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 even no 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.

Categories and Subject Descriptors

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

Keywords

Password-authenticated secret sharing, universal composability

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, and securely (which in some cases means manually) entering strong cryptographic keys on devices is too tedious for most users.

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 [7] 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.

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 [5, 24, 25, 3, 29, 14].

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 [36] 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 a 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 [20] 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 [27] proposed a similar protocol in the password-only setting, i.e., where the user cannot remember public keys. Brainard et al. [6] 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. [30], is a t-out-of-n protocol supporting $t < n$ malicious servers in the public-key setting. Szydlo and Kaliski [37] provided security proofs for slight variants of the two-server protocol by Brainard et al. [6] mentioned earlier. Di Raimondo and Gennaro [17] 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 [4] 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. [28], 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. [1] 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 [12]. 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 [1], 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. [14] 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. [14] 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. [1] 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. [14] 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.

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 [12, 32] 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.) It 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 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 [12], 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 [15].

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 $\mathcal{F}_{2\text{PASS}}$ is given in Figures 1 and 2. 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 $\mathcal{F}_{2\text{PASS}}$ 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 *provides/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 provides input to and obtains output from the adversary and regular protocol machines, while protocol machines can provide input to and receive output from their local subroutines. The adversary can send and receive messages to and from all protocol machines, modeling that it controls all network traffic. Ideal functionalities are special protocol machines that are local to all parties except the adversary, so they interact with regular protocol machines through their input/output tapes and with the adversary through their 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 entry $mark[qid, \mathcal{P}]$ is assigned the value X.

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

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 $\mathcal{S}_1, \mathcal{S}_2$. Also 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 and record nothing.

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 $(\texttt{Stp}, sid, qid, \mathcal{S}, s)$ 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 $(\texttt{Stp}, sid, qid, s)$ 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 $(\texttt{Stp}, sid, qid, \mathcal{U}, s)$ 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 $(\texttt{Stp}, sid, qid, s)$ to $\mathcal{U}$; else, mark it $\texttt{fail}$ for $\mathcal{U}$ and output $(\texttt{Stp}, sid, qid, \texttt{fail})$ to $\mathcal{U}$.

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

Clarification.

Through the *Setup Request* interface, a user $\mathcal{U}$ can initiate the creation of an account u with 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 K and p 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, in the real world, the adversary can always "hijack" the user's 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 pass-

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

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: Upon input $(\texttt{RHjk}, sid, qid', \hat{p}')$ from the adversary $\mathcal{A}$ 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 $(\texttt{RNot}, sid, qid', \mathcal{S}_i)$ 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 $(\texttt{RNot}, sid, qid')$ to $\mathcal{S}_i$. Else, create a record $(\texttt{Perm}, qid', \mathcal{S}_i, \texttt{deny})$, output $(\texttt{Rtr}, sid, qid', \texttt{fail})$ 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 $(\texttt{ARtr}, qid', \cdot, \cdot)$ exists and that no record $(\texttt{Perm}, qid', \mathcal{S}_i, \cdot)$ exists. 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 $\mathcal{S}_i \in \{\mathcal{S}_1, \mathcal{S}_2\}$ and $\mathcal{U}'$ is honest, 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. If records $(\texttt{Perm}, qid', \mathcal{S}_i, \texttt{allow})$ exist for all honest $\mathcal{S}_i \in \{\mathcal{S}_1, \mathcal{S}_2\}$ and $\mathcal{U}'$ is corrupt, then send $(\texttt{Rtr}, sid, qid', c)$ to the adversary.

Retrieve Result Server: Upon receiving $(\texttt{Rtr}, sid, qid', \mathcal{S}_i, a)$ 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 $(\texttt{Rtr}, sid, qid', s)$ 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 $\mathcal{S}_j \in \{\mathcal{S}_1, \mathcal{S}_2\}$, and $p' = p$. Otherwise, $s \leftarrow \texttt{fail}$.

Retrieve Result User: Upon receiving $(\texttt{Rtr}, sid, qid', \mathcal{U}', a, K')$ 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 $(\texttt{Rtr}, sid, qid', K'', s'')$ to $\mathcal{U}'$ and mark query qid' as s'' for $\mathcal{U}'$ where (K'', s'') is

- $(\perp, \texttt{fail})$ if $a = \texttt{deny}$; else,
- $(K', \texttt{succ})$ if $\mathcal{S}_1$ and $\mathcal{S}_2$ are corrupt; else,
- $(K, \texttt{succ})$ if a record $(\texttt{Stp}, p, K)$ exists, $p = p'$, and qid' is marked $\texttt{succ}$ for $\mathcal{S}_1$ and $\mathcal{S}_2$ and is not marked for $\mathcal{A}$; else,
- $(\perp, \texttt{fail})$.

Figure 2: 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 3: Ideal certification functionality.

word and key with its own. The user will always output $\texttt{fail}$ 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 did so as well and 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 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 $(\texttt{RNot}, \ldots)$ message. At this point, the server can apply any external throttling mechanism to decide whether to participate in this retrieval, e.g., by not participating after too many failed attempts. The servers indicate whether they will proceed with the retrieval through the *Retrieve Permission* interface. Only after both 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 moment 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 accounts that are taken.

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. [2].

As mentioned above, security for multiple user accounts is obtained through the JUC theorem [15]. In the multi-session functionality $\hat{\mathcal{F}}_{2\text{PASS}}$, 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 actually 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. [14] 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.

Relation to existing notions. The only existing security notion for 2PASS is due to Bagherzandi et al. [1]. 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 the existing 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 [1]. 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 $(\texttt{CRS}, sid)$ from party $\mathcal{P}$, if no value r is recorded, it chooses and records $r \leftarrow_{\text{R}} D$. It then sends $(\texttt{CRS}, sid, r)$ 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 [13] depicted in Figure 3. 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 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 using randomness r with label $l \in \{0,1\}^*$ we write $c = \mathsf{enc2}_{pk}(m; r; l)$ (if we do not need to refer to the randomness, we simply write $c = \mathsf{enc2}_{pk}(m; *; l)$). When employing these schemes, we assume suitable (implicit) mappings from (tuples of) elements from $\mathbb{G}$ to $\{0,1\}^*$.

3.1 High-Level Idea

The main idea underlying our protocol is similar to the approach of Brainard et al. [6]: 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 (keyg, enc, 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 (essentially that the encryptions were computed correctly). In the description of the protocol we denote these protocols as $\mathrm{ZK}\{(w) : \mathit{predicate}(w, y) = 1\}$ for a proof that a predicate is true w.r.t. to a public value y and a witness value w. 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 $(\mathtt{Stp}, \mathit{sid}, \mathit{qid})$ or $(\mathtt{Rtr}, \mathit{sid}, \mathit{qid})$ and by a sequence number corresponding to 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 (and might be dropped if they cannot be delivered after some timeout). If a server receives a message with a fresh tag $(\mathtt{Stp}, \mathit{sid}, \mathit{qid})$ or $(\mathtt{Rtr}, \mathit{sid}, \mathit{qid})$, and sequence number 1 (message from the user), it starts a new instance of the respective protocol, or drops the message if such an instance 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 $(\mathtt{Stp}, \mathit{sid}, p, K)$, 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 $(\mathtt{Stp}, \mathit{sid}, \mathtt{fail})$. Furthermore, whenever any party receives a message $(\mathtt{Stp}, \mathit{sid}, \mathit{qid})$, it aborts with output $(\mathtt{Stp}, \mathit{sid}, \mathtt{fail})$. The structure of the Setup protocol is depicted in Figure 4; 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 $p_1 \leftarrow_{\mathrm{R}} \mathbb{G}$ and $K_1 \leftarrow_{\mathrm{R}} \mathbb{G}$ 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 $s_1, s_2, \tilde{s}_1, \tilde{s}_2 \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, 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 $F_2 \leftarrow \mathsf{enc2}_{PE_2}((p_2, K_2, s_2, \tilde{s_2},); *; (\mathit{sid}, \mathit{qid}, C_1, \tilde{C}_1, C_2, \tilde{C}_2))$.

(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 $\mathcal{S}_2$.

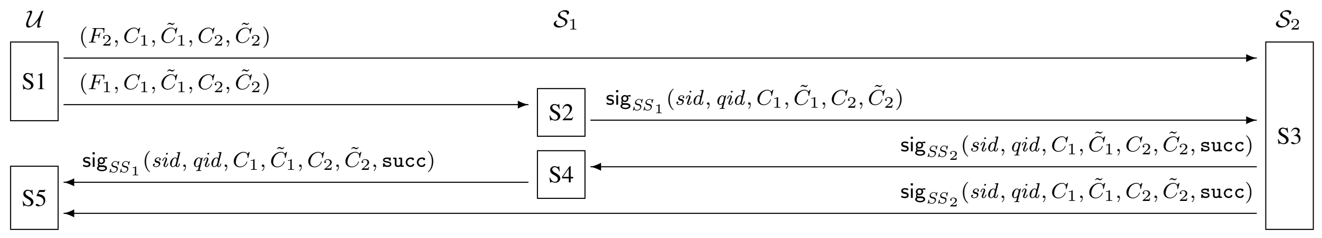
$\mathcal{U}$
$\mathcal{S}_1$
$\mathcal{S}_2$
S1
S2
S3
S4
S5
$(F_2, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$
$(F_1, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$
$\mathsf{sig}_{SS_1}(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2)$
$\mathsf{sig}_{SS_2}(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ})$
$\mathsf{sig}_{SS_1}(sid,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

Figure 4: Communication messages of the Setup **protocol with computation steps Si.**

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 $st_2[u]$ 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 user and first server of acceptance:* Compute signature $\tau_2 \leftarrow \mathsf{sig}_{SS_2}(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ})$ and send τ_2 to $\mathcal{U}$ and $\mathcal{S}_1$.

(h) *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 $(\mathtt{Stp}, sid, qid, \mathtt{succ})$.

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 $\mathsf{ver}_{SS_2}((sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ}), \tau_2) = 1$.

(c) *Inform user of acceptance:* Compute $\tau_1 \leftarrow \mathsf{sig}_{SS_1}(sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ})$ 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 $(\mathtt{Stp}, sid, qid, \mathtt{succ})$.

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 as τ_1 and τ_2, respectively.

(b) *Verify that servers accepted and finalize protocol:* Check that $\mathsf{ver}_{PS_1}((sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ}), \tau_1) = 1$ and that $\mathsf{ver}_{PS_2}((sid, qid, C_1, \tilde{C}_1, C_2, \tilde{C}_2, \mathtt{succ}), \tau_2) = 1$. If so, output $(\mathtt{Stp}, sid, qid, \mathtt{succ})$.

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 5; 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 $(\mathtt{Rtr}, sid, qid', \mathtt{fail})$ in case the party is a server, or with output $(\mathtt{Rtr}, sid, qid', \mathtt{fail})$ 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 $\mathcal{S}_2$.

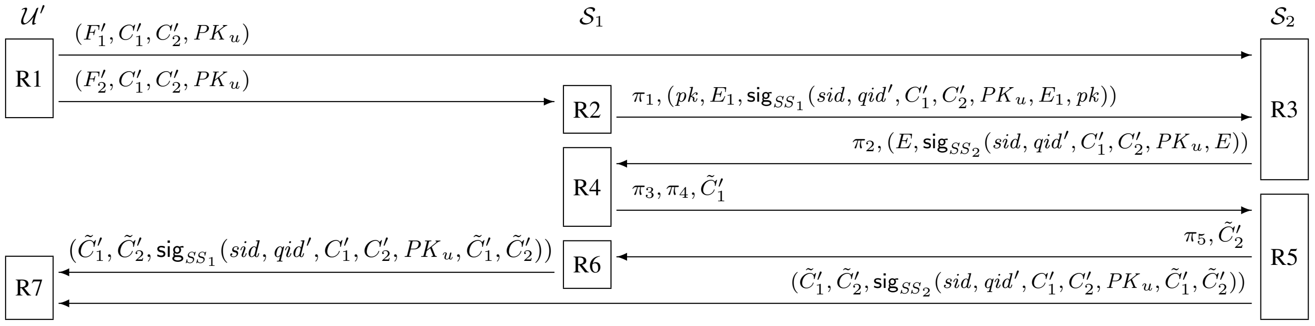
$\mathcal{U}'$
$\mathcal{S}_1$
$\mathcal{S}_2$
R1
R2
R3
R4
R5
R6
R7
(F_1', C_1', C_2', PK_u)
(F_2', C_1', C_2', PK_u)
$\pi_1, (pk, E_1, \mathsf{sig}_{SS_1}(sid, qid', C_1', C_2', PK_u, E_1, pk))$
$\pi_2, (E, \mathsf{sig}_{SS_2}(sid, qid', C_1', C_2', PK_u, E))$
$\pi_3, \pi_4, \tilde{C}_1'$
$\pi_5, \tilde{C}_2'$
$(\tilde{C}_1', \tilde{C}_2', \mathsf{sig}_{SS_1}(sid, qid', C_1', C_2', PK_u, \tilde{C}_1', \tilde{C}_2'))$
$(\tilde{C}_1', \tilde{C}_2', \mathsf{sig}_{SS_2}(sid, qid', C_1', C_2', PK_u, \tilde{C}_1', \tilde{C}_2'))$

Figure 5: 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.**

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 $(\texttt{Rtr}, sid, qid', 1, F_1', C_1', C_2', PK_u)$. 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 $(\texttt{RNot}, sid, qid')$ 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) *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 $\mathcal{S}_2$:

$$\pi_1 := \mathrm{ZK}\{(p_1, p_1', s_1, s_1', r_1) : E_1 = \mathsf{enc}_{pk}(p_1/p_1'; r_1) \wedge C_1 = \mathsf{enc}_{PK}(p_1; s_1) \wedge C_1' = \mathsf{enc}_{PK}(p_1'; s_1')\} .$$

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 $(\texttt{RNot}, sid, qid')$ 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 interact 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:

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

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:

$$\pi_3 := \mathrm{ZK}\{(sk) : 1 = \mathsf{dec}_{sk}(E)\}.$$

(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:

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

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:

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

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

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 $\tau_1' \leftarrow \mathsf{sig}_{SS_1}(sid, qid', C_1', C_2', PK_u, \tilde{C}_1', \tilde{C}_2')$ and send $(\tilde{C}_1', \tilde{C}_2', \tau_1')$ to $\mathcal{U}'$. Output $(\mathtt{Rtr}, sid, qid', \mathtt{succ})$.

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 $(\tilde{C}_1', \tilde{C}_2', \tau_1')$ and $(\tilde{C}_1', \tilde{C}_2', \tau_2')$, respectively.

(b) *Check that both servers agree and verify signatures:* Check that the same ciphertexts $(\tilde{C}_1', \tilde{C}_2')$ are present in the messages from the two servers. Verify that $\mathsf{ver}_{PS_1}((sid, qid', C_1', C_2', PK_u, \tilde{C}_1', \tilde{C}_2'), \tau_1')$ and that $\mathsf{ver}_{PS_2}((sid, qid', C_1', C_2', PK_u, \tilde{C}_1', \tilde{C}_2'), \tau_2')$.

(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 $(\mathtt{Rtr}, sid, qid', K, \mathtt{succ})$.

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 we propose that the Schnorr [35, 33] and Cramer-Shoup [16] schemes be used since the DDH assumption also suffices for their security. We also provide an efficiency analysis.

4.1 ElGamal Encryption

The ElGamal encryption scheme [18] 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 [11, 8]. 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 [8] for details on this.

The most efficient way to make these protocol concurrent zero-knowledge and simulation-sound is by the Fiat-Shamir transformation [19]. In this case, we will have to resort to the random-oracle model [4] for the security proof. To make the resulting non-interactive proofs simulation-sound, it suffices to let the prover include context information as an argument to the random oracle in the Fiat-Shamir transformation, such as the sid, the qid, the protocol step in which the statement is being proven, and a collision-resistant hash of the communication transcript that the prover and verifier have engaged in so far, so that the proof is resistant to a man-in-the-middle attack.

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., [31, 21, 9]). 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, $\mathcal{S}_1$ has generated (pk, sk) as $((y = g^x, g), x) \in ((\mathbb{G}, \mathbb{G}), \mathbb{Z}_q)$. Let $E_1 = (e_{11}, e_{12}) = (p_1/p_1' y^{r_1}, g^{r_1})$, $C_1 = (c_{11}, c_{12}) = (p_1 Y^{s_1}, g^{s_1})$, and $C_1' = (c_{11}', c_{12}') = (p_1' Y^{s_1'}, g^{s_1'})$ with r_1, s_1, s_1' elements of $\mathbb{Z}_q$ be the encryptions computed in the setup and retrieve protocol. Then the proof π_1 can be instantiated with the protocol specified as:

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

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., [8]). 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 $E = (e_1, e_2) = ((e_{11}y^{r_2}p_2/p_2')^z, (e_{12}g^{r_2})^z)$, $C_2 = (c_{21}, c_{22}) = (p_2Y^{s_2}, g^{s_2})$, 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:

$$PK\{(s_2, s_2', r_2, z, \alpha, \beta, \gamma) : \ e_2 = e_{12}^z g^\alpha \ \wedge \ c_{22} = g^{s_2} \ \wedge \ 1 = c_{22}^z g^{-\beta} \ \wedge \ c_{22}' = g^{s_2'} \ \wedge \ 1 = {c_{22}'}^z g^{-\gamma} \ \wedge \ e_1 = (\frac{e_{11}c_{21}}{c_{21}'})^z y^\alpha Y^{-\beta} Y^\gamma\} \ .$$

where by definition $\alpha = zr_2$ and by proof $\beta = zs_2$ and $\gamma = zs_2'$. 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 $\beta = zs_2$ and $\gamma = zs_2'$ 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: $PK\{(x) : \ y = g^x \wedge e_1 = e_2^x\}$. 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 one. Let the encryptions computed in the setup and retrieve protocol be $\tilde{C}_1 = (\tilde{c}_{11}, \tilde{c}_{12}) = (K_1Y^{\tilde{s}_1}, g^{\tilde{s}_1})$, and $\tilde{C}_1' = (\tilde{c}_{11}', \tilde{c}_{12}') = (K_1Y^{\tilde{s}_1'}, g^{\tilde{s}_1'})$. Then the proof π_4 can be realized with the protocol specified as

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

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 [19] to obtain simulation-sound non-interactive proofs in the random-oracle model. The cost of operations other than exponentiations 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 10 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* (keyg, enc, dec) *is semantically secure, the encryption scheme* (keyg2, enc2, dec2) *is CCA2 secure, the signature scheme* (keygsig, sig, ver) *is existentially unforgeable, and the associated proof system is a simulation-sound concurrent zero-knowledge proof, then our protocols* Setup *and* Retrieve *securely realize $\mathcal{F}_{2\text{PASS}}$ in the $\mathcal{F}_{CA}$ and $\mathcal{F}_{CRS}$-hybrid model.*

When instantiated with the ElGamal encryption scheme [18] for (keyg, enc, dec), Cramer-Shoup encryption [16] for (keyg2, enc2, dec2), Schnorr signatures [35, 33] for (keygsig, sig, ver), and the Σ protocols of Section 4 [35, 8], 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 $\mathcal{F}_{2\text{PASS}}$ in the random-oracle and $\mathcal{F}_{CA}$-hybrid model.*

A detailed proof of Theorem 1 is given in the full version of the paper [10]; we provide a proof outline here. First, let us conceptually view all honest participants as a single interactive Turing machine (ITM) called the *challenger*, which obtains all inputs from the environment $\mathcal{E}$ intended for honest parties and which outputs the responses back to $\mathcal{E}$. We 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 real protocol on behalf of the honest participants, with $\mathcal{A}$ as the adversary, so the environment receives the same view as it would in a real execution of the protocol. In the last game, $\mathcal{C}_{10}$ runs the ideal protocol via the ideal functionality on behalf of the honest participants, with the simulator $\mathcal{SIM}$ as the adversary, so the environment receives the same view as it would in an ideal execution. Let $view_i(sid, 1^k)$ denote the view that $\mathcal{E}$ receives when interacting with $\mathcal{C}_i$ for session identifier sid and security parameter k; we will often omit sid and 1^k. We briefly describe each challenger $\mathcal{C}_i$ and why each $view_i$ is indistinguishable from $view_{i-1}$; we refer to the full version [10] for details.

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 whenever it receives some values (PS_i, m, σ) where PS_i is an honest server's signature verification key and σ is a valid signature under PS_i of the message m, and yet m has never been signed by $\mathcal{S}_i$. We have that $view_2 \approx view_1$ by the unforgeability of the signature scheme.

Challenger $\mathcal{C}_3$: Identical to $\mathcal{C}_2$, except that when an honest user uses enc2 to send an encryption of a plaintext m to an honest server, the ciphertext is computed as an encryption of $1^{|m|}$. More concretely, this affects ciphertexts F_i and F_i' sent by an honest user to an honest server $\mathcal{S}_i$ in Steps S1 and R1.

Let's call a setup or retrieve query *intact* if it was initiated by an honest user and the first message $(\texttt{Stp}, sid, qid, 1, \ldots)$ or $(\texttt{Rtr}, sid, qid', 1, \ldots)$ arrives at an honest server unmodified. We call the query *hijacked* if it was initiated by an honest user but these messages were modified in transit, and we call the query *corrupt* if it was either hijacked or intiated by a dishonest user. If this query is intact, then the honest server pretends that the ciphertext F_i or F_i' correctly decrypted to m; otherwise, it decrypts the ciphertext from the modified message and proceeds normally. We can show that $view_2 \approx view_1$ by the CCA-2 security of enc2.

Challenger $\mathcal{C}_4$: Identical to $\mathcal{C}_3$, except that the public key PK in the CRS is generated such that $\mathcal{C}_4$ knows the corresponding secret key SK. Since PK is distributed exactly as in a real CRS, this hop is purely conceptual.

Further, $\mathcal{C}_4$ runs, on the side, a registry $\mathcal{R}$, that in several additional steps will become the ideal functionality $\mathcal{F}$. Whenever the first message of an intact setup query qid is delivered to an honest server, $\mathcal{C}_4$ adds a record $(\texttt{AStp}, qid, \mathcal{U}, p, K)$ to $\mathcal{R}$. Whenever the first message of a corrupt query is delivered to an honest server, $\mathcal{C}_4$ uses the fact that it knows SK to decrypt the ciphertexts $(C_1, \tilde{C}_1, C_2, \tilde{C}_2)$ to recover p and K. It then records $(\texttt{AStp}, qid, \mathcal{U}, p, K)$ in $\mathcal{R}$. The challenger continues running the setup protocol, marking qid as $\texttt{succ}$ or $\texttt{fail}$ for a party $\mathcal{P}$ whenever $\mathcal{P}$ outputs $\texttt{succ}$ of $\texttt{fail}$. If all honest servers output $\texttt{succ}$, it records $(\texttt{Stp}, qid, \mathcal{U}, p, K)$ in $\mathcal{R}$. The existence of the registry $\mathcal{R}$ is internal to $\mathcal{C}_4$ and has no effect on $view_4$.

Challenger $\mathcal{C}_5$: Identical to $\mathcal{C}_4$, except that, whenever an honest party performs a zero-knowledge proof, it uses the zero-knowledge simulator instead of the prover's algorithm, which results in indistinguishable view by indistinguishability of simulation.

Challenger $\mathcal{C}_6$: Identical to $\mathcal{C}_5$, except that when the protocol directs an honest party $\mathcal{H}$ to compute $c = \mathsf{enc}_{PK}(m; r)$ under the CRS public key PK, where (m, r) will never be sent to a dishonest party, $\mathcal{H}$ instead computes $c = \mathsf{enc}_{PK}(1^{|m|}; r)$. 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 party $\mathcal{H}'$ to prove something about the ciphertext c, $\mathcal{C}_6$ will, just as $\mathcal{C}_5$, have $\mathcal{H}'$ run the zero-knowledge simulator for the proof system, instead of the prover, pretending that c is an encryption of m (as m is known to the challenger). We have that $view_6 \approx view_5$ by semantic security of enc and simulation-soundness and indistinguishability of simulation of the ZK proof system.

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 and how it forms the ciphertext $\tilde{C}_2'$ in Step R5.

If the current query qid' is a corrupt query, then $\mathcal{C}_7$ decrypts C_1' and C_2' using SK to recover p'; else the environment explicitly provided p' as input to $\mathcal{C}_7$ on behalf of the honest user. It then looks up the record $(\texttt{Stp}, p, K)$ in its registry $\mathcal{R}$. If $p = p'$, then form E as an encryption of 1; else as an encryption of a random group element. The proof π_2 is executed, as done by all challengers starting with $\mathcal{C}_5$, via the simulator. Further, in Step R5, if query qid' is intact, compute the ciphertext $\tilde{C}_2'$ as an encryption of $1^{|K_2|}$; else (i.e., if this query is corrupt) compute it correctly as an encryption of the key share K_2 established in the setup phase. The proof π_5 is computed via the zero-knowledge simulator.

To prove that $view_7 \approx view_6$, we rely on the simulation-soundness of the proof system and the semantic security of enc.

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 that it computes the ciphertext E_1 in Step R2 as an encryption of 1. If query qid' is corrupt, 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 $p' = p$, the honest server completes the rest of the retrieve protocol as follows: in Step R4 it computes the ciphertext $\tilde{C}_1'$ correctly if the current query qid' is corrupt (so that the dishonest user controlled by the adversary, upon submitting the correct password, will learn the correct key share K_1), and as an encryption of $1^{|K_1|}$ otherwise. The proofs π_3 and π_4 are simulated. Similarly to the previous step, to show that $view_7 \approx view_6$, we rely on the simulation-soundness of the proof system and the semantic security of enc.

Note that at this point, if the user and at least one of the servers is honest, then the only way in which the protocol messages depend on the honest user's input, is on the fact whether the passwords match ($p = p'$) during the retrieve protocol. If they match and the query is corrupt, then the ciphertexts $\tilde{C}_1'$ and $\tilde{C}_2'$ additionally depend on the stored key K. We still want to make sure that for honest queries, the user retrieves the correct key or outputs $\texttt{fail}$ if the passwords match, and outputs $\texttt{fail}$ if they don't match. This we will do in the next step.

Challenger $\mathcal{C}_9$: Same as $\mathcal{C}_8$, except that it halts if one of the following bad events happens: (1) the event that an honest user carries out a retrieval with at least one honest server and the correct password, and at the end of the 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.

The fact that $view_9 \approx view_8$ is shown by an argument about the statements proved by the various zero-knowledge proofs. It relies on the simulation-soundness of the proof system, as well as the fact that, for all challengers starting with $\mathcal{C}_2$, an honest server's signature implies that this server has accepted all preceding proofs.

Challenger $\mathcal{C}_{10}$: In this game, the idea is to give the environment a view that is identical to $view_9$, but to have $\mathcal{C}_{10}$ 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 bookkeeping 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, $\mathcal{C}_{10}$ is now viewed not as a single ITM, but as several ITMs interacting with each other: one 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 parties in Section 2. What remains to do is to describe $\mathcal{SIM}$ and to verify that the resulting $view_{10}$ is identical to $view_9$. The description of $\mathcal{SIM}$ is given in detail in the full version of this paper [10]. In a nutshell, in order to view $\mathcal{C}_9$ as consisting of all these different ITMs, we observe that the protocol messages that the honest parties inside $\mathcal{C}_9$ send out only depend on information about the actual password and key that is provided to the simulator $\mathcal{SIM}$ by $\mathcal{F}$.

Although the way that $\mathcal{C}_{10}$ is structured internally is different from the way $\mathcal{C}_9$ is structured (because $\mathcal{C}_9$ doesn't separate its computation steps into those carried out by $\mathcal{F}$, those carried out by

$\mathcal{SIM}$, and those carried out by honest ideal parties), each message that $\mathcal{C}_{10}$ sends to $\mathcal{A}$ and $\mathcal{E}$ is computed exactly as in $\mathcal{C}_9$, so we have that $view_{10} \approx view_9$.

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.

6. REFERENCES

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

[2] B. Barak, Y. Lindell, and T. Rabin. Protocol initialization for the framework of universal composability. Cryptology ePrint Archive, Report 2004/006, 2004.

[3] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT 2000*.

[4] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93*.

[5] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Symposium on Security and Privacy 1992*.

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

[7] W. E. Burr, D. F. Dodson, E. M. Newton, R. A. Perlner, W. T. Polk, S. Gupta, and E. A. Nabbus. Electronic authentication guideline. NIST Special Publication 800-63-1, 2011.

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

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

[10] J. Camenisch, A. Lysyanskaya, and G. Neven. Practical yet universally composable two-server password-authenticated secret sharing. Cryptology ePrint Archive, 2012.

[11] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In *CRYPTO '97*.

[12] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS 2001*.

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

[14] R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT 2005*.

[15] R. Canetti and T. Rabin. Universal composition with joint state. In CRYPTO 2003.

[16] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[17] M. Di Raimondo and R. Gennaro. Provably secure threshold password-authenticated key exchange. In *EUROCRYPT 2003*.

[18] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO '84*.

[19] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO '86*.

[20] W. Ford and B. S. Kaliski Jr. Server-assisted generation of a strong secret from a password. In *IEEE WETICE 2000*.

[21] J. A. Garay, P. D. MacKenzie, and K. Yang. Strengthening zero-knowledge protocols using signatures. In *EUROCRYPT 2003*.

[22] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2), 1984.

[23] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[24] L. Gong, T. M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, 1993.

[25] S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. *ACM TISSEC*, 2(3):230–268, 1999.

[26] C. Herley, P. C. van Oorschot, and A. S. Patrick. Passwords: If we're so smart, why are we still using them? In *FC 2009*.

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

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

[29] J. Katz, R. Ostrovsky, and M. Yung. Efficient and secure authenticated key exchange using weak passwords. *Journal of the ACM*, 57(1), 2009.

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

[31] P. D. MacKenzie and K. Yang. On simulation-sound trapdoor commitments. In *EUROCRYPT 2004*.

[32] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *ACM CCS 2000*.

[33] D. Pointcheval and J. Stern. Security proofs for signature schemes. In *EUROCRYPT '96*.

[34] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO '91*.

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

[36] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

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