-based protocols is a delicate task. The fact that an adversary can always guess a low-entropy password in an online attack means that there is an inherent non-negligible probability of adversarial success; security must therefore be defined as the adversary's inability to do significantly better than that. The highly distributed setting of multi-user and multi-server protocols further complicates the models and proofs. Secure composition is another issue. All provably secure multi-server protocols mentioned above employ property-based security notions that cover the protocol when executed in isolation, but fail to provide guarantees when the protocol is composed with other protocols and network activity. Composing password-based protocols is particularly delicate because the composition of several protocol may amplify the non-negligible adversarial success. Also, human users are much more likely to leak information about their passwords in their online activities than they are to leak information about about their cryptographic keys.

Our Contributions We propose the first two-server password-authenticated secret sharing (2PASS) scheme in the public-key setting that is provably secure in the universal composability (UC) framework [Can01]. We show that, when considering static corruptions and the fact that an adversarial environment necessarily learns whether a protocol succeeded or failed, our notion implies the only existing 2PASS security definition [BJSL11], but that the converse is not true. The UC framework not only guarantees secure composition in arbitrary network environments, but also, as argued before by Canetti et al. [CHK$^+$05] for the case of single-server PAKE, better addresses many other concerns about property-based definitions for password-based protocols. For example, all property-based definitions assume that passwords are generated outside of the adversary's view according to pre-determined, known, and independent distributions. This does not reflect reality at all: users use the same or related passwords across different services, they share passwords with other users, and constantly leak information about their passwords by using them for other purposes. Rather, our UC security notion follows that of Canetti et al. [CHK$^+$05] in letting the environment dictate the parties' passwords and password guesses. As a result, this approach avoids any assumptions on the distribution of passwords, and at the same time incorporates the non-negligible success of online guessing attacks straight into the model, so that secure protocol composition is guaranteed through the universal composition theorem. As another example, our UC definition allows the adversary to observe authentication sessions by honest users who attempt passwords that are related but not equal to their correct passwords. This is a very common situation that arises every time a user mistypes her password; previous definitions fail to model and, consequently, provide security guarantees in this case.

Our model is also the first to explicitly capture throttling mechanisms, i.e., mechanisms to block accounts after a number of failed authentication attempts, or because a particular server is under attack and deems it prudent to temporarily block an account. As we've seen earlier, throttling is crucial to drive a wedge between the efficiency of online and offline attacks. Throttling is impossible for the PASS scheme of Bagherzandi et al. [BJSL11] since the servers do not learn whether the password was correct. The model and protocol for UC-secure single-server PAKE of Canetti et al. [CHK$^+$05] does not explicitly notify servers about the success or failure of an authentication attempt, although it is mentioned that such functionality can be added with a two-round key-confirmation step. In our model, honest servers can decide at each invocation whether to go through with the protocol based on a prompt from the environment.

In summary, we believe that for password-based protocols, UC security not only gives stronger security guarantees under composition, but is actually a more natural, more practically relevant, and less error-prone approach than property-based definitions. In view of these strong security guarantees, our protocol is surprisingly efficient, as we discuss in Section 4. When instantiated based on the decisional Diffie-Hellman assumption in the random-oracle model, it requires the user to perform eighteen modular exponentiations to set up her account and nineteen to retrieve her stored secret.

We believe that this is an exciting research area, with challenging open problems that include strengthening our protocol to withstand adaptive corruptions, designing a UC-secure 2PASS scheme in the password-only (i.e., non-public-key) model, and building UC-secure protocols for the t-out-of-n case.

Versions of this paper. An extended abstract of this paper previously appeared at ACM CCS 2012. This full version contains a detailed security proof with full descriptions of the challengers (game hops) and the simulator. The protocol presented here differs slightly from the one presented in the extended abstract. Namely, the signatures τ_i and τ'_i in Steps S3–5 and Steps R5–7 now also sign the ciphertexts F_i and F'_i, respectively. While we stress that the original protocol is secure, this minor change in the protocol simplifies the security proof for the case of hijacked queries.

2 Definitions

Although intuitively the security properties we want to capture seem clear, giving a rigorous definition for the problem is a challenging task. Numerous subtleties have to be addressed. For example, where does the password come from? Having the user pick her password at random from a dictionary of a particular size does not accurately model the way users pick their passwords. Can any security still be retained if a user is tricked into trying to retrieve another user's key with her correct password? Do two users get any security if their passwords are correlated in some way that is potentially known to an attacker? Do the servers learn anything when a user mistypes a password?

We define the problem by giving an ideal functionality in the universal-composability (UC) framework [Can01, PW00] that captures all the intuitive security properties required in this scenario. The ideal functionality stores a user's password p and a key K. (Without loss of generality, we assume that the only data that users store on and retrieve from the servers are symmetric encryption keys. With those, users can always encrypt data of arbitrary length and store the resulting ciphertext on an untrusted device or in the cloud.) The ideal functionality only reveals the user's key K when presented with the correct password. It notifies the two servers of all attempts (successful and unsuccessful) to retrieve the key, and allows the servers to interrupt the retrieval whenever they deem necessary. As long as one of the servers is not corrupt, the adversary does not learn anything about the user's password or key, unless it can guess her password.

Following the UC framework, we then require that a protocol must not reveal any more information to an adversary than the ideal functionality does, no matter what values users use for their passwords and keys. This is a very strong definition of security: in particular, a protocol satisfying it is guaranteed to remain secure even when run concurrently with any other protocols.

2.1 Ideal Functionality

Preliminaries. A 2PASS scheme operates in a setting with multiple users $\mathcal{U}_i$, $i = 1, \ldots, U$, multiple servers $\mathcal{S}_j$, $j = 1, \ldots, S$, an adversary $\mathcal{A}$ and the environment $\mathcal{E}$. Users in our protocol are stateless, but each server $\mathcal{S}_j$ maintains an associative array $st_j[\cdot]$ containing its local user directory. The scheme is defined by two interactive protocols Setup and $\mathsf{Retrieve}$. A user $\mathcal{U}_i$ performs the Setup protocol with two servers of its choice $\mathcal{S}_j$ and $\mathcal{S}_k$ to store her secret K under username u and password p. Any user $\mathcal{U}_{i'}$, possibly different from $\mathcal{U}_i$, can recover the secret K by running the Retrieve protocol with $\mathcal{S}_j$ and $\mathcal{S}_k$ using the correct username u and password p.

We assume static Byzantine corruptions, meaning that at the beginning of the game the adversary decides which parties, users and servers alike, are corrupted. From then on, the adversary controls all corrupted parties and cannot corrupt any other parties. The ideal functionality "knows" which participants are honest and which ones are corrupt. Without loss of generality, we assume that there is at least one corrupt user through which the adversary can make setup and retrieve queries. Note that since there is no user authentication other than by passwords, in the real world the adversary can always generate such queries by inserting fake messages into the network.

While our protocol clearly envisages a setting where multiple users can create multiple accounts with any combination of servers of their choice, the UC framework allows us to focus on a single session only, i.e., for a single user account. Security for multiple sessions follows from the UC composition theorem [Can01], or if the different sessions share the same common reference string and PKI (as one would prefer in practice), from the joint-state universal composition (JUC) theorem [CR03].

For the protocol specification and security proof, we can therefore focus on a single user account u that is established with two servers $\mathcal{S}_1$ and $\mathcal{S}_2$. The detailed ideal functionality $\mathcal{F}_{2\mathrm{PASS}}$ is given in Figure 1. The triple $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$ is used as the session identifer, but multiple simultaneous setup and retrieve queries by different users may take place within this session. Each setup and retrieve query within this session has a unique query identifier qid. (See below for further discussion on session and query identifiers.) For compactness of notation, we will from now on refer to the functionality $\mathcal{F}_{2\mathrm{PASS}}$ as $\mathcal{F}$.

We recall that in the UC framework, parties are modeled as interactive Turing machines with two ways of communicating with other machines: reliable, authentic communication via the input and subroutine output tapes, and unreliable communication via the incoming and outgoing communication tapes. The former models local communication between processes and their subroutines; we say that one machine *passes/obtains input/output to/from* another machine. The latter models network communication; we say that one machine *sends/receives* a message to/from another machine. The environment passes input to and obtains output from the adversary and regular protocol machines, while protocol machines can pass input to and receive output from their local subroutines. Protocol machines can only send messages to and receive messages from the adversary, who controls all network traffic. The adversary can deliver these messages arbitrarily, meaning that it can modify, delay, or drop messages at will. An ideal functionality is a special protocol machine that is local to all parties except the adversary. Meaning, it interacts with all regular protocol machines through its input/output tapes and with the adversary through its communication tapes.

The ideal functionality maintains state by creating "records" and by "marking" these records. The state is local to a single instance of $\mathcal{F}$, i.e., for a single session identifier $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$ defining a single user account. The multi-session functionality keeps separate state for each user account. The functionality also keeps a two-dimensional associative array $mark[\cdot,\cdot]$. When we say that query qid is *marked* X for party $\mathcal{P}$, we mean that the entry $mark[qid, \mathcal{P}]$ is assigned the value X.

Clarification. Through the *Setup Request* interface, a user $\mathcal{U}$ can initiate the creation of an account with username u and servers $\mathcal{S}_1$ and $\mathcal{S}_2$ to store a secret K protected with password p. If at least one server is honest, p and K remain hidden from the adversary; if both servers are corrupt, $\mathcal{F}$ sends p and K to the adversary. Since the environment instructs users to create accounts and since the adversary controls the network, multiple setup queries may be going on concurrently. The different queries are distinguished by means of a query identifier qid that $\mathcal{U}$, $\mathcal{S}_1$, and $\mathcal{S}_2$ agree on upfront. (See further discussion below.)

Since agreeing on a query identifier does not mean that a secure channel has been established, the adversary can always "hijack" the query by intercepting the user's network traffic and substituting it with its own. This is modeled by the *Setup Hijack* interface, using which the adversary can replace the password p and key K for a query qid with its own. The user will always output `fail` after a query was hijacked, but the servers do not notice the difference with a regular setup.

The adversary controls when a server or user learns whether the setup succeeded or failed through the *Setup Result Server* and *Setup Result User* interfaces. Once the adversary lets a setup succeed for an honest server, this server will refuse all further setups. The adversary can always make setup transactions fail for a subset of the participants, but the user will only output that setup succeeded if all honest servers also concluded the setup successfully and if the query was not hijacked.

A user $\mathcal{U}'$ (possibly different from $\mathcal{U}$) can recover the secret key K by calling the *Retrieve Request* interface with a password attempt p'. If at least one server is honest, then no party learns p'; if both are corrupt, then

Functionality $\mathcal{F}_{\text{2PASS}}$

Setup Request: Upon input $(\texttt{Stp}, sid, qid, p, K)$ from $\mathcal{U}$, check that $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$ for some $u \in \{0,1\}^*$ and for some server identities $\mathcal{S}_1, \mathcal{S}_2$, and check that query identifier qid is unique. Create a record $(\texttt{AStp}, qid, \mathcal{U}, p, K)$. If $\mathcal{S}_1$ and $\mathcal{S}_2$ are corrupt then send $(\texttt{Stp}, sid, qid, \mathcal{U}, p, K)$ to the adversary. Otherwise, send $(\texttt{Stp}, sid, qid, \mathcal{U})$ to the adversary.

Setup Hijack: Upon input $(\texttt{SHjk}, sid, qid, \hat{p}, \hat{K})$ from the adversary $\mathcal{A}$ for $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$, check that a record $(\texttt{AStp}, qid, \mathcal{U}, p, K)$ exists and that query qid has not been marked for any of $\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{A}\}$. Mark query qid as $\texttt{hjkd}$ for $\mathcal{A}$ and replace record $(\texttt{AStp}, qid, \mathcal{U}, p, K)$ with $(\texttt{AStp}, qid, \mathcal{U}, \hat{p}, \hat{K})$.

Setup Result Server: When receiving $(\texttt{SRlt}, sid, qid, \mathcal{S}, s)$ from the adversary for $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$, for an honest server $\mathcal{S} \in \{\mathcal{S}_1, \mathcal{S}_2\}$, and for $s \in \{\texttt{succ}, \texttt{fail}\}$, check that a record $(\texttt{AStp}, qid, \cdot, \cdot, \cdot)$ exists. If query qid is already marked $\texttt{succ}$ or $\texttt{fail}$ for $\mathcal{S}$, or if some other setup query is already marked $\texttt{succ}$ for $\mathcal{S}$, then do nothing. Else, mark query qid as s for $\mathcal{S}$ and output $(\texttt{SRlt}, sid, qid, s)$ to $\mathcal{S}$. If now query qid is marked $\texttt{succ}$ for all honest servers among $\mathcal{S}_1$ and $\mathcal{S}_2$, then record $(\texttt{Stp}, p, K)$.

Setup Result User: When receiving $(\texttt{SRlt}, sid, qid, \mathcal{U}, s)$ from the adversary for $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$, for an honest user $\mathcal{U}$, and for $s \in \{\texttt{succ}, \texttt{fail}\}$, check that a record $(\texttt{AStp}, qid, \mathcal{U}, \cdot, \cdot)$ exists that is not yet marked for $\mathcal{U}$. If it is marked $\texttt{succ}$ for all honest servers and not marked for $\mathcal{A}$, then mark it s for $\mathcal{U}$ and output $(\texttt{SRlt}, sid, qid, s)$ to $\mathcal{U}$; else, mark it $\texttt{fail}$ for $\mathcal{U}$ and output $(\texttt{SRlt}, sid, qid, \texttt{fail})$ to $\mathcal{U}$.

Retrieve Request: Upon input $(\texttt{Rtr}, sid, qid', p')$ from $\mathcal{U}'$, check that $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$ and that query identifier qid' is unique. Create a record $(\texttt{ARtr}, qid', \mathcal{U}', p')$. If $\mathcal{S}_1$ and $\mathcal{S}_2$ are both corrupt then send $(\texttt{Rtr}, sid, qid', \mathcal{U}', p')$ to the adversary, else send $(\texttt{Rtr}, sid, qid', \mathcal{U}')$ to the adversary.

Retrieve Hijack: When receiving $(\texttt{RHjk}, sid, qid', \hat{p}')$ from the adversary for $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$, check that a record $(\texttt{ARtr}, qid', \mathcal{U}', p')$ exists and that query qid' has not been marked for any of $\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{A}\}$. Mark query qid' as $\texttt{hjkd}$ for $\mathcal{A}$ and replace record $(\texttt{ARtr}, qid', \mathcal{U}', p')$ with $(\texttt{ARtr}, qid', \mathcal{U}', \hat{p}')$.

Retrieve Notification: When receiving $(\texttt{RNtf}, sid, qid', \mathcal{S}_i)$ from the adversary for $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$ and for an honest server $\mathcal{S}_i \in \{\mathcal{S}_1, \mathcal{S}_2\}$, check that a record $(\texttt{ARtr}, qid', \cdot, \cdot)$ exists. If there exists a setup query that is marked $\texttt{succ}$ for $\mathcal{S}_i$ then output $(\texttt{RNtf}, sid, qid')$ to $\mathcal{S}_i$. Else, create a record $(\texttt{Perm}, qid', \mathcal{S}_i, \texttt{deny})$, output $(\texttt{RRlt}, sid, qid', \texttt{fail})$ to $\mathcal{S}_i$, and mark qid' as $\texttt{fail}$ for $\mathcal{S}_i$.

Retrieve Permission: Upon input $(\texttt{Perm}, sid, qid', a)$ from $\mathcal{S}_i \in \{\mathcal{S}_1, \mathcal{S}_2\}$, where $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$ and $a \in \{\texttt{allow}, \texttt{deny}\}$, check that a record $(\texttt{ARtr}, qid', \mathcal{U}', p')$ exists and that no record $(\texttt{Perm}, qid', \mathcal{S}_i, \cdot)$ exists. Create a record $(\texttt{Perm}, qid', \mathcal{S}_i, a)$ and send $(\texttt{Perm}, sid, qid', \mathcal{S}_i, a)$ to the adversary.

If now a record $(\texttt{Perm}, qid', \mathcal{S}_i, \texttt{allow})$ exists for all honest $\mathcal{S}_i \in \{\mathcal{S}_1, \mathcal{S}_2\}$, then send $(\texttt{Rtr}, sid, qid', c, K'')$ to the adversary, where $(c, K'') \leftarrow (\texttt{correct}, K)$ if a record $(\texttt{Stp}, p, K)$ exists, $p' = p$, and either $\mathcal{U}'$ is corrupt or qid' is marked $\texttt{hjkd}$ for $\mathcal{A}$; where $(c, K'') \leftarrow (\texttt{correct}, \perp)$ if a record $(\texttt{Stp}, p, \cdot)$ exists, $p' = p$, $\mathcal{U}'$ is honest, and qid' is not marked for $\mathcal{A}$; and where $(c, K'') \leftarrow (\texttt{wrong}, \perp)$ otherwise.

Retrieve Result Server: Upon receiving $(\texttt{RRlt}, sid, qid', \mathcal{S}_i, a)$ from the adversary for $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$, for an honest server $\mathcal{S}_i \in \{\mathcal{S}_1, \mathcal{S}_2\}$, and for $a \in \{\texttt{allow}, \texttt{deny}\}$, check that records $(\texttt{ARtr}, qid', \cdot, p')$ and $(\texttt{Perm}, qid', \mathcal{S}_i, a_i)$ exist, and that query qid' is not yet marked for $\mathcal{S}_i$.

Output $(\texttt{RRlt}, sid, qid', s)$ to $\mathcal{S}_i$ and mark query qid' as s for $\mathcal{S}_i$, where $s \leftarrow \texttt{succ}$ if $a = \texttt{allow}$, a record $(\texttt{Stp}, p, \cdot)$ exists, records $(\texttt{Perm}, qid', \mathcal{S}_j, \texttt{allow})$ exist for all honest servers $\mathcal{S}_j \in \{\mathcal{S}_1, \mathcal{S}_2\}$, and $p' = p$. Otherwise, $s \leftarrow \texttt{fail}$.

Retrieve Result User: Upon receiving $(\texttt{RRlt}, sid, qid', \mathcal{U}', a, K')$ from the adversary for honest user $\mathcal{U}'$, where $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$, $a \in \{\texttt{allow}, \texttt{deny}\}$, and $\mathcal{S}_i \in \{\mathcal{S}_1, \mathcal{S}_2\}$, check that record $(\texttt{ARtr}, qid', \mathcal{U}', p')$ exists and that query qid' is not yet marked for $\mathcal{U}'$. Output $(\texttt{RRlt}, sid, qid', K'')$ to $\mathcal{U}'$ where

- $K'' \leftarrow \perp$ if $a = \texttt{deny}$; else,
- $K'' \leftarrow K'$ if $\mathcal{S}_1$ and $\mathcal{S}_2$ are corrupt and $a = \texttt{allow}$; else,
- $K'' \leftarrow K$ if qid' is marked $\texttt{succ}$ for $\mathcal{S}_1$ and $\mathcal{S}_2$ and is not marked for $\mathcal{A}$; else,
- $K'' \leftarrow \perp$.

If $K'' = \perp$ then mark query qid' as $\texttt{fail}$ for $\mathcal{U}'$, else mark it as $\texttt{succ}$ for $\mathcal{U}'$.

Figure 1: Ideal functionality for retrieve of 2PASS protocols.

Functionality $\mathcal{F}_{CA}$

Registration: Upon receiving the first message ($\texttt{Register}, sid, v$) from party $\mathcal{P}$, send ($\texttt{Registered}, sid, v$) to the adversary; upon receiving $\texttt{ok}$ from the adversary, and if $sid = \mathcal{P}$ and this is the first request from $\mathcal{P}$, then record the pair $(\mathcal{P}, v)$.

Retrieve: Upon receiving a message ($\texttt{Retrieve}, sid$) from party $\mathcal{P}'$, send ($\texttt{Retrieve}, sid, \mathcal{P}'$) to the adversary, and wait for an $\texttt{ok}$ from the adversary. Then, if there is a recorded pair (sid, v) output ($\texttt{Retrieve}, sid, v$) to $\mathcal{P}'$. Otherwise output ($\texttt{Retrieve}, sid, \perp$) to $\mathcal{P}'$.

Figure 2: Ideal certification functionality.

p' is sent to the adversary. Similarly to setup queries, the adversary can hijack the retrieve query through the *Retrieve Hijack* interface and replace p' with its own $\hat{p}'$.

When the adversary notifies a server of a retrieve request via the *Retrieve Notification* interface, the server outputs a ($\texttt{RNtf}, \ldots$) message. At this point, the server can apply any external throttling mechanism to decide whether to participate in this retrieval, e.g., by refusing to do so after too many failed attempts. The servers indicate whether they proceed with the retrieval through the *Retrieve Permission* interface. Only after all honest servers have allowed the transaction to proceed does the adversary learn whether the password was correct and, if the password is correct and either the user $\mathcal{U}'$ is corrupt or the query was hijacked, also the key K.

The adversary decides at which point the results of the retrieval are delivered to the parties by invoking the *Retrieve Result Server* and *Retrieve Result User* interfaces. The adversary can always make a party fail by setting $a = \texttt{deny}$, even if $p' = p$, but cannot make the retrieval appear successful if $p' \neq p$. This reflects the fact that in the real world, the adversary can always tamper with communication to make a party fail, but cannot force an honest party to succeed, unless he knows the password.

If both servers are corrupt, then the adversary can force the user to succeed with any key K' of the adversary's choice. If at least one server is honest, however, then $\mathcal{F}$ either sends the real recorded key K to $\mathcal{U}'$, or sends it a $\texttt{fail}$ message. The adversary doesn't learn anything about p' or K, and the user can only obtain K if all honest servers participated in the retrieval and the password was correct.

2.2 Discussion

On session and query identifiers. The UC framework imposes that the session identifier *sid* be globally unique. The security proof considers a single instance of the protocol in isolation, meaning that in the security proof, all calls to the ideal functionality have the same *sid*. For 2PASS protocols, the *sid* must be (1) the same for setup and retrieval, so that the ideal functionality can keep state between these phases, and (2) human-memorizable, so that a human user can recover her secret key K based solely on information she can remember. We therefore model *sid* to consist of a user name u and the two server identities $\mathcal{S}_1, \mathcal{S}_2$. Together, these uniquely define a "user account". To ensure that *sid* is unique, servers reject setups for combinations of a username and two servers for which they already have stored state.

Within a single user account (i.e., a single *sid*), multiple setup and retrieve protocol executions may be going on concurrently. To distinguish the different protocol executions, we let the environment specify a unique (within this *sid*) query identifier *qid* when the execution is first initialized by the user. The *qid* need not be human-memorizable, so it can be agreed upon like any session identifier in the UC framework, e.g., by running an initialization protocol that implements $\mathcal{F}_{init}$ as defined by Barak et al. [BLR04].

As mentioned above, security for multiple user accounts is obtained through the JUC theorem [CR03]. In the multi-session functionality $\hat{\mathcal{F}}_{\text{2PASS}}$, the tuple $(u, \mathcal{S}_1, \mathcal{S}_2)$ becomes the sub-session identifier *ssid*, whereas the session identifier *sid* is a unique string that specifies the "universe" in which the multi-session protocol operates, describing for example which CRS to use and which PKI to trust. In practice, the *sid* of the multi-session functionality can be thought of as hardcoded in the software that users use to set up and retrieve their accounts, so that human users need not remember it.

Strengthening the definition. If both servers are corrupt, our ideal functionality hands the password p, the key K, and all password attempts p' to the adversary. Giving away the passwords and key "for free" is a somewhat conservative model for the fact that two corrupt servers can always perform an offline dictionary attack on p—a model that, given the low entropy in human-memorizable passwords and the efficiency of brute-force attacks, is unfortunately quite close to reality. At the same time, it allows for efficient instantiations such as ours that let passwords do what they do best, namely protect against online attacks. One could further strengthen the definition in the spirit of Canetti et al. [CHK$^+$05] by merely giving the adversary access to an offline password testing interface that returns K only when called with the correct password p. Protocols satisfying this stronger notion will have to use a very different and most likely less efficient approach than ours, but would have the benefit of offering some protection when both servers are corrupt but a very strong password is used, and of hiding the password attempt p' when talking to the wrong servers.

Relation to existing notions. The only existing security notion for 2PASS is due to Bagherzandi et al. [BJSL11]. In the static corruption case, if we bear in mind that an adversarial environment will necessarily learn whether the retrieval succeeded or failed, our ideal functionality meets their security definition, so our notion implies it. The notion of Bagherzandi et al. does not imply ours, however, because it fails to capture related-password attacks.

To see why this is true, consider the following (contrived) scheme that satisfies Bagerzandi et al.'s definition but is insecure against a related-password attack. Take a scheme that is secure under the existing notion [BJSL11]. Consider a modified scheme where, if the user's input password starts with 1, the user sends the password in the clear to both servers; else, follow the normal protocol. This scheme still satisfies their definition for the dictionary of passwords starting with 0: their definition does not consider the case when the honest user inputs an incorrect password. It does not satisfy our definition, however: suppose the environment directs a user whose correct password is $0\|p$ to perform a retrieve with password $1\|p$. In the real protocol, a dishonest server involved in the protocol will see the string $1\|p$. In the ideal world, the ideal functionality hides an incorrect password from the servers, and so no simulator will be able to correctly simulate this scenario.

2.3 Setup Assumptions

Our protocol requires two setup assumptions. The first is the availability of a public common reference string (CRS), modeled by an ideal functionality $\mathcal{F}_{CRS}^{D}$ parameterized with a distribution D. Upon receiving input $(\mathtt{CRS}, sid)$ from $\mathcal{P}$, if no value r is recorded, it chooses and records $r \leftarrow_{\mathrm{R}} D$. It then sends $(\mathtt{CRS}, sid, r)$ to $\mathcal{P}$.

The second is the existence of some form of public-key infrastructure where servers can register their public keys and the user can look up these public keys. The user can thus authenticate the servers so that she can be sure that she runs the retrieve protocol with the same servers that she previously ran the setup protocol with. In other words, we assume the availability of the functionality $\mathcal{F}_{CA}$ by Canetti [Can04] depicted in Figure 2. We will design our protocol in a hybrid world where parties can make calls to $\mathcal{F}_{CA}$.

3 Our Protocol

Let GGen be a probabilistic polynomial-time algorithm that on input security parameter 1^k outputs the description of a cyclic group $\mathbb{G}$, its prime order q, and a generator g.

Let $(\mathsf{keyg}, \mathsf{enc}, \mathsf{dec})$ be a semantically secure public-key encryption scheme with message space $\mathbb{G}$; we write $c = \mathsf{enc}_{pk}(m; r)$ to denote that c is an encryption of m with public key pk using randomness r. Our protocol will require this cryptosystem to (1) have committing ciphertexts, so that it can serve as a commitment scheme; (2) have appropriate homomorphic properties (that will become clear in the sequel); (3) have an efficient simulation-sound zero-knowledge proof of knowledge system for proving certain relations among ciphertexts (which properties are needed will be clear in the sequel) and for proving correctness of decryption. The ElGamal cryptosystem [ElG85] satisfies all the properties we need.

Let $(\mathsf{keygsig}, \mathsf{sig}, \mathsf{ver})$ be a signature scheme with message space $\{0,1\}^*$ secure against adaptive message attacks and let $(\mathsf{keyg2}, \mathsf{enc2}, \mathsf{dec2})$ be a CCA2 secure public key encryption scheme with message space $\{0,1\}^*$ that supports labels. To denote an encryption of m with public key pk and label $l \in \{0,1\}^*$, we write $c = \mathsf{enc2}_{pk}(m, l)$. When employing these schemes, we assume suitable (implicit) mappings from (tuples of) elements from $\mathbb{G}$ to $\{0,1\}^*$.

A simulation-sound [Sah99] zero-knowledge protocol is essentially a zero-knowledge proof protocol where the adversary cannot create a new valid proof of a false statement, even after seeing several valid proofs of false statements produced by the simulator. More formally, let $\Phi(v)$ be a predicate over a value v and let $R_\Phi = \{(w, v)\}$ be the associated witness relation so that $\Phi(v) = 1 \Leftrightarrow \exists w : (w, v) \in R_\Phi$. A protocol Π for predicate Φ is a tuple of algorithms $\Pi = (\mathsf{P}, \mathsf{V}, \mathsf{Sim})$ where the prover P on input v and w interacts with the verifier V on input v, denoted $\mathsf{P}(v, w) \leftrightarrow \mathsf{V}(v)$, until the latter outputs 0 or 1. The zero-knowledge simulator Sim, on input a value v and some trapdoor information, produces a valid proof for v. We require that the protocol satisfies the following properties:

Complete: For all $(w, v) \in R_\Phi \;:\; \mathsf{P}(v, w) \leftrightarrow \mathsf{V}(v) = 1$.

Simulation-sound: For all $\mathcal{A}$ running in time polynomial in k, the probability that $\mathcal{A}^{\mathsf{Sim}(\cdot)} \leftrightarrow \mathsf{V}(v) = 1$ for a value v, chosen by $\mathcal{A}$, such that $\Phi(v) = 0$ and v was never queried to $\mathsf{Sim}(\cdot)$, is negligible in k.

Concurrent zero-knowledge: For all $\mathcal{A}$ running in time polynomial in k, $\left|\Pr[\mathcal{A}^{\mathsf{P}(\cdot,\cdot)}] - \Pr[\mathcal{A}^{\mathsf{Sim}'(\cdot,\cdot)}]\right|$ is negligible in k, where $\mathsf{Sim}'(v, w) = \mathsf{Sim}(v)$.

3.1 High-Level Idea

The main idea underlying our protocol is similar to the general approach of Brainard et al. [BJKS03]: in the setup protocol, the user sends secret shares of her password and her key to each of the servers. To retrieve the shares of her key, the user in the retrieve protocol sends new secret shares of her password to the servers. These then run a protocol to determine whether the secrets received in the retrieve protocol and those in the setup protocol are shares of the same password. If so, they send the secret shares of the key to the user.

This basic idea is very simple; the challenge in the design of our protocol is to implement this idea efficiently and in a way that can be proved secure in the UC model. We first explain how this is achieved on a high level and then describe our protocols in detail.

Setup protocol. The servers $\mathcal{S}_1$ and $\mathcal{S}_2$ receive from the user secret shares p_1 and p_2, respectively, of the user's password $p = p_1 p_2$, and, similarly, secret shares K_1 and K_2 of the user's symmetric key $K = K_1 K_2$. To make sure that during the retrieval a malicious server cannot substitute different values for the password and key share, $\mathcal{S}_1$ additionally receives from the user commitments C_2 and $\tilde{C}_2$ of the shares p_2 and K_2, while $\mathcal{S}_2$ is given the opening information $s_2, \tilde{s}_2$ for both commitments. Similarly, $\mathcal{S}_2$ receives two commitments C_1 and $\tilde{C}_1$ to the shares p_1 and K_1, while $\mathcal{S}_1$ is given the corresponding opening information $s_1, \tilde{s}_1$. Later, during the retrieve protocol, the servers will have to prove that they are behaving correctly with respect to these commitments.

To create the commitments and to be able to achieve UC security, we rely on the CRS model by encrypting the values using randomness $s_i, \tilde{s}_i$ under a public key PK given by the CRS, for which nobody knows the corresponding decryption key.

To communicate the secret shares and the opening information to the servers securely, the user will encrypt them under the servers' public keys (which she looks up via the $\mathcal{F}_{CA}$ functionality). This is not enough, however. To prevent a malicious server from substituting different values for the password and key share, we make use of the labels of the CCA2-secure encryption scheme, to bind the encryptions to the specific instance of the protocol, in particular to the commitments C_1, $\tilde{C}_1$, C_2, and $\tilde{C}_2$. To signal to the user that the setup has worked, the servers will send her a signed statement.

Retrieve protocol. The user re-shares the password guess $p' = p'_1 p'_2$ and gives p'_1 and p'_2 to servers $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. In addition, she gives $\mathcal{S}_1$ and $\mathcal{S}_2$ commitments C'_1 and C'_2 to p'_1 and p'_2. She hands the opening information s'_1 for C'_1 to $\mathcal{S}_1$ and s'_2 for C'_2 to $\mathcal{S}_2$. The user also generates an ephemeral key pair (PK_u, SK_u) of a semantically secure encryption scheme and sends the public key to the servers.

Then, $\mathcal{S}_1$ and $\mathcal{S}_2$ jointly compute the following randomized two-party function: on public input (C_1, C_2, C'_1, C'_2) and with each server having his password shares and opening information as private inputs, output 1 if (1) $C_i = \mathsf{enc}(p_i; s_i)$ for $i \in \{1, 2\}$; (2) $C'_i = \mathsf{enc}(p'_i; s'_i)$ for $i \in \{1, 2\}$; (3) $p_1 p_2 = p'_1 p'_2$. Otherwise, output a random element of the group $\mathbb{G}$. If the output is 1, each server sends to the user his share of K encrypted under PK_u.

Let us explain how this two-party computation is done in a way that is both efficient and secure in the UC model. As the first idea, consider the following approach: $\mathcal{S}_1$ forms a ciphertext E_1 of the group element $\delta_1 = p_1/p'_1$, and sends E_1 to $\mathcal{S}_2$. $\mathcal{S}_2$ uses the homomorphic properties of the underlying cryptosystem to obtain $E = E_1 \times E_2$, where E_2 is an encryption of $\delta_2 = p'_2/p_2$. Now E is an encryption of 1 if and only if $p'_1 p'_2 = p_1 p_2$, i.e., if the user's password matches. However, there are three issues: (1) How do $\mathcal{S}_1$ and $\mathcal{S}_2$ decrypt E? (2) How do we make sure that they don't learn anything if the user submitted an incorrect password? (3) How do we make sure that the servers do not deviate from this protocol?

To address (1), we have $\mathcal{S}_1$ generate a temporary public key pk for which it knows the secret key, and so now the ciphertexts E_1, E_2 and E are formed under this temporary public key. This way, $\mathcal{S}_1$ will be able to decrypt E when he receives it. To address (2), our protocol directs $\mathcal{S}_2$ to form E somewhat differently; specifically, by computing $E = (E_1 \times E_2)^z$ for a random $z \in \mathbb{Z}_q$. Now if the password the user has submitted was correct, the decryption of E will still yield 1. However, if it was incorrect, it will be a truly random element of $\mathbb{G}$. Finally, to address (3), $\mathcal{S}_1$ and $\mathcal{S}_2$ must prove to each other, at every step, that the messages they are sending to each other are computed correctly.

As in the Setup protocol, the user encrypts the secret shares and the opening information under the server's public keys (which she looks up via the $\mathcal{F}_{CA}$ functionality). She uses the commitments C'_1, C'_2 and the ephemeral public key PK_u as a label for these ciphertexts. As we will see in the proof of security, owing to the security properties of labelled CCA2 encryption, if the shares are correct the servers can safely use PK_u to encrypt their shares of K. To ensure that the servers encrypt and send the correct shares, they first convince each other that their respective encryptions are consistent with the commitments of the shares received from the user in the Setup protocols. To inform the user of the encryptions' correctness, each server sends to the user a signature of *both* encryptions and the commitments C'_1, C'_2 received just now. Thus a malicious server will be unable to substitute K with a key different from what was stored during setup.

3.2 Protocol Details

We assume that the common reference string functionality $\mathcal{F}_{CRS}$ describes a group $\mathbb{G}$ of prime order q and generator g generated through $\mathsf{GGen}(1^k)$, together with a public key PK of $(\mathsf{keyg}, \mathsf{enc}, \mathsf{dec})$ for which the corresponding secret key is unknown. We also assume the presence of certified public keys for all servers in the system through $\mathcal{F}_{CA}$; we do not require users to have such public keys. More precisely, we assume each server $\mathcal{S}_i$ to have generated key pairs (PE_i, SE_i) and (PS_i, SS_i) for $(\mathsf{keyg2}, \mathsf{enc2}, \mathsf{dec2})$ and $(\mathsf{keygsig}, \mathsf{sig}, \mathsf{ver})$, respectively, and to have registered the public keys by calling $\mathcal{F}_{CA}$ with $(\mathtt{Register}, \mathcal{S}_i, (PE_i, PS_i))$.

Our retrieve protocol requires the servers to prove to each other the validity of some statements that encryptions were correctly computed. In the description of the protocol we denote the proof protocol that a predicate $\Phi(v) = 1$ for a public value v and witness w as $\mathrm{ZK}\{(w) : (w, v) \in R_\Phi\}$. One can attach a "label" λ to a proof protocol by regarding the proof protocol for the conjunction predicate $\Phi(v) = 1 \wedge \mathit{label} = \lambda$; we write $\mathrm{ZK}\{(w) : (w, v) \in R_\Phi\}(\lambda)$ as a shorthand notation. Note that since the label is part of the predicate, simulation-soundness implies that an adversary, after seeing a proof of an invalid statement for one label, cannot create a proof for an invalid statement of a different label. We provide the concrete instantiation of these protocols and the encryption schemes that we use in Section 4. For now we only require that the protocols are concurrent zero-knowledge and simulation-sound proofs. We refer to Section 4 for more details on how this can be achieved.

We assume the following communication and process behavior. The servers are listening on some standard port for protocol messages. As we do not assume secure channels, messages can arrive from anyone. All messages that the parties send to each other are tagged by $(\mathtt{Stp}, \mathit{sid}, \mathit{qid}, i)$ or $(\mathtt{Rtr}, \mathit{sid}, \mathit{qid}, i)$ where i is a sequence number indicating the step in the respective protocol. All other messages received on that port will be dropped. Also dropped are messages that cannot be parsed according to the format for the protocol step corresponding to the tag a message carries and messages which have the same tag as a message that has already been received. The tags are used to route the message to the different protocol instances, and are only delivered to a protocol instance in the order of the sequence number. If they arrive out of sequence, the messages are buffered until they can be delivered in sequence. If a server receives a message from the user with a fresh tag $(\mathtt{Stp}, \mathit{sid}, \mathit{qid})$ or $(\mathtt{Rtr}, \mathit{sid}, \mathit{qid})$, and sequence number 1, it starts a new instance of the respective protocol, or drops the message if an instance qid is already running.

3.2.1 The Setup Protocol

All parties have access to the system parameters including the group $\mathbb{G}$ and the public key PK through $\mathcal{F}_{CRS}$. We assume that each server $\mathcal{S}_i$ keeps internal persistent storage st_i.

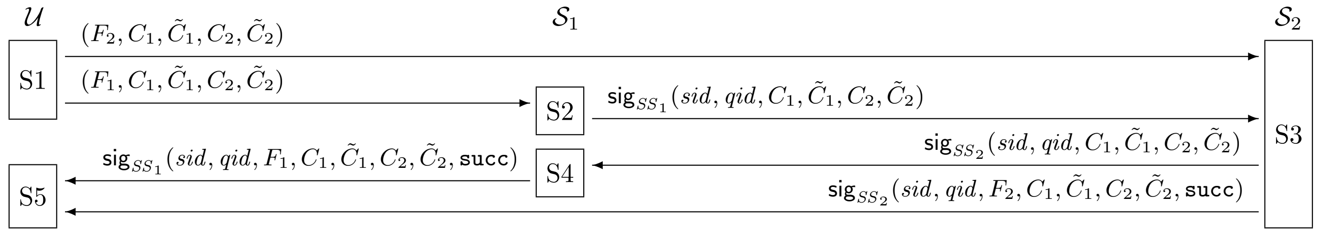
The input to $\mathcal{U}$ is $(\mathtt{Stp}, \mathit{sid}, \mathit{qid}, p, K)$, where $\mathit{sid} = (u, \mathcal{S}_1, \mathcal{S}_2)$, u is the chosen username, p is the user's chosen password, and K the key to be stored. We assume that both p and K are encoded as elements of $\mathbb{G}$. Whenever a test fails, the user or server sends $(\mathtt{Stp}, \mathit{sid}, \mathit{qid}, \mathtt{fail})$ to the other parties and aborts with output $(\mathtt{SRlt}, \mathit{sid}, \mathit{qid}, \mathtt{fail})$. Furthermore, whenever any party receives a message $(\mathtt{Stp}, \mathit{sid}, \mathit{qid}, \mathtt{fail})$, it aborts with output $(\mathtt{SRlt}, \mathit{sid}, \mathit{qid}, \mathtt{fail})$. The structure of the Setup protocol is depicted in Figure 3; the individual steps are as follows.

Step S1*: On input* $(\mathtt{Stp}, \mathit{sid}, \mathit{qid}, p, K)$*, user* $\mathcal{U}$ *performs the following computations.*

(a) *Obtain public keys of the servers and CRS:* Query $\mathcal{F}_{CRS}$ to receive PK and query $\mathcal{F}_{CA}$ with $(\mathtt{Retrieve}, \mathit{sid}, \mathcal{S}_1)$ and $(\mathtt{Retrieve}, \mathit{sid}, \mathcal{S}_2)$ to receive (PE_1, PS_1) and (PE_2, PS_2).

(b) *Compute shares of password and key:* choose $p_1 \leftarrow_R \mathbb{G}$ and $K_1 \leftarrow_R \mathbb{G}$ and compute $p_2 \leftarrow p/p_1$ and $K_2 \leftarrow K/K_1$.

(c) *Encrypt shares under the CRS and the public keys of the servers:* Choose randomness $s_1, s_2, \tilde{s}_1, \tilde{s}_2 \leftarrow_R \mathbb{Z}_q$, encrypt shares of p and K under the CRS as $C_1 \leftarrow \mathsf{enc}_{PK}(p_1; s_1)$, $\tilde{C_1} \leftarrow \mathsf{enc}_{PK}(K_1; \tilde{s}_1)$, $C_2 \leftarrow \mathsf{enc}_{PK}(p_2; s_2)$, and $\tilde{C_2} \leftarrow \mathsf{enc}_{PK}(K_2; \tilde{s}_2)$, and encrypt shares and randomness under the servers' public keys as $F_1 \leftarrow \mathsf{enc2}_{PE_1}((p_1, K_1, s_1, \tilde{s}_1), (\mathit{sid}, \mathit{qid}, C_1, \tilde{C_1}, C_2, \tilde{C_2}))$ and $F_2 \leftarrow \mathsf{enc2}_{PE_2}((p_2, K_2, s_2, \tilde{s_2},), (\mathit{sid}, \mathit{qid}, C_1, \tilde{C_1}, C_2, \tilde{C_2}))$.



$\mathcal{U}$
$\mathcal{S}_1$
$\mathcal{S}_2$
S1
S2
S3
S4
S5
$(F_2, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$
$(F_1, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$
$\mathsf{sig}_{SS_1}(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$
$\mathsf{sig}_{SS_2}(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$
$\mathsf{sig}_{SS_1}(sid, qid, F_1, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ})$
$\mathsf{sig}_{SS_2}(sid, qid, F_2, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ})$

Figure 3: Communication messages of the Setup protocol with computation steps Si.

(d) *Send encryptions to servers:* Send $(F_1, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ to $\mathcal{S}_1$ and $(F_2, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ to $\mathcal{S}_2$.

Step S2*: The first server $\mathcal{S}_1$ proceeds as follows.*

(a) *Receive message from user and check if fresh instance:* Parse the received message as $(\mathtt{Stp}, sid, qid, 1, F_1, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$.
(b) *Obtain public keys of the second server:* Query $\mathcal{F}_{CA}$ with $(\mathtt{Retrieve}, sid, \mathcal{S}_2)$ to receive (PE_2, PS_2).
(c) *Decrypt shares and randomnes:* Decrypt F_1 with label $(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$, which will fail if the label is wrong.
(d) *Verify correct encryption of shares under CRS:* Check whether $C_1 = \mathsf{enc}_{PK}(p_1; s_1)$ and $\tilde{C}_1 = \mathsf{enc}_{PK}(K_1; \tilde{s}_1)$.
(e) *Verify that this is a new instance:* Check that there is no entry $st_1[sid]$ in the state.
(f) *Inform second server that all checks were successful:* Compute the signature $\sigma_1 \leftarrow \mathsf{sig}_{SS_1}(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ and send it to $\mathcal{S}_2$.

Step S3*: The second server $\mathcal{S}_2$ proceeds as follows.*

(a) *Receive message from user and first server:* Parse the message received from $\mathcal{U}$ as $(\mathtt{Stp}, sid, qid, 1, F_2, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ and the message from $\mathcal{S}_1$ as $(\mathtt{Stp}, sid, qid, 2, \sigma_1)$.
(b) *Obtain public keys of $\mathcal{S}_1$:* Send $(\mathtt{Retrieve}, sid, \mathcal{S}_1)$ to $\mathcal{F}_{CA}$ to obtain (PE_1, PS_1).
(c) *Decrypt shares and randomness:* Decrypt F_2 with label $(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$, which will fail if the label is wrong.
(d) *Verify correct encryption of shares under CRS:* Check whether $C_2 = \mathsf{enc}_{PK}(p_2; s_2)$ and $\tilde{C}_2 = \mathsf{enc}_{PK}(K_2; \tilde{s}_2)$.
(e) *Verify that this is a new instance:* Check that there is no entry $st_2[sid]$ in the state.
(f) *Verify first server's confirmation:* Check that $\mathsf{ver}_{PS_1}((sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2), \sigma_1) = 1$.
(g) *Inform first server of acceptance:* Compute signature $\sigma_2 \leftarrow \mathsf{sig}_{SS_2}(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ and send σ_2 to $\mathcal{S}_1$.
(h) *Inform user of acceptance:* Compute signature $\tau_2 \leftarrow \mathsf{sig}_{SS_2}(sid, qid, F_2, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ})$ and send τ_2 to $\mathcal{U}$ and $\mathcal{S}_1$.
(i) *Update state and exit:* Update state $st_2[sid] \leftarrow (PS_1, p_2, K_2, s_2, \tilde{s}_2, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ and output $(\mathtt{SRlt}, sid, qid, \mathtt{succ})$.

Step S4*: The first server $\mathcal{S}_1$ proceeds as follows.*

(a) *Receive message from second server:* Parse the message received from $\mathcal{S}_2$ as τ_2.
(b) *Verify second server's confirmation:* Check that $\mathsf{ver}_{SS_2}((sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2), \sigma_2) = 1$.
(c) *Inform user of acceptance:* Compute $\tau_1 \leftarrow \mathsf{sig}_{SS_1}(sid, qid, F_1, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ})$ and send τ_1 to $\mathcal{U}$.
(d) *Update state and exit:* Update state $st_1[sid] \leftarrow (PS_2, p_1, K_1, s_1, \tilde{s}_1, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ and output $(\mathtt{SRlt}, sid, qid, \mathtt{succ})$.

Step S5*: The user $\mathcal{U}$ proceeds as follows.*

(a) *Receive messages from both servers:* Parse the messages received from $\mathcal{S}_1$ and $\mathcal{S}_2$ as τ_1 and τ_2, respectively.
(b) *Verify that servers accepted and finalize protocol:* Check that $\mathsf{ver}_{PS_1}((sid, qid, F_1, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ}), \tau_1) = 1$ and that $\mathsf{ver}_{PS_2}((sid, qid, F_2, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ}), \tau_2) = 1$. If so, output $(\mathtt{SRlt}, sid, qid, \mathtt{succ})$.

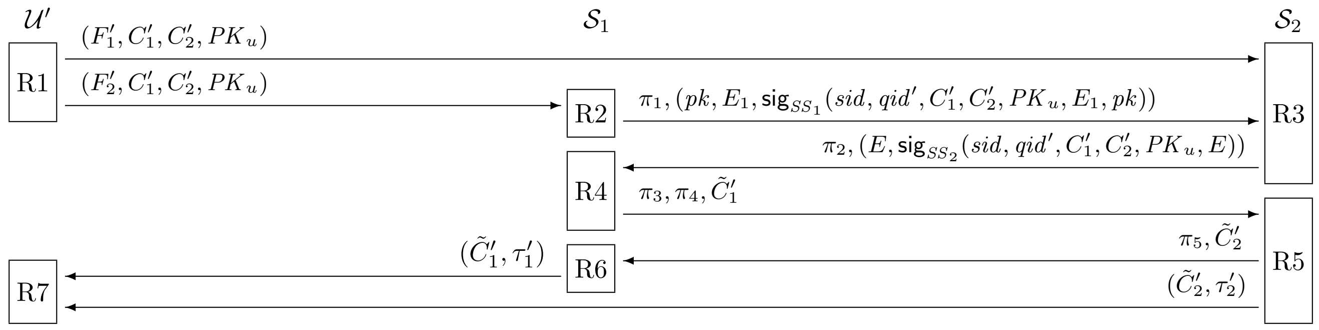

$\mathcal{U}'$
$\mathcal{S}_1$
$\mathcal{S}_2$
(F'_1, C'_1, C'_2, PK_u)
R1
(F'_2, C'_1, C'_2, PK_u)
R2
$\pi_1, (pk, E_1, \mathsf{sig}_{SS_1}(sid, qid', C'_1, C'_2, PK_u, E_1, pk))$
R3
$\pi_2, (E, \mathsf{sig}_{SS_2}(sid, qid', C'_1, C'_2, PK_u, E))$
R4
$\pi_3, \pi_4, \tilde{C}'_1$
$(\tilde{C}'_1, \tau'_1)$
R6
$\pi_5, \tilde{C}'_2$
R5
R7
$(\tilde{C}'_2, \tau'_2)$

Figure 4: Communication messages of the Retrieve protocol with computation steps Ri. In this picture, zero-knowledge proofs are assumed to be non-interactive and thus denoted simply as sending the value π; however, depending on their instantiation, they might be interactive protocols.

3.2.2 The Retrieve Protocol

The input to $\mathcal{U}'$ is $(\mathtt{Rtr}, sid, qid', p')$. The servers $\mathcal{S}_1$ and $\mathcal{S}_2$ have their respective state information $st_1[sid]$ and $st_2[sid]$ as input. The structure of the Retrieve protocol is depicted in Figure 4; the individual steps are as follows. In all steps, whenever a party "fails" or any verification step fails, the party sends $(\mathtt{Rtr}, sid, qid', \mathtt{fail})$ to the other parties and aborts with output $(\mathtt{RRlt}, sid, qid', \mathtt{fail})$ in case the party is a server, or with output $(\mathtt{RRlt}, sid, qid', \perp)$ if it's a user. Furthermore, whenever any party receives a message $(\mathtt{Rtr}, sid, qid', \mathtt{fail})$, it aborts with the same outputs.

Step R1*: On input* $(\mathtt{Rtr}, sid, qid', p')$*, user* $\mathcal{U}'$ *performs the following computations.*

(a) *Obtain public keys of the servers and CRS:* Query $\mathcal{F}_{CRS}$ to receive PK and query $\mathcal{F}_{CA}$ with $(\mathtt{Retrieve}, sid, \mathcal{S}_1)$ and $(\mathtt{Retrieve}, sid, \mathcal{S}_2)$ to receive (PE_1, PS_1) and (PE_2, PS_2).
(b) *Compute shares of password and choose encryption key pair:* Choose $p'_1 \leftarrow_{\mathrm{R}} \mathbb{G}$ and compute $p'_2 \leftarrow p'/p'_1$. Generate $(PK_u, SK_u) \leftarrow \mathsf{keyg}(1^k)$.
(c) *Encrypt shares under the CRS and the servers' public keys:* Choose $s'_1, s'_2 \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and encrypt password shares under the CRS as $C'_1 \leftarrow \mathsf{enc}_{PK}(p'_1; s'_1)$, $C'_2 \leftarrow \mathsf{enc}_{PK}(p'_2; s'_2)$. Encrypt the shares and randomness for both servers as $F'_1 \leftarrow \mathsf{enc2}_{PE_1}((p'_1, s'_1), (sid, qid', C'_1, C'_2, PK_u))$ and $F'_2 \leftarrow \mathsf{enc2}_{PE_2}((p'_2, s'_2), (sid, qid', C'_1, C'_2, PK_u))$.
(d) *Send encryptions to servers:* Send (F'_1, C'_1, C'_2, PK_u) to $\mathcal{S}_1$ and (F'_2, C'_1, C'_2, PK_u) to $\mathcal{S}_2$.

Step R2*: The first server* $\mathcal{S}_1$ *proceeds as follows.*

(a) *Receive message from user, fail if account doesn't exist:* Parse the message received from $\mathcal{U}'$ as $(\mathtt{Rtr}, sid, qid', 1, F'_1, C'_1, C'_2, PK_u)$. If no entry $st_1[sid]$ exists in the state information then fail, else recover $st_1[sid] = (PS_2, p_1, K_1, s_1, \tilde{s}_1, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$.
(b) *Ask environment for permission to continue:* Output $(\mathtt{RNtf}, sid, qid')$ to the environment and wait for an input $(\mathtt{Perm}, sid, qid', a)$ with $a \in \{\mathtt{allow}, \mathtt{deny}\}$. If $a = \mathtt{deny}$ then fail.
(c) *Decrypt share and randomness:* Decrypt F'_1 with label $(sid, qid', C'_1, C'_2, PK_u)$, which will fail if the label is wrong.
(d) *Verify correct encryption of share under CRS:* Check that $C'_1 = \mathsf{enc}_{PK}(p'_1; s'_1)$.
(e) *Generate key pair for homomorphic encryption scheme and encrypt shares' quotient:* Generate $(pk, sk) \leftarrow \mathsf{keyg}(1^k)$, choose $r_1 \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, and compute $E_1 \leftarrow \mathsf{enc}_{pk}(p_1/p'_1; r_1)$.
(f) *Send signed encrypted quotient to second server:* Compute the signature $\sigma'_1 \leftarrow \mathsf{sig}_{SS_1}(sid, qid', C'_1, C'_2, PK_u, E_1, pk)$ and send (pk, E_1, σ'_1) to $\mathcal{S}_2$.
(g) *Prove to second server that* E_1 *is correct:* Perform the following proof protocol with $\mathcal{S}_2$:

$$\pi_1 := \mathrm{ZK}\{(p_1, p'_1, s_1, s'_1, r_1) : \ E_1 = \mathsf{enc}_{pk}(p_1/p'_1; r_1) \wedge C_1 = \mathsf{enc}_{PK}(p_1; s_1) \wedge C'_1 = \mathsf{enc}_{PK}(p'_1; s'_1)\}((sid, qid', 1)) .$$

Step R3*: The second server* $\mathcal{S}_2$ *proceeds as follows.*

(a) *Receive message from user, fail if account doesn't exist:* Parse the message received from $\mathcal{U}'$ as $(\texttt{Rtr}, sid, qid', 1, F_2', C_1', C_2', PK_u)$. If no entry $st_2[sid]$ exists in the saved state then fail, else recover $st_2[sid] = (PS_1, p_2, K_2, s_2, \tilde{s}_2, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$.

(b) *Ask environment for permission to continue:* Output $(\texttt{RNtf}, sid, qid')$ to the environment and wait for an input $(\texttt{Perm}, sid, qid', a)$ with $a \in \{\texttt{allow}, \texttt{deny}\}$. If $a = \texttt{deny}$ then fail.

(c) *Receive message from first server and check proof:* Parse the message from $\mathcal{S}_1$ as $(\texttt{Rtr}, sid, qid', 2, pk, E_1, \sigma_1')$. Furthermore act as a verifier in the proof π_1 with $\mathcal{S}_1$.

(d) *Decrypt password share and randomness:* Decrypt F_2' with label $(sid, qid', C_1', C_2', PK_u)$ and fail if decryption fails.

(e) *Verify share encryption under CRS and first server's signature:* Check that $C_2' = \mathsf{enc}_{PK}(p_2'; s_2')$ and that $\mathsf{ver}_{PS_1}((sid, qid', C_1', C_2', PK_u, E_1, pk), \sigma_1') = 1$.

(f) *Multiply encryption by quotient of own shares:* Choose random $r_2, z \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and compute $E_2 \leftarrow \mathsf{enc}_{pk}(p_2/p_2'; r_2)$ and $E \leftarrow (E_1 \times E_2)^z$.

(g) *Send signed encrypted quotient to first server:* Compute $\sigma_2' \leftarrow \mathsf{sig}_{SS_2}(sid, qid', C_1', C_2', PK_u, E)$ and send (E, σ_2') to $\mathcal{S}_1$.

(h) *Prove to first server that E is correct:* Perform with $\mathcal{S}_1$ the proof protocol:

$$\begin{aligned}\pi_2 \;:=\; & \mathrm{ZK}\{(p_2, p_2', s_2, s_2', r_2, z) :\; E = (E_1 \times \mathsf{enc}_{pk}(p_2'/p_2; r_2))^z \\ & \wedge\; C_2 = \mathsf{enc}_{PK}(p_2; s_2) \;\wedge\; C_2' = \mathsf{enc}_{PK}(p_2'; s_2')\}((sid, qid', 2)) \;.\end{aligned}$$

Step R4*: The first server $\mathcal{S}_1$ proceeds as follows.*

(a) *Receive message from second server and verify proof:* Parse the message from $\mathcal{S}_2$ as (E, σ_2') and interact with $\mathcal{S}_2$ in π_2.

(b) *Verify signature and check $z \neq 0$:* Verify that $\mathsf{ver}_{PS_2}((sid, qid', C_1', C_2', PK_u, E), \sigma_2') = 1$ and that $E \neq \mathsf{enc}_{pk}(1; 0)$.

(c) *Learn whether password matches:* Decrypt E using sk and verify that it decrypts to 1.

(d) *Inform and convince second server of result:* Prove to $\mathcal{S}_2$ that E indeed decrypts to 1 with the protocol:

$$\pi_3 \;:=\; \mathrm{ZK}\{(sk) : 1 = \mathsf{dec}_{sk}(E)\}((sid, qid', 3)) \;.$$

(e) *Verifiably encrypt key share for the user:* Compute ciphertext $\tilde{C}_1' \leftarrow \mathsf{enc}_{PK_u}(K_1; \tilde{s}_1')$ with $\tilde{s}_1' \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and send $\tilde{C}_1'$ to $\mathcal{S}_2$. Prove to $\mathcal{S}_2$ that $\tilde{C}_1'$ encrypts the same key share as $\tilde{C}_1$ from the setup phase:

$$\pi_4 \;:=\; \mathrm{ZK}\{(K_1, \tilde{s}_1, \tilde{s}_1') : \tilde{C}_1 = \mathsf{enc}_{PK}(K_1; \tilde{s}_1) \;\wedge \tilde{C}_1' = \mathsf{enc}_{PK_u}(K_1; \tilde{s}_1')\}((sid, qid', 4)) \;.$$

Step R5*: The second server $\mathcal{S}_2$ proceeds as follows.*

(a) *Receive message from first server and verify proof:* Parse the message from $\mathcal{S}_1$ as $\tilde{C}_1'$ and participate in proofs π_3 and π_4 with $\mathcal{S}_1$.

(b) *Verifiably encrypt key share for the user:* Compute ciphertext $\tilde{C}_2' \leftarrow \mathsf{enc}_{PK_u}(K_2; \tilde{s}_2')$ with $\tilde{s}_2' \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and send $\tilde{C}_2'$ to $\mathcal{S}_1$. Prove to $\mathcal{S}_1$ that $\tilde{C}_2'$ encrypts the same key share as $\tilde{C}_2$ from the setup phase:

$$\pi_5 \;:=\; \mathrm{ZK}\{(K_2, \tilde{s}_2, \tilde{s}_2') : \tilde{C}_2 = \mathsf{enc}_{PK}(K_2; \tilde{s}_2) \;\wedge \tilde{C}_2' = \mathsf{enc}_{PK_u}(K_2; \tilde{s}_2')\}((sid, qid', 5)) \;.$$

(c) *Send signed result to user and finish protocol:* Compute $\tau_2' \leftarrow \mathsf{sig}_{SS_2}(sid, qid', F_1', C_1', C_2', PK_u, \tilde{C}_1', \tilde{C}_2')$ and send $(\tilde{C}_2', \tilde{\tau}_2')$ to $\mathcal{U}'$. Output $(\texttt{RRlt}, sid, qid', \texttt{succ})$.

Step R6*: The first server $\mathcal{S}_1$ proceeds as follows.*

(a) *Receive message from second server and verify proofs:* Parse the message from $\mathcal{S}_2$ as $\tilde{C}_2'$ and interact with it in π_5.

(b) *Send signed result to user and finish protocol:* Compute $\tau_1' \leftarrow \mathsf{sig}_{SS_1}(sid, qid', F_1', C_1', C_2', PK_u, \tilde{C}_1', \tilde{C}_2')$ and send $(\tilde{C}_1', \tau_1')$ to $\mathcal{U}'$. Output $(\texttt{RRlt}, sid, qid', \texttt{succ})$.

Step R7*: The user $\mathcal{U}'$ proceeds as follows.*

(a) *Receive messages from both servers:* Parse the messages from $\mathcal{S}_1$ and $\mathcal{S}_2$ as $(\tilde{C}_1', \tau_1')$ and $(\tilde{C}_2', \tau_2')$, respectively.

(b) *Verify signatures:* Verify that $\mathsf{ver}_{PS_1}((sid, qid', F'_1, C'_1, C'_2, PK_u, \tilde{C}'_1, \tilde{C}'_2), \tau'_1) = 1$ and that $\mathsf{ver}_{PS_2}((sid, qid', F'_2, C'_1, C'_2, PK_u, \tilde{C}'_1, \tilde{C}'_2), \tau'_2) = 1$.

(c) *Compute and output key:* Compute the two key shares $K_1 \leftarrow \mathsf{dec}_{SK_u}(\tilde{C}'_1)$ and $K_2 \leftarrow \mathsf{dec}_{SK_u}(\tilde{C}'_2)$, reconstruct the key as $K \leftarrow K_1 \cdot K_2$, and output $(\mathtt{RRlt}, sid, qid', K)$.

4 Concrete Instantiation

In this section we give constructions of the encryption schemes and zero-knowledge protocols with which our 2PASS protocol can be instantiated. They are secure under the decisional Diffie-Hellman (DDH) assumption; the proofs require the random-oracle model. For the signature scheme and the CCA2-secure encryption, one could for example use Schnorr [Sch91, PS96] signatures and ElGamal encryption with the Fujisaki-Okamoto transformation [ElG85, FO99] be used since the DDH assumption also suffices for their security in the RO model. We also provide an efficiency analysis.

4.1 ElGamal Encryption

The ElGamal encryption scheme [ElG85] assumes a generator g of a group $G = \langle g \rangle$ of prime order q. The secret key x is chosen at random from $\mathbb{Z}_q$. The public key is $y = g^x$. To encrypt a message $m \in G$, select a random r and compute $c_1 \leftarrow y^r m$ and $c_2 \leftarrow g^r$. Output as ciphertext is the tuple (c_1, c_2). To decrypt (c_1, c_2), compute $m \leftarrow c_1/c_2^x$.

It is well known that the ElGamal encryption scheme is CPA secure and is homomorphic: i.e., $E = E_1 \times E_2$ is defined as $(e_1, e_2) = (e_{11}, e_{12}) \times (e_{21}, e_{22}) := (e_{11}e_{21}, e_{12}e_{22})$ and also we define $E^z = (e_1, e_2)^z = (e_1^z, e_2^z)$.

4.2 Zero-Knowledge Proofs and Σ-Protocols

Using the ElGamal encryption scheme will allow us to instantiate the proof protocols in our scheme by well known and efficient Σ-protocols for statements about discrete logarithms in the group $\mathbb{G}$. When referring to the proofs above, use the following notation [CS97, CKY09]. For instance, $PK\{(a, b, c) : y = g^a h^b \wedge \tilde{y} = g^a h^c\}$ denotes 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 $\mathbb{G}$. 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 [CKY09] for details on this.

The most efficient way to make these protocol concurrent zero-knowledge and simulation-sound is by the Fiat-Shamir transformation [FS87]. In this case, we will have to resort to the random-oracle model [BR93] for the security proof. To make the resulting non-interactive proofs simulation-sound, it suffices to let the prover include context information such as the session and query identifiers into the label, and to include the label as an argument to the random oracle in the Fiat-Shamir transformation.

We note, however, that there are alternative methods one could employ instead to make Σ-protocols non-interactive that do not rely on the random oracle model (e.g., [MY04, GMY03, CKS11]). Unfortunately, these methods come with some performance penalty. In our protocol that would impact only the servers, not the user, so should still be very acceptable in practice.

4.3 Concrete ZK Protocols in Our Scheme

As said in the description of our scheme, we assume that the description of a group $\mathbb{G}$ of prime order q and a generator g chosen through $\mathsf{GGen}(1^k)$ is publicly available, together with a public key PK of the cryptosystem $(\mathsf{keyg}, \mathsf{enc}, \mathsf{dec})$. In the following we will further assume that $PK = (Y, g)$ is a public key of the ElGamal encryption scheme.

Proof π_1 in Step R2 of the Retrieve protocol. Suppose that, in Step R2, $\mathcal{S}_1$ generated (pk, sk) as $((y = g^x, g), x) \in ((\mathbb{G}, \mathbb{G}), \mathbb{Z}_q)$. Let the encryptions computed in the setup and retrieve protocol be $E_1 = (e_{11}, e_{12}) = (p_1/p'_1 y^{r_1}, g^{r_1})$,

$C_1 = (c_{11}, c_{12}) = (p_1 Y^{s_1}, g^{s_1})$, and $C_1' = (c_{11}', c_{12}') = (p_1' Y^{s_1'}, g^{s_1'})$ with r_1, s_1, s_1' elements of $\mathbb{Z}_q$. Then the proof π_1 can be instantiated with the protocol specified as:

$$\pi_1 \ := \ PK\{(s_1, s_1', r_1): \ e_{12} = g^{r_1} \wedge c_{12} = g^{s_1} \wedge c_{12}' = g^{s_1'} \wedge \frac{e_{11} c_{11}'}{c_{11}} = y^{r_1} Y^{s_1'} Y^{-s_1}\}((sid, qid', 1)) \ .$$

This protocol requires both the prover and the verifier to compute four exponentiations in $\mathbb{G}$ (note that $\mathbb{G}$ can be an elliptic-curve group).

Let us argue that the protocol indeed proves that E_1 encrypts the quotient of the messages encrypted in C_1 and C_1'. We know that if the prover is successful, then there are values (s_1, s_1', r_1) such that $e_{11} = g^{r_1}$, $c_{11} = g^{s_1}$, $c_{11}' = g^{s_1'}$, and $\frac{e_{11} c_{11}'}{c_{11}} = y^{r_1} Y^{s_1'} Y^{-s_1}$ hold (see e.g., [CKY09]). As we are using the ElGamal encryption scheme, the ciphertexts encrypted in E_1, C_1, and C_1' thus must be $e_{11} y^{-r_1}$, $c_{11} Y^{-s_1}$, and $c_{11}' Y^{-s_1'}$, respectively. The last term of the proof protocol $\frac{e_{11} c_{11}'}{c_{11}} = y^{r_1} Y^{s_1'} Y^{-s_1}$ can be reformed into $e_{11} y^{-r_1} = (c_{11} Y^{-s_1}) / (c_{11}' Y^{-s_1'})$ which amounts to the statement that we claimed.

Proof π_2 in Step R3 of the Retrieve protocol. Let the encryptions computed in the setup and retrieve protocol be $E = (e_1, e_2) = ((e_{11} y^{r_2} p_2 / p_2')^z, (e_{12} g^{r_2})^z)$, $C_2 = (c_{21}, c_{22}) = (p_2 Y^{s_2}, g^{s_2})$, and $C_2' = (c_{21}', c_{22}') = (p_2' Y^{s_2'}, g^{s_2'})$ with $z, r_2, s_2, s_2' \in \mathbb{Z}_q$. Then the proof π_2 can be instantiated with the protocol specified as:

$$\begin{aligned} \pi_2 \ &:= \ PK\{(s_2, s_2', z, \alpha, \beta, \gamma): \ e_2 = e_{12}^z g^\alpha \ \wedge \ c_{22} = g^{s_2} \ \wedge 1 = c_{22}^z g^{-\beta} \ \wedge \ c_{22}' = g^{s_2'} \\ &\wedge 1 = {c_{22}'}^z g^{-\gamma} \ \wedge \ e_1 = (\frac{e_{11} c_{21}}{c_{21}'})^z y^\alpha Y^{-\beta} Y^\gamma\}((sid, qid', 2)) \ . \end{aligned}$$

where by definition $\alpha = z r_2$ and by proof $\beta = z s_2$ and $\gamma = z s_2'$. Let's again show that this proof protocol is indeed a proof that E is an encryption of a random power of the plaintext in E_1 (let's call it $\tilde{m}$) times the quotient of the plaintexts in C_2' and C_2 (let's call them m' and m, respectively). Again, from the properties of the proof protocol we know that there exist values $s_2, s_2', r_2, z, \alpha, \beta, \gamma$ so that the terms in the protocol specification hold. Now from $c_{22} = g^{s_2}$, $1 = c_{22}^z g^{-\beta}$ $c_{22}' = g^{s_2'}$ and $1 = {c_{22}'}^z g^{-\gamma}$ we can conclude that $\beta = z s_2$ and $\gamma = z s_2'$ holds. Further, the ciphertexts encrypted in C_2, and C_2' thus must be $m := c_{21} Y^{-s_2}$, and $m' := c_{21}' Y^{-s_2'}$, respectively. From the proof term $e_1 = (\frac{e_{11} c_{21}}{c_{21}'})^z y^\alpha Y^{-\beta} Y^\gamma$ we can derive that $e_1 = e_{11}^z (m/m')^z y^\alpha$.

Also, let r_1 be the value such that $e_{12} := g^{r_1}$ and let $\tilde{m} := e_{11} y^{-r_1}$. Thus, $e_2 = e_{12}^z g^\alpha = g^{r_1 z + \alpha}$ and $e_1 = e_{11}^z (m/m')^z y^\alpha = \tilde{m}^z y^{-r_1 z} (m/m')^z y^\alpha$. We can write $e_1 = (\tilde{m} m / m')^z y^{-r_1 z + \alpha}$ which means that E is indeed an encryption of $(\tilde{m} m / m')^z$ as we claimed.

Proof π_3 in Step R4 of the Retrieve protocol. The proof π_3 showing that the encryption $E = (e_1, e_2)$ decrypts to 1 (w.r.t. the public/secret key pair $(pk, sk) = ((y = g^x, g), x)$ that $\mathcal{S}_1$ has generated in Step R2 of the retrieve protocol) can be implemented with the following protocol specification:

$$\pi_3 \ := \ PK\{(x): \ y = g^x \ \wedge \ e_1 = e_2^x\}((sid, qid', 3)) \ .$$

It is not very hard to see that this protocol indeed shows that E encrypts to 1.

Proofs π_4 and π_5 in Steps R4 and R5. The proofs in these two steps are essentially the same (just the indices are different), so we describe only the first. Let the user's public key $PK_u = (Y_u, g)$ and let the encryptions computed in the setup and retrieve protocol be $\tilde{C}_1 = (\tilde{c}_{11}, \tilde{c}_{12}) = (K_1 Y^{\tilde{s}_1}, g^{\tilde{s}_1})$, and $\tilde{C}_1' = (\tilde{c}_{11}', \tilde{c}_{12}') = (K_1 Y_u^{\tilde{s}_1'}, g^{\tilde{s}_1'})$. Then the proof π_4 can be realized with the protocol specified as

$$\pi_4 \ := \ PK\{(\tilde{s}_1, \tilde{s}_1'): \ \tilde{c}_{12} = g^{\tilde{s}_1} \ \wedge \ \tilde{c}_{12}' = g^{\tilde{s}_1'} \ \wedge \ \frac{\tilde{c}_{11}'}{\tilde{c}_{11}} = Y_u^{\tilde{s}_1'} Y^{-\tilde{s}_1}\}((sid, qid', 4)) \ .$$

It is not hard to see that this protocol indeed proves that the two ciphertexts encrypt the same plaintext.

4.4 Efficiency Analysis

Let us count the number of exponentiations in the group $\mathbb{G}$ when our protocol is instantiated as suggested above and using the Fiat-Shamir transformation [FS87] to obtain simulation-sound non-interactive proofs in the random-oracle model. We neglect operations other than exponentiations because their cost is insignificant in comparison. The user has to perform 18 exponentiations in the Setup protocol and 19 exponentiations in the Retrieve protocol. Each server has to do 11 exponentiations in the Setup protocol. In the Retrieve protocol, $\mathcal{S}_1$ and $\mathcal{S}_2$ need to do 26 and 30 exponentiations, respectively. (Note that some of the exponentiations by the servers could be optimized as they are part of multi-base exponentiations.) Finally we note that an elliptic-curve group can be used for $\mathbb{G}$ and that our protocols do not require secure channels and hence avoid the additional cost of setting these up.

The communication costs are as follows: the user sends to each server 16 group elements and receives 1 group element from each in the Setup protocol. The user sends 11 group elements to and receives 5 group elements from each server in the Retrieve protocol. The servers send to each other 1 group elements in the Setup protocol and 6 (resp. 5) group elements, 6 (resp. 9) exponents, and 3 (resp. 2) hash values in the Retrieve protocol.

Therefore, our protocol is efficient enough to be useful in practice.

5 Security Analysis

Theorem 1 *If the encryption scheme* $(\mathsf{keyg}, \mathsf{enc}, \mathsf{dec})$ *is semantically secure, the encryption scheme* $(\mathsf{keyg2}, \mathsf{enc2}, \mathsf{dec2})$ *is CCA2 secure, the signature scheme* $(\mathsf{keygsig}, \mathsf{sig}, \mathsf{ver})$ *is existentially unforgeable, and the associated proof system is a simulation-sound concurrent zero-knowledge proof, then the* Setup *and* Retrieve *protocols securely realize* $\mathcal{F}_{2\mathrm{PASS}}$ *in the* $\mathcal{F}_{CA}$ *and* $\mathcal{F}_{CRS}$*-hybrid model.*

When instantiated with the ElGamal encryption scheme [ElG85] for $(\mathsf{keyg}, \mathsf{enc}, \mathsf{dec})$, ElGamal encryption with the Fujisaki-Okamoto transformation [FO99] for $(\mathsf{keyg2}, \mathsf{enc2}, \mathsf{dec2})$, Schnorr signatures [Sch91, PS96] for $(\mathsf{keygsig}, \mathsf{sig}, \mathsf{ver})$, and the Σ protocols of Section 4 [Sch91, CKY09], by the UC composition theorem and the security of the underlying building blocks we have the following corollary:

Corollary 1 *Under the decisional Diffie-Hellman assumption for the group associated with* GGen*, the* Setup *and* Retrieve *protocols as instantiated above securely realize* $\mathcal{F}_{2\mathrm{PASS}}$ *in the random-oracle and* $\mathcal{F}_{CA}$*-hybrid model.*

5.1 Sequence of Games

Proof of Theorem 1 Let us conceptually view all honest participants as a single interactive Turing machine, which we will call the *challenger*, that obtains all inputs from the environment intended for honest parties and that outputs the responses back to the environment. We will define a series of games with a series of challengers; the challenger corresponding to game i is denoted $\mathcal{C}_i$. In the first game, $\mathcal{C}_1$ receives as input the value $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$ and runs our protocol on behalf of the honest participants, and with $\mathcal{A}$ as the adversary, so the environment $\mathcal{E}$ receives the same view as it would receive in the real execution of the protocol. In the last game, $\mathcal{C}_{11}$ runs the ideal protocol (via the ideal functionality) on behalf of the honest participants, and with the simulator $\mathcal{SIM}$ (which we will describe in the sequel) as the adversary, so the environment receives the same view as it would in the ideal execution. Let $view_i(sid, 1^k)$ denote the view that the environment $\mathcal{E}$ receives when interacting with $\mathcal{C}_i$ for session identifier sid and security parameter k; we will often omit sid and 1^k. We will show that, for each i, $1 \leq i < 11$, the view $view_i$ is (computationally) indistinguishable from the view $view_{i+1}$ it receives when interacting with $\mathcal{C}_{i+1}$, denoted $view_i \approx view_{i+1}$. The indistinguishability of the real world and the ideal world then follows through a hybrid argument.

Challenger $\mathcal{C}_1$: The challenger runs all honest parties with the real protocol with all the inputs coming directly from the environment. Therefore, $view_1$ is identical to the view that $\mathcal{E}$ receives when honest participants execute our protocol.

Challenger $\mathcal{C}_2$: Identical to $\mathcal{C}_1$, except that it halts (and therefore acts visibly different from $\mathcal{C}_1$) whenever an honest party receives a valid signature σ under the public key PS_i of an honest server $\mathcal{S}_i$ on a message that was never signed by $\mathcal{S}_i$. We have that $view_2 \approx view_1$, because otherwise a straightforward reduction can break the security of the underlying signature scheme.

In particular, this means that in a setup protocol, if at least one of the servers is honest and an honest user successfully ends the setup protocol, then the honest server's records reflect the same values $(C_1, C_2, \tilde{C}_1, \tilde{C}_2)$ as the user sent in Step S1, because the server signed the same values in Step S3 or S4. Likewise, in a retrieve protocol, if at least one server is honest and an honest user recovers a key $K \neq \perp$, then the honest server performed the protocol using the same values (C_1', C_2') that the user sent to it in Step R1. Moreover, if at least one server is honest, then the absence of forged signatures ensures that if an honest user successfully recovers a key in the retrieval protocol, then all honest servers have verified the validity of the proofs provided by the other server.

Challenger $\mathcal{C}_3$: Identical to $\mathcal{C}_2$, except in the case when an honest participant uses $\mathsf{enc2}$ to send an encryption of a plaintext m to an honest server: in this case, the ciphertext is computed as an encryption of $1^{|m|}$. In particular, if an honest user interacts with an honest server $\mathcal{S}_i$, this is done for ciphertexts F_i sent as part of a message $(\mathtt{Stp}, \mathit{sid}, \mathit{qid}, 1, F_i, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ in Step S1 and for ciphertexts F'_i sent as part of a message $(\mathtt{Rtr}, \mathit{sid}, \mathit{qid}', 1, F'_i, C'_1, C'_2, PK_u)$ in Step R1.

We call a setup or retrieve query *intact for server* $\mathcal{S}_i$ if it was initiated by an honest user and the first message $(\mathtt{Stp}, \mathit{sid}, \mathit{qid}, 1, \ldots)$ or $(\mathtt{Rtr}, \mathit{sid}, \mathit{qid}', 1, \ldots)$ arrives at the honest server $\mathcal{S}_i$ unmodified. We call the query *hijacked for* $\mathcal{S}_i$ if it was initiated by an honest user but this message was modified in transit. We call the query *corrupt for* $\mathcal{S}_i$ if it was either hijacked or initiated by a dishonest user. If the query is intact, then the honest server $\mathcal{S}_i$ pretends that the ciphertext F_i or F'_i correctly decrypted to m, meaning that $\mathcal{S}_i$ continues executing the real protocol as if it decrypted to m. Otherwise, it decrypts the ciphertext from the modified message and continues the protocol with the actual decrypted value.

We have that $\mathit{view}_3 \approx \mathit{view}_2$, because otherwise $(\mathsf{keyg2}, \mathsf{enc2}, \mathsf{dec2})$ is not a CCA2-secure encryption scheme by the following hybrid argument. We define a series of hybrids; the first hybrid agrees with $\mathcal{C}_2$, the last one with $\mathcal{C}_3$. Let $\ell(k)$ be an upper bound on the total number of ciphertexts F_j and F'_j that honest participants compute using $\mathsf{enc2}$ and send to each other. For $0 \leq i \leq \ell(k)$, hybrid i forms the first i such ciphertexts as an encryption of $1^{|m|}$, and the remaining ones correctly, as directed by the protocol.

Suppose that the environment could distinguish two neighboring hybrids i and $i+1$. Then we set up a reduction that breaks labeled CCA2 security of $\mathsf{enc2}$. The reduction receives as input a public key PE for $\mathsf{enc2}$ and can interact with the CCA2 challenger which will decrypt ciphertexts of its choice, except the challenge ciphertext. The reduction assigns the public key PE to one of the honest servers at random. Up until it needs to compute the $(i+1)$st ciphertext, it interacts with $\mathcal{E}$ the way that $\mathcal{C}_3$ would, except that it does not have the decryption key for the selected server. But whenever it needs to decrypt a ciphertext submitted by the adversary $\mathcal{A}$, it simply forwards it to the decryption oracle. When it is time to compute the $i+1^{st}$ ciphertext, if this ciphertext needs to be computed under the public key other than PE, the reduction gives up. (Note, however, that this happens with probability at most 1/2 and independently of the environment's view so far.) Otherwise, it will ask the CCA2 challenger for a challenge ciphertext, suggesting two possibilities for the plaintext: the real plaintext m that hybrid i would encrypt, and the plaintext $1^{|m|}$ that hybrid $i+1$ would encrypt. After the challenge ciphertext c has been issued, the reduction uses it as the $i+1^{st}$ ciphertext F_j or F'_j. If the query is left intact, then it proceeds as if F_j or F'_j decrypted to m and interacts with the environment the way $\mathcal{C}_3$ would, except that whenever it needs to decrypt a ciphertext from the adversary, it forwards it to the decryption oracle.

What remains to be shown is why ciphertexts that are part of corrupt queries can always be queried to the decryption oracle. The ciphertexts F_1 and F_2 in the setup protocol are formed using $(\mathit{sid}, \mathit{qid}, C_1, C_2, \tilde{C}_1, \tilde{C}_2)$ as a label. Suppose that $\mathcal{S}_1$ is honest, and an honest user is running the setup protocol, and the ciphertext F_1 happens to be the $(i+1)$st ciphertext, so the reduction has the honest user use the ciphertext $F_1 = c$, and send $(\mathtt{Stp}, \mathit{sid}, \mathit{qid}, 1, F_1, C_1, C_2, \tilde{C}_1, \tilde{C}_2)$ to $\mathcal{S}_1$. Suppose at some later point, $\mathcal{S}_1$ receives a message $(\mathtt{Stp}, \mathit{sid}, \mathit{qid}, 1, \ldots)$ from the adversary. If the query is intact for $\mathcal{S}_1$, i.e., the entire message was received correctly, then the challenger treats F_1 as an encryption of m and has $\mathcal{S}_1$ act accordingly. If the query is hijacked for $\mathcal{S}_1$, then either the ciphertexts $C_1, C_2, \tilde{C}_1, \tilde{C}_2$ are different, or F_1 is different. In the former case, $\mathcal{S}_1$ is allowed to use the decryption oracle to decrypt $F_1 = c$ since the label is different. In the latter case, it can also use the decryption oracle, since the ciphertext is different. Finally, if the adversary were to reuse F_1 in a different hijacked query $\mathit{qid}' \neq \mathit{qid}$, then also $\mathcal{S}_1$ can use the decryption oracle since the decryption label is different. We can argue the same for the ciphertext F_2 of the setup protocol as well as the ciphertexts F'_1 and F'_2 of the retrieve protocol.

Challenger $\mathcal{C}_4$: Identical to $\mathcal{C}_3$, except that, whenever an honest party performs a zero-knowledge proof, it uses the zero-knowledge simulator instead of the prover's algorithm. Note that, depending on the instantiation of the zero-knowledge proofs, this may involve setting up the CRS or random oracle in such a way that the challenger can run the simulator. Indistinguishability from $\mathcal{C}_3$ follows by a straightforward reduction from the zero-knowledge property of the proof system.

Challenger $\mathcal{C}_5$: Identical to $\mathcal{C}_4$, except in the following case: when an honest user $\mathcal{U}$ is directed by the protocol to compute a ciphertext $c = \mathsf{enc}_{PK}(m; r)$ under the public key PK in the CRS, where (m, r) will never be encrypted to a dishonest party, $\mathcal{U}$ instead computes $c = \mathsf{enc}_{PK}(1; r)$, where 1 is the unity group element. More concretely, this change affects the ciphertexts C_i, $\tilde{C}_i$, C'_i that an honest user sends to an honest server $\mathcal{S}_i$ in Steps S1 and R1. Whenever the protocol directs an honest server $\mathcal{S}_i$ to prove something about the ciphertext c, $\mathcal{C}_5$ will, just as $\mathcal{C}_4$, have $\mathcal{S}_i$ run the zero-knowledge simulator, but now to prove the false statement that $c = \mathsf{enc}_{PK}(m; r)$.

Here and in all future games, we condition the analysis on the event that the adversary does not create a new valid zero-knowledge proof of a false statement. A hybrid argument over the proofs produced by the adversary can be used that this holds under the simulation soundness of the proof protocol. Conditioning on this event, challenger $\mathcal{C}_5$ is indistinguishable from $\mathcal{C}_4$ by the semantic security of $(\mathsf{keyg}, \mathsf{enc}, \mathsf{dec})$ through a simple hybrid argument. Therefore, $view_5 \approx view_4$.

Challenger $\mathcal{C}_6$: Identical to $\mathcal{C}_5$, except that: (1) the public key "in the sky" PK is generated so that $\mathcal{C}_6$ knows the corresponding secret key SK. Since PK is distributed exactly as in a real CRS, this hop is purely conceptional.

Further, $\mathcal{C}_6$ is running, on the side, what will, in several additional steps, become the ideal functionality $\mathcal{F}$; for now we will think of it as a registry $\mathcal{R}$ of what has happened so far corresponding to the *sid* of this session (recall that a challenger receives the value *sid* as input). The existence of the registry $\mathcal{R}$ is internal to $\mathcal{C}_6$ and has no effect on $view_6$.

Whenever a setup query *qid* using password p and key K remains intact for an honest server $\mathcal{S}_i$, $\mathcal{C}_6$ creates the record $(\texttt{AStp}, qid, \mathcal{U}, p, K)$ and adds it to the registry $\mathcal{R}$. It then continues running the setup protocol with $\mathcal{S}_1$ and $\mathcal{S}_2$ on behalf of the honest user and marks this query *qid* as `succ` for an honest server $\mathcal{S}_i$ when $\mathcal{S}_i$ outputs $(\texttt{SRlt}, sid, qid, \texttt{succ})$, or marks it `fail` for $\mathcal{S}_i$ if something goes wrong. When the query is marked `succ` for all honest servers of $\mathcal{S}_1$ and $\mathcal{S}_2$, $\mathcal{C}_6$ stores a record $(\texttt{Stp}, p, K)$ in $\mathcal{R}$. Further, when $\mathcal{U}$ finished the protocol successfully, $\mathcal{C}_6$ marks the query *qid* as `succ` for $\mathcal{U}$.

Whenever the first message of a corrupt setup query arrives at an honest server $\mathcal{S}_i$, $\mathcal{C}_6$ uses SK to decrypt the ciphertexts $(C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ and recover p and K from them. It then stores a record $(\texttt{AStp}, qid, \mathcal{U}, p, K)$ in $\mathcal{R}$ and marks query *qid* as `succ` for $\mathcal{S}_i$ when $\mathcal{S}_i$ outputs $(\texttt{SRlt}, sid, qid, \texttt{succ})$. Note that two honest servers may receive different ciphertexts $(C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ for a corrupt query, leading to two different $(\texttt{AStp}, \ldots)$ records to be created. However, given that the adversary does not forge signatures, the honest servers will never conclude such a protocol successfully.

Similarly, whenever the first message of a corrupt retrieve query arrives at an honest server $\mathcal{S}_i$, $\mathcal{C}_6$ uses SK to decrypt the ciphertexts C_1' and C_2' to recover p', creates a record $(\texttt{ARtr}, qid', \mathcal{U}', p')$ and marks this query qid' as `succ` for $\mathcal{S}_i$ when $\mathcal{S}_i$ outputs $(\texttt{RRlt}, sid, qid', s)$.

It is important to note that, starting with $\mathcal{C}_5$, whenever at least one of the servers is honest, the protocol messages that our challengers generate on behalf of honest users are distributed independently of the actual passwords and keys that this honest user receives as input from the environment. The same holds for messages generated on behalf of honest servers in the setup protocol. However, in the retrieve protocol, messages generated on behalf of the honest servers might still depend on the input password in two specific ways: (1) an honest $\mathcal{S}_1$ will pretend that (C_1, C_1') decrypt to (p_1, p_1') and will use these values when computing the ciphertext E_1 in Step R2; and (2) an honest $\mathcal{S}_2$ will, similarly, use (p_2, p_2') in computing E in Step R3. The next three game hops will eliminate this dependence, as well as the dependence that messages computed in steps R4 and R5 have on the key stored and successfully retrieved by an honest user.

Challenger $\mathcal{C}_7$: Suppose that an honest server $\mathcal{S}_2$ is engaged with a server $\mathcal{S}_1$ in a $\mathsf{Retrieve}$ attempt for a particular (sid, qid'). $\mathcal{C}_7$ differs from $\mathcal{C}_6$ in how it computes the ciphertext E in Step R3.

First, note that $\mathcal{C}_7$ can look up if its registry $\mathcal{R}$ has a record $(\texttt{Stp}, p, K)$ (see the description of $\mathcal{C}_6$ for the definition of $\mathcal{R}$). If no such record exists, then the honest server $\mathcal{S}_2$ could not have accepted in the setup protocol, and so the retrieve protocol will end there (and there will be no deviation from what $\mathcal{C}_6$ would have done). Else, if it exists: if the current query qid' is a corrupt query for $\mathcal{S}_2$, then decrypt C_1' and C_2' to recover p'; else the environment explicitly provided p' as input to $\mathcal{C}_7$ on behalf of the honest user. If $p = p'$, then form E as a random encryption of 1; else as an encryption of a random value. The proof is executed, as done by all the challengers starting with $\mathcal{C}_4$, via the zero-knowledge simulator.

If we condition on the adversary's being unable to prove a false statement (which happens with overwhelming probability), then $view_7$ is identical to $view_6$, because the proof π_1 guarantees that E_1 is an encryption of p_1/p_1', and so following $\mathcal{S}_2$'s protocol would make E an encryption of 1 whenever the passwords match, and an encryption of a random value whenever they don't. By simulation soundness, we conditioned on an event which happens with all but negligible probability, and therefore $view_7 \approx view_6$.

Challenger $\mathcal{C}_8$: Suppose that an honest server $\mathcal{S}_1$ is engaged with $\mathcal{S}_2$ in a $\mathsf{Retrieve}$ attempt for a particular (sid, qid'). $\mathcal{C}_8$ differs from $\mathcal{C}_7$ in how it computes the ciphertext E_1 in Step R2: it always computes E_1 as an encryption of 1. If the current query qid' is corrupt for $\mathcal{S}_1$, then $\mathcal{C}_8$ recovers p' by decrypting C_1' and C_2'; else p' is known to it. In Step R4, if $p' \neq p$, it simply fails; if $p' = p$, it continues the protocol as if the encryption E received from $\mathcal{S}_2$ decrypts to 1.

Conditioning again on the adversary's being unable to prove a false statement, we can construct a reduction that shows that, if $view_8$ is distinguishable from $view_7$, then the homomorphic encryption scheme is not semantically secure. The reduction will use a standard hybrid argument; the challenge ciphertext will either be an encryption of the value p_1/p_1' or an encryption of 1. Let C_2 be the encryption of p_2 that $\mathcal{S}_1$ stored in its state at the end of its successful setup phase, and let p_2' either be the actual decryption of the ciphertext C_2' (if this is a corrupt query) or the pretended decryption of C_2' (if this is an intact query). The soundness of the proofs guarantees that, whenever E_1 is an encryption of p_1/p_1', E can only decrypt to 1 when $p_1/p_1' = p_2/p_2'$ or when $z = 0$; the former case means that $p' = p$, i.e., the password match, the latter case is ruled out by the test that $E \neq \mathsf{enc}_{pk}(1; 0)$.

Challenger $\mathcal{C}_9$: This challenger differs from the previous one in how an honest server $\mathcal{S}_i$ forms the ciphertext $\tilde{C}_i'$ in Steps R4 and R5.

If a retrieve query qid' originates from an honest user and is intact for $\mathcal{S}_i$, then $\mathcal{C}_9$ computes the ciphertext $\tilde{C}_i'$ as an encryption of 1. Otherwise, i.e., if this retrieve query is corrupt but the passwords matched, $\mathcal{C}_9$ proceeds as follows. If the account was set up through a corrupt query, then $\mathcal{C}_9$ computes $\tilde{C}_i'$ as an encryption of the correct share K_i contained in $\tilde{C}_i$ at the time of set up, which $\mathcal{C}_9$ decrypted using the secret key SK. If the account was set up by an honest user with an intact query, we distinguish between the case where one server is corrupt and the case that both servers are honest. In the first case, let $\mathcal{S}_i$ be honest and $\mathcal{S}_j$ be corrupt. If this is the first time that a retrieve with the correct password is performed, then $\mathcal{C}_9$ encrypts a random share $K_i \leftarrow_{\mathrm{R}} \mathbb{G}$ and simulates the proofs π_4 or π_5; in all subsequent queries, it uses the same share K_i. In the second case, where both servers are honest, if this is the first time that a retrieve with the correct password is performed, $\mathcal{C}_9$ looks up the record $(\mathtt{Stp}, \cdot, K)$ in $\mathcal{R}$, chooses a random $K_1 \leftarrow_{\mathrm{R}} \mathbb{G}$, computes $K_2 \leftarrow K/K_1$, computes $\tilde{C}_i'$ as an encryption of K_i, and simulates the proofs π_4 and π_5. In all subsequent retrieve queries, it computes $\tilde{C}_i'$ using the same share K_i.

Note that for the first corrupt retrieve query with a correct password, the ciphertext $\tilde{C}_i'$ is distributed exactly as in $\mathcal{C}_8$, namely as the encryption of the correct share K_i if the account was set up through a corrupt query; as the encryption of a random share K_i if the account was set up through an intact honest query and only $\mathcal{S}_i$ is honest; and as the encryption of random shares of the real key K if the account was set up through an intact honest query and both servers are honest. Moreover, in all subsequent corrupt retrieve queries with a correct password, the same share K_i is encrypted as in the first such query, as in $\mathcal{C}_8$.

We have left to show that if the retrieve query originates from an honest server and is intact for $\mathcal{S}_i$, computing $\tilde{C}_i'$ as an encryption of 1 does not distort the adversary's view. If $\mathcal{C}_9$ uses the correct public key PK_u that was generated by the simulated honest user, then $view_9 \approx view_8$ by the semantic security of the encryption scheme $(\mathsf{keyg}, \mathsf{enc}, \mathsf{dec})$. We have left to argue that the challenger uses the correct PK_u. This follows from the very definition of an intact query: the query is considered intact if the first message $(\mathtt{Rtr}, sid, qid', 1, F_i', C_1', C_2', PK_u)$ that an honest server $\mathcal{S}_i$ receives is identical to what the honest user sent. Therefore, as long as the query is intact, $\mathcal{C}_9$ always encrypts $\tilde{C}_i'$ under the correct public key PK_u created by the honest user.

Note that at this point, if at least one of the servers is honest, then the protocol messages for intact queries are distributed independently of the honest user's input, both in the setup and in the retrieval protocols, except for the one bit of information whether $p = p'$. We still want to make sure that in this case, an honest user who provides the correct password retrieves the correct key, and one who does not provide the correct password, does not retrieve anything; this we will do in the next step.

Challenger $\mathcal{C}_{10}$: Same as $\mathcal{C}_9$, except that it halts if one of the following bad events happen: (1) the event that an honest user carries out a retrieval with at least one honest server and the correct password, and in the end of the retrieval protocol the user outputs a key K' that is not equal to K stored in the registry $\mathcal{R}$; or (2) the event that an honest user carries out a retrieval with at least one honest server and an incorrect password, but successfully ends the retrieval protocol with some key K'; or (3) the event that in a corrupt retrieve query with an incorrect password, an honest server $\mathcal{S}_i$ encrypts his key share K_i under PK_u; or (4) the event that in a hijacked setup or retrieve query, the honest user completes the protocol successfully.

Suppose that (1) an honest user performs a retrieve with some password p' and outputs a key K'. Let us consider the case when $\mathcal{S}_1$ is adversarial, while $\mathcal{S}_2$ is honest. Since $\mathcal{S}_2$ is honest and signed the values $(C_1', C_2', PK_u, \tilde{C}_1', \tilde{C}_2')$ in signature τ_2', we know that $\mathcal{S}_2$ successfully verified the proofs π_1, π_3 and π_4. (The signature τ_2' is not a forgery because otherwise the challenger would have halted – see the description of $\mathcal{C}_2$.)

We know that if $\mathcal{S}_2$ accepted the proofs π_1 and π_3, then the password defined by C_1', C_2' must have matched that defined by C_1, C_2 (see Challenger $\mathcal{C}_8$). By the simulation soundness of the proof π_4, which was verified by the honest $\mathcal{S}_2$, ciphertext $\tilde{C}_1'$ contains the same plaintext K_1 as the ciphertext $\tilde{C}_1$ that $\mathcal{S}_2$ stored during the successful setup protocol. We also know that the challenger simulates the honest server and user in such a way that the user will pretend that the ciphertext $\tilde{C}_2'$ will decrypt to K_2, the same plaintext that it pretended to be present in $\tilde{C}_2$ during the setup. Therefore, by simulation soundness of the proof system, the user will output the real key $K = K_1 K_2$.

If, on the other hand, in the case (2) that the password p' doesn't match, then $\mathcal{S}_2$ will never create a signed ciphertext $\tilde{C}_2'$, but rather output `fail` at the latest in Step R5 after verifying π_3, because it replaced the ciphertext E with the encryption of a random group element, so that, conditioned on the adversary not being able to create valid proofs of false statements, $\mathcal{S}_1$ will not be able to create a convincing proof π_3 that E decrypts to 1.

In summary, $\mathcal{S}_2$ will only sign $(C_1', C_2', PK_u, \tilde{C}_1', \tilde{C}_2')$ if the password p' defined by C_1', C_2' matches the password p defined by the ciphertexts C_1, C_2 that it recorded during setup, and if the key K' defined by $\tilde{C}_1', \tilde{C}_2'$ matches the key K defined by the ciphertexts $\tilde{C}_1, \tilde{C}_2$ that it recorded during setup. Therefore, if the password is correct, an honest user will either receive the correct key or output `fail`; if the password is false, then an honest user will always output `fail`.

The case when $\mathcal{S}_2$ is corrupt and $\mathcal{S}_1$ is honest is analogous and relies on the fact that an honest server's signature on the message that the user receives in Step R7 guarantees that the honest server has verified proofs π_2 and π_5, and on the fact that if the password is incorrect, $\mathcal{S}_1$ replaces E_1 with an encryption of a random group element, causing E to be an encryption of a random group element as well.

To argue that case (3) cannot occur, assume that a corrupt retrieve query is made with C_1', C_2' that decrypt to shares of an incorrect password $p' \neq p$. The challenger will decrypt the password shares p_1', p_2' from C_1', C_2' and manipulate the ciphertext E_1 (if $\mathcal{S}_1$ is honest) and/or E (if $\mathcal{S}_2$ is honest) to make sure that honest servers conclude that the password is incorrect and output `fail` in Steps R5 and/or R6.

Finally, (4) cannot occur because the signatures τ_i received in Step S5 and the signatures τ_i' received in Step R7 authenticate the full first message of the protocol as it was received by server $\mathcal{S}_i$. If the query was hijacked for $\mathcal{S}_i$, then the first message received by $\mathcal{S}_i$ was different than what the honest user sent out, so the signature τ_i or τ_i' will not verify correctly.

Challenger $\mathcal{C}_{11}$: In this game, the idea is to give the environment a view that is identical to view_{10}, but to have $\mathcal{C}_{11}$ internally run the full-fledged ideal functionality $\mathcal{F}$, and to have all the honest participants run the ideal protocol with $\mathcal{F}$; interaction with the adversary will now be based solely on what $\mathcal{F}$ sends to the ideal-world adversary. To this end, we turn the registry $\mathcal{R}$ into the internal book-keeping of $\mathcal{F}$: $\mathcal{F}$ keeps track of what the correct password is, and at what stage various attempts at setup and retrieve currently are. Essentially, $\mathcal{C}_{11}$ is now viewed not as a single interactive Turing machine (ITM), but as several ITMs interacting with each other: there is an ITM that executes the ideal functionality $\mathcal{F}$; a "dummy" ITM for each ideal-world honest participant that simply relays messages between the environment and $\mathcal{F}$; and an ITM $\mathcal{SIM}$ that talks to $\mathcal{F}$ on behalf of the ideal-world adversary and to $\mathcal{A}$ on behalf of the honest participants.

We have already described $\mathcal{F}$ and the ideal honest parties in Section 2. What remains to do is to describe $\mathcal{SIM}$ and to verify that the resulting view_{11} is identical to view_{10}. The description of $\mathcal{SIM}$ is given in detail in Section 5.2. In a nutshell, in order to view $\mathcal{C}_{10}$ as consisting of all these different ITMs, we observe that the protocol messages that the honest parties inside $\mathcal{C}_{10}$ send out either do not depend on the correct passwords and keys at all, so these protocol messages can be computed by the simulator $\mathcal{SIM}$ without access to the actual records, or they depend on the mere fact whether $p' = p$ and the registered key K, both of which are provided by $\mathcal{F}$ to $\mathcal{SIM}$ when they are needed.

Although the way that $\mathcal{C}_{11}$ is structured internally is different from the way $\mathcal{C}_{10}$ is structured (because $\mathcal{C}_{10}$ is not separating its computation steps into those carried out by $\mathcal{F}$, $\mathcal{SIM}$, and the honest ideal parties), each message $\mathcal{C}_{11}$ sends to $\mathcal{A}$ and $\mathcal{E}$ is computed identically to the messages that $\mathcal{C}_{10}$ sends to them, so this hop is purely conceptual. ∎

5.2 The Simulator

Here we describe the simulator $\mathcal{SIM}$ defined as part of $\mathcal{C}_{11}$ that acts as an adversary in the ideal world against the ideal functionality and that uses the real-world adversary $\mathcal{A}$ in such a way that the environment cannot distinguish whether it's interacting with $\mathcal{A}$ and the honest parties in the real world or with $\mathcal{SIM}$ and dummy parties in the ideal world.

We describe the simulator for a single session $sid = (u, \mathcal{S}_1, \mathcal{S}_2)$, i.e., for a single account with username u registered with servers $\mathcal{S}_1$ and $\mathcal{S}_2$. In the descriptions below, we denote by "$\mathcal{P}$" the simulator-created real-world machines corresponding to ideal-world participant $\mathcal{P}$. During the entire execution, the simulator relays all inputs from the environment to the real-world adversary $\mathcal{A}$ and relays all outputs of the adversary back to the environment. The simulator first generates a key pair $(PK, SK) \leftarrow_{\mathrm{R}} \mathsf{keyg}(1^k)$ and answers all queries to $\mathcal{F}_{CRS}$ with PK, so that it knows the decryption key for the CRS. It also generates key pairs $(PE_i, SE_i) \leftarrow_{\mathrm{R}} \mathsf{keyg2}(1^k)$ and $(PS_i, SS_i) \leftarrow_{\mathrm{R}} \mathsf{keygsig}(1^k)$ for all simulated honest servers "$\mathcal{S}_i$" and responds to queries $(\mathtt{Retrieve}, \mathcal{S}_i)$ to $\mathcal{F}_{CA}$ with $(\mathtt{Retrieve}, \mathcal{S}_i, (PS_i, SS_i))$. For all other queries, it honestly executes the code of the functionality $\mathcal{F}_{CA}$.

5.2.1 Setup by an Honest User

There are three cases: (a) both servers are honest, (b) both servers are corrupt, and (c) one of the servers is corrupt.

(a) Both servers honest The setup protocol gets activated when the environment provides an input $(\mathtt{Stp}, sid, qid, p, K)$ to an honest user $\mathcal{U}$, who relays it to $\mathcal{F}$. As a result, $\mathcal{SIM}$ receives $(\mathtt{Stp}, sid, qid, \mathcal{U})$ from $\mathcal{F}$. $\mathcal{SIM}$ will simulate all parties "$\mathcal{U}$", "$\mathcal{S}_1$", and "$\mathcal{S}_2$". $\mathcal{SIM}$ starts the setup protocol with the modifications described in Challengers $\mathcal{C}_1$ through $\mathcal{C}_{11}$—recall that these modifications guarantee that the messages that honest participants exchange are distributed independently of p and K) for "$\mathcal{U}$". It therefore computes ciphertexts F_i, C_i, and $\tilde{C}_i$ as encryptions of ones of the correct length. The simulator also records $(\mathtt{AStp}, qid, \perp, \perp)$.

The first server that receives a message $(\mathtt{Stp}, sid, qid, 1, F_i, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$. Recall that if this message is identical to the message sent by "$\mathcal{U}$", then we call this query qid *intact* for $\mathcal{S}_i$, otherwise we call it *hijacked* for $\mathcal{S}_i$.

If the query is intact for $\mathcal{S}_i$, then each simulated server "$\mathcal{S}_i$" continues the protocol but skips the Steps S2(c-d) or S3(c-d) where F_i is decrypted and tested against C_i and $\tilde{C}_i$. When "$\mathcal{S}_i$" outputs $(\mathtt{SRlt}, sid, qid, s)$, then $\mathcal{SIM}$ sends $(\mathtt{SRlt}, sid, qid, \mathcal{S}_i, s)$ to $\mathcal{F}$ and marks qid as s for $\mathcal{S}_i$. If now qid is marked $\mathtt{succ}$ for $\mathcal{S}_1$ and $\mathcal{S}_2$, then the simulator records $(\mathtt{Stp}, \perp, \perp)$. When the simulated user "$\mathcal{U}$" outputs $(\mathtt{SRlt}, sid, qid, s)$, then $\mathcal{SIM}$ sends $(\mathtt{SRlt}, sid, qid, \mathcal{U}, s)$ to $\mathcal{F}$.

Note that because honest users in the real protocol only output $\mathtt{succ}$ after having received signed success statements from both servers, $\mathcal{SIM}$ can correctly steer the honest users in the ideal world, where the ideal functionality imposes that users only output $\mathtt{succ}$ if the query is intact and after both servers have output $\mathtt{succ}$. Also note that, if the setup was successful, the simulator has a record $(\mathtt{Stp}, \perp, \perp)$, while $\mathcal{F}$ has recorded $(\mathtt{Stp}, p, K)$ for the actual password p and key K.

If the query is hijacked for $\mathcal{S}_i$, then the server "$\mathcal{S}_i$" who received the first message $(\mathtt{Stp}, sid, qid, 1, \ldots)$ uses the secret key SK corresponding to PK in the CRS to decrypt the ciphertexts $C_1, \tilde{C}_1, C_2, \tilde{C}_2$ to obtain shares $\hat{p}_1, \hat{K}_1, \hat{p}_2, \hat{K}_2$. It computes $\hat{p} \leftarrow \hat{p}_1 \hat{p}_2$ and $\hat{K} \leftarrow \hat{K}_1 \hat{K}_2$ and sends a message $(\mathtt{SHjk}, sid, qid, \hat{p}, \hat{K})$ to $\mathcal{F}$. It also creates a record $(\mathtt{AStp}, qid, \hat{p}, \hat{K})$ and marks qid as $\mathtt{hjkd}$ for $\mathcal{A}$. Other than that, "$\mathcal{S}_i$" simply follows the real protocol.

When any of the simulated servers "$\mathcal{S}_j$" outputs $(\mathtt{SRlt}, sid, qid, s)$, then $\mathcal{SIM}$ sends $(\mathtt{SRlt}, sid, qid, \mathcal{S}_j, s)$ to $\mathcal{F}$ and marks qid as s for $\mathcal{S}_i$. If now qid is marked $\mathtt{succ}$ for $\mathcal{S}_1$ and $\mathcal{S}_2$, then the simulator records $(\mathtt{Stp}, \hat{p}, \hat{K})$. When the simulated user "$\mathcal{U}$" outputs $(\mathtt{SRlt}, sid, qid, \mathtt{fail})$, then $\mathcal{SIM}$ sends $(\mathtt{SRlt}, sid, qid, \mathcal{U}, \mathtt{fail})$ to $\mathcal{F}$.

Note that "$\mathcal{U}$" will never output $\mathtt{succ}$ if the query is hijacked for $\mathcal{S}_i$, because the signature τ_i received in Step S5 will not verify correctly. Also note that if all honest servers successfully finished the protocol, $\mathcal{SIM}$ will have a record $(\mathtt{Stp}, p, K)$ for the same password p and key K as recorded in the ideal functionality $\mathcal{F}$.

(b) Both servers corrupt As soon as the environment provides input $(\mathtt{Stp}, sid, qid, p, K)$ to $\mathcal{U}$, $\mathcal{SIM}$ receives $(\mathtt{Stp}, sid, qid, \mathcal{U}, p, K)$ from $\mathcal{F}$. $\mathcal{SIM}$ simulates only "$\mathcal{U}$", which interacts with the servers controlled by $\mathcal{A}$. The simulator lets "$\mathcal{U}$" perform a setup protocol on input $(\mathtt{Stp}, sid, qid, p, K)$. When user "$\mathcal{U}$" outputs a message $(\mathtt{SRlt}, sid, qid, s)$, $\mathcal{SIM}$ outputs $(\mathtt{SRlt}, sid, qid, \mathcal{U}, s)$ to $\mathcal{F}$. The simulator does not record anything; note that the ideal functionality $\mathcal{F}$ also doesn't create a record $(\mathtt{Stp}, \cdot, \cdot)$ if both servers are corrupt.

(c) One server corrupt Let $\mathcal{S}_i$ be honest and $\mathcal{S}_j$ be corrupt. When the environment provides input $(\mathtt{Stp}, sid, qid, p, K)$ to $\mathcal{U}$, $\mathcal{SIM}$ receives $(\mathtt{Stp}, sid, qid, \mathcal{U})$ from $\mathcal{F}$. $\mathcal{SIM}$ simulates the real-world user "$\mathcal{U}$" and server "$\mathcal{S}_i$" in interaction with $\mathcal{A}$, who plays the role of $\mathcal{S}_j$. "$\mathcal{U}$" executes the Setup protocol with random shares $\hat{p}_1, \hat{p}_2, \hat{K}_1, \hat{K}_2 \leftarrow_{\mathrm{R}} \mathbb{G}$ but with F_i, C_i, and $\tilde{C}_i$ being encryptions of ones (see the description of challengers $\mathcal{C}_3$ and $\mathcal{C}_5$).

If the query is intact for $\mathcal{S}_i$, i.e., the message $(\texttt{Stp}, sid, qid, 1, F_i, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ that "$\mathcal{S}_i$" receives is identical to what was sent by "$\mathcal{U}$", then $\mathcal{SIM}$ creates a record $(\texttt{AStp}, qid, \bot, \bot)$ and lets "$\mathcal{S}_i$" continue the setup protocol, pretending that all ciphertexts decrypted correctly and simulating all proofs. If, on the other hand, the query is hijacked, then $\mathcal{SIM}$ decrypts the ciphertexts $C_1, \tilde{C}_1, C_2, \tilde{C}_2$ using SK to recover $\hat{p}_1$, $\hat{K}_1$, $\hat{p}_2$, and $\hat{K}_2$, respectively, computes $\hat{p} \leftarrow \hat{p}_1\hat{p}_2$ and $\hat{K} \leftarrow \hat{K}_1\hat{K}_2$, and sends a message $(\texttt{SHjk}, sid, qid, \hat{p}, \hat{K})$ to $\mathcal{F}$. It also creates a record $(\texttt{AStp}, qid, \hat{p}, \hat{K})$ and marks qid as $\texttt{hjkd}$ for $\mathcal{A}$.

When "$\mathcal{S}_i$" outputs $(\texttt{SRlt}, sid, qid, s)$, $\mathcal{SIM}$ sends $(\texttt{SRlt}, sid, qid, \mathcal{S}_i, s)$ to $\mathcal{F}$. If $s = \texttt{succ}$ then it looks up record $(\texttt{AStp}, qid, p, K)$ and records $(\texttt{Stp}, p, K)$. When "$\mathcal{U}$" outputs $(\texttt{SRlt}, sid, qid, s')$, then $\mathcal{SIM}$ sends $(\texttt{SRlt}, sid, qid, \mathcal{U}, s')$ to $\mathcal{F}$.

Note that, if $\mathcal{S}_i$ outputs $\texttt{succ}$, we have that $(p, K) = (\bot, \bot)$ if the query is intact and $(p, K) = (\hat{p}, \hat{K})$ if the query is hijacked. In the latter case, $\mathcal{SIM}$'s record matches that of $\mathcal{F}$, but in the former case it does not.

5.2.2 Setup by a Dishonest User

Again, there are three cases: (a) both servers are honest, (b) both servers are corrupt, and (c) one of the servers is corrupt.

(a) Both servers honest We have to ensure that $\mathcal{F}$ will record the correct p and K if the setup protocol succeeds and that the outputs that $\mathcal{F}$ sends to the ideal-world $\mathcal{S}_1$ and $\mathcal{S}_2$ are correct in terms of content as well as timing with respect to the real-world $\mathcal{S}_1$ and $\mathcal{S}_2$. To this end, $\mathcal{SIM}$ will run simulated parties "$\mathcal{S}_1$" and "$\mathcal{S}_2$" that follow the normal Setup protocol. However, when a server "$\mathcal{S}_i$" receives a message $(\texttt{Stp}, sid, qid, 1, F_i, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ from $\mathcal{A}$ as coming from real-world user $\mathcal{U}$, then $\mathcal{SIM}$ decrypts ciphertexts $C_1, \tilde{C}_1, C_2, \tilde{C}_2$ using the secret key SK corresponding to PK in the CRS to obtain shares p_1, K_1, p_2, K_2, computes $p \leftarrow p_1p_2$ and $K \leftarrow K_1K_2$, and sends a message $(\texttt{Stp}, sid, qid, p, K)$ to $\mathcal{F}$ as coming from ideal-world user $\mathcal{U}$. (This actually causes the $\texttt{Stp}$ message to be sent twice for the same qid, once for "$\mathcal{S}_1$" and once for "$\mathcal{S}_2$", but the message to arrive last is simply ignored by the functionality. The decrypted password and key may even be different for both cases, but then the honest servers "$\mathcal{S}_i$" will output $\texttt{fail}$ anyway.) It also creates a record $(\texttt{AStp}, qid, p, K)$.

When a server "$\mathcal{S}_i$" outputs $(\texttt{SRlt}, sid, qid, s)$, then $\mathcal{SIM}$ sends $(\texttt{SRlt}, sid, qid, \mathcal{S}_i, s)$ to $\mathcal{F}$ and marks qid as s for $\mathcal{S}_i$. If now qid is marked $\texttt{succ}$ for $\mathcal{S}_1$ and $\mathcal{S}_2$, then the simulator looks up record $(\texttt{AStp}, qid, p, K)$ and records $(\texttt{Stp}, p, K)$.

Note that the real protocol will never output $\texttt{succ}$ without having received the first message, so the setup result is always delivered after a valid setup request. Also note that $\mathcal{SIM}$ will have a record $(\texttt{Stp}, p, K)$ for the same password p and key K as is recorded in the ideal functionality $\mathcal{F}$.

(b) Both servers corrupt This case is entirely internal to the adversary, and there is nothing for the simulator to do.

(c) One server corrupt Let $\mathcal{S}_i$ be honest and $\mathcal{S}_j$ be corrupt. $\mathcal{SIM}$ simulates a real-world party "$\mathcal{S}_i$" which interacts with the adversarial $\mathcal{U}$ and $\mathcal{S}_j$. Its strategy is similar to case (a) above. Namely, when "$\mathcal{S}_i$" receives from $\mathcal{U}$ a message $(\texttt{Stp}, sid, qid, 1, F_i, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ for "$\mathcal{S}_i$", it decrypts the shares p_1, K_1, p_2, K_2 using SK, computes $p \leftarrow p_1p_2$ and $K \leftarrow K_1K_2$, sends a message $(\texttt{Stp}, sid, qid, p, K)$ to $\mathcal{F}$, and creates a record $(\texttt{AStp}, qid, p, K)$. When "$\mathcal{S}_i$" outputs $(\texttt{SRlt}, sid, qid, s)$, then $\mathcal{SIM}$ sends $(\texttt{SRlt}, sid, qid, \mathcal{S}_i, s)$ to $\mathcal{F}$. If $s = \texttt{succ}$, then the simulator looks up record $(\texttt{AStp}, qid, p, K)$ and records $(\texttt{Stp}, p, K)$.

Note that also in this case, the record $(\texttt{Stp}, p, K)$ kept by $\mathcal{SIM}$ is identical to that kept by $\mathcal{F}$.

Records of the Simulator The simulator $\mathcal{SIM}$ will have a single record $(\texttt{Stp}, \cdot, \cdot)$ if at least one server is honest and if all honest servers successfully completed the setup for some setup query qid. This matches the way $\mathcal{F}$ maintains a record $(\texttt{Stp}, \cdot, \cdot)$, but the contents of the record held by $\mathcal{SIM}$ and $\mathcal{F}$ may be different. Namely, $\mathcal{SIM}$ will have the correct password and key on file if the setup succeeded for a corrupt query (i.e., a query that was initiated by a dishonest user, or a query initiated by an honest user but the first message was modified on the network), but will have $\bot$ in their place if it succeeded for an intact query.

For our protocol, when both servers are honest, it is possible that "$\mathcal{S}_2$" successfully completed setup in Step S3 but that "$\mathcal{S}_1$" either didn't yet reach Step S4 or even failed in Step S4 (due to the adversary delaying or tampering with the message coming from "$\mathcal{S}_2$", respectively). In this case, $\mathcal{SIM}$ will have registered state for "$\mathcal{S}_2$" and will have marked qid as $\texttt{succ}$ for $\mathcal{S}_2$, but not for $\mathcal{S}_1$. This matches $\mathcal{F}$'s records at the same point, which will have qid marked as $\texttt{succ}$ for $\mathcal{S}_2$ but unmarked or marked as $\texttt{fail}$ for $\mathcal{S}_1$.

Note that due to Step S4, the inverse situation where "$\mathcal{S}_1$" accepted a setup that "$\mathcal{S}_2$" did not (yet) accept, cannot occur. Consequently, it also cannot occur that "$\mathcal{S}_1$" and "$\mathcal{S}_2$" accepted different setup queries.

5.2.3 Retrieve by an Honest User

Consider an honest user carrying out the retrieval protocol. We again have three cases: (a) both servers are honest, (b) both servers are corrupt, and (c) one server is corrupt. Note that the user who performs the retrieve may not be the same as the user who created the account.

(a) Both servers honest Here $\mathcal{SIM}$ simulates the real-world user "$\mathcal{U}'$" and both real-world servers "$\mathcal{S}_1$" and "$\mathcal{S}_2$"; the real-world adversary $\mathcal{A}$ controls the network between them. The protocol starts when $\mathcal{SIM}$ receives $(\mathtt{Rtr}, sid, qid', \mathcal{U}')$ from the ideal functionality $\mathcal{F}$. $\mathcal{SIM}$ lets "$\mathcal{U}'$" executes the Retrieve protocol with the simulated honest servers "$\mathcal{S}_1$" and "$\mathcal{S}_2$" as follows: the step R1 is computed as per modifications described in Challengers $\mathcal{C}_3$ and $\mathcal{C}_5$, i.e., with ciphertexts F_i and C'_i encrypting ones instead of the real password. Assuming that the adversary delivers messages sent between honest participants, the servers "$\mathcal{S}_1$" and "$\mathcal{S}_2$" also follow the protocol, with the modifications described below.

If the message $(\mathtt{Rtr}, sid, qid', 1, F'_1, C'_1, C'_2, PK_u)$ arrives at server "$\mathcal{S}_i$" intact, then $\mathcal{SIM}$ continues the simulation as follows. When "$\mathcal{S}_i$" outputs $(\mathtt{RNtf}, sid, qid')$, $\mathcal{SIM}$ sends $(\mathtt{RNtf}, sid, qid', \mathcal{S}_i)$ to $\mathcal{F}$. Whenever $\mathcal{SIM}$ receives $(\mathtt{Perm}, sid, qid', \mathcal{S}_i, a)$ from $\mathcal{F}$, $\mathcal{SIM}$ provides "$\mathcal{S}_i$" with input $(\mathtt{Perm}, sid, qid', a)$.

If $\mathcal{SIM}$ receives $(\mathtt{Perm}, sid, qid', \mathcal{S}_i, \mathtt{deny})$ from $\mathcal{F}$ for at least one of $\mathcal{S}_1, \mathcal{S}_2$ (this corresponds to the event when a server is directed by the environment not to participate in the retrieval because, for example, too many unsuccessful attempts have been made to retrieve for this sid), then the simulated protocol will fail eventually.

If the environment permits both ideal servers to continue (i.e., if $\mathcal{F}$ obtains input $(\mathtt{Perm}, sid, qid', \mathcal{S}_i, \mathtt{allow})$ for both $\mathcal{S}_1$ and $\mathcal{S}_2$), then $\mathcal{SIM}$ receives an additional message $(\mathtt{Rtr}, sid, qid', c, \perp)$ indicating whether the password provided by the ideal-world user $\mathcal{U}'$ is correct. If $c = \mathtt{wrong}$ then $\mathcal{SIM}$ continues executing the modified protocol by letting "$\mathcal{S}_1$" replace ciphertext E_1 with an encryption of 1, letting "$\mathcal{S}_2$" replace E with an encryption of a random group element, causing "$\mathcal{S}_1$" to fail in Step R4(c), and by letting both servers simulate all proofs (see $\mathcal{C}_4$, $\mathcal{C}_7$, and $\mathcal{C}_8$). If $c = \mathtt{correct}$, then "$\mathcal{S}_1$" and "$\mathcal{S}_2$" replace E_1, E, $\tilde{C}'_1$, and $\tilde{C}'_2$ with encryptions of ones, respectively, and fake all proofs (see $\mathcal{C}_7$, $\mathcal{C}_8$, and $\mathcal{C}_9$).

If the message $(\mathtt{Rtr}, sid, qid', 1, F'_1, C'_1, C'_2, PK_u)$ received by server "$\mathcal{S}_i$" is not the same as was sent by the user, however, then the query was hijacked by the adversary. In this case, $\mathcal{SIM}$ decrypts $\hat{p}'_1$ and $\hat{p}'_2$ from C'_1 and C'_2 using SK, computes $\hat{p}' \leftarrow \hat{p}'_1\hat{p}'_2$, and sends a message $(\mathtt{RHjk}, sid, qid', \hat{p}')$ to $\mathcal{F}$.

It then continues the simulation of "$\mathcal{S}_1$" and "$\mathcal{S}_2$" similar to the case of a retrieve by a corrupt user described in Section 5.2.4(a). Namely, when "$\mathcal{S}_i$" outputs $(\mathtt{RNtf}, sid, qid')$, then $\mathcal{SIM}$ sends $(\mathtt{RNtf}, sid, qid', \mathcal{S}_i)$ to $\mathcal{F}$. If $\mathcal{SIM}$ receives two messages $(\mathtt{Perm}, sid, qid', \mathcal{S}_1, \mathtt{allow})$ and $(\mathtt{Perm}, sid, qid', \mathcal{S}_2, \mathtt{allow})$, then it also receives a message $(\mathtt{Rtr}, sid, qid', c, K)$ from $\mathcal{F}$ indicating whether the password is correct, and if it is correct, including the key K.

If $c = \mathtt{wrong}$ then $\mathcal{SIM}$ lets "$\mathcal{S}_1$" replace ciphertext E_1 with an encryption of 1, lets "$\mathcal{S}_2$" replace E with an encryption of a random group element, lets "$\mathcal{S}_1$" fail in Step R4(c), and simulates all proofs. If $c = \mathtt{correct}$, then $\mathcal{SIM}$ lets "$\mathcal{S}_1$" and "$\mathcal{S}_2$" replace E_1 and E with encryptions of ones and lets them fake all proofs (see $\mathcal{C}_7$ and $\mathcal{C}_8$). If this account was set up through a corrupt setup query, then "$\mathcal{S}_1$" and "$\mathcal{S}_2$" have correct shares K_1 and K_2 of the actual key K in their state information. In this case, the ciphertexts $\tilde{C}'_1$ and $\tilde{C}'_2$ are constructed normally, containing the same key shares K_1 and K_2 as registered in the state information during setup. If this account was setup through an intact setup query from an honest user and this is the first time that a dishonest user retrieves it with the correct password, then $\mathcal{SIM}$ chooses $K_1 \leftarrow_{\mathrm{R}} \mathbb{G}$, computes $K_2 \leftarrow K/K_1$, stores these shares in the states of "$\mathcal{S}_1$" and "$\mathcal{S}_2$", respectively, and encrypts them to create ciphertexts $\tilde{C}'_1$ and $\tilde{C}'_2$, simulating all proofs. For all subsequent retrieve queries by a dishonest user with the correct password, the same key shares K_1 and K_2 are used, as described in challenger $\mathcal{C}_9$.

Both for intact and hijacked queries, $\mathcal{SIM}$ continues the rest of the protocol as follows. When "$\mathcal{S}_i$" outputs $(\mathtt{RRlt}, sid, qid', s)$, then $\mathcal{SIM}$ sends $(\mathtt{RRlt}, sid, qid', \mathcal{S}_i, a)$ to $\mathcal{F}$, where $a \leftarrow \mathtt{allow}$ if $s = \mathtt{succ}$ and $a \leftarrow \mathtt{deny}$ if $s = \mathtt{fail}$. Note that $s = \mathtt{succ}$ can only occur when $c = \mathtt{correct}$, i.e., after $\mathcal{F}$ communicated to $\mathcal{SIM}$ that the password provided by $\mathcal{U}'$ in the ideal world is correct. Hence, by sending $(\mathtt{RRlt}, sid, qid', \mathcal{S}_i, \mathtt{allow})$ to $\mathcal{F}$, the ideal-world $\mathcal{S}_i$ will also output $(\mathtt{RRlt}, sid, qid', \mathtt{succ})$.

When "$\mathcal{U}'$" outputs $(\mathtt{RRlt}, sid, qid', s)$, then $\mathcal{SIM}$ sends $(\mathtt{RRlt}, sid, qid', \mathcal{U}', a)$ to $\mathcal{F}$, where $a \leftarrow \mathtt{allow}$ if $s = \mathtt{succ}$ and $a \leftarrow \mathtt{deny}$ if $s = \mathtt{fail}$.

(b) Both servers corrupt In this case $\mathcal{SIM}$ simulates "$\mathcal{U}'$". First, $\mathcal{SIM}$ receives $(\texttt{Rtr}, sid, qid', \mathcal{U}', p')$ from $\mathcal{F}$. It starts the honest retrieve protocol for "$\mathcal{U}'$" on input $(\texttt{Rtr}, sid, qid', p')$. When the user "$\mathcal{U}'$" eventually outputs $(\texttt{RRlt}, sid, qid', K', s)$, then $\mathcal{SIM}$ sends $(\texttt{RRlt}, sid, qid', \mathcal{U}', \texttt{deny}, \perp)$ to $\mathcal{F}$ if $(K', s) = (\perp, \texttt{fail})$, or sends $(\texttt{RRlt}, sid, qid', \mathcal{U}', \texttt{allow}, K')$ otherwise.

(c) One server corrupt Let $\mathcal{S}_i$ be honest and $\mathcal{S}_j$ be corrupt. $\mathcal{SIM}$ receives $(\texttt{Rtr}, sid, qid', \mathcal{U}')$ from $\mathcal{F}$. $\mathcal{SIM}$ creates real-world "$\mathcal{U}'$" and "$\mathcal{S}_i$" and talks on their behalf to $\mathcal{S}_j$, played by $\mathcal{A}$. In the simulation of "$\mathcal{U}'$", the simulator picks $p'_j \leftarrow_{\mathrm{R}} \mathbb{G}$, it replaces C'_i and F'_i with encryptions of ones (see $\mathcal{C}_3$ and $\mathcal{C}_5$).

We first consider the case of an intact query for $\mathcal{S}_i$, i.e., where the first message $(\texttt{Rtr}, sid, qid', 1, F'_1, C'_1, C'_2, PK_u)$ received by "$\mathcal{S}_i$" is the same as what was sent by "$\mathcal{U}'$". When "$\mathcal{S}_i$" outputs $(\texttt{RNtf}, sid, qid')$, then $\mathcal{SIM}$ sends $(\texttt{RNtf}, sid, qid', \mathcal{S}_i)$ to $\mathcal{F}$. When $\mathcal{SIM}$ receives $(\texttt{Perm}, sid, qid', \mathcal{S}_i, a)$ from $\mathcal{F}$, then $\mathcal{SIM}$ provides input $(\texttt{Perm}, sid, qid', a)$ to "$\mathcal{S}_i$".

If $a = \texttt{allow}$, meaning that the environment let $\mathcal{S}_i$ proceed with the retrieve protocol, then $\mathcal{SIM}$ as the ideal-world adversary receives a message $(\texttt{Rtr}, sid, qid', c, \perp)$ from $\mathcal{F}$, indicating whether the password submitted by $\mathcal{U}'$ in the ideal world was correct. At this point, "$\mathcal{S}_i$" proceeds with the protocol with the adversarial $\mathcal{S}_j$ according to challengers $\mathcal{C}_7$, $\mathcal{C}_8$, and $\mathcal{C}_9$. Namely, if $\mathcal{S}_i = \mathcal{S}_1$ and $c = \texttt{correct}$, then "$\mathcal{S}_1$" replaces E_1 and $\tilde{C}'_1$ with encryptions of ones, pretends that E decrypted to 1, and simulates all proofs. If $\mathcal{S}_i = \mathcal{S}_1$ and $c = \texttt{wrong}$, then "$\mathcal{S}_1$" replaces E_1 with encryptions of ones, simulates all proofs, and in Step R4 fails. If $\mathcal{S}_i = \mathcal{S}_2$ and $c = \texttt{wrong}$, then "$\mathcal{S}_2$" replaces E and $\tilde{C}'_2$ with encryptions of ones and simulates all proofs. If $\mathcal{S}_i = \mathcal{S}_2$ and $c = \texttt{correct}$, then "$\mathcal{S}_2$" replaces E with the encryption of a random group element, causing it to fail in Step R5.

When "$\mathcal{S}_i$" outputs $(\texttt{RRlt}, sid, qid', s)$, then $\mathcal{SIM}$ sends $(\texttt{RRlt}, sid, qid', \mathcal{S}_i, a)$ to $\mathcal{F}$, where $a \leftarrow \texttt{allow}$ if $s = \texttt{succ}$ and $a \leftarrow \texttt{deny}$ if $s = \texttt{fail}$. When the simulated user "$\mathcal{U}'$" eventually outputs $(\texttt{RRlt}, sid, qid', K', s')$, $\mathcal{SIM}$ sends $(\texttt{RRlt}, sid, qid', \mathcal{U}', a, \perp)$ to $\mathcal{F}$, where $a \leftarrow \texttt{allow}$ if $s' = \texttt{succ}$ and $a \leftarrow \texttt{deny}$ if $s' = \texttt{fail}$.

In the case that the query is hijacked for $\mathcal{S}_i$, i.e., where the first message $(\texttt{Rtr}, sid, qid', 1, F'_i, C'_1, C'_2, PK_u)$ received by "$\mathcal{S}_i$" was modified in transit over the network, $\mathcal{SIM}$ proceeds as follows. It decrypts $\hat{p}'_1$ and $\hat{p}'_2$ from C'_1 and C'_2 using SK, computes $\hat{p}' \leftarrow \hat{p}'_1\hat{p}'_2$, and sends a message $(\texttt{RHjk}, sid, qid', \hat{p}')$ to $\mathcal{F}$.

It then continues the simulation of "$\mathcal{S}_i$" similar to the case of a retrieve by a corrupt user described in Section 5.2.4(c). Namely, when "$\mathcal{S}_i$" outputs $(\texttt{RNtf}, sid, qid')$, then $\mathcal{SIM}$ sends $(\texttt{RNtf}, sid, qid', \mathcal{S}_i)$ to $\mathcal{F}$. If $\mathcal{SIM}$ receives a message $(\texttt{Perm}, sid, qid', \mathcal{S}_i, \texttt{allow})$, then it also receives a message $(\texttt{Rtr}, sid, qid', c, K)$ from $\mathcal{F}$ indicating whether the password is correct, and if it is correct, including the key K.

If $\mathcal{S}_i = \mathcal{S}_1$ and $c = \texttt{wrong}$ then $\mathcal{SIM}$ lets "$\mathcal{S}_1$" replace ciphertext E_1 with an encryption of 1 and lets "$\mathcal{S}_1$" fail in Step R4(c), simulating all proofs. If $\mathcal{S}_i = \mathcal{S}_1$ and $c = \texttt{correct}$ then $\mathcal{SIM}$ lets "$\mathcal{S}_1$" replace E_1 with an encryption of 1 and lets it fake all proofs, including the proof π_3 that E decrypted to one (see $\mathcal{C}_7$ and $\mathcal{C}_8$). If $\mathcal{S}_i = \mathcal{S}_2$ and $c = \texttt{wrong}$ then $\mathcal{SIM}$ lets "$\mathcal{S}_2$" replace ciphertext E with an encryption of a random group element, causing the adversarially controlled server $\mathcal{S}_1$ to fail. If $\mathcal{S}_i = \mathcal{S}_2$ and $c = \texttt{correct}$ then $\mathcal{SIM}$ lets "$\mathcal{S}_2$" replace E with an encryption of 1 and lets it fake all proofs.

If this account was set up through a corrupt setup query, then "$\mathcal{S}_i$" has a share K_i in its state information, so it creates the ciphertext $\tilde{C}'_i$ as an encryption of K_i. If this account was setup through an intact setup query from an honest user and this is the first time that a dishonest user retrieves it with the correct password, then $\mathcal{SIM}$ chooses $K_i \leftarrow_{\mathrm{R}} \mathbb{G}$, stores K_i in the states of "$\mathcal{S}_i$", and computes $\tilde{C}'_i$ as an encryption of K_i. For all subsequent corrupt retrieve queries, it uses the same key share K_i, as described in challenger $\mathcal{C}_9$.

5.2.4 Retrieve by a Dishonest User

Again, there are three cases: (a) both servers are honest; (b) both servers are corrupt; (c) one of the servers is corrupt.

(a) Both servers honest The simulator $\mathcal{SIM}$ will run simulated parties "$\mathcal{S}_1$" and "$\mathcal{S}_2$" that will interact with an adversarial user played by $\mathcal{A}$.

When a server "$\mathcal{S}_i$" receives the first message $(\texttt{Rtr}, sid, qid', 1, F'_i, C'_1, C'_2, PK_u)$, the simulator decrypts C'_1 and C'_2 using the CRS secret key SK to obtain p'_1, p'_2, and computes $p' \leftarrow p'_1 p'_2$. It also retrieves from its state, if it has any, the ciphertexts $(C_1, C_2, \tilde{C}_1, \tilde{C}_2)$ that "$\mathcal{S}_i$" stored at the time when setup was run for this sid. (If setup has not been completed yet, then in accordance with the protocol, $\mathcal{S}_i$ will fail, causing the other server to fail later as well.) $\mathcal{SIM}$ then passes an input $(\texttt{Rtr}, sid, qid', p')$ on behalf of $\mathcal{U}'$ to $\mathcal{F}$. (Note that for each of the servers $\mathcal{S}_1$ and $\mathcal{S}_2$, a separate input $(\texttt{Rtr}, sid, qid', \cdot)$ will be generated, but the input arriving last will be ignored by $\mathcal{F}$. The two inputs may even be for different passwords p' if the first messages arriving at both servers contain different ciphertexts C_1', C_2'. This doesn't influence the view of the adversary or environment, however, since these protocols will conclude in Step R3(e) that the password was wrong anyway.)

When "$\mathcal{S}_i$" outputs $(\texttt{RNtf}, sid, qid')$, then $\mathcal{SIM}$ sends $(\texttt{RNtf}, sid, qid', \mathcal{S}_i)$ to $\mathcal{F}$. Whenever $\mathcal{F}$ sends $(\texttt{Perm}, sid, qid', \mathcal{S}_i, a)$ to $\mathcal{SIM}$, $\mathcal{SIM}$ provides "$\mathcal{S}_i$" with input $(\texttt{Perm}, sid, qid', a)$. If $\mathcal{SIM}$ received two messages $(\texttt{Perm}, sid, qid', \mathcal{S}_i, \texttt{allow})$, then it also receives a message $(\texttt{Rtr}, sid, qid', c, K)$ from $\mathcal{F}$. If $c = \texttt{wrong}$ then $\mathcal{SIM}$ lets "$\mathcal{S}_1$" replace ciphertext E_1 with an encryption of 1, lets "$\mathcal{S}_2$" replace E with an encryption of a random group element, lets "$\mathcal{S}_1$" fail in Step R4(c), and simulates all proofs. If $c = \texttt{correct}$, then $\mathcal{SIM}$ lets "$\mathcal{S}_1$" and "$\mathcal{S}_2$" replace E_1 and E with encryptions of ones and lets them fake all proofs (see $\mathcal{C}_7$ and $\mathcal{C}_8$).

If this account was setup through a corrupt setup query, then "$\mathcal{S}_1$" and "$\mathcal{S}_2$" have correct shares K_1 and K_2 of the actual key K in their state information. In this case, the ciphertexts $\tilde{C}_1'$ and $\tilde{C}_2'$ are constructed normally, containing the same key shares K_1 and K_2 as registered in the state information during setup.

If this account was setup through an honest setup query and this is the first time that a corrupt query retrieves it with the correct password, then $\mathcal{SIM}$ chooses $K_1 \leftarrow_{\mathrm{R}} \mathbb{G}$, computes $K_2 \leftarrow K/K_1$, stores these shares in the states of "$\mathcal{S}_1$" and "$\mathcal{S}_2$", respectively, encrypts them to create ciphertexts $\tilde{C}_1'$ and $\tilde{C}_2'$, and simulates all proofs. For all subsequent corrupt retrieve queries with the correct password, the same key shares K_1 and K_2 are used, as described in challenger $\mathcal{C}_9$.

When "$\mathcal{S}_i$" outputs $(\texttt{RRlt}, sid, qid', s)$, then $\mathcal{SIM}$ sends $(\texttt{RRlt}, sid, qid', \mathcal{S}_i, a)$ to $\mathcal{F}$, where $a \leftarrow \texttt{allow}$ if $s = \texttt{succ}$ and $a \leftarrow \texttt{deny}$ if $s = \texttt{fail}$.

(b) Both servers corrupt This case is entirely internal to the adversary, so there is nothing for the simulator to do.

(c) One server corrupt Suppose first that $\mathcal{S}_1$ is corrupt and $\mathcal{S}_2$ honest. The simulator $\mathcal{SIM}$ will run a simulated party "$\mathcal{S}_2$" as shown in the description of $\mathcal{C}_8$ and $\mathcal{C}_9$: When "$\mathcal{S}_2$" receives from the real-world user $\mathcal{U}'$ (controlled by $\mathcal{A}$) a message $(\texttt{Rtr}, sid, qid', 1, F_1', C_1', C_2', PK_u)$, the simulator decrypts C_1' and C_2' using the CRS secret key SK to obtain p_1', p_2', computes $p' \leftarrow p_1' p_2'$, and provides input $(\texttt{Rtr}, sid, qid', p')$ to $\mathcal{F}$ on behalf of $\mathcal{U}'$.

When "$\mathcal{S}_2$" outputs $(\texttt{RNtf}, sid, qid')$, then $\mathcal{SIM}$ sends $(\texttt{RNtf}, sid, qid', \mathcal{S}_2)$ to $\mathcal{F}$. When $\mathcal{SIM}$ receives $(\texttt{Perm}, sid, qid', \mathcal{S}_2, a)$ from $\mathcal{F}$, $\mathcal{SIM}$ inputs $(\texttt{Perm}, sid, qid', a)$ to "$\mathcal{S}_2$".

If $a = \texttt{allow}$ then $\mathcal{SIM}$ receives an additional message $(\texttt{Rtr}, sid, qid', c, K)$ from $\mathcal{F}$. If $c = \texttt{wrong}$, then "$\mathcal{S}_2$" acts the way described in $\mathcal{C}_7$ when the passwords do not match (i.e., makes E an encryption of a random group element and simulates all proofs). If $c = \texttt{correct}$, then "$\mathcal{S}_2$" acts the way described in $\mathcal{C}_7$ when the passwords match (i.e. it replaces the ciphertext E in Step R3 with an encryption of 1 and simulates the zero-knowledge proof π_2).

If this account was setup through a corrupt setup query, then "$\mathcal{S}_2$" has a correct key share K_2 of the actual key K in its state. In this case, it constructs the ciphertext $\tilde{C}_2'$ normally, containing the same key share K_2.

If this account was setup through an honest setup query and this is the first time that a corrupt query retrieves it with the correct password, then $\mathcal{SIM}$ decrypts $\tilde{C}_1$ to obtain K_1, computes $K_2 \leftarrow K/K_1$, stores K_2 in the state of "$\mathcal{S}_2$", encrypts it to create ciphertext $\tilde{C}_2'$, and simulates all proofs. For all subsequent corrupt retrieve queries with the correct password, the same key share K_2 is used.

When "$\mathcal{S}_2$" outputs $(\texttt{RRlt}, sid, qid', s)$, $\mathcal{SIM}$ sends a message $(\texttt{RRlt}, sid, qid', \mathcal{S}_2, a)$ to $\mathcal{F}$, where $a \leftarrow \texttt{allow}$ if $s = \texttt{succ}$ and $a \leftarrow \texttt{deny}$ if $s = \texttt{fail}$. Note that "$\mathcal{S}_2$" will only succeed if $\mathcal{F}$ previously communicated that the password was correct.

The case that $\mathcal{S}_1$ is honest and $\mathcal{S}_2$ is corrupt is analogous, with "$\mathcal{S}_1$" doing what we described in $\mathcal{C}_9$.

6 Conclusion

We have presented a protocol that allows a user to securely store a secret with two servers and retrieve from them using a human- memorizable password. If at least one of the servers is honest, the secret and password are protected from all attacks except on- line dictionary attacks; the latter, however, can be detected and countered by the honest server. Our protocol is reasonably efficient and provably secure in the UC framework, guaranteeing that it remains secure in arbitrary usage contexts and for arbitrary password distributions. Open problems include strengthening our protocol to withstand adaptive corruptions, designing a UC-secure 2PASS scheme in the password-only (i.e., non-public-key) model, and to build UC-secure protocols for the t-out-of-n case.

Acknowledgments

We thank Kristiyan Haralambiev, Stas Jarecki, Anja Lehmann, Victor Shoup, and the anonymous referees for their valuable feedback and comments on earlier versions of this work. This work was supported in part by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II. The first and third authors were supported in part by EC Grant Agreement 257782 ABC4Trust. The second author is supported by the United States NSF grants 1012060, 0964379, 0831293.

References

[BDN+11] William E. Burr, Donna F. Dodson, Elaine M. Newton, Ray A. Perlner, W. Timothy Polk, Sarbari Gupta, and Emad A. Nabbus. Electronic authentication guideline. NIST Special Publication 800-63-1, 2011.

[BJKS03] John Brainard, Ari Juels, Burton S. Kaliski Jr., and Michael Szydlo. A new two-server approach for authentication with short secrets. In *Proceedings of the 12th USENIX Security Symposium (SECURITY 2003)*, pages 201–214, Washington, DC, USA, August 5–9, 2003. USENIX Association.

[BJSL11] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 433–444. ACM, 2011.

[BLR04] Boaz Barak, Yehuda Lindell, and Tal Rabin. Protocol initialization for the framework of universal composability. Cryptology ePrint Archive, Report 2004/006, 2004. `http://eprint.iacr.org/`.

[BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, May 1992.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 139–155. Springer, May 2000.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73. ACM Press, November 1993.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145. IEEE Computer Society Press, October 2001.

[Can04] Ran Canetti. Universally composable signature, certification, and authentication. In *17th Computer Security Foundations Workshop*, page 219. IEEE Computer Society, 2004.

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 404–421. Springer, May 2005.

[CKS11] Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 449–467. Springer, 2011.

[CKY09] Jan Camenisch, Aggelos Kiayias, and Moti Yung. On the portability of generalized schnorr proofs. In Antoine Joux, editor, *EUROCRYPT*, pages 425–442, 2009.

[CR03] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281. Springer, August 2003.

[CS97] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In Burt Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume 1296 of *Lecture Notes in Computer Science*, pages 410–424. Springer Verlag, 1997.

[DG03] Mario Di Raimondo and Rosario Gennaro. Provably secure threshold password-authenticated key exchange. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 507–523. Springer, May 2003.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology — CRYPTO '84*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Verlag, 1985.

[FK00] Warwick Ford and Burton S. Kaliski Jr. Server-assisted generation of a strong secret from a password. In *9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*, pages 176–180, Gaithersburg, MD, USA, June 4–16, 2000. IEEE Computer Society.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, August 1999.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer Verlag, 1987.

[GLNS93] Li Gong, T. Mark A. Lomas, Roger M. Needham, and Jerome H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, June 1993.

[GMY03] Juan A. Garay, Philip D. MacKenzie, and Ke Yang. Strengthening zero-knowledge protocols using signatures. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 177–194. Springer, May 2003.

[HK99] Shai Halevi and Hugo Krawczyk. Public-key cryptography and password protocols. *ACM Transactions on Information and System Security*, 2(3):230–268, 1999.

[Jab01] David P. Jablon. Password authentication using multiple servers. In David Naccache, editor, *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 344–360. Springer, April 2001.

[KMTG05] Jonathan Katz, Philip D. MacKenzie, Gelareh Taban, and Virgil D. Gligor. Two-server password-only authenticated key exchange. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05: 3rd International Conference on Applied Cryptography and Network Security*, volume 3531 of *Lecture Notes in Computer Science*, pages 1–16. Springer, June 2005.

[KOY09] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient and secure authenticated key exchange using weak passwords. *Journal of the ACM*, 57(1), 2009.

[MSJ02] Philip D. MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 385–400. Springer, August 2002.

[MY04] Philip D. MacKenzie and Ke Yang. On simulation-sound trapdoor commitments. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 382–400. Springer, 2004.

[PS96] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 387–398. Springer Verlag, 1996.

[PW00] Birgit Pfitzmann and Michael Waidner. Composition and integrity preservation of secure reactive systems. In *Proc. 7th ACM Conference on Computer and Communications Security*, pages 245–254. ACM press, November 2000.

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *40th Annual Symposium on Foundations of Computer Science*, pages 543–553. IEEE Computer Society Press, October 1999.

[Sch91] Claus P. Schnorr. Efficient signature generation for smart cards. *Journal of Cryptology*, 4(3):239–252, 1991.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SK05] Michael Szydlo and Burton S. Kaliski Jr. Proofs for two-server password authentication. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 227–244. Springer, February 2005.