Two-Server Password-Authenticated Secret Sharing UC-Secure Against Transient Corruptions *

Jan Camenisch[1], Robert R. Enderlein[1,2], and Gregory Neven[1]

[1] IBM Research – Zurich, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland
[2] Department of Computer Science, ETH Zürich, CH-8092 Zürich, Switzerland

Abstract. Protecting user data entails providing authenticated users access to their data. The most prevalent and probably also the most feasible approach to the latter is by username and password. With password breaches through server compromise now reaching billions of affected passwords, distributing the password files and user data over multiple servers is not just a good idea, it is a dearly needed solution to a topical problem. Threshold password-authenticated secret sharing (TPASS) protocols enable users to share secret data among a set of servers so that they can later recover that data using a single password. No coalition of servers up to a certain threshold can learn anything about the data or perform an offline dictionary attack on the password. Several TPASS protocols have appeared in the literature and one is even available commercially. Although designed to tolerate server corruptions, unfortunately none of these protocols provide details, let alone security proofs, about how to proceed when a compromise actually occurs. Indeed, they consider static corruptions only, which for instance does not model real-world adaptive attacks by hackers. We provide the first TPASS protocol that is provably secure against adaptive server corruptions. Moreover, our protocol contains an efficient recovery procedure allowing one to re-initialize servers to recover from corruption. We prove our protocol secure in the universal-composability model where servers can be corrupted adaptively at any time; the users' passwords and secrets remain safe as long as both servers are not corrupted at the same time. Our protocol does not require random oracles but does assume that servers have certified public keys.

Keywords: Universal composability, threshold cryptography, passwords, transient corruptions.

1 Introduction

Properly protecting our digital assets still is a major challenge today. Because of their convenience, we protect access to our data almost exclusively by passwords, despite their inherent weaknesses. Indeed, not a month goes by without the announcement of another major password breach in the press. In 2013, hundreds of millions of passwords were stolen through server compromises, including massive breaches at Adobe, Evernote, LivingSocial, and Cupid Media. In August 2014, more than one billion passwords from more than 400,000 websites were reported stolen by a single crime ring. Barring some technical blunders on the part of Adobe, most of these passwords were properly salted and hashed. But even the theft of password hashes is detrimental to the security of a system. Indeed, the combination of weak human-memorizable passwords (NIST estimates sixteen-character passwords to contain only 30 bits of entropy [5]) and the blazing efficiency of brute-force dictionary attacks (currently testing up to 350 billion guesses per second on a rig of 25 GPUs [20]) mean that any password of which a hash was leaked should be considered cracked.

Stronger password hash functions [32] only give a linear security improvement, in the sense that the required effort from the attacker increases at most with the same factor as the honest server is willing to spend on password verification. Since computing password hashes is the attacker's core business, but only a marginal activity to a respectable web server, the former probably has the better hardware and software for the job.

* This is the full version of a paper that appeared at the 18th International Conference on Practice and Theory in Public-Key Cryptography (PKC 2015). The final publication is available at `link.springer.com`.

A much better approach to password-based authentication, first suggested by Ford and Kaliski [19], is to distribute the capability to test passwords over multiple servers. The idea is that no single server by itself stores enough information to allow it to test whether a password is correct and therefore to allow an attacker to mount an offline dictionary attack after having stolen the information. Rather, each server stores an information-theoretic share of the password and engages in a cryptographic protocol with the user and the other servers to test password correctness. As long as less than a certain threshold of servers are compromised, the password and the stored data remain secure.

Building on this approach, several threshold password-authenticated key exchange (TPAKE) protocols have since appeared in the literature [19, 24, 28, 4, 16, 34, 26, 25], where, if the password is correct, the user shares a different secret key with each of the servers after the protocol. Finally addressing the problem of protecting user data, threshold password-authenticated secret sharing (TPASS) protocols [1, 10, 9, 25] combine data protection and user authentication into a single protocol. They enable the password-authenticated user to reconstruct a strong secret, which can then be used for further cryptographic purposes, e.g., decrypting encrypted data stored in the cloud. An implementation of the protocol by Brainard et al. [4] is commercially available as EMC's *RSA Distributed Credential Protection* (DCP) [17].

Unfortunately, none of the protocols proposed to date provide a satisfying level of security. Indeed, for protocols that are meant to resist server compromise, the research papers are surprisingly silent about what needs to be done when a server actually gets corrupted and how to recover from such an event. The work by Di Raimondo and Gennaro [16] is the only one to mention the possibility to extend their protocol to provide proactive security by refreshing the shares between time periods; unfortunately, no details are provided. The RSA DCP product description [17] mentions a re-randomization feature that "can happen proactively on an automatic schedule or reactively, making information taken from one server useless in the event of a detected breach." This feature is not described in any of the underlying research papers [4, 34], however, and neither is a security proof known. Taking only protocols with provable security guarantees into account, the existing ones can protect against servers that are malicious from the beginning, but do not offer any guarantees against adaptive corruptions. The latter is a much more realistic setting, modelling for instance servers getting compromised by malicious hackers. This state of affairs is rather troubling, given that the main threats to password security today, and arguably, the whole *raison d'être* of TPAKE/TPASS schemes, come from the latter type of attacks.

One would hope to be able to strengthen existing protocols with ideas from proactive secret sharing [21] to obtain security against adaptive corruptions, but this task is not straightforward and so far neither the resulting protocol details nor the envisaged security properties have ever been spelled out. Indeed, designing cryptographic protocols secure against adaptive corruptions is much more difficult than against static corruptions. One difficulty thereby is that in the security proof the simulator must generate network traffic for honest parties *without* knowing their inputs, but, once the party is corrupted, must be able to produce realistic state information that is consistent with the now revealed actual inputs as well as the previously simulated network traffic. Generic multiparty computation protocols secure against adaptive corruption can be applied, but these are too inefficient. In fact, evaluating a single multiplication gate in the most efficient two-party computation protocol secure against adaptive corruptions [7] is more than three times slower than a full execution of the dedicated protocol we present here.

Our contributions. We provide the first threshold password-authenticated secret sharing protocol that is provably secure against *adaptive* corruptions, assuming data can be securely erased, which in this setting is a standard and also realistic assumption. Our protocol is a two-server protocol in the public-key setting, meaning that servers have trusted public keys, but users do not. We do not require random oracles. We

also describe a *recovery procedure* that servers can execute to recover from corruption and to renew their keys assuming a trusted backup is available. The security of the password and the stored secret is preserved as long as both servers are never corrupted simultaneously.

We prove our protocol secure in the universal composability (UC) framework [11, 12]. The very relevant advantages of composable security notions for the particular case of password-based protocols have been argued before [13, 10]; we briefly summarize them here. In composable notions, the passwords for honest users, as well as their password attempts, are provided by the environment. Passwords and password attempts can therefore be distributed arbitrarily and even dependently, reflecting real users who may choose the same or similar passwords for different accounts. It also correctly models typos made by honest users when entering their passwords: all property-based notions in the literature limit the adversary to seeing transcripts of honest users authenticating with their correct password, so in principle security breaks down as soon as a user mistypes the password. Finally, composable definitions absorb the inherent polynomial success probability of the adversary into the functionality. Thus, security is retained when the protocol is composed with other protocols, in particular, protocols that use the stored secret as a key. In contrast, composition of property-based notions with non-negligible success probabilities is problematic because the adversary's advantage may be inflated. Also, strictly speaking, the security provided by property-based notions is guaranteed only if a protocol is used in isolation.

Our construction uses the same basic approach as the TPASS protocols of Brainard et al. [4] and Camenisch et al. [10]. During the setup phase, the user generates shares of his key and password and sends them to the servers (together with some commitments that will later be used in the retrieve phase). During the retrieve phase, the servers run a subprotocol with the user to verify the latter's password attempt using the commitments and shares obtained during setup. If the verification succeeds, the servers send the shares of the key back to the user, who can then reconstruct the key. Furthermore, the correctness of all values exchanged is enforced by zero-knowledge proofs. Like the recent work of Camenisch et al. [9], we do not require the user to share the password during the retrieve phase but run a dedicated protocol to verify whether the provided password equals the priorly shared one. This offers additional protection for the user's password in case he mistakenly tries to recover his secret from servers different from the ones he initially shared his secret with. During setup, the user can be expected to carefully choose his servers, but retrieval happens more frequently and possibly from different devices, leaving more room for error.

The novelty of our protocol lies in how we transform the basic approach into an efficient protocol secure against an adaptive adversary. The crux here is that parties should never be committed to their inputs but at the same time must prove that they perform their computation correctly. We believe that the techniques we use in our protocol to achieve this are of independent interest when building other protocols that are UC-secure against adaptive corruptions. First, instead of using (binding) encryptions to transmit integers between parties, we use a variant of Beaver and Haber's non-committing encryption based on one-time pads (OTP) [3]: the sender first commits to a value with a mixed trapdoor commitment scheme [7] and then encrypts both the value and the opening with the OTP. This enables the recipient to later prove statements about the encrypted value. Second, our three-party password-checking protocol achieves efficiency by transforming commitments with shared opening information into an Elgamal-like encryption of the same value under a shared secret key. To be able to simulate the servers' state if they get corrupted during the protocol execution, each pair of parties needs to temporarily re-encrypt the ciphertext with a key shared between them.

Finally, we note that our protocol is well within reach of a practical implementation: users and servers have to perform a few hundred exponentiations each, which translates to an overall computation time of less than 0.1 seconds per party.

2 Our Ideal Functionality $\mathcal{F}_{\mathbf{2pass}}$

We now describe on a high level our ideal functionality $\mathcal{F}_{\text{2pass}}$ for two-server password-authenticated secret sharing, secure against transient corruptions. We provide the formal definition of $\mathcal{F}_{\text{2pass}}$ in the GNUC variant [22] of the UC framework [11] in the Appendix in §A. $\mathcal{F}_{\text{2pass}}$ is reminiscent of similar functionalities by Camenisch et al. [10, 9], the main differences being our modifications to handle transient corruptions. We compare the ideal functionalities in §E.1.

The functionality $\mathcal{F}_{\text{2pass}}$ involves two servers, $\mathcal{P}$ and $\mathcal{Q}$, and a plurality of users. We chose to define $\mathcal{F}_{\text{2pass}}$ for a single user account, specified by the session id sid. Multiple accounts can be realized by multiple instances of $\mathcal{F}_{\text{2pass}}$ or with a multi-session realization of $\mathcal{F}_{\text{2pass}}$. The session identifier sid consists of $(pid_{\mathcal{P}}, pid_{\mathcal{Q}}, (\mathbb{G}, q, g), uacc, ssid)$, i.e., the identity of the two servers, the description of a group of prime order q with generator g, the name of the user account $uacc$ (any string), and an arbitrary suffix $ssid$. Only the parties with identities $pid_{\mathcal{P}}$ and $pid_{\mathcal{Q}}$ can provide input in the role of $\mathcal{P}$ and $\mathcal{Q}$, respectively, to $\mathcal{F}_{\text{2pass}}$. When starting a fresh query, any party can provide input in the role of a user to $\mathcal{F}_{\text{2pass}}$; for subsequent inputs in that query, $\mathcal{F}_{\text{2pass}}$ ensures it comes from the same party; additionally, $\mathcal{F}_{\text{2pass}}$ does not disclose the identity of the user to the servers.

$\mathcal{F}_{\text{2pass}}[sid]$ reacts to a set of instructions, each requiring the parties to send multiple inputs to $\mathcal{F}_{\text{2pass}}$ in a specific order. The main instructions are Setup, Retrieve, and Refresh. Additionally $\mathcal{F}_{\text{2pass}}$ reacts to instructions modelling dishonest behavior, namely Corrupt, Recover, and Hijack. $\mathcal{F}_{\text{2pass}}$ may process multiple queries (instances of instructions) concurrently. A query identifier qid is used to distinguish between separate executions of the main instructions. We now provide a summary of the instructions. We refer to Figure 1 for a high-level definition of $\mathcal{F}_{\text{2pass}}$ and to §A for the full formalization.

With the *Setup* instruction, a user sets up the user account by submitting a key k and a password p to $\mathcal{F}_{\text{2pass}}$ for storage, protected under the password. This instruction can be run only once, which we enforce by fixing qid to "Setup". With the *Retrieve* instruction, any user can then retrieve that k provided her submitted password attempt a is correct, i.e., $a = p$, and the servers are willing to participate in this query. Giving the server the choice to refuse to participate in a query is important to counter online password guessing attacks. $\mathcal{F}_{\text{2pass}}$ allows for the adaptive corruption of users and servers with the *Corrupt* instruction, and for recovery from corruption of servers at any time with the *Recover* instruction. Servers should run the *Refresh* instruction whenever they recover from corruption or at regular intervals; in the real protocol, the two servers re-randomize their state in this instruction and thereby clear the residual knowledge $\mathcal{A}$ might have. If both servers are corrupted at the same time or sequentially with no Refresh in between, the adversary $\mathcal{A}$ will learn the current key and password (k, p) and is allowed to set them to different values. Finally, recall that in our realization of $\mathcal{F}_{\text{2pass}}$, the first message from the user to the servers is not authenticated. $\mathcal{A}$ can therefore learn the qid from that message, drop the message, and send his own message to the servers with that qid. We model this attack in $\mathcal{F}_{\text{2pass}}$ with the *Hijack* instruction. Servers will not notice this attack, but the user will conclude his query failed.

Our $\mathcal{F}_{\text{2pass}}$ functionality gives the following security guarantees: k and p are protected from $\mathcal{A}$ as long as at least one server is honest and no corrupt user is able to correctly guess the password. Furthermore, if at least one server is honest, no offline password guessing attacks are possible. Honest servers can limit online guessing attacks by limiting Retrieve queries after too many failed attempts. Finally, an honest user's password attempt a remains hidden even if a Retrieve query is directed at two corrupt servers.

$\mathcal{F}_{\text{2pass}}$ processes the instructions as follows. $\mathcal{F}_{\text{2pass}}$ accepts inputs and messages only for a specific sid. It further checks that the sid has the correct format. Whenever $\mathcal{F}_{\text{2pass}}$ receives an input from a party it will eventually send a message to $\mathcal{A}$ containing the identity of the party, the type of input, sid, qid, and—if applicable—sends out delayed messages. [a]

Setup: The user inputs $\langle$Setup, sid, qid = "Setup", $p, k\rangle$ to $\mathcal{F}_{\text{2pass}}$ and the two servers each input[b] $\langle$ReadySetup, sid, qid = "Setup"$\rangle$ to $\mathcal{F}_{\text{2pass}}$. $\mathcal{F}_{\text{2pass}}$ then sends a public delayed message $\langle$Done, sid, $qid\rangle$ to the user and each of the two servers.

Retrieve: To start, the user inputs $\langle$Retrieve, sid, qid, $a\rangle$ to $\mathcal{F}_{\text{2pass}}$, and the two servers each input $\langle$ReadyRetrieve, sid, $qid\rangle$ to $\mathcal{F}_{\text{2pass}}$. $\mathcal{F}_{\text{2pass}}$ waits for a message $\langle$Lock, sid, $qid\rangle$ from $\mathcal{A}$, and then replies whether the user's password attempt was correct by sending $\langle$Lock, sid, qid, $b\rangle$ to $\mathcal{A}$—where $b = 1$ if $a = p$ and $b = 0$ otherwise. $\mathcal{F}_{\text{2pass}}$ then sends a public delayed message $\langle$Delivered, sid, qid, $b\rangle$ to the two servers, and a private delayed message $\langle$Deliver, sid, qid, $k'\rangle$ to the user, where $k' = k$ if $a = p$, and $k' = \varepsilon$ otherwise.

Corrupt: When a party becomes corrupt, the party's ideal peer will input $\langle$Corrupt, $sid\rangle$ to $\mathcal{F}_{\text{2pass}}$. Recall that $\mathcal{A}$ thereafter obtains control of the corrupted party's input to and output from $\mathcal{F}_{\text{2pass}}$. $\mathcal{A}$ may prevent a subsequent Refresh query from succeeding in case the server later recovers from corruption—in a real protocol, $\mathcal{A}$ may tamper with the server's internal state. If both servers are corrupted at the same time (or corrupted in sequence with no Refresh query in between), $\mathcal{F}_{\text{2pass}}$ will send (k, p) to $\mathcal{A}$ and allow $\mathcal{A}$ to provide arbitrary replacement values. That is, $\mathcal{A}$ can force $\mathcal{F}_{\text{2pass}}$ to return arbitrary values to the user if the latter interacts with two corrupted servers in a Retrieve query.

Recover: When a party recovers from corruption, the party's ideal peer will input $\langle$Recover, $sid\rangle$ to $\mathcal{F}_{\text{2pass}}$. $\mathcal{F}_{\text{2pass}}$ then stops accepting input and messages for all currently running Setup and Retrieve queries, and will not accept any further Setup and Retrieve queries until a Refresh query suceeds.

Refresh: To start a Refresh query, each server inputs $\langle$Refresh, sid, $qid\rangle$ to $\mathcal{F}_{\text{2pass}}$. While this query is in progress, no further Setup, Retrieve, and Refresh queries are accepted, and currently running queries are dropped. Once it has received a message from both servers, $\mathcal{F}_{\text{2pass}}$ sends $\langle$RefreshDone, sid, $qid\rangle$ as public delayed messages to the two servers. $\mathcal{F}_{\text{2pass}}$ then resumes accepting new queries. Note that while a server was corrupted, $\mathcal{A}$ might have prevented it from completing this Refresh query.

Hijack: Just after a user provided its first input to $\mathcal{F}_{\text{2pass}}$ in a Setup or Retrieve query and before $\mathcal{A}$ sends anything to $\mathcal{F}_{\text{2pass}}$ for the same query, $\mathcal{A}$ has the option of stealing the id of the query by sending a $\langle$HijackSetup, sid, qid, p, $k\rangle$ or $\langle$HijackRetrieve, sid, qid, $a\rangle$ message, respectively, to $\mathcal{F}_{\text{2pass}}$. In that case, $\mathcal{F}_{\text{2pass}}$ ignores the user's first message and runs the query with $\mathcal{A}$ instead of the user, with the qid chosen by the user but input—(p, k) or a—provided by $\mathcal{A}$.

[a] Messages from an ideal functionality to a party are direct outputs, unless they are specified to be delayed outputs. In the latter case, $\mathcal{F}_{\text{2pass}}$ notifies $\mathcal{A}$ it wishes to send the message and waits for a confirmation by $\mathcal{A}$ before actually sending out the message. A public delayed output means that $\mathcal{A}$ learns the message; a private message means that $\mathcal{A}$ will learn only the type of the message and the recipient.

[b] The GNUC coventions forbid that $\mathcal{F}_{\text{2pass}}$ sends a message to the servers at this point, as the servers might not yet exist.

Fig. 1: High-level definition of $\mathcal{F}_{\text{2pass}}$. See the text for explanations, and see §A for the full formalization.

3 Preliminaries

In this section, we introduce the notation used throughout this paper, give the ideal functionalities and cryptographic building blocks we use as subroutines in our construction, and provide a refresher on corruption models in the UC framework.

3.1 Notation

Let $\eta \geq 80$ be the security parameter. Let ε denote the empty string. If $\mathbb{S}$ is a set, then $s \xleftarrow{\$} \mathbb{S}$ means we set s to a random element of that set. If A is a probabilistic polynomial-time (PPT) algorithm, then $y \xleftarrow{\$} \mathrm{A}(x)$ means we assign y to the output of $\mathrm{A}(x)$ when run with fresh random coins on input x. If s is a bitstring, then by $|s|$ we denote the length of s. If $\mathcal{U}$ and $\mathcal{P}$ are parties, and Sub is a two-party protocol, then by $(out_{\mathcal{U}}; out_{\mathcal{P}}) \xleftarrow{\$} \langle \mathcal{U}.\mathrm{Sub}(in_{\mathcal{U}}), \mathcal{P}.\mathrm{Sub}(in_{\mathcal{P}})\rangle(in_{\mathcal{UP}})$ we denote the simultaneous execution of the protocol by the two parties, on common input $in_{\mathcal{UP}}$, with $\mathcal{U}$'s additional private input $in_{\mathcal{U}}$, with $\mathcal{P}$'s additional private input $in_{\mathcal{P}}$, and where $\mathcal{U}$'s output is $out_{\mathcal{U}}$ and $\mathcal{P}$'s output is $out_{\mathcal{P}}$. We use an analogue notation for three-party protocols.

We use the following arrow-notation: $\overset{publicData}{\dashrightarrow}$ to denote the transmission of public data over a channel that the two parties have already established between themselves (we discuss how such a channel is established in more detail later). When we write $(\text{📦} : dataToErase)$ next to such an arrow, we mean that the value $dataToErase$ is securely erased before the public data is transmitted. When we write $\left[secretData\right]_{\text{🔒}}$ on such an arrow, we mean that $secretData$ is sent in a non-committing encrypted form. All these transmissions must be secure against adaptive corruptions in the erasure model.

3.2 Ideal Functionalities that we Use as Subroutines

We now describe the ideal functionalities we use as subroutines in our construction. These are authenticated channels ($\mathcal{F}_{\mathrm{ac}}$), one-side-authenticated channels ($\mathcal{F}_{\mathrm{osac}}$), zero-knowledge proofs of existence ($\mathcal{F}_{\mathrm{gzk}}$), and common reference strings ($\mathcal{F}_{\mathrm{crs}}^{D}$).

Authenticated channels. Let $\mathcal{F}_{\mathrm{ac}}[sid]$ be a single-use authenticated channel [22]. In our construction, we allow only servers to communicate among themselves using $\mathcal{F}_{\mathrm{ac}}[sid]$. We recall the formal definition in §B.2.

One-side-authenticated channels. Let $\mathcal{F}_{\mathrm{osac}}[sid]$ be a multi-use channel where only one party, the server, authenticates himself towards the other party, the client. The server has the guarantee that in a given session all messages come from the same client. Note that the first message from the client to the server is not authenticated and can be modified (*hijacked*) by the adversary—the original client will be excluded from the rest of the interaction. We provide a formal definition in §B.3. We also refer to the work of Barak et al. [2] for a formal treatment of communication without or with partial authentication. A realization of $\mathcal{F}_{\mathrm{osac}}[sid]$ is out of scope, but not hard to construct.

Zero-knowledge proofs of knowledge and existence. Let $\mathcal{F}_{\mathrm{gzk}}[sid]$ be the zero-knowledge functionality supporting proofs of existence [8], also called "gullible" zero-knowledge proofs. These proofs of existence are cheaper than the corresponding proofs of knowledge, but they impose limitations on the simulator $\mathcal{S}$ in the security proof. In a realization of $\mathcal{F}_{\mathrm{gzk}}$, the prover reveals the statement to be proven only in the *last* message. This is crucial for our construction, as this allows the prover to *erase* (📦) witnesses and other data before disclosing the statement to be proven. We recall the formal definition [8] in §B.4.

Notation. When specifying the predicate to be proven, we use a combination of the Camenisch-Stadler notation [15] and the notation introduced by Camenisch, Krenn, and Shoup [8]; for example:

$$\mathcal{F}_{\text{gzk}}[sid]\{(\rangle\!\alpha, \beta\ ;\ \exists\gamma) : y = g^{\gamma} \wedge z = g^{\alpha}k^{\beta}h^{\gamma}\}$$

is used for proving the existence of the discrete logarithm to the base g, and of a representation of z to the bases g, k, and h such that the h-part of this representation is equal to the discrete logarithm of y to the base g. Furthermore, knowledge of the g-part and the k-part of the representation is proven. Variables quantified by $\rangle\!$ (knowledge) can be extracted by the simulator $\mathcal{S}$ in the security proof, while variables quantified by $\exists$ (existence) cannot.

By writing a proof on an arrow: $\overset{\pi_0}{\dashrightarrow}$ we denote the performance of such an interactive zero-knowledge proof protocol secure against adaptive corruptions with erasures. If additional public or secret data is written on the arrow, or data to be erased besides the arrow, then this data is transmitted with, or erased before, respectively, the last message of the proof protocol (cf. §3.1). The predicate of the proof may depend on that data.

Proofs with two verifiers. Let $\mathcal{F}_{\text{gzk}}^{\text{2v}}[sid]$ be the three-party ideal functionality to denote the parallel execution of two independent zero-knowledge proofs with the same prover and same specification, but two different verifiers. The prover waits for a reply from both verifiers before sending out the last message of each proof. This gives the prover the opportunity to erase the same witnesses in both proofs. We provide a formal definition in §B.5. The proof that the special composition theorem by Camenisch, Krenn, and Shoup [8] holds also for $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ is very similar to the proof that it holds for $\mathcal{F}_{\text{gzk}}$ and is omitted.

Common reference string. Let $\mathcal{F}_{\text{crs}}^{D}[sid]$ be a common reference string (CRS) functionality, which provides a CRS distributed according to some distribution D. We make use of two distributions in this paper: $\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$ provides a uniform CRS over $\mathbb{G}^3$ and $\mathcal{F}_{\text{crs}}^{\text{gzk}}$ provides a CRS as required by Camenisch et al.'s protocol π, the intended realization of $\mathcal{F}_{\text{gzk}}$ [8]. We provide a formal definition in §B.1.

3.3 Cryptographic Building Blocks of Our Construction

Our construction makes use of two cryptographic building blocks: a CCA2-secure encryption scheme, and a homomorphic mixed trapdoor commitment scheme.

CCA2-secure encryption. We denote the key generation function $(pk, sk, kgr) \xleftarrow{\$} \text{Gen}(1^{\eta})$, where kgr is the randomness that was used to generate the key pair. We denote the encryption function $(e, er) \xleftarrow{\$} \text{Enc}(pk, pt, l)$ that takes as input a public key pk, a plaintext $pt \in \{0,1\}^*$, and a label $l \in \{0,1\}^*$; and outputs the ciphertext e and the randomness er used to encrypt. The corresponding decryption function $pt \xleftarrow{\$} \text{Dec}(sk, e, l)$ takes as input the secret key sk, the ciphertext e, and the label l. We require the scheme to be secure against adaptive chosen ciphertext attacks [33]. An example of such an encryption scheme is Cramer-Shoup encryption in a hybrid setting over a group $\mathbb{G}$ of prime order q [15, §5.2]. To accommodate the label l in the encryption function, it must be added as an additional input to the hash function used during encryption.

Homomorphic mixed trapdoor (HMT) commitment. An HMT commitment scheme [7] is a computationally binding equivocable homomorphic commitment scheme, constructed from Pedersen commitments [31]. It works well with proofs of *existence* using $\mathcal{F}_{\text{gzk}}$, resulting in an efficiency gain in our protocol compared to a construction using plain Pedersen commitments, which would have to use proofs of *knowledge*. We provide a high-level overview of HMT commtiments here and recall the definition of HMT commitments in §C.

HMT commitments operate in a group $\mathbb{G}$ of prime order q (with generator g) where the decision Diffie-Hellman (DDH) problem is hard. They implicitly use a CRS (h, y, w) provided by $\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$. By $(c, o) \xleftarrow{\$} \text{Com}(s)$ we denote the function that takes as input a value $s \in \mathbb{Z}_q$ to be committed, and outputs a commitment c and an opening $o \in \mathbb{Z}_q$ to the commitment. We will also use the notation $c \leftarrow \text{Com}(s, o)$, where the opening is chosen outside the function. The commitments are homomorphic with respect to addition over $\mathbb{Z}_q$: i.e., $c * c' = \text{Com}(s + s', o + o')$. With a trapdoor to the CRS it is possible to efficiently equivocate commitments. Finally, we note that it is possible to extract a Pedersen commitment pc from a commitment c, we denote this operation by $pc := y^s h^o \leftarrow \text{PedC}(c)$.

3.4 Corruption in the UC Model

The UC model defines several types of party corruptions, the most important being *static*, *adaptive*, and *transient* corruptions. In protocols secure against static party corruptions, parties are either honest or corrupt from the start of the protocol and do not change their corruption status. In protocols secure against adaptive corruptions, parties can become corrupted at any time; once corrupted, they remain so for the rest of the protocol. Finally, transient corruptions [11] are similar to the adaptive corruptions, but parties can *recover* from corruption and regain their security.

In the following we discuss the modelling of transient corruptions in the UC framework, how one can use ideal functionalities designed for adaptive corruptions in a protocol designed for transient corruptions, and finally we discuss a particular problem that appears in protocols secure against adaptive or transient corruptions: the selective decommitment problem.

Modelling transient corruptions in real/hybrid protocols. We now recall how corruption and recovery is modelled in real/hybrid protocols.

Corruption of a party. When a party becomes corrupted, all of its internal state excluding the parts that were explicitly erased (🗑) is handed over to the adversary $\mathcal{A}$. $\mathcal{A}$ then controls that party. The ideal functionalities that were used as subroutines are notified of the corruption, and may provide additional information or capabilities to $\mathcal{A}$. Note that $\mathcal{A}$ can always choose to let a corrupted party follow the honest protocol, but passively monitor the party's internal state.

Recovery from corruption. $\mathcal{A}$ may cede control from a party. When doing that, $\mathcal{A}$ may specify a new internal state for the party. We then say that the party formally recovered. In real life, a party might know it recovered if it detected a breach and has restored from backup.

In most protocols however, formal recovery is not enough: the adversary still knows parts of the internal state of the formally recovered party. To allow the party to effectively recover its security, it must take additional steps, e.g., notify its subroutines (and stop using the subroutines that cannot handle recovery) and run a protocol-specific *Refresh* instruction. The party might thereby drop all currently running queries.

A party initiates a Refresh query to modify its internal state so that firstly it is synchronized with the other protocol participants, and so that secondly $\mathcal{A}$'s knowledge of the old state does not interfere with security of the new state. Parties should initiate a Refresh query when they formally recover from corruption. (If parties cannot detect formal recovery, they should run Refresh periodically.) The Refresh query might fail if the state of the party is inconsistent with that of the others. The party might also not necessarily recover its security even after succesful completion of the query, e.g., because all other participants are corrupted. Note that the security of a party is fully restored (if at all) only after Refresh completes: in the grey zone bewteen formal recovery and completion of Refresh, the party must not run any queries other than Refresh.

Using ideal functionalities designed for the adaptive type in a transient-secure hybrid protocol. Protocols secure against transient corruptions may use ideal functionalities as subroutines that were designed to handle adaptive corruptions, e.g., $\mathcal{F}_{\text{ac}}$, $\mathcal{F}_{\text{osac}}$, $\mathcal{F}_{\text{gzk}}$, and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$: upon formal recovery, the party must stop using all instances of these ideal functionalities. Thereby, it has to abort all currently running queries. Thereafter, it has to use fresh instances of these ideal functionalities for running the Refresh query, and all subsequent queries.

The selective decommitment problem. Hofheinz demonstrated that it is impossible to prove any protocol secure against adaptive corruptions (and thus, against transient corruptions) that uses perfectly binding commitments or (binding) encryptions to commit to or to encrypt the parties' input, respectively [23]. Let us expand on this. For example, assume that in a protocol a user $\mathcal{U}$ with an input i must send out a binding commitment c or an encryption e depending on i, e.g., $(c, o) = \text{Com}(i)$ or $(e, er) = \text{Enc}(pk, i, l)$. The simulator $\mathcal{S}$ in the security proof must be able to simulate the honest $\mathcal{U}$ without knowing her input i, i.e., $\mathcal{S}$ must send c or e to the adversary $\mathcal{A}$, containing some value that is most likely different from i. If $\mathcal{U}$ then gets corrupted, $\mathcal{S}$ must produce an internal state for $\mathcal{U}$, namely the opening o or the randomness er used to encrypt and—if applicable—the secret key sk, that is consistent with both her real input i and the values c or e already sent out to the adversary. However, due to the binding nature of the commitment and encryption, and unless it could predict i, $\mathcal{S}$ cannot find an internal state for $\mathcal{U}$ consistent with these values and therefore the security proof will not go through.

We explain how we avoid the selective decommitment problem in our protocol in Section 4.2.

4 Our Construction of TPASS Secure Against Transient Corruptions

In this section we present our realization Π_{2pass} of the $\mathcal{F}_{\text{2pass}}$ ideal functionality in the ($\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$, $\mathcal{F}_{\text{osac}}$, $\mathcal{F}_{\text{ac}}$, $\mathcal{F}_{\text{gzk}}$, $\mathcal{F}_{\text{gzk}}^{2\text{v}}$)-hybrid setting. Our Π_{2pass} protocol further uses a CCA2-secure cryptosystem and an HMT commitment scheme. As for $\mathcal{F}_{\text{2pass}}$, we describe Π_{2pass} for a single user account only, i.e., each instance of Π_{2pass} uses a fixed sid.

We start this section by discussing the high-level ideas of our construction. We then elaborate on the novel core ideas in our construction, before providing the detailed construction, and we comment on a multi-session version of Π_{2pass} that uses a constant size CRS. We finish by providing an estimate of the computational and communication complexity of Π_{2pass} in both the standard and random oracle models, and compare it with the complexity of related work.

4.1 High Level Approach of our TPASS Protocol

Our protocol Π_{2pass} implements the Setup, Retrieve, and Refresh instructions of $\mathcal{F}_{\text{2pass}}$. An adversary can hijack a Setup or Retrieve query through the $\mathcal{F}_{\text{osac}}$ subroutine. The other instructions of $\mathcal{F}_{\text{2pass}}$ are purely conceptual for the security proof. At a high level, the realizations of the Setup and Retrieve instructions of Π_{2pass} are reminiscent of the schemes by Camenisch et al. [10, 9] and Brainard et al. [4]: during Setup, the user generates shares of his key and password and sends them to the servers (together with some commitments that will later be used in Retrieve). During Retrieve, the servers run a subprotocol with the user to verify the latter's password attempt using the commitments and shares obtained in Setup. If the verification succeeds, the servers send the shares of the key back to the user, who can then reconstruct the key. Furthermore, the correctness of all values exchanged is enforced by zero-knowledge proofs. To deal with transient corruptions, our Π_{2pass} needs to implement the Refresh instruction, which allows the servers to re-randomize their shares of the key and password and thereby to re-secure their states when one of them is recovering from corruption. Naturally, prior schemes do not have a Refresh instruction as they do not provide security against transient corruptions.

The novelties of our construction arise from how we turn this basic approach into a scheme that is secure against adaptive and transient corruptions and at the same time efficient enough to be considered for practical deployment.

4.2 Key Ideas of our TPASS Protocol

We now present the key ideas that make it possible for our TPASS protocol to be secure against transient corruptions. These ideas are novel and of independent interest.

Three-party computation for determining equality to zero. The core subprotocol ChkPwd is depicted in Figure 2. To check if the password attempt a input by the user during a Retrieve query matches the stored password $p = p_{\mathcal{P}} + p_{\mathcal{Q}}$, the user and the two servers engage in a three-party computation to check if $\delta := p_{\mathcal{P}} + p_{\mathcal{Q}} - a \stackrel{?}{=} 0$, where $p_{\mathcal{P}}$ and $p_{\mathcal{Q}}$ are the shares stored by the respective servers. For efficiency reasons, it does not make sense to base that protocol on a generic multiparty computation protocol. Indeed, running one Retrieve query in our protocol is more than 3.7 times faster than evaluating a single multiplication gate in the best generic two-party computation protocol that is secure against adaptive corruptions [7] (see §E.3).

The first observation is that a commitment in the HMT scheme we use essentially consists of a *pair* of Pedersen commitments. Thus, while all components need to be considered to prove that a commitment is formed correctly, it is often sufficient to consider just one component later when doing computations with them. Now, based on this, a first idea for the desired subprotocol would be as follows. The servers' commitments $cp_{\mathcal{P}}$ and $cp_{\mathcal{Q}}$ to the shares of the password are distributed to all the parties, who then generate a commitment on the sum of the two shares using the homomorphic property of HMT commitments, and extract the first component thereof to obtain a value

$$C := \mathrm{PedC}(cp_{\mathcal{P}} * cp_{\mathcal{Q}}) = y^{p_{\mathcal{P}}+p_{\mathcal{Q}}} h^{op_{\mathcal{P}}+op_{\mathcal{Q}}} ,$$

where y and h are part of the CRS. That value is an equivocable Pedersen commitment to $p := p_{\mathcal{P}} + p_{\mathcal{Q}}$ with equivocation trapdoor $\log_y h$. Given C, the user subtracts his password attempt a from that commitment:

$$B := Cy^{-a} = y^{\delta} h^{op_{\mathcal{P}}+op_{\mathcal{Q}}}.$$

We now consider the Elgamal "ciphertext" $\big(A := h^{-1}, B\big)$, which is an encryption of y^{δ} under the shared secret key $(-op_{\mathcal{P}} - op_{\mathcal{Q}})$ with fixed randomness -1. This ciphertext is then passed from $\mathcal{U}$ to $\mathcal{P}$, from $\mathcal{P}$ to $\mathcal{Q}$, and then from $\mathcal{Q}$ back to $\mathcal{P}$, where at each step, the sender exponentiates that ciphertext by a non-zero random number $r_{\mathcal{U}}$, $r_{\mathcal{P}}$, and $r_{\mathcal{Q}}$, respectively, thereby multiplying the plaintext by that random number. Also, if possible, the sender will partially decrypt the ciphertext by removing $op_{\mathcal{P}}$ or $op_{\mathcal{Q}}$: $\mathcal{U}$ computes

$$\big(A_{\mathcal{U}}, D_{\mathcal{U}}\big) := \big(A^{r_{\mathcal{U}}}, B^{r_{\mathcal{U}}}\big) = \big(h^{-r_{\mathcal{U}}}, y^{\delta * r_{\mathcal{U}}} h^{(op_{\mathcal{P}}+op_{\mathcal{Q}})r_{\mathcal{U}}}\big)$$

and sends it to $\mathcal{P}$, $\mathcal{P}$ computes

$$\big(A_{\mathcal{P}}, D_{\mathcal{P}}\big) := \big(A_{\mathcal{U}}^{r_{\mathcal{P}}}, D_{\mathcal{U}}^{r_{\mathcal{P}}} A_{\mathcal{P}}^{op_{\mathcal{P}}}\big) = \big(h^{-r_{\mathcal{U}} r_{\mathcal{P}}}, y^{\delta * r_{\mathcal{U}} r_{\mathcal{P}}} h^{op_{\mathcal{Q}} r_{\mathcal{U}} r_{\mathcal{P}}}\big)$$

and sends it to $\mathcal{Q}$, and $\mathcal{Q}$ computes

$$\big(A_{\mathcal{Q}}, B_{\mathcal{Q}}\big) := \big(A_{\mathcal{P}}^{r_{\mathcal{Q}}}, D_{\mathcal{P}}^{r_{\mathcal{Q}}} A_{\mathcal{Q}}^{op_{\mathcal{Q}}}\big) = \big(h^{-r_{\mathcal{U}} r_{\mathcal{P}} r_{\mathcal{Q}}}, y^{\delta * r_{\mathcal{U}} r_{\mathcal{P}} r_{\mathcal{Q}}}\big)$$

and sends it to $\mathcal{P}$. If in the end the result $B_{\mathcal{Q}}$ is the neutral element, then $\delta = 0$, and the password was correct.

Unfortunately, this first idea doesn't quite work: if $\delta = 0$, $D_{\mathcal{U}}$ fixes a value for $(op_{\mathcal{P}} + op_{\mathcal{Q}})$ and $D_{\mathcal{P}}$ fixes a value for $op_{\mathcal{Q}}$. Thus $cp_{\mathcal{P}}$ and $cp_{\mathcal{Q}}$, together with $D_{\mathcal{U}}$ and $D_{\mathcal{P}}$ form unequivocable statistically binding commitments to $p_{\mathcal{P}}$ and $p_{\mathcal{Q}}$. This causes a selective decommitment problem. Our solution is to blind the values $D_{\mathcal{U}}$ and $D_{\mathcal{P}}$ with non-committing random shifts $s_{\mathcal{UP}}$, $s_{\mathcal{UQ}}$, and $s_{\mathcal{PQ}}$ as follows, thereby circumventing the problem. $\mathcal{U}$ chooses $s_{\mathcal{UP}}$ and $s_{\mathcal{UQ}}$, and sends them to $\mathcal{P}$ and $\mathcal{Q}$, respectively, in a non-

Fig. 2: Subroutine ChkPwd: the servers check if $\mathcal{U}$'s password attempt a is equal to the password $p_{\mathcal{P}}+p_{\mathcal{Q}}$. See the next figure for the instantiation of the zero-knowledge proofs.

$\pi_3 := \mathcal{F}_{\text{gzk}}[sid, qid, 3]\{(\rightarrow p_{\mathcal{P}}, op_{\mathcal{P}}) : cp_{\mathcal{P}} = \text{Com}(p_{\mathcal{P}}, op_{\mathcal{P}})\}.$

$\pi_4 := \mathcal{F}_{\text{gzk}}[sid, qid, 4]\{(\rightarrow p_{\mathcal{Q}}, op_{\mathcal{Q}}) : cp_{\mathcal{Q}} = \text{Com}(p_{\mathcal{Q}}, op_{\mathcal{Q}})\}.$

$\pi_5 := \mathcal{F}^{2\text{v}}_{\text{gzk}}[sid, qid, cp_{\mathcal{P}}, cp_{\mathcal{Q}}, 5]\{(\rightarrow a, \sigma\ ;\ \exists \rho, \beta) :$

$$h = A_{\mathcal{U}}^{\rho} \wedge C = (B_{\mathcal{U}}^{-1})^{\rho} y^{a} h^{\sigma} \wedge (cs_{\mathcal{UP}} * cs_{\mathcal{UQ}}) = \text{Com}(\sigma, \beta)$$

$\}$, where $\sigma := s_{\mathcal{UP}} + s_{\mathcal{UQ}}$, $\rho := -1/r_{\mathcal{U}}$, and $\beta := os_{\mathcal{UP}} + os_{\mathcal{UQ}}$.

$\mathcal{U}$ runs two proofs, one with $\mathcal{P}$ and one with $\mathcal{Q}$, in parallel: she performs the erasures and sends out the last message of both proofs only *after* she received the second message of the proof from both servers (see *Proofs with two verifiers* in §3.2).

$\pi_6 := \mathcal{F}_{\text{gzk}}[sid, qid, cp_{\mathcal{P}}, cp_{\mathcal{Q}}, cs_{\mathcal{UP}}, cs_{\mathcal{UQ}}, A_{\mathcal{U}}, B_{\mathcal{U}}, 6]\{(\exists p_{\mathcal{P}}, op_{\mathcal{P}}, r_{\mathcal{P}}, \sigma, \beta) :$

$$A_{\mathcal{P}} = A_{\mathcal{U}}^{r_{\mathcal{P}}} \wedge A_{\mathcal{P}} \neq g^0 \wedge B_{\mathcal{P}} = B_{\mathcal{U}}^{r_{\mathcal{P}}} A_{\mathcal{P}}^{op_{\mathcal{P}}+\sigma} \wedge$$
$$cp_{\mathcal{P}} = \text{Com}(p_{\mathcal{P}}, op_{\mathcal{P}}) \wedge \left(cs_{\mathcal{PQ}} * cs_{\mathcal{UP}}^{-1}\right) = \text{Com}(\sigma, \beta)$$

$\}$, where $\sigma := s_{\mathcal{PQ}} - s_{\mathcal{UP}}$ and $\beta := os_{\mathcal{PQ}} - os_{\mathcal{UP}}$.

$\pi_7 := \mathcal{F}_{\text{gzk}}[sid, qid, cs_{\mathcal{PQ}}, A_{\mathcal{P}}, B_{\mathcal{P}}, 7]\{(\exists p_{\mathcal{Q}}, op_{\mathcal{Q}}, r_{\mathcal{Q}}, \sigma, \beta) :$

$$A_{\mathcal{Q}} = A_{\mathcal{P}}^{r_{\mathcal{Q}}} \wedge A_{\mathcal{Q}} \neq g^0 \wedge B_{\mathcal{Q}} = B_{\mathcal{P}}^{r_{\mathcal{Q}}} A_{\mathcal{Q}}^{op_{\mathcal{Q}}-\sigma} \wedge$$
$$cp_{\mathcal{Q}} = \text{Com}(p_{\mathcal{Q}}, op_{\mathcal{Q}}) \wedge (cs_{\mathcal{UQ}} * cs_{\mathcal{PQ}}) = \text{Com}(\sigma, \beta)$$

$\}$, where $\sigma := s_{\mathcal{UQ}} + s_{\mathcal{PQ}}$ and $\beta := os_{\mathcal{UQ}} + os_{\mathcal{PQ}}$.

Fig. 3: Instantiation of zero-knowledge proofs for ChkPwd.

committing manner. $\mathcal{U}$ then generates $B_\mathcal{U}$ by multiplying $D_\mathcal{U}$ with the blinding factor $A_\mathcal{U}^{s_{\mathcal{UP}}+s_{\mathcal{UQ}}}$, i.e.,

$$\big(A_\mathcal{U}, B_\mathcal{U}\big) := \big(A^{r_\mathcal{U}}, B^{r_\mathcal{U}} A_\mathcal{U}^{s_{\mathcal{UP}}+s_{\mathcal{UQ}}}\big) = \big(h^{-r_\mathcal{U}}, y^{\delta * r_\mathcal{U}} h^{(op_\mathcal{P}+op_\mathcal{Q}-s_{\mathcal{UP}}-s_{\mathcal{UQ}})r_\mathcal{U}}\big)$$

and sends $B_\mathcal{U}$ instead of $D_\mathcal{U}$ to $\mathcal{P}$. The ciphertext $(A_\mathcal{U}, B_\mathcal{U})$ is now encrypted under the shared key $(s_{\mathcal{UP}} + s_{\mathcal{UQ}} - op_\mathcal{P} - op_\mathcal{Q})$. Similarily, $\mathcal{P}$ chooses $s_{\mathcal{PQ}}$ and sends it to $\mathcal{Q}$. $\mathcal{P}$ generates $B_\mathcal{P}$ like $D_\mathcal{P}$ but uses $B_\mathcal{U}$ instead of $D_\mathcal{U}$ in the formula and multiplies the result by $A_\mathcal{P}^{-s_{\mathcal{UP}}+s_{\mathcal{PQ}}}$, i.e.,

$$\big(A_\mathcal{P}, B_\mathcal{P}\big) := \big(A_\mathcal{U}^{r_\mathcal{P}}, B_\mathcal{U}^{r_\mathcal{P}} A_\mathcal{P}^{op_\mathcal{P}-s_{\mathcal{UP}}+s_{\mathcal{PQ}}}\big) = \big(h^{-r_\mathcal{U} r_\mathcal{P}}, y^{\delta * r_\mathcal{U} r_\mathcal{P}} h^{(op_\mathcal{Q}-s_{\mathcal{UQ}}-s_{\mathcal{PQ}})r_\mathcal{U} r_\mathcal{P}}\big)$$

and sends $B_\mathcal{P}$ to $\mathcal{Q}$ instead of $D_\mathcal{P}$, i.e., the ciphertext $(A_\mathcal{P}, B_\mathcal{P})$ is now encrypted under the shared key $(s_{\mathcal{UQ}} + s_{\mathcal{PQ}} - op_\mathcal{Q})$. Finally $\mathcal{Q}$ computes $B_\mathcal{Q}$ differently by replacing $D_\mathcal{P}$ by $B_\mathcal{P}$ in the formula and multiplying the result by $A_\mathcal{Q}^{-s_{\mathcal{UQ}}-s_{\mathcal{PQ}}}$, i.e.,

$$\big(A_\mathcal{Q}, B_\mathcal{Q}\big) := \big(A_\mathcal{P}^{r_\mathcal{Q}}, B_\mathcal{P}^{r_\mathcal{Q}} A_\mathcal{Q}^{op_\mathcal{Q}-s_{\mathcal{UQ}}-s_{\mathcal{PQ}}}\big) = \big(h^{-r_\mathcal{U} r_\mathcal{P} r_\mathcal{Q}}, y^{\delta * r_\mathcal{U} r_\mathcal{P} r_\mathcal{Q}}\big).$$

At the end of each step, the parties prove to each other in zero-knowledge that they computed their values correctly; whereby the parties use the trick explained in the next paragraph to refer to $s_{\mathcal{UP}}$, $s_{\mathcal{UQ}}$, and $s_{\mathcal{PQ}}$ in the proofs. These proofs also allow the simulator to extract a, $p_\mathcal{P}$, $p_\mathcal{Q}$, $op_\mathcal{P}$, $op_\mathcal{Q}$, and $(s_{\mathcal{UP}} + s_{\mathcal{PQ}})$ in the security proof.

Transmission of secrets for later use in proofs. In the protocol just described, $\mathcal{U}$ must send the value $s_{\mathcal{UP}}$ to $\mathcal{P}$ in a non-committing manner and all parties must be able to prove knowledge of that same value in subsequent zero-knowledge proofs. Simply having $\mathcal{U}$ encrypt $s_{\mathcal{UP}}$ is not sufficient, because $\mathcal{P}$ can later not prove knowledge of the encrypted $s_{\mathcal{UP}}$ in proofs. A similar situation also arises in other parts of our protocol, for example in the Setup instruction when $\mathcal{U}$ must send a share $p_\mathcal{P}$ to the password to $\mathcal{P}$ in a non-committing manner.

In a setting that considers only static corruptions, such problems are often solved by requiring $\mathcal{U}$ to send a Pedersen commitment $cs_{\mathcal{UP}}$ to $s_{\mathcal{UP}}$ to all parties, and to send $s_{\mathcal{UP}}$ and the opening $os_{\mathcal{UP}}$ to the commitment to $\mathcal{P}$, encrypted under $\mathcal{P}$'s public key. Thus, with $cs_{\mathcal{UP}}$, $\mathcal{P}$ can later prove that it correctly used $s_{\mathcal{UP}}$ in its computations.

When dealing with adaptive or transient corruptions, this does not work: the encryption of $s_{\mathcal{UP}}$ causes a selective decommitment problem. Instead, we have $\mathcal{U}$ generate an equivocable commitment $cs_{\mathcal{UP}}$ to $s_{\mathcal{UP}}$ with opening $os_{\mathcal{UP}}$, then establish a one-time pad (OTP) with $\mathcal{P}$, and then encrypt both $s_{\mathcal{UP}}$ and $os_{\mathcal{UP}}$ with the OTP. $\mathcal{U}$ then sends the resulting ciphertext to $\mathcal{P}$ in any convenient manner (in this specific example, $\mathcal{U}$ sends it as part of proof protocol π_5 in Figure 2 that actually uses the values $s_{\mathcal{UP}}$, $os_{\mathcal{UP}}$, and $cs_{\mathcal{UP}}$ in some indirect form; in the Setup instruction where she needs to send $p_\mathcal{P}$ to $\mathcal{P}$ in a non-committing manner, $\mathcal{U}$ sends the ciphertext to $\mathcal{P}$ directly). Afterwards, $\mathcal{P}$ can refer to $s_{\mathcal{UP}}$ in zero-knowledge proofs by means of $cs_{\mathcal{UP}}$, e.g., $\mathcal{F}_{\text{gzk}}[sid]\{(\exists s_{\mathcal{UP}}, os_{\mathcal{UP}}) : cs_{\mathcal{UP}} = \text{Com}(s_{\mathcal{UP}}, os_{\mathcal{UP}})\}$. This approach will allow $\mathcal{S}$ to equivocate $s_{\mathcal{UP}}$, provided that no extra dependencies on the opening $os_{\mathcal{UP}}$ are introduced in other protocol steps (the first idea of the three-party protocol above describes the problems when such an extra dependency is introduced on $op_\mathcal{P}$).

4.3 Detailed Construction of $\Pi_{\mathbf{2pass}}$ in the Standard Model (with Erasures)

We now give the full details of the instructions of our protocol and their respective subprotocols. Let us start with five remarks. First, we implicitly assume that all parties query $\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$ to obtain a CRS (h, y, w) whenever they need it. Second, all commitments Com must be realized with HMT commitments (see §3.3). Using Pedersen commitments instead would require expensive zero-knowledge proofs of *knowledge* in the protocol, thereby massively increasing the computational complexity. Third, we assume that for each query the user establishes a single instance of a one-side-authenticated channel $\mathcal{F}_{\text{osac}}[(sid, qid), \mathcal{P}]$ and $\mathcal{F}_{\text{osac}}[(sid, qid), \mathcal{Q}]$ with each respective server; all communication denoted by arrows: ⇢, and all communication inside the zero-knowledge functionalities $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ happen through

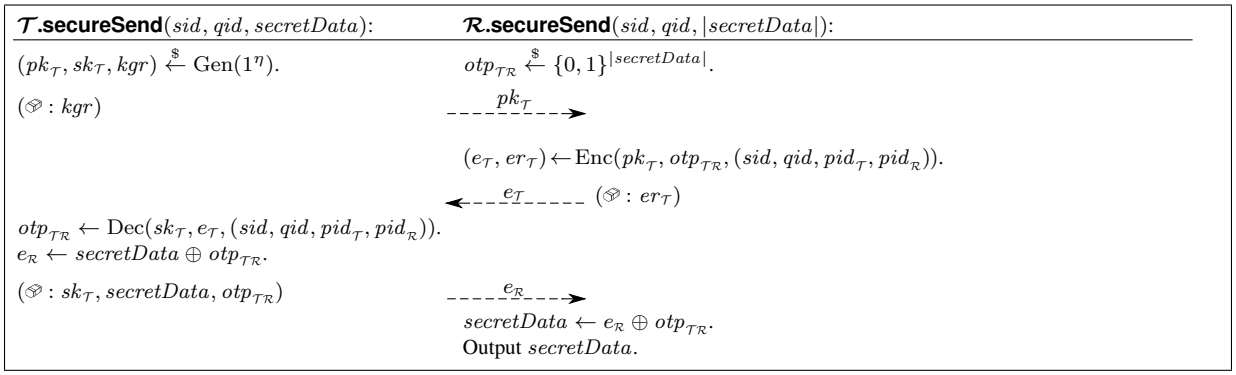


Fig. 4: Subroutine secureSend, the realization of $\overset{[secretData]_{\text{🔒}}}{\dashrightarrow}$: a party $\mathcal{T}$ (user or server) sends $secretData$ to $\mathcal{R}$ (user or server) in a non-committing encrypted form.

that instance.[3] The two servers communicate with each other through regular authenticated channels $\mathcal{F}_{\mathrm{ac}}[(sid, qid), \mathcal{P}, \mathcal{Q}, ssid]$. Fourth, parties can send data in a non-committing and confidential manner, i.e., secure against adaptive corruptions, by using the secureSend subroutine depicted in Figure 4. We denote such communication by: $[secretData]_{\text{🔒}}$ (cf. §3.1). The parties establish a one-time pad (OTP) with each other, encrypt the data with that OTP, and erase the OTP before sending the ciphertext [3]. Fifth, we implicitly assume that a party aborts a query without output if any check fails.

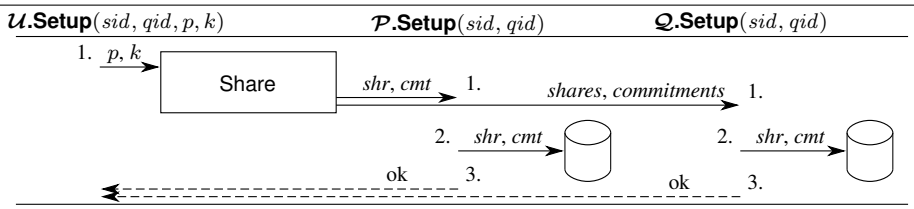
The Setup instruction. Recall that the goal of the Setup instruction is for a user to set up an account $uacc$ with the two servers $\mathcal{P}$ and $\mathcal{Q}$ and store a key $k \in \mathbb{Z}_q$ protected under a password $p \in \mathbb{Z}_q$ therein. The servers will silenty abort a Setup query if the user account has already been established.

When a user $\mathcal{U}$ receives an input $\langle$Setup, $sid = (pid_{\mathcal{P}}, pid_{\mathcal{Q}}, (\mathbb{G}, q, g), uacc, ssid)$, $qid =$ "Setup", $p, k\rangle$ from the environment $\mathcal{Z}$, she starts a Setup query. Each of the servers starts a Setup query when he receives an input $\langle$ReadySetup, sid, $qid\rangle$ from $\mathcal{Z}$. As the first step of the Setup query, $\mathcal{U}$ distributes shares of k and p to both servers using the Share subprotocol. In that subprotocol, the user establishes an OTP with each server and encrypts the shares with the respective OTPs in order to circumvent the *selective decommitment problem* [23]. Finally, the servers store their shares as their internal state and send an acknowledgement back to the user. See Figure 5. At the end of the Setup query, each of the three parties outputs $\langle$Done, sid, $qid\rangle$ to $\mathcal{Z}$.

The Share subprotocol Setup uses is depicted in Figure 6. In that subprotocol $\mathcal{U}$ splits her inputs p and k into random additive shares $p_{\mathcal{P}} + p_{\mathcal{Q}} := p$ and $k_{\mathcal{P}} + k_{\mathcal{Q}} := k$, and sends $(p_{\mathcal{P}}, k_{\mathcal{P}})$ to $\mathcal{P}$ and sends $(p_{\mathcal{Q}}, k_{\mathcal{Q}})$ to $\mathcal{Q}$. She commits to all shares and sends all commitments to both servers; additionally she sends the openings for a server's shares to the respective server; thus enabling the servers to later perform zero-knowledge proofs about their shares and the commitments to them. The servers then ensure they got the same commitments and prove to each other that they know their shares. In π_2, $\mathcal{Q}$ also proves to $\mathcal{P}$ that he knows the opening $op_{\mathcal{Q}}$ corresponding to his share of the password: this is needed so that $\mathcal{S}$ can properly simulate $B_{\mathcal{P}} = (A_{\mathcal{P}})^{s_{\mathcal{UQ}}+s_{\mathcal{PQ}}-op_{\mathcal{Q}}}$ in ChkPwd (we note that $\mathcal{S}$ does not need to know the value $op_{\mathcal{P}}$ from π_1 at this point).

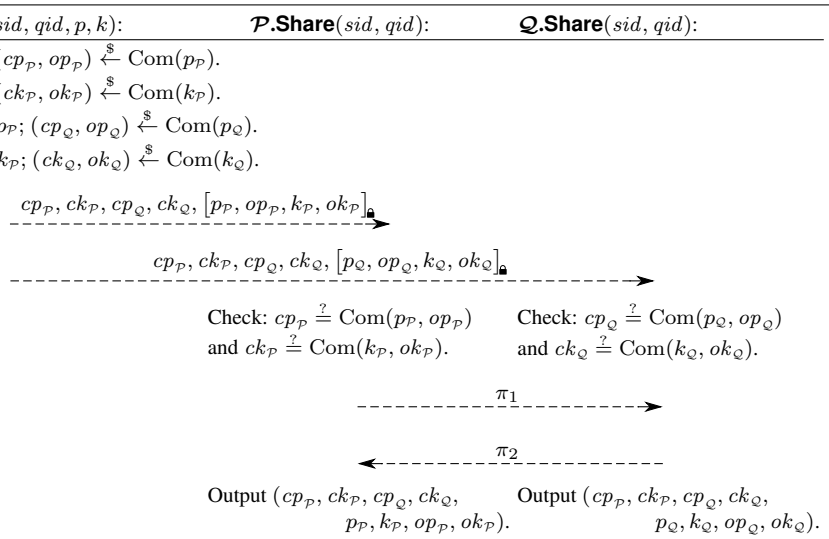
The Retrieve instruction. Recall that the goal of the Retrieve instruction is for a user (not necessarily the same as during Setup) to retrieve the key k, contingent upon her holding a correct password attempt $a \in \mathbb{Z}_q$.

[3] Refer to Barak et al. [2] for details about modelling communication with partial authentication in the UC model.



1. $\mathcal{U}$ generates shares and commitments to her password and key and sends them to the servers:
   $(\varepsilon; (cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{P}}, k_{\mathcal{P}}, op_{\mathcal{P}}, ok_{\mathcal{P}}); (cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{Q}}, k_{\mathcal{Q}}, op_{\mathcal{Q}}, ok_{\mathcal{Q}})) \xleftarrow{\$}$
   $\langle \mathcal{U}.\mathsf{Share}(p, k), \mathcal{P}.\mathsf{Share}(), \mathcal{Q}.\mathsf{Share}() \rangle (sid, qid)$.
2. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ stores $(cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{R}}, k_{\mathcal{R}}, op_{\mathcal{R}}, ok_{\mathcal{R}})$ into his long-term storage.
3. The servers send an acknowledgement to $\mathcal{U}$.

Fig. 5: Setup instruction: $\mathcal{U}$ distributedly stores a key k protected under a password p on two servers $\mathcal{P}$ and $\mathcal{Q}$.

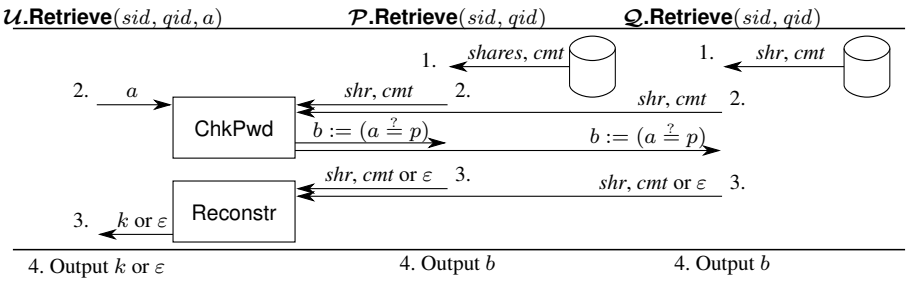


Instantiation of zero-knowledge proofs:

$\pi_1 := \mathcal{F}_{\mathrm{gzk}}[sid, qid, cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, 1]\{(\nearrow p_{\mathcal{P}}, k_{\mathcal{P}}\,;\, \exists op_{\mathcal{P}}, ok_{\mathcal{P}}) :$
$cp_{\mathcal{P}} = \mathrm{Com}(p_{\mathcal{P}}, op_{\mathcal{P}}) \wedge ck_{\mathcal{P}} = \mathrm{Com}(k_{\mathcal{P}}, ok_{\mathcal{P}})\}$.

$\pi_2 := \mathcal{F}_{\mathrm{gzk}}[sid, qid, cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, 2]\{(\nearrow p_{\mathcal{Q}}, k_{\mathcal{Q}}, op_{\mathcal{Q}}\,;\, \exists ok_{\mathcal{Q}}) :$
$cp_{\mathcal{Q}} = \mathrm{Com}(p_{\mathcal{Q}}, op_{\mathcal{Q}}) \wedge ck_{\mathcal{Q}} = \mathrm{Com}(k_{\mathcal{Q}}, ok_{\mathcal{Q}})\}$.

Fig. 6: Subroutine Share: $\mathcal{U}$ generates shares to her password p and key k, and sends them to the servers.



1. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ retrieves $\left(cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{R}}, k_{\mathcal{R}}, op_{\mathcal{R}}, ok_{\mathcal{R}}\right)$ from his long-term storage.
2. $\mathcal{U}$, $\mathcal{P}$, and $\mathcal{Q}$ run a three-party protocol to determine if the password attempt is correct:
   $(\varepsilon,b,b) \stackrel{\$}{\leftarrow} \langle \mathcal{U}.\mathsf{ChkPwd}(a), \mathcal{P}.\mathsf{ChkPwd}\left(cp_{\mathcal{P}}, cp_{\mathcal{Q}}, p_{\mathcal{P}}, op_{\mathcal{P}}\right), \mathcal{Q}.\mathsf{ChkPwd}\left(cp_{\mathcal{P}}, cp_{\mathcal{Q}}, p_{\mathcal{Q}}, op_{\mathcal{Q}}\right)\rangle(sid, qid)$.
3. – If $b = 1$ (i.e., $a = p$: the password attempt was correct), the servers send the key $k' = k$ to $\mathcal{U}$:
   $(k', \varepsilon, \varepsilon) \stackrel{\$}{\leftarrow} \langle \mathcal{U}.\mathsf{Reconstr}(), \mathcal{P}.\mathsf{Reconstr}\left(ck_{\mathcal{P}}, ck_{\mathcal{Q}}, k_{\mathcal{P}}, ok_{\mathcal{P}}\right), \mathcal{Q}.\mathsf{Reconstr}\left(ck_{\mathcal{P}}, ck_{\mathcal{Q}}, k_{\mathcal{Q}}, ok_{\mathcal{Q}}\right)\rangle(sid, qid)$.
   – Else if $b = 0$, the servers send a empty value $k' = \varepsilon$ instead:
   $\left(k'; \varepsilon; \varepsilon\right) \stackrel{\$}{\leftarrow} \langle \mathcal{U}.\mathsf{Reconstr}(), \mathcal{P}.\mathsf{Reconstr}(\varepsilon, \varepsilon, \varepsilon, \varepsilon), \mathcal{Q}.\mathsf{Reconstr}(\varepsilon, \varepsilon, \varepsilon, \varepsilon)\rangle(sid, qid)$.

Fig. 7: Retrieve instruction: $\mathcal{U}$ retrieves the key k if she provides the correct password.

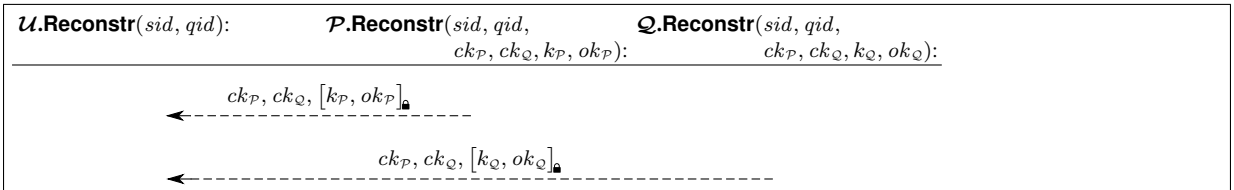


Fig. 8: Subroutine Reconstr: the servers send their commitments and shares of the key to $\mathcal{U}$ so that she may reconstruct her key k.

When a user $\mathcal{U}$ receives an input $\langle$Retrieve, sid, qid, $a\rangle$ with the same sid as during Setup from $\mathcal{Z}$, she starts a Retrieve query. Each of the servers starts a Retrieve query when he receives an input $\langle$ReadyRetrieve, sid, $qid\rangle$ from $\mathcal{Z}$. The servers may refuse to service the query if they for instance suspect that an online password guessing attack is in progress, e.g., if they have processed too many failed Retrieve queries for that user account already. As many policies for throttling down can be envisaged, we decided not to include the policy in our model but rather to let $\mathcal{Z}$ decide: if the server should refuse service, $\mathcal{Z}$ does not provide the initial input $\langle$ReadyRetrieve, sid, $qid\rangle$. The Retrieve instruction runs as follows and is depicted in Figure 7. The servers start a Retrieve query by retrieving their internal state. The user and the servers then engage in a three-party computation to determine whether $\delta := p_{\mathcal{P}} + p_{\mathcal{Q}} - a \stackrel{?}{=} 0$, i.e., whether the password attempt is correct, using the ChkPwd subprotocol. If the password is correct, the servers send their shares of the key back to the user using the Reconstr subprotocol; if wrong, they send back ε. At the end of the Retrieve query, $\mathcal{U}$ outputs $\langle$Deliver, sid, qid, $k'\rangle$ to $\mathcal{Z}$, and each server outputs $\langle$Delivered, sid, qid, $b\rangle$ to $\mathcal{Z}$—where $k' = k$ and $b = 1$ if the password attempt was correct, else $k' = \varepsilon$ and $b = 0$.

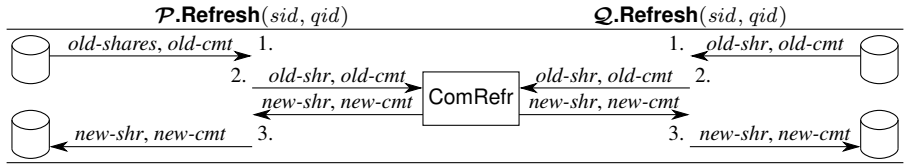
We now describe the two subprotocols that the Retrieve instruction uses. ChkPwd was already explained in §4.2 and was depicted in Figure 2. Reconstr is depicted in Figure 8. In this subprotocol, each server sends his share of the key ($k_{\mathcal{P}}$ or $k_{\mathcal{Q}}$) and the corresponding opening to $\mathcal{U}$. Both servers also send her the two commitments to the shares of the key. The user checks that she received the same commitments from both servers, that the shares and openings are correct, and reconstructs the key $k := k_{\mathcal{P}} + k_{\mathcal{Q}}$. The servers may send ε instead to denote a failed password attempt; in that case $\mathcal{U}$ outputs ε.

In both the ChkPwd and the Reconstr subprotocols, $\mathcal{U}$ needs to send data in a non-committing and confidential manner to $\mathcal{P}$. Instead of generating the OTPs for each subprotocol separately, the two parties could generate a single OTP of double the length in one operation and use the first half of the OTP during ChkPwd and the second half during Reconstr. This optimization would save one key generation (for the CCA2-secure cryptosystem), one encryption, and one decryption. The same optimization can be applied between $\mathcal{U}$ and $\mathcal{Q}$.

The Refresh instruction. In the Refresh instruction, the servers re-randomize their shares and generate new commitments to them. This ensures that $\mathcal{A}$ no longer has any knowledge about the internal state of a party who recovered from corruption. Servers execute a Refresh query immediately after they formally recover from corruption (see §3.4). Upon starting a Refresh query, the servers abort all running Setup and Retrieve queries and stop accepting new ones. Upon completion of the Refresh query, they resume acceptance of new Setup and Retrieve queries.

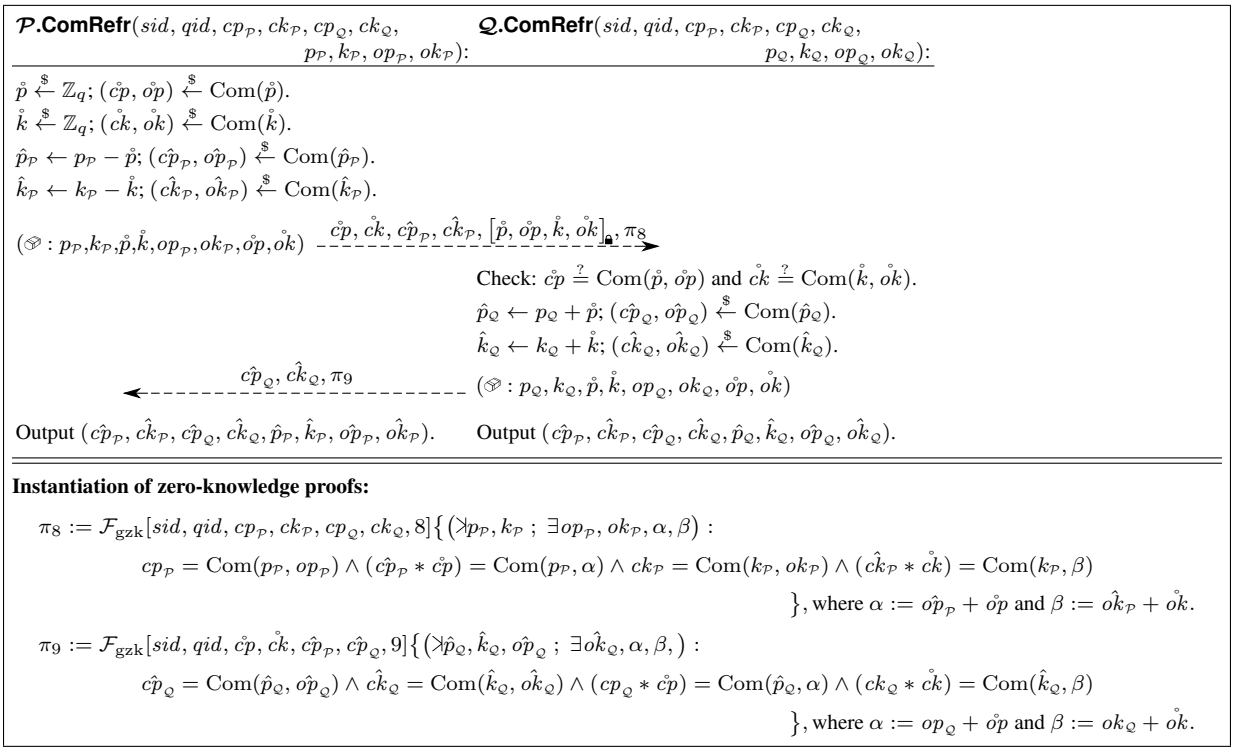
When a server receives an input $\langle$Refresh, sid, $qid\rangle$ with the same sid as during Setup from $\mathcal{Z}$, he starts the Refresh instruction. The Refresh protocol runs as follows and is depicted in Figure 9. The servers start by recovering their internal state. The servers then re-randomize their shares of the password and key using the ComRefr subprotocol. Finally both servers store their new internal state. At the end of the protocol, each server outputs $\langle$RefreshDone, sid, $qid\rangle$ to $\mathcal{Z}$.

The Refresh instruction uses the ComRefr subprotocol, depicted in Figure 10, the goal of which is for both servers $\mathcal{P}$ and $\mathcal{Q}$ to re-randomize their respective shares $(p_{\mathcal{P}}, k_{\mathcal{P}})$ and $(p_{\mathcal{Q}}, k_{\mathcal{Q}})$. $\mathcal{P}$ randomly selects two offsets $\mathring{p}$ and $\mathring{k}$ and subtracts them from his shares. $\mathcal{P}$ then commits to the offsets and his new shares. $\mathcal{P}$ proves to $\mathcal{Q}$ that all operations were done correctly. As part of the proof, $\mathcal{P}$ sends all the commitments and a ciphertext that contains the offsets and the corresponding openings encrypted under an OTP to $\mathcal{Q}$. $\mathcal{Q}$ likewise updates his shares and generates new commitments to them. $\mathcal{Q}$ proves to $\mathcal{P}$ that all operations were done honestly and that he knows the opening $o\hat{p}_{\mathcal{Q}}$ corresponding to his new share of



1. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ retrieves $\big(cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{R}}, k_{\mathcal{R}}, op_{\mathcal{R}}, ok_{\mathcal{R}}\big)$ from his long-term storage.
2. The servers re-randomize their shares:
$\big((\hat{cp}_{\mathcal{P}}, \hat{ck}_{\mathcal{P}}, \hat{cp}_{\mathcal{Q}}, \hat{ck}_{\mathcal{Q}}, \hat{p}_{\mathcal{P}}, \hat{k}_{\mathcal{P}}, \hat{op}_{\mathcal{P}}, \hat{ok}_{\mathcal{P}}); (\hat{cp}_{\mathcal{P}}, \hat{ck}_{\mathcal{P}}, \hat{cp}_{\mathcal{Q}}, \hat{ck}_{\mathcal{Q}}, \hat{p}_{\mathcal{Q}}, \hat{k}_{\mathcal{Q}}, \hat{op}_{\mathcal{Q}}, \hat{ok}_{\mathcal{Q}})\big) \xleftarrow{\$}$
$\big\langle \mathcal{P}.\mathsf{ComRefr}\big(p_{\mathcal{P}}, k_{\mathcal{P}}, op_{\mathcal{P}}, ok_{\mathcal{P}}\big), \mathcal{Q}.\mathsf{ComRefr}\big(p_{\mathcal{Q}}, k_{\mathcal{Q}}, op_{\mathcal{Q}}, ok_{\mathcal{Q}}\big)\big\rangle\big(sid, qid, cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}\big)$.
3. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ stores $\big(\hat{cp}_{\mathcal{P}}, \hat{ck}_{\mathcal{P}}, \hat{cp}_{\mathcal{Q}}, \hat{ck}_{\mathcal{Q}}, \hat{p}_{\mathcal{R}}, \hat{k}_{\mathcal{R}}, \hat{op}_{\mathcal{R}}, \hat{ok}_{\mathcal{R}}\big)$ into his long-term storage.

Fig. 9: Refresh instruction: the servers re-randomize their internal state.



Instantiation of zero-knowledge proofs:

$$\pi_8 := \mathcal{F}_{\mathrm{gzk}}[sid, qid, cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, 8]\big\{\big(\nexists p_{\mathcal{P}}, k_{\mathcal{P}}\ ;\ \exists op_{\mathcal{P}}, ok_{\mathcal{P}}, \alpha, \beta\big)\ :$$
$$cp_{\mathcal{P}} = \mathrm{Com}(p_{\mathcal{P}}, op_{\mathcal{P}}) \wedge (\hat{cp}_{\mathcal{P}} * \mathring{cp}) = \mathrm{Com}(p_{\mathcal{P}}, \alpha) \wedge ck_{\mathcal{P}} = \mathrm{Com}(k_{\mathcal{P}}, ok_{\mathcal{P}}) \wedge (\hat{ck}_{\mathcal{P}} * \mathring{ck}) = \mathrm{Com}(k_{\mathcal{P}}, \beta)$$
$$\big\}, \text{where } \alpha := \hat{op}_{\mathcal{P}} + \mathring{op} \text{ and } \beta := \hat{ok}_{\mathcal{P}} + \mathring{ok}.$$

$$\pi_9 := \mathcal{F}_{\mathrm{gzk}}[sid, qid, \mathring{cp}, \mathring{ck}, \hat{cp}_{\mathcal{P}}, \hat{cp}_{\mathcal{Q}}, 9]\big\{\big(\nexists \hat{p}_{\mathcal{Q}}, \hat{k}_{\mathcal{Q}}, \hat{op}_{\mathcal{Q}}\ ;\ \exists \hat{ok}_{\mathcal{Q}}, \alpha, \beta, \big)\ :$$
$$\hat{cp}_{\mathcal{Q}} = \mathrm{Com}(\hat{p}_{\mathcal{Q}}, \hat{op}_{\mathcal{Q}}) \wedge \hat{ck}_{\mathcal{Q}} = \mathrm{Com}(\hat{k}_{\mathcal{Q}}, \hat{ok}_{\mathcal{Q}}) \wedge (cp_{\mathcal{Q}} * \mathring{cp}) = \mathrm{Com}(\hat{p}_{\mathcal{Q}}, \alpha) \wedge (ck_{\mathcal{Q}} * \mathring{ck}) = \mathrm{Com}(\hat{k}_{\mathcal{Q}}, \beta)$$
$$\big\}, \text{where } \alpha := op_{\mathcal{Q}} + \mathring{op} \text{ and } \beta := ok_{\mathcal{Q}} + \mathring{ok}.$$

Fig. 10: Subroutine ComRefr: the servers generate new commitments and shares of the password and key based on the old ones.

the password (for the same reason as in Share: $\mathcal{S}$ needs $\hat{op}_{\mathcal{Q}}$ when simulating $B_{\mathcal{P}}$ in ChkPwd). As part of the proof, $\mathcal{Q}$ sends the new commitments to $\mathcal{P}$.

4.4 Constructing a Multi-Session $\Pi_{\mathbf{2pass}}$ with Constant-Size CRS

In order to handle multiple user accounts, one can run multiple independent sessions of Π_{2pass}. With that first approach, security is guaranteed by direct application of the UC composition theorem. Each session however needs an independent copy of $\mathcal{F}_{\mathrm{crs}}^{\mathbb{G}^3}$. In §D.4 we argue that using the *same* instance of $\mathcal{F}_{\mathrm{crs}}^{\mathbb{G}^3}$ for all the otherwise independent sessions is secure as well. Informally, the second approach works because the CRS is used chiefly by the HMT commitments, which are all bound to sid by the zero-knowledge proofs. Further, the JUC theorem [14] guarantees that all instances in the realizations of $\mathcal{F}_{\mathrm{gzk}}$ and $\mathcal{F}_{\mathrm{gzk}}^{\mathrm{2v}}$ can use the same instance of $\mathcal{F}_{\mathrm{crs}}^{\mathrm{gzk}}$.

4.5 Computational and Communication Complexity in the Standard Model

The sum of the computation time of all parties for Setup, Retrieve, and Refresh queries is less than 0.08, 0.16, and 0.09 seconds for 80/1248-bit security[4] on modern computers,[5] and the communication complexity is 5, 7, and 3 round trips (when combining messages wherever possible), respectively. For the Setup instruction, 43 elements of $\mathbb{Z}_q$, 56 elements of $\mathbb{G}$, 12 elements of $\mathbb{Z}_n$, and 4 elements of $\mathbb{Z}_{n^2}$ are transmitted over plain/TCP channels in our preferred embodiment, corresponding to roughly 5.2 kilobytes for 80/1248-bit security when $\mathbb{G}$ is an elliptic curve. For the Retrieve instruction, 73.5, 99, 16, and 6 elements of $\mathbb{Z}_q$,$\mathbb{G}$,$\mathbb{Z}_n$, and $\mathbb{Z}_{n^2}$ are transmitted respectively (8 kB). For the Refresh instruction, 34, 46, 10, and 4 elements of $\mathbb{Z}_q$,$\mathbb{G}$,$\mathbb{Z}_n$, and $\mathbb{Z}_{n^2}$ are transmitted respectively (4.5 kB). Due to the fact that our protocol is secure against adaptive corruptions, it is computationally more expensive than a standard-model instantiation of the CLN protocol [10] (i.e., with *interactive* zero-knowledge proofs): our Retrieve queries are about 10 and 2.6 times slower for users and servers, respectively; and more data is transferred; however the number of round trips is identical. See §E.3 for a detailed analysis.

4.6 Construction of $\Pi_{\mathbf{2pass}}$ in the Random-Oracle Model

Our Π_{2pass} can be improved in several ways when security in the random-oracle model only is sufficient. First, one can transform all interactive zero-knowledge proofs into non-interactive ones using the Fiat-Shamir transformation [18] in combination with encryption to a public key in the CRS for online extraction [30]. Second, one can replace our secureSend protocol by Nielsen's NINCE [29]. Third, one can use faster encryption and signature algorithms. This improves the computational complexity of our Setup, Retrieve, and Refresh queries by only about 15%, 25%, and 6% but the number of communication rounds is now much smaller: 3, 3, and 2 round trips, respectively. Compared to CLN [10], the computational complexity of our Retrieve queries are then about 11 and 3.7 times larger for users and servers, respectively; the number of round trips is the same. Compared to 1-out-of-2 CLLN [9], the computational complexity of our Retrieve queries are about 2.6 and 4.1 times larger for users and servers, respectively, but need 2 round trips less: if the network delay is large then our protocol is faster than CLLN. See §E.4 for a detailed analysis.

5 Proof Sketch

For reasons of space, we provide the security proof in §D and explain only the main ideas here.

We use the standard approach for proving the security of UC protocols: we construct a straight-line simulator $\mathcal{S}$ such that for all polynomial-time bounded environments and all polynomial-time bounded adversaries $\mathcal{A}$ it holds that the environment $\mathcal{Z}$ cannot distinguish its interaction with $\mathcal{A}$ and Π_{2pass} in the $(\mathcal{F}_{\mathrm{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\mathrm{osac}}, \mathcal{F}_{\mathrm{ac}}, \mathcal{F}_{\mathrm{gzk}}, \mathcal{F}_{\mathrm{gzk}}^{\mathrm{2v}})$-hybrid *real* world from its interaction with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{2pass}}$ in the *ideal* world. We prove this statement by defining a sequence of intermediate *hybrid* worlds (the first one being the real world and the last one the ideal world) and showing that $\mathcal{Z}$ cannot distinguish between any two consecutive hybrid worlds.

The main difficulties in constructing $\mathcal{S}$ (and accordingly in designing our protocol to allow us to address those difficulties) are as follows: *1)* $\mathcal{S}$ has to extract the inputs of all corrupted parties from the interaction with them; *2)* $\mathcal{S}$ has to compute and send commitments and ciphertexts to the corrupted parties on behalf of the honest parties without knowing the latter's inputs, i.e., $\mathcal{S}$ needs to commit and encrypt dummy values; *3)* but when an honest party gets corrupted mid-protocol, $\mathcal{S}$ has to provide $\mathcal{A}$ with the full *non-erased* intermediate state of that party, in particular the opening of commitments that

[4] The subgroup size $|q|$ is 2*80 bits and the RSA modulus size $|n|$ is 1248 bits.

[5] When using the GNU MP (GMP) bignum library on 64-bit Linux on a computer with an Intel Core i7 Q720 1.60GHz CPU.

were sent out and the randomness used to compute encryptions that were sent out (if these value need to be retained by a party).

To address the first difficulty, recall that parties are required to perform proofs of *knowledge* of their shares upon their first use in the protocol. $\mathcal{S}$ can therefore recover the inputs of all corrupted parties with the help of $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$. The commitments and proofs of *existence* with $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ ensure that the corrupted parties are unable to alter their inputs mid-protocol.

The second and third difficulty we address as follows. In general, $\mathcal{S}$ runs honest parties with random input and adjusts their internal state as follows when it learns the correct values. When $\mathcal{S}$ is told by $\mathcal{F}_{\text{2pass}}$ whether the password attempt was correct in a Retrieve query, it can generate credible values B_u, $B_{\mathcal{P}}$, and $B_{\mathcal{Q}}$ in the ChkPwd subroutine because $\mathcal{S}$ can recover the opening values op from dishonest servers through $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$. When a user gets corrupted during Setup, or both servers get corrupted, $\mathcal{S}$ can recover the actual password and key associated with the user account from $\mathcal{F}_{\text{2pass}}$ and then needs to equivocate all relevant commitments and encryptions sent earlier to the corrupted parties. This is also the case when a user gets corrupted during Retrieve, where $\mathcal{S}$ is also allowed to recover the actual password attempt. $\mathcal{S}$ can equivocate such commitments, with the help of the trapdoor, and equivocate the ciphertexts containing the openings of commitments it sent between two honest parties by altering the one-time pads. By the time a one-time pad is used, the decryption keys and randomness used to establish it have been erased and so they can be changed to equivocate. Additionally, $\mathcal{S}$ never needs to reveal the randomness used inside the ChkPwd subroutine, in particular because $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ allow for the erasure of witnesses *before* delivering the statement to be proven to the other party. The rest of the security proof is rather straightforward.

6 Conclusion

We presented the first TPASS protocol secure against adaptive corruptions and where servers can recover from corruptions in a provably secure way. Our protocol involves two servers, and security for the user is guaranteed as long as at most one server is corrupted at any time. Our protocol is efficient enough to be well within reach of a practical implementation. Designing an efficient protocol in the more general t-out-of-n setting is an interesting open problem.

Acknowledgements. We are grateful to the anonymous reviewers of all earlier versions of this paper for their comments, and thank Anja Lehmann for many helpful discussions. This work was supported by the European Community through the Seventh Framework Programme (FP7), under grant agreement n°321310 for the project PERCY.

References

1. A. Bagherzandi, S. Jarecki, N. Saxena, Y. Lu. Password-protected secret sharing. In *ACM CCS 2011*, pages 433–444.
2. B. Barak, R. Canetti, Y. Lindell, R. Pass, T. Rabin. Secure Computation without Authentication. In *CRYPTO 2005*, pages 361–377.
3. Donald Beaver, Stuart Haber. Cryptographic Protocols Provably Secure Against Dynamic Adversaries. In *EUROCRYPT 1991*, pages 307–323.
4. J. Brainard, A. Juels, B. Kaliski, M. Szydlo. A new two-server approach for authentication with short secrets. In *USENIX SECURITY 2003*, pages 201–214.
5. W. Burr, D. Dodson, E. Newton, R. Perlner, W. Polk, S. Gupta, E. Nabbus. Electronic authentication guideline. NIST Special Publication 800-63-1, 2011.
6. J. Camenisch, R. R. Enderlein, G. Neven. Two-Server Password-Authenticated Secret Sharing UC-Secure Against Transient Corruptions. *IACR Cryptology ePrint Archive*, 2015:006.
7. J. Camenisch, R. R. Enderlein, V. Shoup. Practical and Employable Protocols for UC-Secure Circuit Evaluation over $\mathbb{Z}_n$. In *ESORICS 2013*, pages 19–37.
8. J. Camenisch, S. Krenn, V. Shoup. A Framework for Practical Universally Composable Zero-Knowledge Protocols. In *ASIACRYPT 2011*, pages 449–467.
9. J. Camenisch, A. Lehmann, A. Lysyanskaya, G. Neven. Memento: How to Reconstruct Your Secrets from a Single Password in a Hostile Environment. In *CRYPTO 2014*, pages 256–275.
10. J. Camenisch, A. Lysyanskaya, G. Neven. Practical Yet Universally Composable Two-Server Password-Authenticated Secret Sharing. In *ACM CCS 2012*, pages 525–536.
11. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. *IACR Cryptology ePrint Archive*, 2000:67.
12. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. *FOCS 2001*, pages 136–145.
13. R. Canetti, S. Halevi, J. Katz, Y. Lindell, P. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT 2005*, pages 404–421.
14. R. Canetti, T. Rabin. Universal composition with joint state. In *CRYPTO 2003*, pages 265–281.
15. R. Cramer, V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO 1998*, pages 13–25.
16. M. Di Raimondo, R. Gennaro. Provably secure threshold password-authenticated key exchange. In *EUROCRYPT 2003*, pages 507–523.
17. EMC Corporation. RSA Distributed Credential Protection. `http://www.emc.com/security/rsa-distributed-credential-protection.htm`.
18. A. Fiat, A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO 1986*, pages 186–194.
19. W. Ford, B. Kaliski. Server-assisted generation of a strong secret from a password. In *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*, pages 176–180.
20. J. Gosney. Password cracking HPC. Passwords^12 Conference, 2012.
21. A. Herzberg, S. Jarecki, H. Krawczyk, M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO 1995*, pages 339–352.
22. D. Hofheinz, V. Shoup. GNUC: A new universal composability framework. *IACR Cryptology ePrint Archive*, 2011:303.
23. D. Hofheinz. Possibility and impossibility results for selective decommitments. *J. Cryptology*, 24(3):470–516, 2011.
24. D. Jablon. Password authentication using multiple servers. In *CT-RSA 2001*, pages 344–360.
25. S. Jarecki, A. Kiayias, H. Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT 2014*, pages 233–253.
26. J. Katz, P. MacKenzie, G. Taban, V. Gligor. Two-server password-only authenticated key exchange. In *J. of Computer and System Sciences* 78(2): 651–669, 2012.
27. S. Krenn. *Bringing zero-knowledge proofs of knowledge to practice*. PhD thesis, 2012.
28. P. MacKenzie, T. Shrimpton, M. Jakobsson. Threshold password-authenticated key exchange. In *CRYPTO 2002*, pages 385–400.
29. J. B. Nielsen. Separating random oracle proofs from complexity theoretic proofs: the non-committing encryption case. In *CRYPTO 2002*, pages 111–126.
30. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT 1999*, pages 223–238.
31. T. P. Pedersen, B. Pfitzmann. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO 1991*, pages 129–140.

32. N. Provos, D. Mazières. A future-adaptable password scheme. In *USENIX 1999, FREENIX Track*, pages 81–91.
33. C. Rackoff, D. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. *CRYPTO 1991*, pages 433–444.
34. M. Szydlo, B. Kaliski. Proofs for two-server password authentication. In *CT-RSA 2005*, pages 227–244.

A Our Ideal Functionality $\mathcal{F}_{\mathbf{2pass}}$

In this section, we provide a full definition of our $\mathcal{F}_{\text{2pass}}$ ideal functionality which we already briefly described in §2. We use the GNUC [22] formalisms to define $\mathcal{F}_{\text{2pass}}$, but it is easy to adapt our ideal functionality to any universal composability framework.

As we model the single-session variant of $\mathcal{F}_{\text{2pass}}$, the session id $\underline{sid}$ is fixed. Recall that sid comprises the identity of the two servers $pid_{\mathcal{P}}$ and $pid_{\mathcal{Q}}$, the description of a group $(\mathbb{G}, q, g)$ of prime order q, the name of the user account $uacc$, and a suffix $ssid$. Each instance of $\mathcal{F}_{\text{2pass}}$ checks that $\underline{sid}$ is of the correct format when first invoked. Also, only messages with the correct $sid = \underline{sid}$ are accepted.

Interfaces. $\mathcal{F}_{\text{2pass}}$ is a four-interface system:

- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{U}$-interface, connected to the ideal peers of the users. This interface is multiplexed; we assume that headers are added to all messages to enable proper routing.
- The $\mathcal{P}$-interface, connected to the ideal peer of the first server $\mathcal{P}$.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the second server $\mathcal{Q}$.

State. The ideal functionality is stateful and maintains the following data structures. We underline these datastructures to distinguish them from local variables.

- $\underline{Seen}$: associative array between $\{0,1\}^*$ and a subset of $\{0,1\}^*$. Keeps track of which messages were accepted per qid.
- $\underline{Corrupted}$: a set of party ids (servers and users). Keeps track of who is currently corrupted.
- $\underline{SetupDone}$: subset of $\{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$. Keeps track of which server successfully completed the Setup query.
- $\underline{JustRecovered}$: a subset of $\{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$. Keeps track of which servers have formally recovered from corruption, but not yet completed the subsequent Refresh query.
- $\underline{Refreshing}$: a subset of $\{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$. Keeps track of which servers have started doing a Refresh query.
- $\underline{RefreshPeriod}^{\mathcal{P}}$ and $\underline{RefreshPeriod}^{\mathcal{Q}}$: integers. Keep track of the refresh period (the time between two Refresh queries) of each server.
- $\underline{QidPeriod}^{\mathcal{P}}$ and $\underline{QidPeriod}^{\mathcal{Q}}$: associative arrays between $\{0,1\}^*$ and an integer. Keep track of which qids belong to which refresh period.
- $\underline{CorruptedIn}$: associative array between an integer and a subset of $\{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$. Keeps track of which servers where corrupted in a given refresh period.
- $\underline{p}^{\mathcal{P}}$ and $\underline{p}^{\mathcal{Q}}$: elements of $\mathbb{Z}_q \cup \{\perp, \triangle\}$. The password stored by the user, in the view of the given server. (Normally both values are equal. The values are unequal in case a server recovered from corruption in an altered state. All queries including the Refresh query will be blocked if the values are unequal. The symbol $\triangle$ is represents the fact that the simulator does not yet know the value of the stored password, however it must provide a concrete value before the end of the Refresh query.)
- $\underline{k}^{\mathcal{P}}$ and $\underline{k}^{\mathcal{Q}}$: elements of $\mathbb{Z}_q \cup \{\perp, \triangle\}$. The key stored by the user, in the view of the given server. (Same comments as for $\underline{p}^{\mathcal{P}}$ and $\underline{p}^{\mathcal{Q}}$.)
- $\underline{a}$: associative array between $\{0,1\}^*$ and an element of $\mathbb{Z}_q \cup \{\perp\}$. The password attempts per qid.
- $\underline{k}'$: associative array between $\{0,1\}^*$ and an element of $\mathbb{Z}_q \cup \{\perp, \varepsilon\}$. The key to be returned to the user per qid, where ε means the user input an incorrect password attempt.
- $\underline{U}$: associative array between $\{0,1\}^*$ and a user in the system. Keeps track of which user initiated a query. If $\underline{U}[qid] = \mathcal{A}$, $\mathcal{F}_{\text{osac}}$ sends/receives messages on the network interface instead of the $\mathcal{U}$-interface as written for the query qid.

The default value of these are as follows: associative arrays are initially empty. If no value is associated to a given key in an array, then $\perp$ is returned. Sets are initially empty. Integers are initially 0. All other values are initially $\perp$.

Aborting all queries during a refresh. In $\mathcal{F}_{\text{2pass}}$, we enforce that once an honest server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ starts a Refresh query, all other queries are aborted. To simplify the exposition, we define the following predicates: let $\text{AcceptNewQid}(qid, \mathcal{R})$ denote the predicate $(pid_{\mathcal{R}} \in \underline{Corrupted} \vee (pid_{\mathcal{R}} \notin \underline{Refreshing} \wedge pid_{\mathcal{R}} \notin \underline{JustRecovered} \wedge \underline{QidPeriod}^{\mathcal{R}}[qid] = \perp))$ and let $\text{AcceptQid}(qid, \mathcal{R})$ denote the predicate $(pid_{\mathcal{R}} \in \underline{Corrupted} \vee (pid_{\mathcal{R}} \notin \underline{Refreshing} \wedge pid_{\mathcal{R}} \notin \underline{JustRecovered} \wedge \underline{QidPeriod}^{\mathcal{R}}[qid] = \underline{RefreshPeriod}^{\mathcal{R}}))$. The first predicate checks that qid has not yet been seen by a server $\mathcal{R}$. The second predicate checks that qid has already been seen, and that there was no Refresh query between now and the moment qid was first seen. Both predicates also check that $\mathcal{R}$ is not currently blocking new queries due to recently formally recovering from corruption or due to currently running a Refresh query. Both predicates can be overriden by a corrupt $\mathcal{R}$.

Reacting to Messages. Our $\mathcal{F}_{\text{2pass}}$ reacts to messages as follows.

Setup instruction. Recall that the Setup instruction allows a user to store her password and key in the ideal functionality. The user starts by privately inputing her key and password to $\mathcal{F}_{\text{2pass}}$ (Message 1). The servers have to state that they are ready to execute a Setup query by notifying $\mathcal{F}_{\text{2pass}}$ (Message 2). (Messages 1 and 2 can happen in any order.) After these two steps, $\mathcal{F}_{\text{2pass}}$ sends a public delayed acknowledgement to the servers (Message 3), and finally sends a public delayed acknowledgement to the user (Message 4).

1. Receive $\langle$Setup, $\underline{sid}$, $qid, p, k\rangle$ on $\mathcal{U}$ (from a user $pid_{\mathcal{U}}$),
   where $qid =$ "Setup", $p \in \mathbb{Z}_q$, and $k \in \mathbb{Z}_q$,
   such that {"Setup", "Retrieve"} $\cap\ \underline{Seen}[qid] = \emptyset$:

   Insert "Setup" into $\underline{Seen}[qid]$.
   Record the user: $\underline{U}[qid] \leftarrow pid_{\mathcal{U}}$. Save the user's input: $\underline{p}^{\mathcal{P}} \leftarrow p$, $\underline{p}^{\mathcal{Q}} \leftarrow p$, $\underline{k}^{\mathcal{P}} \leftarrow k$, and $\underline{k}^{\mathcal{Q}} \leftarrow k$.
   Send $\langle$Setup, $\underline{sid}$, qid, $\mathcal{U}\rangle$ on network.

2. Receive $\langle$ReadySetup:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on $\mathcal{R}$,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $qid =$ "Setup", and $\text{AcceptNewQid}(qid, \mathcal{R})$,
   such that {"ReadySetup:$\mathcal{R}$", "ReadyRetrieve:$\mathcal{R}$", "Refresh:$\mathcal{R}$"} $\cap\ \underline{Seen}[qid] = \emptyset$:

   Insert "ReadySetup:$\mathcal{R}$" into $\underline{Seen}[qid]$.
   Set $\underline{QidPeriod}^{\mathcal{R}}[qid] \leftarrow \underline{RefreshPeriod}^{\mathcal{R}}$.
   Send $\langle$ReadySetup:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on network.

3. Receive $\langle$Done:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on network,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ and $\text{AcceptQid}(qid, \mathcal{R})$,
   such that {"Done:$\mathcal{R}$"} $\cap\ \underline{Seen}[qid] = \emptyset$,
   and {"Setup", "ReadySetup:$\mathcal{P}$", "ReadySetup:$\mathcal{Q}$"} $\subset \underline{Seen}[qid]$:

   Insert "Done:$\mathcal{R}$" into $\underline{Seen}[qid]$.
   Insert $pid_{\mathcal{R}}$into $\underline{SetupDone}$.
   Send $\langle$Done:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on $\mathcal{R}$.

4. Receive $\langle$Done, $\underline{sid}$, $qid\rangle$ on network,
   such that {"Done"} $\cap$ $\underline{Seen}[qid] = \emptyset$
   and {"Done:$\mathcal{P}$", "Done:$\mathcal{Q}$"} $\subset$ $\underline{Seen}[qid]$:

   Insert "Done" into $\underline{Seen}[qid]$.
   Send $\langle$Done, $\underline{sid}$, $qid\rangle$ on $\mathcal{U}$ (to user $\underline{U}[qid]$).

Retrieve instruction. Recall that the Retrieve instruction allows a user to recover the key stored in $\mathcal{F}_{\text{2pass}}$ provided she knows the correct password. The user starts by privately inputing her password attempt to $\mathcal{F}_{\text{2pass}}$ (Message 5). The servers have to state that they are ready to execute a Retrieve query and willing to service the user's query by notifying $\mathcal{F}_{\text{2pass}}$ (Message 6). (Messages 5 and 6 can happen in any order.) The adversary is the first to learn of the result of the password check: by sending a lock message to $\mathcal{F}_{\text{2pass}}$, the latter tells the former the result of that check (Message 7). After these two steps, $\mathcal{F}_{\text{2pass}}$ sends a public delayed message to the servers with the result of the check (Message 8), and finally sends a public delayed message to the user containing the key or an empty message in case the password was wrong (Message 9).

5. Receive $\langle$Retrieve, $\underline{sid}$, $qid, a\rangle$ on $\mathcal{U}$ (from user $pid_{\mathcal{U}}$),
   where $a \in \mathbb{Z}_q$,
   such that {"Retrieve", "Setup"} $\cap$ $\underline{Seen}[qid] = \emptyset$:

   Insert "Retrieve" into $\underline{Seen}[qid]$.
   Record the user: $\underline{U}[qid] \leftarrow \mathcal{U}$. Save the user's input: $\underline{a}[qid] \leftarrow a$.
   Send $\langle$Retrieve, $\underline{sid}$, qid, $\mathcal{U}\rangle$ on network.

6. Receive $\langle$ReadyRetrieve:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on $\mathcal{R}$,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $\text{AcceptNewQid}(qid, \mathcal{R})$, and $pid_{\mathcal{R}} \in (\underline{SetupDone} \cup \underline{Corrupted})$,
   such that {"ReadyRetrieve:$\mathcal{R}$", "ReadySetup:$\mathcal{R}$", "Refresh:$\mathcal{R}$"} $\cap$ $\underline{Seen}[qid] = \emptyset$:

   Insert "ReadyRetrieve:$\mathcal{R}$" into $\underline{Seen}[qid]$.
   Set $\underline{QidPeriod}^{\mathcal{R}}[qid] \leftarrow \underline{RefreshPeriod}^{\mathcal{R}}$.
   Send $\langle$ReadyRetrieve:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on network.

7. Receive $\langle$Lock, $\underline{sid}$, $qid\rangle$ on network,
   where $\underline{p}^{\mathcal{P}} = \underline{p}^{\mathcal{Q}}$, $\underline{k}^{\mathcal{P}} = \underline{k}^{\mathcal{Q}}$, $\underline{p}^{\mathcal{P}} \neq \triangle$, and $\underline{k}^{\mathcal{P}} \neq \triangle$,
   such that {"Lock"} $\cap$ $\underline{Seen}[qid] = \emptyset$,
   and {"Retrieve", "ReadyRetrieve:$\mathcal{P}$", "ReadyRetrieve:$\mathcal{Q}$"} $\subset$ $\underline{Seen}[qid]$:

   Insert "Lock" into $\underline{Seen}[qid]$.
   If $\underline{a}[qid] = \underline{p}^{\mathcal{P}}$, then set $b \leftarrow 1$ and $\underline{k}'[qid] \leftarrow \underline{k}^{\mathcal{P}}$; else set $b \leftarrow 0$ and $\underline{k}'[qid] \leftarrow \varepsilon$.
   Send $\langle$Lock, $\underline{sid}$, $qid, b\rangle$ on network.

8. Receive $\langle$Delivered:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on network,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ and $\text{AcceptQid}(qid, \mathcal{R})$,
   such that {"Delivered:$\mathcal{R}$"} $\cap$ $\underline{Seen}[qid] = \emptyset$,
   and {"Lock"} $\subset$ $\underline{Seen}[qid]$:

Insert "Delivered:$\mathcal{R}$" into $\underline{Seen}[qid]$.
If $\underline{k}'[qid] \neq \varepsilon$, then set $b \leftarrow 1$; else set $b \leftarrow 0$.
Send $\langle$Delivered:$\mathcal{R}$, $\underline{sid}$, qid, $b\rangle$ on $\mathcal{R}$.

9. Receive $\langle$Deliver, $\underline{sid}$, $qid\rangle$ on network,
such that {"Deliver"} $\cap$ $\underline{Seen}[qid] = \emptyset$
and {"Delivered:$\mathcal{P}$", "Delivered:$\mathcal{Q}$"} $\subset$ $\underline{Seen}[qid]$:

Insert "Deliver" into $\underline{Seen}[qid]$.
Send $\langle$Deliver, $\underline{sid}$, qid, $\underline{k}'[qid]\rangle$ on $\mathcal{U}$ (to user $\underline{U}[qid]$).

Refresh instruction. Recall that the Refresh instruction allows the servers to jointly re-randomize their internal states and thereby clear the residual knowledge that $\mathcal{A}$ might have. The servers have to state that they are ready to execute a Refresh query by publicly notifying $\mathcal{F}_{\text{2pass}}$ (Message 10). Afterwards, $\mathcal{F}_{\text{2pass}}$ sends a public delayed acknowledgement to the servers (Message 11). We note that while Refresh is active, $\mathcal{F}_{\text{2pass}}$ accepts no other queries and drops all incomplete queries.

10. Receive $\langle$Refresh:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on $\mathcal{R}$,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $pid_{\mathcal{R}} \in (\underline{SetupDone} \cup \underline{Corrupted})$, and $pid_{\mathcal{R}} \notin \underline{Refreshing}$,
such that {"Refresh:$\mathcal{R}$", "ReadySetup:$\mathcal{R}$", "ReadyRetrieve:$\mathcal{R}$"} $\cap$ $\underline{Seen}[qid] = \emptyset$:

Insert "Refresh:$\mathcal{R}$" into $\underline{Seen}[qid]$.
Insert $pid_{\mathcal{R}}$into $\underline{Refreshing}$. If $\underline{Corrupted} \neq \emptyset$, then $\underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}} + 1] \leftarrow \underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}}]$.
Send $\langle$Refresh:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on network.

11. Receive $\langle$RefreshDone:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $\underline{p}^{\mathcal{P}} = \underline{p}^{\mathcal{Q}}$, $\underline{k}^{\mathcal{P}} = \underline{k}^{\mathcal{Q}}$, $\underline{p}^{\mathcal{P}} \neq \triangle$, and $\underline{k}^{\mathcal{Q}} \neq \triangle$,
such that {"RefreshDone:$\mathcal{R}$"} $\cap$ $\underline{Seen}[qid] = \emptyset$,
and {"Refresh:$\mathcal{P}$", "Refresh:$\mathcal{Q}$"} $\subset$ $\underline{Seen}[qid]$:

Insert "RefreshDone:$\mathcal{R}$" into $\underline{Seen}[qid]$.
Increment $\underline{RefreshPeriod}^{\mathcal{R}}$ by 1. Remove $pid_{\mathcal{R}}$from $\underline{Refreshing}$ and $\underline{JustRecovered}$.
Send $\langle$RefreshDone:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on $\mathcal{R}$.

Corruption. We now present all instructions that have to do with corruption, hijacking, and recovery from corruption. Servers (Message 12) and users (Message 13) can be corrupted if $\mathcal{F}_{\text{2pass}}$ receives a special corrupt message from the environment. In our protocol, the first message of the user can be hijacked by $\mathcal{A}$; in $\mathcal{F}_{\text{2pass}}$ this is modelled by allowing $\mathcal{A}$ to take over the query and the user doesn't continue with the query (Messages 14 and 15). $\mathcal{F}_{\text{2pass}}$ allows $\mathcal{A}$ the following behaviour. If the user is corrupt, $\mathcal{A}$ can recover her input (Messages 16 and 17). If both servers were corrupt in the same refresh period (between Refresh queries), then the user's password and key are exposed (Message 18). If both servers are corrupt at the same time, then $\mathcal{A}$ may make $\mathcal{F}_{\text{2pass}}$ return whatever it wants during Retrieve (Message 19). If a server is corrupt, $\mathcal{A}$ can modify its internal state (Message 20)—note that unless that state is consistent across both servers, none of the queries will work—, here we note that $\mathcal{A}$ may set the state to a special symbol $\triangle$ instead of providing a value directly. If a server's state is $\triangle$, $\mathcal{A}$ may set the real state

at a later point in time (Message 21). This models the fact that $\mathcal{S}$ may not know the value of the saved password or key if $\mathcal{A}$ sets the servers to an inconsistent state. Note that $\mathcal{F}_{\text{2pass}}$ will not complete any queries while in that state. Finally, the environment may uncorrupt servers by sending a special Recover message (Message 22). We note that $\mathcal{A}$ has had the chance to specify the internal state of the recovered server before Recover is called with the previous messages.

12. Receive $\langle$Corrupt, $\underline{sid}\rangle$ on $\mathcal{R}$,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ and $pid_{\mathcal{R}} \notin \underline{Corrupted}$:

    Insert $pid_{\mathcal{R}}$into $\underline{Corrupted}$ and into $\underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}}]$.
    If $pid_{\mathcal{R}} \in \underline{Refreshing}$, then $\underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}} + 1] \leftarrow \underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}}]$.
    Send $\langle$Corrupt, $\underline{sid}, \mathcal{R}\rangle$ on network.

    In the GNUC model, $\mathcal{F}_{\text{2pass}}$ also sends out invitations for all applicable ExposeSetup messages on network. This is due to the modelling of the runtime of Turing machines in GNUC. In other models, this is not necessary.

13. Receive $\langle$CorruptUser, $\underline{sid}\rangle$ on $\mathcal{U}$ (from a user $pid_{\mathcal{U}}$),
where $\mathcal{U} \notin \underline{Corrupted}$:

    Insert $pid_{\mathcal{U}}$into $\underline{Corrupted}$.
    Send $\langle$CorruptUser, $\underline{sid}, pid_{\mathcal{U}}\rangle$ on network.

    In the GNUC model, $\mathcal{F}_{\text{2pass}}$ also sends out invitations for all applicable ExposeUserSetup and ExposeUserRetrieve messages on network.

    To simplify the presentation, we do not allow users to become uncorrupted.

14. Receive $\langle$HijackSetup, $\underline{sid}, qid, p, k\rangle$ on network,
such that {"HijackSetup", "Done:$\mathcal{P}$", "Done:$\mathcal{Q}$"} $\cap$ $\underline{Seen}[qid] = \emptyset$
and {"Setup"} $\subset$ $\underline{Seen}[qid]$:

    Insert "HijackSetup" into $\underline{Seen}[qid]$.
    Change $\underline{U}[qid] \leftarrow \mathcal{A}$. Change the input: $\underline{p}^{\mathcal{P}} \leftarrow p$, $\underline{p}^{\mathcal{Q}} \leftarrow p$, $\underline{k}^{\mathcal{P}} \leftarrow k$, and $\underline{k}^{\mathcal{Q}} \leftarrow k$.
    Send $\langle$HijackSetup, $\underline{sid}, qid\rangle$ on network.

15. Receive $\langle$HijackRetrieve, $\underline{sid}, qid, a\rangle$ on network,
such that {"HijackRetrieve", "Lock"} $\cap$ $\underline{Seen}[qid] = \emptyset$
and {"Retrieve"} $\subset$ $\underline{Seen}[qid]$:

    Insert "HijackRetrieve" into $\underline{Seen}[qid]$.
    Change $\underline{U}[qid] \leftarrow \mathcal{A}$. Change the input: $\underline{a}[qid] \leftarrow a$.
    Send $\langle$HijackRetrieve, $\underline{sid}, qid\rangle$ on network.

16. Receive $\langle$ExposeUserSetup, $\underline{sid}$, $qid\rangle$ on network,
where $\underline{U}[qid] \in \underline{Corrupted}$,
such that {"ExposeUserSetup", "Done:$\mathcal{P}$", "Done:$\mathcal{Q}$"} $\cap$ $\underline{Seen}[qid] = \emptyset$,
and {"Setup"} $\subset$ $\underline{Seen}[qid]$:

    Insert "ExposeUserSetup" into $\underline{Seen}[qid]$.
    Send $\langle$ExposeUserSetup, $\underline{sid}$, qid, $\underline{p}^{\mathcal{P}}$, $\underline{k}^{\mathcal{P}}\rangle$ on network.

17. Receive $\langle$ExposeUserRetrieve, $\underline{sid}$, $qid\rangle$ on network,
where $\underline{U}[qid] \in \underline{Corrupted}$,
such that {"ExposeUserRetrieve", "Lock"} $\cap$ $\underline{Seen}[qid] = \emptyset$,
and {"Retrieve"} $\subset$ $\underline{Seen}[qid]$:

    Insert "ExposeUserRetrieve" into $\underline{Seen}[qid]$.
    Send $\langle$ExposeUserRetrieve, $\underline{sid}$, qid, $\underline{a}[qid]\rangle$ on network.

18. Receive $\langle$ExposeSetup, $\underline{sid}$, $qid\rangle$ on network,
where $\underline{RefreshPeriod}^{\mathcal{P}} = \underline{RefreshPeriod}^{\mathcal{Q}}$, $\underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{P}}] = \{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$, $\underline{p}^{\mathcal{P}} = \underline{p}^{\mathcal{Q}}$, and $\underline{k}^{\mathcal{P}} = \underline{k}^{\mathcal{Q}}$,
such that {"ExposeSetup"} $\cap$ $\underline{Seen}[qid] = \emptyset$,
and {"Setup"} $\subset$ $\underline{Seen}[qid]$:

    Insert "ExposeSetup" into $\underline{Seen}[qid]$.
    Send $\langle$ExposeSetup, $\underline{sid}$, qid, $\underline{p}^{\mathcal{P}}$, $\underline{k}^{\mathcal{P}}\rangle$ on network.

19. Receive $\langle$ModifyRetrieveResponse, $\underline{sid}$, qid, $k\rangle$ on network,
where $\underline{Corrupted} = \{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$ and $k \in \mathbb{Z}_q \cup \{\varepsilon\}$:

    Replace: $\underline{k}'[qid] \leftarrow k$.
    Send $\langle$ModifyRetrieveResponse, $\underline{sid}$, $qid\rangle$ on network.

20. Receive $\langle$ModifySetup:$\mathcal{R}$, $\underline{sid}$, p, k, $b\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $pid_{\mathcal{R}} \in \underline{Corrupted}$, $p \in \mathbb{Z}_q \cup \{\perp, \triangle\}$, $k \in \mathbb{Z}_q \cup \{\perp, \triangle\}$, and $b \in \mathbb{Z}_2$:

    Replace: $\underline{p}^{\mathcal{R}} \leftarrow p$, $\underline{k}^{\mathcal{R}} \leftarrow k$. If $b = 1$ then add $pid_{\mathcal{R}}$to $\underline{SetupDone}$; else remove $pid_{\mathcal{R}}$from $\underline{SetupDone}$.
    Send $\langle$ModifySetup:$\mathcal{R}$, $\underline{sid}\rangle$ on network.

21. Receive $\langle$FalseMemory:$\mathcal{R}$, $\underline{sid}$, qid, p, $k\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $\underline{p}^{\mathcal{R}} = \triangle$, $\underline{k}^{\mathcal{R}} = \triangle$, $p \in \mathbb{Z}_q$, and $k \in \mathbb{Z}_q$:

    Replace: $\underline{p}^{\mathcal{R}} \leftarrow p$, $\underline{k}^{\mathcal{R}} \leftarrow k$.
    Send $\langle$FalseMemory:$\mathcal{R}$, $\underline{sid}\rangle$ on network.

22. Receive $\langle$Recover, $\underline{sid}, \mathcal{R}\rangle$ on $\mathcal{R}$,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ and $pid_{\mathcal{R}} \in \underline{Corrupted}$:

If $\underline{Corrupted} = \{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$, then let $b \leftarrow \max(\underline{RefreshPeriod}^{\mathcal{P}}, \underline{RefreshPeriod}^{\mathcal{Q}}) + 1$, $\underline{RefreshPeriod}^{\mathcal{P}} \leftarrow b$, $\underline{RefreshPeriod}^{\mathcal{Q}} \leftarrow b$, and $\underline{CorruptedIn}[b] \leftarrow \{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$.
Remove $pid_{\mathcal{R}}$from $\underline{Corrupted}$ and from $\underline{Refreshing}$.
If $pid_{\mathcal{R}} \in \underline{SetupDone}$, then insert $pid_{\mathcal{R}}$into $\underline{JustRecovered}$; else remove $pid_{\mathcal{R}}$from $\underline{JustRecovered}$.
Send $\langle$Recover, $\underline{sid}, \mathcal{R}\rangle$ on network.

To simplify the presentation, we chose not to model the fact that the recovered server might accept qids he has seen already or reject qids that he has not yet seen. Fixing this issue is tedious but not difficult. In practice where qids are chosen at random by a protocol preceeding ours, this issue is moot.

B Auxilliary Ideal Functionalities

In this section, we recall the definition of the ideal functionalities we use as subroutines in our realization Π_{2pass}, namely: the common reference string ideal functionality ($\mathcal{F}_{\text{crs}}^{D}$), the ideal functionalities for authenticated channels ($\mathcal{F}_{\text{ac}}$) and for one-sided–authenticated channels ($\mathcal{F}_{\text{osac}}$), and the special ideal functionality for zero-knowledge proofs of existence for one verifier ($\mathcal{F}_{\text{gzk}}$) and two verifiers ($\mathcal{F}_{\text{gzk}}^{\text{2v}}$). We adapted some of the functionalities found in the litterature to the GNUC model, and modified some other functionalities to suit the needs of our protocol.

B.1 Common Reference Strings $\mathcal{F}_{\text{crs}}^{D}$

Here we describe the ideal functionality for common reference strings $\mathcal{F}_{\text{crs}}^{D}$ for a distribution D. Recall that we make use of two distributions in this paper: $\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$ outputs a CRS uniformly distributed over $\mathbb{G}^3$ and $\mathcal{F}_{\text{crs}}^{\text{gzk}}$ outputs a CRS according to the distribution required by Camenisch et al.'s protocol π for zero-knowledge proofs of existance [8]. The structure of $\mathcal{F}_{\text{crs}}^{D}$ is the same as what is defined in Canetti's UC paper [11], but adapted to the GNUC model:

Interfaces. $\mathcal{F}_{\text{crs}}^{D}$ is a two-interface system:
- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{U}$-interface, connected to the ideal peers of all the parties. This interface is multiplexed; we assume that headers are added to all messages to enable proper routing.

State. The ideal functionality is stateful and maintains the following data structures:
- $\underline{Seen}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{x}$: the stored CRS, initially $\underline{x} = \perp$.
We model the single-session variant of $\mathcal{F}_{\text{crs}}^{D}$, the session id $sid = \underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\text{crs}}^{D}$ reacts to messages as follows.

1. Receive $\langle$GetCRS:$pid, sid\rangle$ on $\mathcal{U}$ (from a party pid),
such that $\{$"GetCRS:pid"$\} \cap \underline{Seen} = \emptyset$:

Insert "GetCRS:pid" into $\underline{Seen}$.
If $\underline{x} = \perp$, then randomly choose $\underline{x}$ according to distribution D.
Send $\langle$GetCRS:$pid, sid, \underline{x}\rangle$ on network.

2. Receive $\langle$Deliver:$pid, sid\rangle$ on network,
   such that {"Deliver:pid"} $\cap$ $\underline{Seen} = \emptyset$
   and {"GetCRS:pid"} $\subset$ $\underline{Seen}$:

   Insert "Deliver:pid" into $\underline{Seen}$.
   Send $\langle$Deliver:$pid, sid, \underline{x}\rangle$ on $\mathcal{U}$ (to party pid).

B.2 Authenticated Channels $\mathcal{F}_{\mathbf{ac}}$

Here we describe the ideal functionality for single-use authenticated channels $\mathcal{F}_{\mathrm{ac}}$. The structure is the same as the one defined in Hofheinz and Shoup's GNUC paper [22]:

Interfaces. $\mathcal{F}_{\mathrm{ac}}$ is a three-interface system:
- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{P}$-interface, connected to the ideal peer of the sender.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the receiver.

State. The ideal functionality is stateful and maintains the following data structures:
- $\underline{Seen}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{x}$: the message that is to be sent, where $\underline{x} \in \{0,1\}^*$.
  We model the single-session variant of $\mathcal{F}_{\mathrm{ac}}$, the session id $sid = \underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\mathrm{ac}}$ reacts to messages as follows.

1. Receive $\langle$Send, $sid, x\rangle$ on $\mathcal{P}$,
   such that {"Send"} $\cap$ $\underline{Seen} = \emptyset$:

   Insert "Send" into $\underline{Seen}$.
   Store the message: $\underline{x} \leftarrow x$.
   Send $\langle$Send, $sid, \underline{x}\rangle$ on network.

2. Receive $\langle$Ready, $sid\rangle$ on $\mathcal{Q}$,
   such that {"Ready"} $\cap$ $\underline{Seen} = \emptyset$:

   Insert "Ready" into $\underline{Seen}$.
   Send $\langle$Ready, $sid\rangle$ on network.

3. Receive $\langle$Done, $sid\rangle$ on network,
   such that {"Done"} $\cap$ $\underline{Seen} = \emptyset$
   and {"Send", "Ready"} $\subset$ $\underline{Seen}$:

   Insert "Done" into $\underline{Seen}$.
   Send $\langle$Done, $sid\rangle$ on $\mathcal{P}$.

4. Receive $\langle$Deliver, $sid, x\rangle$ on network,
   where $x = \underline{x}$,
   such that {"Deliver"} $\cap$ $\underline{Seen} = \emptyset$,
   and {"Send", "Ready"} $\subset$ $\underline{Seen}$:

Insert "Deliver" into $\underline{Seen}$.
Send $\langle$Deliver, $sid, \underline{x}\rangle$ on $\mathcal{Q}$.

5. Receive $\langle$Corrupt:$\mathcal{R}, sid\rangle$ on $\mathcal{R}$,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
such that $\{$"Corrupt:$\mathcal{R}$"$\} \cap \underline{Seen} = \emptyset$:

   Insert "Corrupt:$\mathcal{R}$" into $\underline{Seen}$.
   Send $\langle$Corrupt:$\mathcal{R}, sid\rangle$ on network.

6. Receive $\langle$Reset, $sid, x\rangle$ on network,
such that $\{$"Reset"$\} \cap \underline{Seen} = \emptyset$
and $\{$"Corrupt:$\mathcal{P}$"$\} \subset \underline{Seen}$:

   Insert "Reset" into $\underline{Seen}$.
   Store a new message: $\underline{x} \leftarrow x$.
   Send $\langle$Reset, $sid\rangle$ on network.

B.3 One-Sided–Authenticated Channels $\mathcal{F}_{\text{osac}}$

Here we describe our ideal functionality for multi-use one-sided–authenticated channels $\mathcal{F}_{\text{osac}}$. The structure is similar to the regular $\mathcal{F}_{\text{ac}}$ with the obvious extensions for multi-use, but we added an additional Hijack instruction to model the fact that the first message from the user is not authenticated.

Interfaces. $\mathcal{F}_{\text{osac}}$ is a three-interface system:
- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{U}$-interface, connected to the ideal peers of the users. This interface is multiplexed; we assume that headers are added to all messages to enable proper routing.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the initial receiver.

State. The ideal functionality is stateful and maintains the following data structures:
- $\underline{Seen}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{xc}$: an associative array between an integer and the message that is to be sent to the server.
- $\underline{xs}$: an associative array between an integer and the message that is to be sent to the user.
- $\underline{U}$: a user in the system. Keeps track of which user initiated a query. If $\underline{U} = \mathcal{A}$, $\mathcal{F}_{\text{osac}}$ sends/receives messages on the network interface instead of the $\mathcal{U}$-interface as written.
  We model the single-session variant of $\mathcal{F}_{\text{osac}}$, the session id $sid = \underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\text{osac}}$ reacts to messages as follows.

Message from user to server. For messages that the user sends to the server, $\mathcal{F}_{\text{osac}}$ proceeds similarily to a multi-session $\mathcal{F}_{\text{ac}}$, except that the first message might be hijacked by $\mathcal{A}$.

1. Receive $\langle$Send:c:0, $sid, x\rangle$ on $\mathcal{U}$ (from a user pid_u),
such that $\{$"Send:c:0"$\} \cap \underline{Seen} = \emptyset$:

   Insert "Send:c:0" into $\underline{Seen}$.
   Store the message: $\underline{xc}[0] \leftarrow x$. Record the user: $\underline{U} \leftarrow pid_u$.
   Send $\langle$Send:c:0, $sid, x\rangle$ on network.

2. Receive $\langle$Send:c:$qid, sid, x\rangle$ on $\mathcal{U}$ (from the user $\underline{U}$),
   where $qid \in \mathbb{N}^*$,
   such that {"Send:c:qid"} $\cap$ $\underline{Seen} = \emptyset$,
   and {"Done:c:$(qid-1)$"} $\subset$ $\underline{Seen}$:

   Insert "Send:c:qid" into $\underline{Seen}$.
   Store the message: $\underline{xc}[qid] \leftarrow x$.
   Send $\langle$Send:c:$qid, sid, x\rangle$ on network.

3. Receive $\langle$Ready:c:$0, sid\rangle$ on $\mathcal{Q}$,
   such that {"Ready:c:0"} $\cap$ $\underline{Seen} = \emptyset$:

   Insert "Ready:c:0" into $\underline{Seen}$.
   Send $\langle$Ready:c:$0, sid\rangle$ on network.

4. Receive $\langle$Ready:c:$qid, sid\rangle$ on $\mathcal{Q}$,
   where $qid \in \mathbb{N}^*$,
   such that {"Ready:c:qid"} $\cap$ $\underline{Seen} = \emptyset$,
   and {"Deliver:c:$(qid-1)$"} $\subset$ $\underline{Seen}$:

   Insert "Ready:c:qid" into $\underline{Seen}$.
   Send $\langle$Ready:c:$qid, sid\rangle$ on network.

5. Receive $\langle$Done:c:$qid, sid\rangle$ on network,
   such that {"Done:c:qid"} $\cap$ $\underline{Seen} = \emptyset$
   and {"Send:c:qid", "Ready:c:qid"} $\subset$ $\underline{Seen}$:

   Insert "Done:c:qid" into $\underline{Seen}$.
   Send $\langle$Done:c:$qid, sid\rangle$ on $\mathcal{U}$ (to the user $\underline{U}$).

6. Receive $\langle$Deliver:c:$qid, sid, x\rangle$ on network,
   where $x = \underline{xc}[qid]$,
   such that {"Deliver:c:qid"} $\cap$ $\underline{Seen} = \emptyset$,
   and {"Send", "Ready"} $\subset$ $\underline{Seen}$:

   Insert "Deliver:c:qid" into $\underline{Seen}$.
   Send $\langle$Deliver:c:$qid, sid, \underline{xc}[qid]\rangle$ on $\mathcal{Q}$.

Message from server to user. For messages that the server sends to the user, $\mathcal{F}_{\text{osac}}$ proceeds similarly as for a multi-session $\mathcal{F}_{\text{ac}}$, i.e., there is no hijack. The server can start sending messages to the user only after it received the first message from the user.

7. Receive $\langle$Send:s:$0, sid, x\rangle$ on $\mathcal{Q}$,
   such that {"Send:s:0"} $\cap$ $\underline{Seen} = \emptyset$
   and {"Deliver:c:0"} $\subset$ $\underline{Seen}$:

Insert "Send:s:0" into $\underline{Seen}$.
Store the message: $\underline{xs}[0] \leftarrow x$.
Send $\langle$Send:s:0, $sid, x\rangle$ on network.

8. Receive $\langle$Send:s:$qid, sid, x\rangle$ on $\mathcal{Q}$,
   where $qid \in \mathbb{N}^*$,
   such that {"Send:s:qid"} $\cap$ $\underline{Seen} = \emptyset$,
   and {"Done:s:$(qid - 1)$"} $\subset$ $\underline{Seen}$:

   Insert "Send:s:qid" into $\underline{Seen}$.
   Store the message: $\underline{xs}[qid] \leftarrow x$.
   Send $\langle$Send:s:$qid, sid, x\rangle$ on network.

9. Receive $\langle$Ready:s:0, $sid\rangle$ on $\mathcal{U}$ (from the user $\underline{U}$),
   such that {"Ready:s:0"} $\cap$ $\underline{Seen} = \emptyset$
   and {"Done:c:0"} $\subset$ $\underline{Seen}$:

   Insert "Ready:s:0" into $\underline{Seen}$.
   Send $\langle$Ready:s:0, $sid\rangle$ on network.

10. Receive $\langle$Ready:s:$qid, sid\rangle$ on $\mathcal{U}$ (from the user $\underline{U}$),
    where $qid \in \mathbb{N}^*$,
    such that {"Ready:s:qid"} $\cap$ $\underline{Seen} = \emptyset$,
    and {"Deliver:s:$(qid - 1)$"} $\subset$ $\underline{Seen}$:

    Insert "Ready:s:qid" into $\underline{Seen}$.
    Send $\langle$Ready:s:$qid, sid\rangle$ on network.

11. Receive $\langle$Done:s:$qid, sid\rangle$ on network,
    such that {"Done:s:qid"} $\cap$ $\underline{Seen} = \emptyset$
    and {"Send:s:qid", "Ready:s:qid"} $\subset$ $\underline{Seen}$:

    Insert "Done:s:qid" into $\underline{Seen}$.
    Send $\langle$Done:s:$qid, sid\rangle$ on $\mathcal{Q}$.

12. Receive $\langle$Deliver:s:$qid, sid, x\rangle$ on network,
    where $x = \underline{xs}[qid]$,
    such that {"Deliver:s:qid"} $\cap$ $\underline{Seen} = \emptyset$,
    and {"Send", "Ready"} $\subset$ $\underline{Seen}$:

    Insert "Deliver:s:qid" into $\underline{Seen}$.
    Send $\langle$Deliver:s:$qid, sid, \underline{xs}[qid]\rangle$ on $\mathcal{U}$ (to user $\underline{U}$).

Corruption. $\mathcal{F}_{\text{osac}}$ reacts to corruption, reset, and hijack messages as follows.

13. Receive $\langle$Corrupt:$\mathcal{Q}, sid\rangle$ on $\mathcal{Q}$,
    such that {"Corrupt:$\mathcal{Q}$"} $\cap$ $\underline{Seen} = \emptyset$:

    Insert "Corrupt:$\mathcal{Q}$" into $\underline{Seen}$.
    Send $\langle$Corrupt:$\mathcal{Q}, sid\rangle$ on network.

14. Receive $\langle$Corrupt:$\mathcal{U}, sid\rangle$ on $\mathcal{U}$ (from the user $\underline{U}$),
    such that {"Corrupt:$\mathcal{U}$"} $\cap$ $\underline{Seen} = \emptyset$
    and {"Send:c:0"} $\subset$ $\underline{Seen}$:

    Insert "Corrupt:$\mathcal{U}$" into $\underline{Seen}$.
    Send $\langle$Corrupt:$\mathcal{U}, sid\rangle$ on network.

15. Receive $\langle$Reset:c:$qid, sid, x\rangle$ on network,
    where $qid \in \mathbb{N}$,
    such that {"Reset:c:qid"} $\cap$ $\underline{Seen} = \emptyset$,
    and {"Corrupt:$\mathcal{U}$"} $\subset$ $\underline{Seen}$:

    Insert "Reset:c:qid" into $\underline{Seen}$.
    Store a new message: $\underline{xc}[qid] \leftarrow x$.
    Send $\langle$Reset:c:$qid, sid\rangle$ on network.

16. Receive $\langle$Reset:s:$qid, sid, x\rangle$ on network,
    where $qid \in \mathbb{N}$,
    such that {"Reset:s:qid"} $\cap$ $\underline{Seen} = \emptyset$,
    and {"Corrupt:$\mathcal{Q}$"} $\subset$ $\underline{Seen}$:

    Insert "Reset:s:qid" into $\underline{Seen}$.
    Store a new message: $\underline{xs}[qid] \leftarrow x$.
    Send $\langle$Reset:s:$qid, sid\rangle$ on network.

17. Receive $\langle$Hijack, $sid, x\rangle$ on network,
    such that {"Hijack", "Done:c:0", "Deliver:c:0"} $\cap$ $\underline{Seen} = \emptyset$
    and {"Send:c:0"} $\subset$ $\underline{Seen}$:

    Insert "Hijack" into $\underline{Seen}$.
    Change $\underline{U}[qid] \leftarrow \mathcal{A}$. Store a new message: $\underline{xc}[qid] \leftarrow x$.
    Send $\langle$Hijack, $sid, x\rangle$ on network.

Realization. $\mathcal{F}_{\text{osac}}$ can be realized given a certificate authority, a CCA-2 secure public-key cryptosystem, an existentially unforgeable message authentication code (MAC), and insecure channels. We do not provide a realization here, as we consider it out of scope. A realization of $\mathcal{F}_{\text{osac}}$ might be reminiscent of the TLS protocol [**?**] with server certificates.

We note that the parties perform the following expensive operations: the client does one encryption using the CCA-2 cryptosystem, and the server one decryption.

B.4 Zero-Knowledge Proofs of Existence for One Verifier $\mathcal{F}_{\text{gzk}}$

The functionality $\mathcal{F}_{\text{gzk}}$ is a tool which allows one to simplify the security proof of protocols which use zero-knowledge proofs of *existence*. This functionality was proposed by Camenisch, Krenn, and Shoup [8]. The two major differences between $\mathcal{F}_{\text{gzk}}$ and the traditional functionality for zero-knowledge proofs of knowledge (e.g., the one defined by Hofheinz and Shoup [22] in the GNUC model) is that the former *1)* does not check its inputs and *2)* does not allow the adversary to extract the witnesses quantified by $\exists$.

One must be careful with this functionality, since it is not intended to be used like a regular ideal functionality. Indeed the functionality is quite useless by itself. However, by using the special composition theorem by Camenisch, Krenn, and Shoup, one can prove that if the $\mathcal{F}_{\text{gzk}}$-hybrid protocol is secure against a weak class of environments called *nice* environments, and the protocol is such that honest provers never try to prove incorrect statements, then the modified protocol in which all instances of $\mathcal{F}_{\text{gzk}}$ have been replaced by the zero-knowledge protocol π described by Camenisch, Krenn, and Shoup is secure in the UC-sense.

In the definition here, unlike the definition of Camenisch et al., the adversary learns the statement that was proven, i.e., authenticated (or one-sided–authenticated) channels are sufficient in the realization of π. We note that the special composition theorem of Camenisch et al. still holds despite this change and the translation to the GNUC framework.

$\mathcal{F}_{\text{gzk}}$ is designed to be used in a setting where adaptive corruption with erasures are allowed; and $\mathcal{F}_{\text{gzk}}$ is parameterized by a binary predicate $R : \big(x, (w_{\nrightarrow}, w_{\exists})\big) \mapsto \{0,1\}$, and a leakage function $\ell : (x, w_{\nrightarrow}) \mapsto \{0,1\}^*$.

Interfaces. $\mathcal{F}_{\text{gzk}}$ is a three-interface system:
- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{P}$-interface, connected to the ideal peer of the prover.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the verifier.

State. The ideal functionality is stateful and maintains the following data structures:
- $\underline{Seen}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{x}$: the statement that is to be proven. This can be used as the first argument to the binary predicate R.
- $\underline{w_{\nrightarrow}}$: the witnesses whose knowledge is proven. This and the witnesses whose existence is proven can be used as the second argument to the binary predicate R.

We model the single-session variant of $\mathcal{F}_{\text{gzk}}$, the session id $sid = \underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\text{gzk}}$ reacts to messages as follows.

1. Receive $\langle \text{Send}, sid, x, w_{\nrightarrow}, w_{\exists} \rangle$ on $\mathcal{P}$,
   such that $\{\text{“Send”}\} \cap \underline{Seen} = \emptyset$:

   Insert “Send” into $\underline{Seen}$.
   Store the instance and all witnesses quantified by $\nrightarrow$: $\underline{x} \leftarrow x$ and $\underline{w_{\nrightarrow}} \leftarrow w_{\nrightarrow}$.
   Send $\langle \text{Send}, sid, \ell(x, w_{\nrightarrow}) \rangle$ on network.

   The ideal functionality $\mathcal{F}_{\text{gzk}}$, being *gullible*, does not check if the predicate holds, i.e., if $R\big(x, (w_{\nrightarrow}, w_{\exists})\big) \stackrel{?}{=} 1$. Usually in an $\mathcal{F}_{\text{gzk}}$-hybrid protocol, the environment is restricted to being *nice*, and the honest parties never prove false statements, so $\mathcal{F}_{\text{gzk}}$ should never see false statements.

2. Receive $\langle$Ready, $sid\rangle$ on $\mathcal{Q}$,
   such that {"Ready"} $\cap$ $\underline{Seen} = \emptyset$:

   Insert "Ready" into $\underline{Seen}$.
   Send $\langle$Ready, $sid\rangle$ on network.

3. Receive $\langle$Lock, $sid\rangle$ on network,
   such that {"Lock"} $\cap$ $\underline{Seen} = \emptyset$
   and {"Send", "Ready"} $\subset$ $\underline{Seen}$:

   Insert "Lock" into $\underline{Seen}$.
   Send $\langle$Lock, $sid, \underline{x}\rangle$ on network.

4. Receive $\langle$Done, $sid\rangle$ on network,
   such that {"Done"} $\cap$ $\underline{Seen} = \emptyset$
   and {"Lock"} $\subset$ $\underline{Seen}$:

   Insert "Done" into $\underline{Seen}$.
   Send $\langle$Done, $sid\rangle$ on $\mathcal{P}$.

5. Receive $\langle$Deliver, $sid, \underline{x}, L\rangle$ on network,
   where $L = \ell(\underline{x}, \underline{w_{\triangleright}}) \vee$ "Corrupt:$\mathcal{Q}$" $\in \underline{Seen}$,
   such that {"Deliver"} $\cap$ $\underline{Seen} = \emptyset$,
   and {"Lock"} $\subset$ $\underline{Seen}$:

   Insert "Deliver" into $\underline{Seen}$.
   Send $\langle$Deliver, $sid, \underline{x}\rangle$ on $\mathcal{Q}$.

6. Receive $\langle$Corrupt:$\mathcal{R}$, $sid\rangle$ on $\mathcal{R}$,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
   such that {"Corrupt:$\mathcal{R}$"} $\cap$ $\underline{Seen} = \emptyset$:

   Insert "Corrupt:$\mathcal{R}$" into $\underline{Seen}$.
   Send $\langle$Corrupt:$\mathcal{R}$, $sid\rangle$ on network.

   In the GNUC model, for $\mathcal{R} = \mathcal{P}$, $\mathcal{F}_{\text{gzk}}$ also sends an invitation for the Expose message if applicable.

7. Receive $\langle$Reset, $sid, x, w_{\triangleright}, w_{\exists}\rangle$ on network,
   such that {"Reset", "Lock"} $\cap$ $\underline{Seen} = \emptyset$
   and {"Corrupt:$\mathcal{P}$"} $\subset$ $\underline{Seen}$:

   Insert "Reset" into $\underline{Seen}$.
   Store the instance and all witnesses quantified by $\triangleright$: $\underline{x} \leftarrow x$ and $\underline{w_{\triangleright}} \leftarrow w_{\triangleright}$.
   Send $\langle$Reset, $sid\rangle$ on network.

8. Receive $\langle$Expose, $sid\rangle$ on network,
   such that {"Expose", "Lock"} $\cap$ $\underline{Seen} = \emptyset$
   and {"Send", "Corrupt:$\mathcal{P}$"} $\subset$ $\underline{Seen}$:

   Insert "Expose" into $\underline{Seen}$.
   Send $\langle$Expose, $sid, \underline{x}, \underline{w_{\triangleright}}\rangle$ on network.

B.5 Zero-Knowledge Proofs of Existence for Two Verifiers $\mathcal{F}_{\text{gzk}}^{\text{2v}}$

Our functionality $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ is very similar to $\mathcal{F}_{\text{gzk}}$. It is intended to be used when the prover needs to simultaneously prove the same statement to two different verifiers (and erase the same values in both proofs). An honest execution of $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ is similar to running two sessions of $\mathcal{F}_{\text{gzk}}$ in parallel with the same statement and with two different verifiers, except that both sessions share a single Lock message. Of course, if the prover is corrupted, he can prove different statements to the two verifiers.

Interfaces. $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ is a four-interface system:
- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{U}$-interface, connected to the ideal peer of the prover.
- The $\mathcal{P}$-interface, connected to the ideal peer of the first verifier.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the second verifier.

State. The ideal functionality is stateful and maintains the following data structures:
- $\underline{Seen}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{x}^{\mathcal{P}}, \underline{x}^{\mathcal{Q}}$: the statement that is to be proven. This can be used as the first argument to the binary predicate R.
- $\underline{w_{\triangleright}}^{\mathcal{P}}, \underline{w_{\triangleright}}^{\mathcal{Q}}$: the witnesses whose knowledge is proven. This and the witnesses whose existence is proven can be used as the second argument to the binary predicate R.
  We model the single-session variant of $\mathcal{F}_{\text{gzk}}^{\text{2v}}$, the session id $sid = \underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ reacts to messages as follows.

1. Receive $\langle$Send:$\mathcal{R}$, sid, x, $w_{\triangleright}$, $w_{\exists}\rangle$ on $\mathcal{U}$,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
   such that $\{$"Send:$\mathcal{R}$"$\} \cap \underline{Seen} = \emptyset$:

   Insert "Send:$\mathcal{R}$" into $\underline{Seen}$.
   Store the instance and all witnesses quantified by $\triangleright$: $\underline{x}^{\mathcal{R}} \leftarrow x$ and $\underline{w_{\triangleright}}^{\mathcal{R}} \leftarrow w_{\triangleright}$.
   Send $\langle$Send:$\mathcal{R}$, sid, $\ell(x, w_{\triangleright})\rangle$ on network.

   The ideal functionality $\mathcal{F}_{\text{gzk}}^{\text{2v}}$, being *gullible*, does not check if the predicate holds, i.e., if $R\big(x, (w_{\triangleright}, w_{\exists})\big) \stackrel{?}{=} 1$. Usually in an $\mathcal{F}_{\text{gzk}}^{\text{2v}}$-hybrid protocol, the environment is restricted to being *nice*, and the honest parties never prove false statements, so $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ should never see false statements.

2. Receive $\langle$Ready:$\mathcal{R}$, $sid\rangle$ on $\mathcal{R}$,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
   such that $\{$"Ready:$\mathcal{R}$"$\} \cap \underline{Seen} = \emptyset$:

   Insert "Ready:$\mathcal{R}$" into $\underline{Seen}$.
   Send $\langle$Ready:$\mathcal{R}$, $sid\rangle$ on network.

3. Receive $\langle$Lock:$\mathcal{R}$, $sid\rangle$ on network,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\} \wedge \big((\{$"Send:$\mathcal{P}$", "Send:$\mathcal{Q}$", "Ready:$\mathcal{P}$", "Ready:$\mathcal{Q}$"$\} \subset \underline{Seen} \wedge \underline{x}^{\mathcal{P}} = \underline{x}^{\mathcal{Q}}) \vee$ ("Corrupt:$\mathcal{U}$" $\in \underline{Seen}$)$\big)$,
   such that $\{$"Lock:$\mathcal{R}$"$\} \cap \underline{Seen} = \emptyset$,
   and $\{$"Send:$\mathcal{R}$", "Ready:$\mathcal{R}$"$\} \subset \underline{Seen}$:

Insert "Lock:$\mathcal{R}$" into $\underline{Seen}$.
Send $\langle$Lock:$\mathcal{R}$, sid, $\underline{x}^{\mathcal{R}}\rangle$ on network.

When the user is honest, we make sure here that the two protocols are synchronized. This way, in the realization, the user can erase data in both protocols simultaneously.

4. Receive $\langle$Done:$\mathcal{R}$, $sid\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\} \wedge \big((\{\text{"Lock:}\mathcal{P}\text{"}, \text{"Lock:}\mathcal{Q}\text{"}\} \subset \underline{Seen}) \vee (\text{"Corrupt:}\mathcal{U}\text{"} \in \underline{Seen})\big)$,
such that $\{\text{"Done:}\mathcal{R}\text{"}\} \cap \underline{Seen} = \emptyset$,
and $\{\text{"Lock:}\mathcal{R}\text{"}\} \subset \underline{Seen}$:

   Insert "Done:$\mathcal{R}$" into $\underline{Seen}$.
   Send $\langle$Done:$\mathcal{R}$, $sid\rangle$ on $\mathcal{U}$.

5. Receive $\langle$Deliver:$\mathcal{R}$, sid, $\underline{x}^{\mathcal{R}}$, $L\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\} \wedge L = \ell(\underline{x}^{\mathcal{R}}, \underline{w}_{\nrightarrow}^{\mathcal{R}}) \vee$ "Corrupt:$\mathcal{R}$" $\in \underline{Seen} \wedge \big((\{\text{"Lock:}\mathcal{P}\text{"}, \text{"Lock:}\mathcal{Q}\text{"}\} \subset \underline{Seen}) \vee (\text{"Corrupt:}\mathcal{U}\text{"} \in \underline{Seen})\big)$,
such that $\{\text{"Deliver:}\mathcal{R}\text{"}\} \cap \underline{Seen} = \emptyset$,
and $\{\text{"Lock:}\mathcal{R}\text{"}\} \subset \underline{Seen}$:

   Insert "Deliver:$\mathcal{R}$" into $\underline{Seen}$.
   Send $\langle$Deliver:$\mathcal{R}$, sid, $\underline{x}^{\mathcal{R}}\rangle$ on $\mathcal{R}$.

6. Receive $\langle$Corrupt:$\mathcal{T}$, $sid\rangle$ on $\mathcal{T}$,
where $\mathcal{T} \in \{\mathcal{U}, \mathcal{P}, \mathcal{Q}\}$,
such that $\{\text{"Corrupt:}\mathcal{T}\text{"}\} \cap \underline{Seen} = \emptyset$:

   Insert "Corrupt:$\mathcal{T}$" into $\underline{Seen}$.
   Send $\langle$Corrupt:$\mathcal{T}$, $sid\rangle$ on network.

In the GNUC model, for $\mathcal{T} = \mathcal{U}$, $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ also sends an invitation for the Expose message if applicable.

7. Receive $\langle$Reset:$\mathcal{R}$, sid, x, $w_{\nrightarrow}$, $w_{\exists}\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
such that $\{\text{"Reset:}\mathcal{R}\text{"}, \text{"Lock"}\} \cap \underline{Seen} = \emptyset$,
and $\{\text{"Corrupt:}\mathcal{U}\text{"}\} \subset \underline{Seen}$:

   Insert "Reset:$\mathcal{R}$" into $\underline{Seen}$.
   Store the instance and all witnesses quantified by $\nrightarrow$: $\underline{x}^{\mathcal{R}} \leftarrow x$ and $\underline{w}_{\nrightarrow}^{\mathcal{R}} \leftarrow w_{\nrightarrow}$.
   Send $\langle$Reset:$\mathcal{R}$, $sid\rangle$ on network.

8. Receive $\langle$Expose:$\mathcal{R}$, $sid\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
such that $\{\text{"Expose:}\mathcal{R}\text{"}, \text{"Lock"}\} \cap \underline{Seen} = \emptyset$,
and $\{\text{"Send"}, \text{"Corrupt:}\mathcal{U}\text{"}\} \subset \underline{Seen}$:

   Insert "Expose:$\mathcal{R}$" into $\underline{Seen}$.
   Send $\langle$Expose:$\mathcal{R}$, sid, $\underline{x}^{\mathcal{R}}$, $\underline{w}_{\nrightarrow}^{\mathcal{R}}\rangle$ on network.

Realization. $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ is realized by running two independent instances of the π protocol by Camenisch, Krenn, and Shoup [8]—one instance with each verifier. However, the prover waits until he got a reply from both verifiers before erasing the witnesses and sending out the last message in each proof instance.

C Homomorphic Mixed Trapdoor Commitments

We now recall the definition and a construction of the homomorphic mixed trapdoor (HMT) commitment scheme [7], which we use in our protocol as a building block for constructing UC commitment schemes [**?**]. This scheme works well with proofs of *existence* using $\mathcal{F}_{\text{gzk}}$, resulting in an efficiency gain in the overall protocol. We adapt Camenisch et al.'s definition and construction of HMT commitments in a group of composite order [7] to work instead in a group $\mathbb{G}$ of prime order q (with generator g) where the decision Diffie-Hellman (DDH) problem is hard.

An HMT commitment scheme is a commitment scheme that is either perfectly hiding and equivocable or statistically binding, depending on the distribution of the CRS. When used with zero-knowledge proofs, HMT commitment schemes are similar to UC commitments based on Pedersen commitments [**?**] in that *1)* the simulator $\mathcal{S}$ can equivocate commitments in the security proof without being caught, even if he has to provide all randomness used to generate the commitment to the adversary; and *2)* $\mathcal{S}$ can use an adversary who equivocates commitments[6] to solve a hard cryptographic problem. However, unlike UC commitments based on Pedersen commitments, in HMT commitments *3)* $\mathcal{S}$ *does not always need to extract the openings or the committed values* from $\mathcal{F}_{\text{gzk}}$.

Definition. Let $crs_i \xleftarrow{\$} \text{CRSGen}_i(\mathbb{G}, q, g)$ for $i \in \{0, 1\}$ be two PPT algorithms that generate parameters for a commitment scheme. If $i = 0$, the commitment scheme is perfectly hiding (computationally binding) and, if $i = 1$, the commitment scheme is statistically binding (computationally hiding). For the perfect-hiding setting, let $(crs'_0, t) \xleftarrow{\$} \text{CRSGen}'_0(\mathbb{G}, q, g)$ be the function that additionally outputs a trapdoor t and such that crs_0 and crs'_0 have the same distribution. It is required that crs_0 and crs_1 are computationally indistinguishable.

Let $(c, o) \xleftarrow{\$} \text{Com}_{crs_i}(s)$ be the function that takes as input a value $s \in \mathbb{Z}_q$ to be committed, and outputs a commitment c and an opening $o \in \mathbb{Z}_q$ to the commitment. We will also use the notation $c \leftarrow \text{Com}_{crs_i}(s, o)$, where the opening is chosen outside the function. We denote the verification of a commitment by $c \stackrel{?}{=} \text{Com}_{crs_i}(s, o)$. The commitments are homomorphic with respect to addition over $\mathbb{Z}_q$: i.e., $c * c' = \text{Com}_{crs_i}(s + s', o + o')$. With the trapdoor t it is possible to efficiently equivocate commitments in the perfect-hiding setting: $\text{Com}_{crs_i}(s, o) = \text{Com}_{crs_i}(s', (s - s') * t + o)$.

In the sequel, we drop the subscript crs_i of Com if it is clear which parameters need to be used.

Construction. In the construction based on Elgamal, CRSGen_i runs as follows: select x, μ, and t at random from $\mathbb{Z}_q$; compute $h \leftarrow g^x$, $y \leftarrow g^{i*\mu} h^t$, and $w \leftarrow g^t$; and finally output $crs_i \leftarrow (h, y, w)$. The function CRSGen'_0 is like CRSGen_0 but additionally outputs t. Thus for $i = 1$, (y, w) is an Elgamal encryption of g^μ with respect to the public key (g, h), and for $i = 0$, (y, w) is an encryption of g^0. In practice, for $i = 1$ it is possible to randomly sample h, y, and w from $\mathbb{G}$ or to obtain (h, y, w) from $\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$.

The commitment function $\text{Com}_{crs_i}(s)$ is constructed as follows: select o at random from $\mathbb{Z}_q$ and compute $c \leftarrow (y^s h^o, w^s g^o)$. The commitment is a re-randomized encryption of $g^{i*\mu*s}$. Notice that the first element of c is a Pedersen commitment to s; we denote the extraction of that commitment pc as follows: $pc \leftarrow \text{PedC}(c) := y^s h^o$.

[6] As the commitment scheme is malleable, the protocol designer must take into account that the adversary might base his commitments on the simulator's commitments. Such problems can usually be avoided by requiring that for all *new* commitments, a proof of *knowledge* of the committed value is performed.

D Security Proof

In this section we prove that our protocol Π_{2pass} securely realizes the ideal functionality $\mathcal{F}_{\text{2pass}}$. We proceed as follows: we start by stating the main theorem and a number of lemmas, and then prove the main theorem. We then proceed to prove the main lemma in two steps: first, we describe the construction of a simulator $\mathcal{S}$, and then prove that $\mathcal{S}$ meets the requirements of the main lemma. Finally, we comment on multi-session realizations of $\mathcal{F}_{\text{2pass}}$ that use a constant-size CRS.

D.1 Security Proof

Recall that Π_{2pass} is a $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{gzk}}, \mathcal{F}_{\text{gzk}}^{\text{2v}}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}})$-hybrid protocol. Let $\Pi_{\text{2pass}}^{\pi/\mathcal{F}_{\text{gzk}}}$ be the $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{crs}}^{\text{gzk}}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}})$-hybrid protocol in which every instance of $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ in Π_{2pass} has been replaced by the zero-knowledge protocol π described in Camenisch, Krenn, and Shoup's paper [8]. To prove our scheme secure, we need to prove the following theorem:

Theorem 1. *There exists a simulator $\mathcal{S}$, such that for all polynomial-time-bounded environments $\mathcal{Z}$ and the dummy adversary $\mathcal{A}$:*

$$\text{Exec}(\Pi_{\text{2pass}}^{\pi/\mathcal{F}_{\text{gzk}}}, \mathcal{A}, \mathcal{Z}) \approx \text{Exec}(\mathcal{F}_{\text{2pass}}, \mathcal{S}, \mathcal{Z}).$$

In the theorem above, $\text{Exec}(\Pi_{\text{2pass}}^{\pi/\mathcal{F}_{\text{gzk}}}, \mathcal{A}, \mathcal{Z})$ denotes the binary random variable given by the output of $\mathcal{Z}$ when interacting with $\mathcal{A}$ and $\Pi_{\text{2pass}}^{\pi/\mathcal{F}_{\text{gzk}}}$ in the $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{crs}}^{\text{gzk}}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}})$-hybrid world, and where the randomness is taken over the random coins of $\mathcal{Z}$ and the random coins internal to $\Pi_{\text{2pass}}^{\pi/\mathcal{F}_{\text{gzk}}}$; and analogously for $\text{Exec}(\mathcal{F}_{\text{2pass}}, \mathcal{S}, \mathcal{Z})$ in the *ideal world*. The symbol $\approx$ means statistically close. The setting can be visualized in Figure 11.

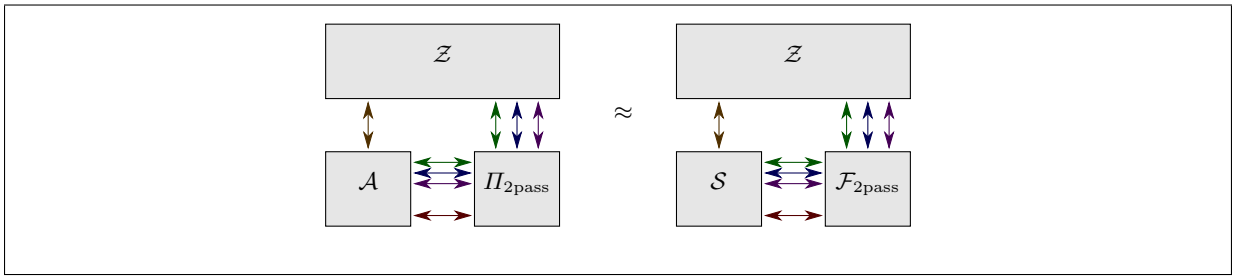


Fig. 11: A visualization of Theorem 1.

To prove the theorem, we need the following two definitions and lemmas:

Definition 1. *A* nice *environment is an environment that never asks $\mathcal{A}$ to submit a false statement to $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ [8].*

Definition 2. *A $\mathcal{F}_{\text{gzk}}$-*friendly *protocol is a protocol in which honest parties acting as provers only prove true statements with $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ [27].*

Lemma 1. *There exists a simulator $\mathcal{S}$ that does not extract the witnesses quantified by $\exists$ in any $\mathcal{F}_{\text{gzk}}$ and in any $\mathcal{F}_{\text{gzk}}^{\text{2v}}$, such that for all polynomial-time-bounded* nice *environments $\mathcal{Z}$ and the dummy adversary $\mathcal{A}$:*

$$\text{Exec}(\Pi_{\text{2pass}}, \mathcal{A}, \mathcal{Z}) \approx \text{Exec}(\mathcal{F}_{\text{2pass}}, \mathcal{S}, \mathcal{Z}).$$

Lemma 2. *Π_{2pass} is a $\mathcal{F}_{\text{gzk}}$-friendly protocol.*

Proof of Lemma 1. In §D.2 we construct a simulator $\mathcal{S}$, and in §D.3 we prove that is satisfies the requirements of Lemma 1.

Proof of Lemma 2. One can see by inspection that Π_{2pass} is a $\mathcal{F}_{\text{gzk}}$-friendly protocol, i.e., that honest parties only prove true statements with $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{\text{2v}}$.

Proof of Theorem 1. Since we prove in §D.2 that the simulator $\mathcal{S}$ we construct in §D.2 satisfies the requirements of Lemma 1 and because Lemma 2 holds, we apply the special composition theorem of Camenisch et al. [8, 27]. (Note: this composition theorem was not proven for $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ or for the GNUC model, but it is easy to adapt their proof to handle that case.)

Conclusion. From Theorem 1, we can conclude that the $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{crs}}^{\text{gzk}}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}})$-hybrid protocol $\Pi_{\text{2pass}}^{\pi/\mathcal{F}_{\text{gzk}}}$ is a secure realization of the ideal functionality $\mathcal{F}_{\text{2pass}}$, and is universally composable.

D.2 Construction of the Simulator

Notation and Modelling. We adopt the convention that the ideal functionalities in the $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}}, \mathcal{F}_{\text{gzk}}, \mathcal{F}_{\text{gzk}}^{\text{2v}})$-hybrid "real" world (and which are controlled by $\mathcal{S}$) are surrounded by quotes: "$\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$", "$\mathcal{F}_{\text{osac}}$", "$\mathcal{F}_{\text{ac}}$", "$\mathcal{F}_{\text{gzk}}$", "$\mathcal{F}_{\text{gzk}}^{\text{2v}}$". Note that $\mathcal{S}$ does not have to run these ideal functionalities honestly, it just needs to ensure that the messages it sends on their behalf are indistinguishable from an honest execution.

Simulator. The simulator $\mathcal{S}$ is an ten-interface system, with five external and five internal interfaces. We also use quotes to designate internal interfaces of $\mathcal{S}$. These interfaces are the $\mathcal{Z}$-, $\mathcal{A}$-, $\mathcal{P}$-, $\mathcal{Q}$-, and $\mathcal{U}$-interfaces on one hand, and the "$\mathcal{Z}$"-, "$\mathcal{A}$"-, "$\mathcal{P}$"-, "$\mathcal{Q}$"-, and "$\mathcal{U}$"-interfaces on the other hand. See Figure 12.

The simulator $\mathcal{S}$ runs one instance of the adversary $\mathcal{A}$ internally. $\mathcal{S}$ connects to the environment through its external $\mathcal{Z}$-interface. It communicates with $\mathcal{F}_{\text{2pass}}$ through four external interfaces: the $\mathcal{A}$-, the $\mathcal{P}$-, the $\mathcal{Q}$-, and the (multiplexed) $\mathcal{U}$-interfaces. The $\mathcal{A}$-interface is connected to the network interface of $\mathcal{F}_{\text{2pass}}$, it is through this interface that $\mathcal{S}$ sends the messages in the role of the ideal adversary to $\mathcal{F}_{\text{2pass}}$ and expects to receive the messages destined to the ideal adversary. The latter three interfaces are connected to the ideal peer of the respective party; such an interface becomes active only when the corresponding party is corrupted. The simulator interacts with $\mathcal{A}$ through its five internal interfaces: the "$\mathcal{Z}$"-, the "$\mathcal{A}$"-, the "$\mathcal{P}$"-, the "$\mathcal{Q}$"-, and the (multiplexed) "$\mathcal{U}$"-interfaces. Through the "$\mathcal{Z}$" interface, $\mathcal{S}$ must simulate the messages from the environment. Through the "$\mathcal{A}$" interface, the simulator must simulate all traffic between $\mathcal{A}$ and the network interface of "$\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$", "$\mathcal{F}_{\text{osac}}$", "$\mathcal{F}_{\text{ac}}$", "$\mathcal{F}_{\text{gzk}}$", and "$\mathcal{F}_{\text{gzk}}^{\text{2v}}$". Through the latter three interfaces, the simulator must simulate the ideal peers of the respective parties; similarly to above, such an interface becomes active only when the corresponding party is corrupted.

Ideal peers. Each ideal peer is a three-interface system. The IO-interface of the ideal peer is connected to the environment in the ideal world. $\mathcal{S}$ also simulates ideal peers for each of the ideal subroutines of Π_{2pass} for the sake of $\mathcal{A}$, the IO-interface of these ideal peers is then connected to Π_{2pass}. The subroutine-interface is connected to an ideal functionality. The network interface is connected to the adversary or the simulator. When the party corresponding to the ideal peer is honest, the ideal peer fowards all messages between the IO-interface to the subroutine-interface in both directions, i.e., the environment/protocol communicates directly with the ideal functionality. When the party is corrupted, the ideal peer forwards all messages from the IO-interface and the subroutine-interface to the network interface, and forwards all messages from the network interface to either the IO-interface or the subroutine-interface (we assume

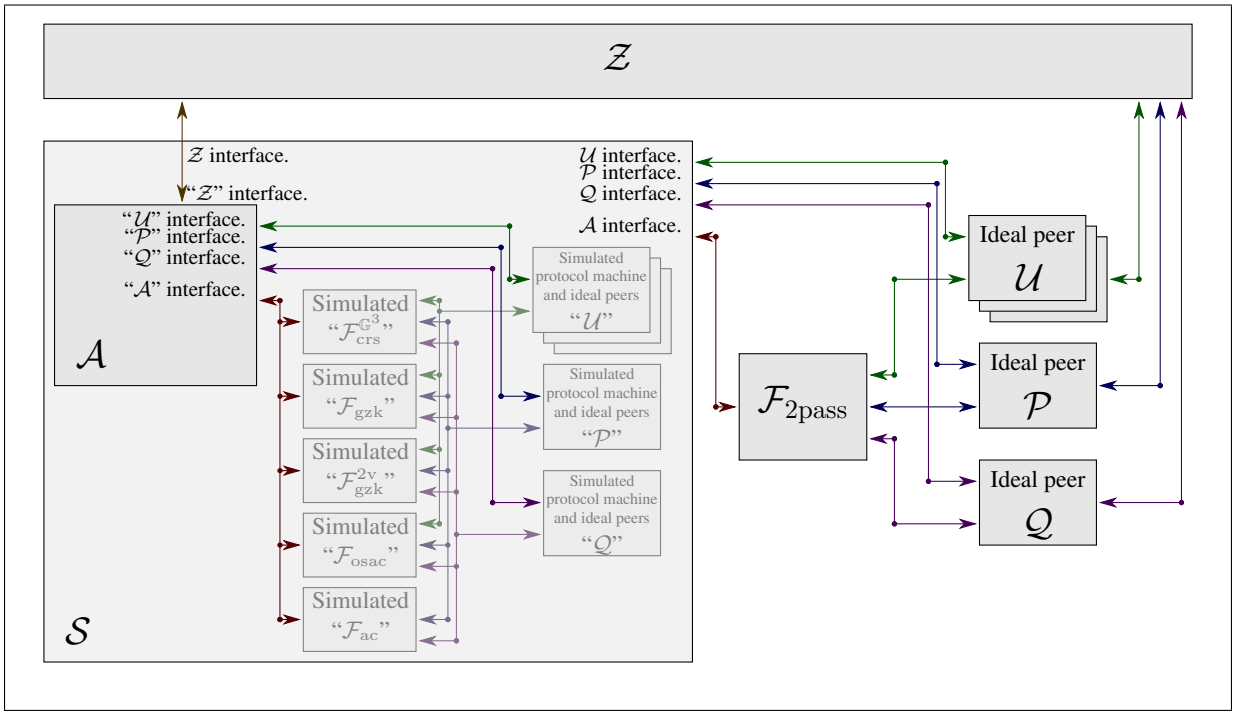


Fig. 12: The interfaces of the simulator $\mathcal{S}$.

that there is some sort of header that indicates where the message must be routed to); i.e., the adversary/simulator has direct access to the ideal functionality, learns the input of the ideal peer, and provides the output.

When an ideal peer receives a special $\langle \text{Corrupt}, \ldots \rangle$ message from the IO-interface, it forwards this message on the subroutine-interface and considers itself corrupted. When a corrupted ideal peer receives a special $\langle \text{Recover}, \ldots \rangle$ message from the network interface, it forwards this message on the subroutine-interface and considers itself formally recovered.

Protocol machines. A protocol machine is a multi-interface system. The IO-interface of the protocol machine is connected to the environment or another protocol machine. Each of zero or more subroutine-interfaces is connected to an ideal peer or a protocol machine. The network interface is connected to the adversary. Protocol machines excute the code of honest parties. When they receive a special corrupt message from the IO-interface they send a message containing their internal state on the network interface; thereafter they act as forwarders between the network interface and the IO- and subroutine-interfaces as for the ideal peers (the adversary is then supposed to send a corrupt message via that corrupted machine to all its subroutines). The adversary may request to change the internal state of corrupted protocol machines through the network interface in case machines recover from corruption. When they receive a special recovery message from the IO-interface, they stop forwarding traffic for the adversary and resume normal operation; however the protocol machines must use a new set of ideal peers and ideal functionalities in case the latter donot support recovery from corruption.

We now describe how to construct $\mathcal{S}$.

Environment interface. $\mathcal{S}$ forwards all messages between its $\mathcal{Z}$-interface and its "$\mathcal{Z}$"-interface in both directions, i.e., it relays all messages between $\mathcal{Z}$ and $\mathcal{A}$.

Party interfaces. When a party is honest, no messages are sent through the party interfaces. When a party becomes corrupted, and after $\mathcal{S}$ has handed the (simulated) internal state of that party to $\mathcal{Z}$, $\mathcal{S}$ relays all messages coming from the external interface (e.g., the $\mathcal{P}$-interface) to the internal corresponding interface (e.g., the "$\mathcal{P}$"-interface), and relays all messages from the internal interface destined for the environment to the external interface; this means that $\mathcal{A}$ receives the party's input and provides the party's output directly to $\mathcal{Z}$.

Common reference string. Upon the first query to "$\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$", $\mathcal{S}$ chooses a common reference string with CRSGen'_0, so that $\mathcal{S}$ knows the trapdoor Trap which will enable it to equivocate all commitments it makes on behalf of "$\mathcal{U}$", "$\mathcal{P}$" and "$\mathcal{Q}$".

General behavior of $\mathcal{S}$. In general, $\mathcal{S}$ simulates the ideal functionalities "$\mathcal{F}_{\text{ac}}$", "$\mathcal{F}_{\text{osac}}$", "$\mathcal{F}_{\text{gzk}}$", and "$\mathcal{F}_{\text{gzk}}^{2\text{v}}$" honestly, and simulates the ideal peers and protocol machines "$\mathcal{U}$", "$\mathcal{P}$", and "$\mathcal{Q}$" honestly. In fact, when $\mathcal{S}$ knows the correct input of the parties (which we can get either though the ideal peers in the ideal world or by extracting information from "$\mathcal{F}_{\text{gzk}}$"/"$\mathcal{F}_{\text{gzk}}^{2\text{v}}$"), it is easy to see how $\mathcal{S}$ proceeds. We emphasize that $\mathcal{S}$ never needs to send any input to "$\mathcal{F}_{\text{gzk}}$"/"$\mathcal{F}_{\text{gzk}}^{2\text{v}}$" on behalf of parties it controls, i.e., $\mathcal{S}$ can make proofs of false statements with "$\mathcal{F}_{\text{gzk}}$"/"$\mathcal{F}_{\text{gzk}}^{2\text{v}}$" or do proofs of knowledge even though it doesn't know the correct witnesses. In the remainder of this subsection, we will describe what $\mathcal{S}$ does when it doesn't know the correct inputs of the parties and is forced to lie.

Adjustements when $\mathcal{S}$ doesn't know the parties' input. When $\mathcal{S}$ does not know the input of some parties and must nevertheless produce output that depends on said input, $\mathcal{S}$ performs the following adjustments:

Setup.
- *Everybody honest:* $\mathcal{S}$ proceeds as if the user's input was random.
- *one server $\mathcal{R}$ corrupt ($\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$), others honest:* $\mathcal{S}$ proceeds as if the user's input was random.
- ($\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt,* $\mathcal{U}$ *honest:* $\mathcal{S}$ learns the user's input by sending $\langle \text{ExposeSetup}, \cdots \rangle$ to $\mathcal{F}_{\text{2pass}}$.)
- ($\mathcal{U}$ *corrupt, others honest:* $\mathcal{S}$ runs Setup queries honestly. $\mathcal{S}$ recovers $\mathcal{U}$'s input during Share.)
- ($\mathcal{U}$ *and one server $\mathcal{R}$ corrupt ($\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$), other server honest:* $\mathcal{S}$ runs Setup queries honestly. In Share, $\mathcal{S}$ recovers one of $\mathcal{U}$'s shares directly, the other through "$\mathcal{F}_{\text{gzk}}$".)
- ($\mathcal{U}$, $\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt:* The simulation is internal to the adversary.)

Retrieve. Upon receiving $\langle \text{Lock}, \cdots \rangle$ from $\mathcal{F}_{\text{2pass}}$, $\mathcal{S}$ knows whether $\delta = 0$ or not.
- *Everybody honest:* In general, $\mathcal{S}$ proceeds as if the user's input was random. $\mathcal{S}$ sends out random values $A_{\mathcal{U}}$, $B_{\mathcal{U}}$, $A_{\mathcal{P}}$, $B_{\mathcal{P}}$, and $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{Q}} = g^0$, otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{Q}}$.
- $\mathcal{P}$ *corrupt, others honest:* In general, $\mathcal{S}$ proceeds as if the user's input was random. $\mathcal{S}$ sends out random values $A_{\mathcal{U}}$, $B_{\mathcal{U}}$, and $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{Q}} = g^0$, otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{Q}}$.
- $\mathcal{Q}$ *corrupt, others honest:* In general, $\mathcal{S}$ proceeds as if the user's input was random. $\mathcal{S}$ sends out random values $A_{\mathcal{U}}$, $B_{\mathcal{U}}$, $A_{\mathcal{P}}$, and $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{P}} = (A_{\mathcal{P}})^{s_{\mathcal{U}\mathcal{Q}} + s_{\mathcal{P}\mathcal{Q}} - op_{\mathcal{Q}}}$ on behalf of $\mathcal{P}$ (at this point, $\mathcal{S}$ knows the value of $op_{\mathcal{Q}}$ through "$\mathcal{F}_{\text{gzk}}[\cdots, 2]$" from Share in the Setup query or through "$\mathcal{F}_{\text{gzk}}[\cdots, 9]$" from ComRefr in the Refresh query), otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{P}}$.
- $\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt,* $\mathcal{U}$ *honest:* Upon receiving $\langle \text{Lock}, \cdots \rangle$ from $\mathcal{F}_{\text{2pass}}$, $\mathcal{S}$ knows whether $\delta = 0$ or not. $\mathcal{S}$ sends out a random value $A_{\mathcal{U}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{U}} = (A_{\mathcal{U}})^{s_{\mathcal{U}\mathcal{P}} + s_{\mathcal{U}\mathcal{Q}} - op_{\mathcal{P}} - op_{\mathcal{Q}}}$ on behalf of $\mathcal{U}$ (at this point, $\mathcal{S}$ knows the value of $op_{\mathcal{P}}$ and $op_{\mathcal{Q}}$ because of the earlier "$\mathcal{F}_{\text{gzk}}[\cdots, 3]$" and "$\mathcal{F}_{\text{gzk}}[\cdots, 4]$" in ChkPwd), otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{U}}$.
- $\mathcal{U}$ *corrupt, others honest:* $\mathcal{S}$ sends out random values $A_{\mathcal{P}}$, $B_{\mathcal{P}}$, and $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{Q}} = g^0$, otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{Q}}$.

- $\mathcal{U}$ *and* $\mathcal{P}$ *corrupt,* $\mathcal{Q}$ *honest:* $\mathcal{S}$ sends out a random value $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{Q}} = g^0$, otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{Q}}$.
- $\mathcal{U}$ *and* $\mathcal{Q}$ *corrupt,* $\mathcal{P}$ *honest:* $\mathcal{S}$ sends out a random value $A_{\mathcal{P}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{P}} = (A_{\mathcal{P}})^{s_{\mathcal{U}\mathcal{Q}} + s_{\mathcal{P}\mathcal{Q}} - op_{\mathcal{Q}}}$ (at this point, $\mathcal{S}$ knows the value of $op_{\mathcal{Q}}$ through "$\mathcal{F}_{\text{gzk}}[\cdots, 2]$" from Share in the Setup query or through "$\mathcal{F}_{\text{gzk}}[\cdots, 9]$" from ComRefr in the Refresh query), otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{P}}$.
- ($\mathcal{U}$, $\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt:* The simulation is internal to the adversary.)

Refresh. $\mathcal{S}$ can run Refresh queries honestly, even if it doesn't know the correct value of the shares of the servers.

Adjustements upon corruption of $\mathcal{U}$. When a user $\mathcal{U}$ gets corrupted, $\mathcal{S}$ needs to perform the following adjustments, depending at which point in the simulation the corruption happened.

Share *after 1st Send of "$\mathcal{F}_{\text{osac}}$".* No adjustments needed, since most of the state was erased.

ChkPwd *after Lock of "$\mathcal{F}^{2\text{v}}_{\text{gzk}}[\cdots, 5]$".* No adjustments needed, since most of the state was erased.

Reconstr *after 1st Deliver of "$\mathcal{F}_{\text{osac}}$".* If not done already, immediately send $\langle \text{Deliver}, \cdots \rangle$ on the $\mathcal{U}$ interface to recover output.
- $\mathcal{P}$ *and* $\mathcal{Q}$ *honest:* adjust $k_{\mathcal{Q}}$ to match output of $\mathcal{U}$. $\mathcal{S}$ will need to adjust $ok_{\mathcal{Q}}$ (with help of trapdoor), and the OTP used to transmit it.
- $\mathcal{P}$ *corrupt,* $\mathcal{Q}$ *honest:* adjust $k_{\mathcal{Q}}$ to match output of $\mathcal{U}$. $\mathcal{S}$ will need to adjust $ok_{\mathcal{Q}}$ (with trapdoor), and the OTP used to transmit it.
- $\mathcal{P}$ *honest,* $\mathcal{Q}$ *corrupt:* adjust $k_{\mathcal{P}}$ to match output of $\mathcal{U}$. $\mathcal{S}$ will need to adjust $ok_{\mathcal{P}}$ (with trapdoor), and the OTP used to transmit it.
- $\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt:* there is nothing to adjust.

We note that the values $k_{\mathcal{P}}$, $k_{\mathcal{Q}}$, $ok_{\mathcal{P}}$, and $ok_{\mathcal{Q}}$ are fixed after the corruption of the first user in the Retrieve query that succeeded in retrieving the key. When re-adjusting these values after a subsequent corruption of a user in the Retrieve query, the values will not change again.

Adjustments upon corruption of $\mathcal{P}$. When $\mathcal{P}$ gets corrupted, $\mathcal{S}$ needs to perform the following adjustments, depending at which point in the simulation the corruption happened.

Share *after Deliver of "$\mathcal{F}_{\text{osac}}$".*
- $\mathcal{U}$ *was honest during setup,* $\mathcal{U}$ *was always honest during retrieve,* $\mathcal{Q}$ *honest:* No adjustments needed.
- $\mathcal{U}$ *was honest during setup,* $\mathcal{U}$ *was always honest during retrieve,* $\mathcal{Q}$ *corrupt:* Adjust $p_{\mathcal{P}}$ and $k_{\mathcal{P}}$ to match $\mathcal{U}$'s input. $\mathcal{S}$ also needs to adjust $op_{\mathcal{P}}$ and $ok_{\mathcal{P}}$ (with help of trapdoor) and the OTPs used to transmit those.
- $\mathcal{U}$ *was honest during setup,* $\mathcal{U}$ *was corrupted at least once during retrieve,* $\mathcal{Q}$ *honest:* $\mathcal{S}$ needs to adjust the OTP used to transmit the shares of the key. The values $k_{\mathcal{P}}$ and $ok_{\mathcal{P}}$ were already adjusted when $\mathcal{U}$ was corrupted during Retrieve.
- $\mathcal{U}$ *was honest during setup,* $\mathcal{U}$ *was corrupted at least once during retrieve,* $\mathcal{Q}$ *corrupt:* Adjust $p_{\mathcal{P}}$ to match $\mathcal{U}$'s input. The values $k_{\mathcal{P}}$ and $ok_{\mathcal{P}}$ were already adjusted when $\mathcal{U}$ was corrupted during Retrieve. $\mathcal{S}$ also needs to adjust $op_{\mathcal{P}}$ (with help of trapdoor) and the OTPs used to transmit messages between Alice and Bob.
- $\mathcal{U}$ *was corrupt during setup:* No adjustments needed.

ChkPwd *after Deliver of "$\mathcal{F}^{2\text{v}}_{\text{gzk}}[\cdots, 5]$".*
- $\mathcal{U}$ *and* $\mathcal{Q}$ *honest:* Nothing to adjust.
- $\mathcal{U}$ *corrupted:* Nothing to adjust.

– *$\mathcal{U}$ honest, $\mathcal{Q}$ corrupted:* If $\delta = 0$: adjust $s_{\mathcal{UP}}$ so that $B_{\mathcal{U}} = (A_{\mathcal{U}})^{s_{\mathcal{UP}}+s_{\mathcal{UQ}}-op_{\mathcal{P}}-op_{\mathcal{Q}}}$. $\mathcal{S}$ also needs to adjust $os_{\mathcal{UP}}$ (with trapdoor) and the OTP used to transmit it.

ChkPwd after Deliver of "$\mathcal{F}_{\text{gzk}}[\cdots, 6]$".
– *$\mathcal{Q}$ honest:* Nothing to adjust, since parts of the state were erased.
– *$\mathcal{Q}$ corrupted:* Nothing to adjust.

Reconstr after Send of "$\mathcal{F}_{\text{osac}}$".
– *$\mathcal{U}$ and $\mathcal{Q}$ honest:* Nothing to adjust.
– *$\mathcal{U}$ corrupted:* Nothing to adjust.
– *$\mathcal{U}$ honest, $\mathcal{Q}$ corrupted:* Adjust $k_{\mathcal{P}}$ to match correct output of $\mathcal{U}$. $\mathcal{S}$ will need to adjust $ok_{\mathcal{P}}$ (with trapdoor) and the OTP used to transmit it.

Adjustments upon corruption of $\mathcal{Q}$. When $\mathcal{Q}$ gets corrupted, $\mathcal{S}$ needs to perform the following adjustments, depending at which point in the simulation the corruption happened.

Share after Deliver of "$\mathcal{F}_{\text{osac}}$". Similar as for $\mathcal{P}$.

ChkPwd after Deliver of "$\mathcal{F}^{2\text{v}}_{\text{gzk}}[\cdots, 5]$".
– *$\mathcal{U}$ honest, $\mathcal{P}$ corrupted:* If $\delta = 0$: adjust $s_{\mathcal{UQ}}$ so that $B_{\mathcal{U}} = (A_{\mathcal{U}})^{s_{\mathcal{UP}}+s_{\mathcal{UQ}}-op_{\mathcal{P}}-op_{\mathcal{Q}}}$. $\mathcal{S}$ also needs to adjust $os_{\mathcal{UQ}}$ (with trapdoor) and the OTP used to transmit it.
– *Other cases:* Nothing to adjust.

ChkPwd after Deliver of "$\mathcal{F}_{\text{gzk}}[\cdots, 6]$".
– *$\mathcal{P}$ honest:* Nothing to adjust.
– *$\mathcal{P}$ corrupted:* If $\delta = 0$: adjust $s_{\mathcal{PQ}}$ so that $B_{\mathcal{P}} = (A_{\mathcal{P}})^{s_{\mathcal{UQ}}+s_{\mathcal{PQ}}-op_{\mathcal{Q}}}$. $\mathcal{S}$ also needs to adjust $os_{\mathcal{PQ}}$ (with trapdoor) and the OTP used to transmit it.

Reconstr after Send of "$\mathcal{F}_{\text{osac}}$". Similar as for $\mathcal{P}$.

D.3 Proof of Indistinguishability

In this subsection, we prove that the ideal world with $\mathcal{S}$ as defined above and the real world with an arbitrary adversary are indistinguishable.

We are going to define a sequence of games Game_1 to Game_8, as described by Shoup [**?**]. In the first game, everything is distributed as in the protocol Π_{2pass}, whereas in the last game everything is distributed as in the ideal world $\mathcal{F}_{\text{2pass}}$. By the piling-up lemma, the advantage of the environment $\mathcal{Z}$ in distinguishing between the first and the last game is less than the sum of the advantages in distinguishing consecutive games Game_i and Game_{i+1}. We are going to prove that $\mathcal{Z}$ has only a negligible advantage in the latter, based either on a reduction to a hard cryptographic problem, or by *failure events* happening with negligible probability. As the number of games we consider is polynomial w.r.t. the security parameter, the overall advantage of $\mathcal{Z}$ in distinguishing between the real world and ideal world setting is negligible.

We must stress that in all intermediate games, the simulator $\mathcal{S}_i$ receives the input of all honest parties. We only require that the simulator of the last game does not make use of these inputs, so that it may also be used in the ideal world setting.

As we mentioned earlier, we also note that $\mathcal{S}_i$ is not allowed to extract the witnesses whose *existence* is proven (i.e., witnesses quantified by $\exists$) from "$\mathcal{F}_{\text{gzk}}$"/"$\mathcal{F}^{2\text{v}}_{\text{gzk}}$".

Game_1. As observed in the previous paragraph, $\mathcal{S}_1$ receives the input of all honest parties. $\mathcal{S}_1$ runs all parties honestly, and runs "$\mathcal{F}^{\mathbb{G}^3}_{\text{crs}}$", "$\mathcal{F}_{\text{osac}}$", "$\mathcal{F}_{\text{ac}}$", "$\mathcal{F}_{\text{gzk}}$", and "$\mathcal{F}^{2\text{v}}_{\text{gzk}}$" honestly. By construction, this setting is perfectly indistinguishable from the ($\mathcal{F}^{\mathbb{G}^3}_{\text{crs}}$, $\mathcal{F}_{\text{osac}}$, $\mathcal{F}_{\text{ac}}$, $\mathcal{F}_{\text{gzk}}$, $\mathcal{F}^{2\text{v}}_{\text{gzk}}$)-hybrid *real* world Π_{2pass}.

Game$_2$. $\mathcal{S}_2$ runs like $\mathcal{S}_1$, except that it aborts if the adversary manages to open a commitment to two different values. $\mathcal{S}_2$ detects that case when:

- A corrupt $\mathcal{P}$ uses a different share $p_\mathcal{P}$ or $k_\mathcal{P}$ in any "$\mathcal{F}_{\text{gzk}}$" with a honest party, than the value it received from honest $\mathcal{U}$.
- A corrupt $\mathcal{P}$ uses a different values of shares $p_\mathcal{P}$ or $k_\mathcal{P}$ in subsequent runs of "$\mathcal{F}_{\text{gzk}}$".
- A corrupt $\mathcal{Q}$ uses a different share $p_\mathcal{Q}$ or $k_\mathcal{Q}$ in any "$\mathcal{F}_{\text{gzk}}$" with a honest party, than the value it received from honest $\mathcal{U}$.
- A corrupt $\mathcal{Q}$ uses a different values of shares $p_\mathcal{Q}$ or $k_\mathcal{Q}$ in subsequent runs of "$\mathcal{F}_{\text{gzk}}$".
- A corrupt $\mathcal{U}$ sends a value $B_\mathcal{U}$ in "$\mathcal{F}_{\text{gzk}}^{\text{2v}}[\cdots, 5]$" to an honest $\mathcal{P}$ or $\mathcal{Q}$ that is incompatible with $s_{\mathcal{UP}}$ or $s_{\mathcal{UQ}}$, respectively. (Here $\mathcal{S}_2$ doesn't need to extract those values, it can simply decrypt $B_\mathcal{U}$ and see if δ is equal or not equal to zero as expected.)
- A corrupt $\mathcal{P}$ sends a value $B_\mathcal{P}$ to honest $\mathcal{Q}$ in "$\mathcal{F}_{\text{gzk}}[\cdots, 6]$" that is incompatible with $s_{\mathcal{UP}}$ or $s_{\mathcal{PQ}}$ or $p_\mathcal{P}$. (Here $\mathcal{S}_2$ doesn't need to extract those values, it can simply decrypt $B_\mathcal{P}$ and see if δ is equal or not equal to zero as expected.)
- A corrupt $\mathcal{Q}$ sends a value $B_\mathcal{Q}$ to an honest $\mathcal{P}$ in "$\mathcal{F}_{\text{gzk}}[\cdots, 7]$" that is incompatible with $s_{\mathcal{UQ}}$ or $s_{\mathcal{PQ}}$ or $p_\mathcal{Q}$. (Here $\mathcal{S}_2$ doesn't need to extract those values, it can simply see if δ is equal or not equal to zero as expected.)

The probability that $\mathcal{S}_2$ aborts is at most the probability that the commitment was not binding after all (recall that the ideal functionalities $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ provide perfect soundness), which is negligible.

Game$_3$. $\mathcal{S}_3$ runs like $\mathcal{S}_2$, except that when secureSend is run between two honest parties, the parties use a different one-time-pad than the one that was encrypted in $e_\mathcal{T}$. Recall that both honest parties have deleted the randomness and decryption key for that ciphertext by the time the one-time-pad is first put into use.

The advantage of $\mathcal{Z}$ in distinguishing between Game$_3$ and Game$_2$ is at most the advantage of a polynomial-time environment in the CCA-2 security game of Enc, which is negligible.

Game$_4$. $\mathcal{S}_4$ runs like $\mathcal{S}_3$, except that when ChkPwd is run by an honest $\mathcal{U}$ and whenever $\delta \neq 0$, $\mathcal{S}_4$ chooses $A_\mathcal{U}$ and $B_\mathcal{U}$ at random from $\mathbb{G}$. $\mathcal{S}_4$ will make proofs of false statements with $\mathcal{F}_{\text{gzk}}^{\text{2v}}$.

We now argue that the advantage that $\mathcal{Z}$ has in distinguishing between Game$_4$ and Game$_3$ is negligible under the DDH assumption. We do this by running a hybrid arguments over all retrieve queries.

In the hybrid j, the simulator $\mathcal{S}_{3,j}$ behaves like $\mathcal{S}_4$ for the first j queries, and like $\mathcal{S}_3$ for the following queries.

We now construct a distinguisher $\mathcal{S}$ that operates with an environment which tries to distinguish between hybrid $j-1$ and j: the simulator $\mathcal{S}$ gets a tuple $(h, y, Y, Z) \in \mathbb{G}^4$ where either:

- all four elements are randomly sampled (*left* setting); or
- the four elements form a DDH tuple (*right* setting), i.e. $Y = h^{r_\mathcal{U}}$ and $Z = y^{r_\mathcal{U}}$.

$\mathcal{S}$ chooses w at random, and sets the CRS to (h, y, w). In the first $j-1$ Retrieve queries, $\mathcal{S}$ behaves like $\mathcal{S}_4$. In queries $j+1$ and following, $\mathcal{S}$ behaves like $\mathcal{S}_3$. In jth query, whenever $\delta \neq 0$ and whenever $\mathcal{U}$ is honest, $\mathcal{S}$ recovers $op_\mathcal{P}$ from "$\mathcal{F}_{\text{gzk}}[\cdots, 3]$" if needed, recovers $op_\mathcal{Q}$ from "$\mathcal{F}_{\text{gzk}}[\cdots, 4]$" if needed, sets $A_\mathcal{U} \leftarrow Y^{-1}$, sets $B_\mathcal{U} \leftarrow Z^\delta (A_\mathcal{U})^{s_{\mathcal{UP}}+s_{\mathcal{UQ}}-op_\mathcal{P}-op_\mathcal{Q}}$, and makes a proof of a false statement in the subsequent $\mathcal{F}_{\text{gzk}}$. $\mathcal{S}$ then outputs whatever the environment outputs. Notice how in the *left* setting, the distribution of all values is like in hybrid j, and in the *right* setting, the distribution of all values is like in the hybrid $j-1$. Also note that since $r_\mathcal{U}$ is erased before the last message of "$\mathcal{F}_{\text{gzk}}^{\text{2v}}[\cdots, 5]$", $\mathcal{S}_4$ will not get into trouble if $\mathcal{U}$ is corrupted.

The advantage of $\mathcal{S}$ in the DDH-game is therefore equal or better than the advantage of the environment in distinguishing between the two hybrids. The former being negligible by assumption, the

latter must also be negligible. Since the number of queries is polynomial, the overall advantage of the environment in distinguishing between Game_4 and Game_3 is negligible.

Game_5. $\mathcal{S}_5$ runs like $\mathcal{S}_4$, except that when ChkPwd is run by an honest $\mathcal{P}$ and whenever $\delta \neq 0$, $\mathcal{S}_5$ chooses $A_\mathcal{P}$ and $B_\mathcal{P}$ at random from $\mathbb{G}$. $\mathcal{S}_5$ will make proofs of false statements with $\mathcal{F}_{\text{gzk}}$.

We now argue that the advantage that $\mathcal{Z}$ has in distinguishing between Game_5 and Game_4 is negligible under the DDH assumption. We do this by running a hybrid arguments over all retrieve queries.

In the hybrid j, the simulator $\mathcal{S}_{4,j}$ behaves like $\mathcal{S}_5$ for the first j queries, and like $\mathcal{S}_4$ for the following queries.

We now construct a distinguisher $\mathcal{S}$ that operates with an environment which tries to distinguish between hybrid $j-1$ and j: the simulator $\mathcal{S}$ gets a tuple $(h, y, Y, Z) \in \mathbb{G}^4$ where either:

- all four elements are randomly sampled (*left* setting); or
- the four elements form a DDH tuple (*right* setting), i.e. $Y = h^{r_\mathcal{U} r_\mathcal{P}}$ and $Z = y^{r_\mathcal{U} r_\mathcal{P}}$.

$\mathcal{S}$ chooses w at random, and sets the CRS to (h, y, w). In the first $j-1$ Retrieve queries, $\mathcal{S}$ behaves like $\mathcal{S}_5$. In queries $j+1$ and following, $\mathcal{S}$ behaves like $\mathcal{S}_4$. In jth query, whenever $\delta \neq 0$ and whenever $\mathcal{P}$ is honest, $\mathcal{S}$ recovers $s_{\mathcal{UQ}}$ from "$\mathcal{F}^{2\text{v}}_{\text{gzk}}[\cdots, 5]$" if needed, ($op_\mathcal{Q}$ was recovered in "$\mathcal{F}_{\text{gzk}}[\cdots, 2]$" in Share during Setup or in "$\mathcal{F}_{\text{gzk}}[\cdots, 9]$" in ComRefr during Refresh) sets $A_\mathcal{P} \leftarrow Y^{-1}$, sets $B_\mathcal{P} \leftarrow Z^\delta (A_\mathcal{P})^{s_{\mathcal{PQ}} + s_{\mathcal{UQ}} - op_\mathcal{Q}}$, and makes a proof of a false statement in the subsequent $\mathcal{F}_{\text{gzk}}$. $\mathcal{S}$ then outputs whatever the environment outputs. Notice how in the *left* setting, the distribution of all values is like in hybrid j, and in the *right* setting, the distribution of all values is like in the hybrid $j-1$. Also note that since $r_\mathcal{P}$ is erased before the last message of "$\mathcal{F}_{\text{gzk}}[\cdots, 6]$", $\mathcal{S}_5$ will not get into trouble if $\mathcal{P}$ is corrupted.

The advantage of $\mathcal{S}$ in the DDH-game is therefore equal or better than the advantage of the environment in distinguishing between the two hybrids. The former being negligible by assumption, the latter must also be negligible. Since the number of queries is polynomial, the overall advantage of the environment in distinguishing between Game_5 and Game_4 is negligible.

Game_6. $\mathcal{S}_6$ runs like $\mathcal{S}_5$, except that when ChkPwd is run by an honest $\mathcal{Q}$ and whenever $\delta \neq 0$, $\mathcal{S}_6$ chooses $A_\mathcal{Q}$ and $B_\mathcal{Q}$ at random from $\mathbb{G}$. $\mathcal{S}_6$ will make proofs of false statements with $\mathcal{F}_{\text{gzk}}$.

We now argue that the advantage that $\mathcal{Z}$ has in distinguishing between Game_6 and Game_5 is negligible under the DDH assumption. We do this by running a hybrid arguments over all retrieve queries.

In the hybrid j, the simulator $\mathcal{S}_{5,j}$ behaves like $\mathcal{S}_6$ for the first j queries, and like $\mathcal{S}_5$ for the following queries.

We now construct a distinguisher $\mathcal{S}$ that operates with an environment which tries to distinguish between hybrid $j-1$ and j: the simulator $\mathcal{S}$ gets a tuple $(h, y, Y, Z) \in \mathbb{G}^4$ where either:

- all four elements are randomly sampled (*left* setting); or
- the four elements form a DDH tuple (*right* setting), i.e. $Y = h^{r_\mathcal{U} r_\mathcal{P} r_\mathcal{Q}}$ and $Z = y^{r_\mathcal{U} r_\mathcal{P} r_\mathcal{Q}}$.

$\mathcal{S}$ chooses w at random, and sets the CRS to (h, y, w). In the first $j-1$ Retrieve queries, $\mathcal{S}$ behaves like $\mathcal{S}_6$. In queries $j+1$ and following, $\mathcal{S}$ behaves like $\mathcal{S}_5$. In jth query, whenever $\delta \neq 0$ and whenever $\mathcal{Q}$ is honest, $\mathcal{S}$ sets $A_\mathcal{Q} \leftarrow Y^{-1}$, sets $B_\mathcal{Q} \leftarrow Z^\delta$, and makes a proof of a false statement in the subsequent $\mathcal{F}_{\text{gzk}}$. $\mathcal{S}$ then outputs whatever the environment outputs. Notice how in the *left* setting, the distribution of all values is like in hybrid j, and in the *right* setting, the distribution of all values is like in the hybrid $j-1$. Also note that since $r_\mathcal{Q}$ is erased before the last message of "$\mathcal{F}_{\text{gzk}}[\cdots, 7]$", $\mathcal{S}_6$ will not get into trouble if $\mathcal{Q}$ is corrupted.

The advantage of $\mathcal{S}$ in the DDH-game is therefore equal or better than the advantage of the environment in distinguishing between the two hybrids. The former being negligible by assumption, the latter must also be negligible. Since the number of queries is polynomial, the overall advantage of the environment in distinguishing between Game_6 and Game_5 is negligible.

Game$_7$. $\mathcal{S}_7$ runs like $\mathcal{S}_6$, except that now it chooses the CRS with CRSGen'_0 instead of CRSGen_1 upon the first query to "$\mathcal{F}_{\mathrm{crs}}^{\mathbb{G}^3}$". The commitment scheme is now perfectly hiding, and $\mathcal{S}_7$ can now efficiently equivocate commitments using the trapdoor information.

The advantage that $\mathcal{Z}$ has in distinguishing between Game$_7$ and Game$_6$ is equal to its advantage in breaking the semantic security game of Elgamal encryption, which is negligible by assumption.

Game$_8$. $\mathcal{S}_8$ runs like $\mathcal{S}$ described in §D.2.

This is a purely conceptual change, so $\mathcal{Z}$ has no advantage in distinguishing between Game$_8$ and Game$_7$.

qed.

D.4 Multi-Session Realization with Constant-Size CRS

If one wants to support multiple user accounts with our system, one can simply run multiple copies of Π_{2pass}. However, in that case each instance of Π_{2pass} must run with a different instance of the CRS. In this subsection we argue that running multiple copies of Π_{2pass} with the *same* CRS, which we will write as $\widehat{\Pi_{\mathrm{2pass}}}$, is actually secure as well, i.e., $\widehat{\Pi_{\mathrm{2pass}}}$ realizes $\widehat{\mathcal{F}_{\mathrm{2pass}}}$, the multi-session version of $\mathcal{F}_{\mathrm{2pass}}$.

It's easy to see that most steps of the proof in the previous subsection carry over to the multi-session case as well: the simulator simply has to run simulations for all the sessions in parallel. The proof of indistinguishability is similar as well: the hop between Game 6 and 7 remains the same; for all other game hops, we can use a hybrid argument over each session to prove indistinguishability between consecutive games.

We note that even though the plain HMT commitments are malleable, a party never accepts a commitment unless it has verified a zero-knowledge proof that the committer knows the value that was committed to; the adversary can thus not base any of its commitments on commitments made by the simulator without causing the simulation to abort.

Finally we note that the session id never appears explicitly in the proof in the previous section. This is actually not a problem in the multi-session case: the session id appears implicitly as part of the sid of ideal functionalities used as subroutines; there is thus no danger of confusing sessions.

E Comparison with Related Work

In this section, we compare the ideal functionalities, the protocol constructions, and the runtime of our protocol against those by Camenisch, Lysyanskaya, and Neven (CLN) [10] and by Camenisch, Lehmann, Lysyanskaya, and Neven (CLLN) [9]. For the t-out-of-n CLLN protocol, we consider the special case that $(t, n) = (1, 2)$.

E.1 Comparison of Ideal Functionalities

In the following, we compare our ideal functionality $\mathcal{F}_{\mathrm{2pass}}$ with those of the CLN and CLLN protocols. On a high level, our $\mathcal{F}_{\mathrm{2pass}}$ is similar to both functionalities, in that we also model Setup and Retrieve instructions and let $\mathcal{A}$ hijack queries. We note the following differences: *1)* Our $\mathcal{F}_{\mathrm{2pass}}$ allows for adaptive corruptions and recovery from corruption, not just static corruptions. *2)* Our $\mathcal{F}_{\mathrm{2pass}}$ on one hand and the CLN- and CLLN-$\mathcal{F}_{\mathrm{2pass}}$ on the other all give the servers the option to refuse to service a Retrieve request, but model this option differently. In CLN- and CLLN-$\mathcal{F}_{\mathrm{2pass}}$, the servers are activated by the ideal functionality whenever a user wants to perform a retrieval; the servers then contact the environment and ask for permission to continue. In our $\mathcal{F}_{\mathrm{2pass}}$, the ideal functionality does not activate the servers

directly as required by the GNUC conventions:[7] instead, the servers activate $\mathcal{F}_{\text{2pass}}$. In order to allow the environment to decide whether the servers should service the request, the servers in our $\mathcal{F}_{\text{2pass}}$ don't need to explicity send a message to the environment to ask for permission to continue: the environment can simply refuse to provide the input that activates the server. *3)* Finally, similarly to the CLLN model but unlike the CLN one, the user's password attempt is protected during Retrieve in our $\mathcal{F}_{\text{2pass}}$ no matter the corruption status of the servers. Thus, if the user by mistake talks to the wrong servers for Retrieve, she is still safe, while the CLN functionality in this case hands the password over the adversary.

E.2 Comparison of our Construction

Comparison with the CLN-protocol. As we already pointed out in §2, the Setup and Retrieve instructions in both follow a similar structure, and we will thus mainly focus on the differences between the two.

In the CLN-protocol, the user's password and key are group elements instead of integers. This allows for an efficient way to enable the simulator $\mathcal{S}$ to extract the user's input from commitments sent by the user. Unlike the CLN protocol, our protocol must perform more expensive zero-knowledge that allow the simulator $\mathcal{S}$ to extract that input.

In the CLN-protocol, the user never performs any zero-knowledge proofs. This means that unlike our protocol (and the CLLN-protocol), the user's password attempt is not protected in case he contacts two corrupted servers.

The CLN-protocol [10] is secure against *static* corruptions only, while our protocol allows for *adaptive* corruptions. Due to the selective decommitment problem, which affects only protocols secure against adaptive corruptions, our protocol and the CLN-protocol further differ on three counts: *1)* in our protocol, parties need to establish OTPs among themselves, which is not needed in the CLN-protocol. We need this extra step to be able to encrypt in a non-committing way. *2)* In the CLN-protocol, the user and the servers communicate using perfectly-binding commitments. The servers don't need to prove knowledge of their shares to each other like we do at the end of Share, as $\mathcal{S}$ can extract the password and key from the perfectly-binding commitments. *3)* Our ChkPwd subroutine is different than the corresponding protocol in CLN, since we cannot allow the servers to send committing ciphertexts to each other.

Comparison with the CLLN-protocol. The recent CLLN-protocol [9] is also secure against static corruptions only, and thus the points in the previous paragraph also apply. Their protocol uses non-interactive zero-knowledge proofs extensively, and it is not clear how to instantiate their protocol in the standard model without resorting to impractical generic non-interactive zero-knowledge proofs. Furthermore, their protocol needs additional communication rounds compared to the random-oracle–version of our protocol.

E.3 Comparison of Computational Complexity for the Standard Model Constructions

In Table 1 we provide an estimate of the computational complexity of our protocol and compare it with the complexity of the CLN-protocol (adapted to the standard model) [10]. We re-did the estimation for CLN using a slightly different way of counting multi-exponentiations, so our numbers are slightly different from those provided in the original paper.

We also provide an estimate of the computation time when run with the "smallest general purpose" security level of the Ecrypt-II recommendations [**?**] ($\eta = 80$, $\log_2 q = 2\eta = 160$, $\log_2 n = 1248$, where $n = p' * p''$ is a safe RSA modulus) on a standard laptop with a 64-bit operating system using the GMP Multiple Precision Library.

[7] In the GNUC model, ideal functionalities are not allowed to activate a party from which they never received a message. In practice, a higher-level protocol will take care of notifying the servers of incoming messages.

We used the following runtime estimates for the basic building blocks, and assume the runtime of exponentiations scales linearly depending on the bitlength of the exponent:

- Let $\mathsf{exp}.\mathbb{G}$ be the runtime of exponentiation in $\mathbb{G}$ per bit of the exponent. For $\eta = 80$ and for a subgroup of the integers modulo a large prime, we use the estimate $\mathsf{exp}.\mathbb{G} = 1.42$ µs (i.e., a full exponentiation takes $2\eta \cdot \mathsf{exp}.\mathbb{G} = 227$ µs).
- Let $\mathsf{exp.n}$ be the runtime per bit of exponentiation modulo n. For $\eta = 80$ we use the estimate $\mathsf{exp.n} = 1.42$ µs (i.e., a full exponentiation takes $\log_2 n \cdot \mathsf{exp.n} = 1820$ µs).
- Let $\mathsf{exp.p}$ be the runtime per bit of exponentiation modulo p' or p''. For $\eta = 80$ we use the estimate $\mathsf{exp.p} = 0.42$ µs (i.e., an RSA decryption with Chinese remainder theorem takes $2((\log_2 n)/2 \cdot \mathsf{exp.p}) = 538$ µs).
- Let $\mathsf{exp.n^2}$ be the runtime per bit of exponentiation modulo n^2. For $\eta = 80$ we use the estimate $\mathsf{exp.n^2} = 5.14$ µs (i.e., a Paillier encryption $(n+1)^x r^n \pmod{n^2}$ takes time $\log_2 n \cdot \mathsf{exp.n^2} = 6580$ µs).
- Let $\mathsf{tprime}.2\eta$ be the average runtime to generate a prime of size 2η. For $\eta = 80$ we use the estimate $\mathsf{tprime}.2\eta = 329$ µs.

We did not consider the optimizations that are possible when running multibase exponentiations, or using precomputations for fixed bases. We chose to ignore the runtime of "fast" operations: additions, multiplications, inversions, and symmetric cryptographic operations. We also ignored the network delay: our Setup protocol requires one additional roundtrip than the CLN's, and both our and the CLN's Retrieve protocol have the same number of roundtrips. We also do not consider the various setup costs, such as generating a fresh RSA modulus for the signature, which need to be done once only and can be done ahead of time.

We chose the following standard-model implementations of primitives:

- For CCA-2 secure encryption in both our and the CLN-protocol, we use the Cramer-Shoup cryptosystem [15].
- For the signature scheme in the CLN-protocol, we use the Cramer-Shoup signature scheme [**?**].

We treated all zero-knowledge proofs in the CLN-protocols as proofs of existence, since $\mathcal{S}$ can extract the key and password from the El-Gamal ciphertexts. For the zero-knowledge proofs of knowledge we use in our protocol, we fit up to three witnesses into the plaintext of the verifiable encryption: Given witnesses $0 \leq a, b, c < q$, we encrypt $(a + q^2 b + q^4 c)$; by adding a range proofs that $-q^2/2 < a, b, c < q^2/2$ to the statement of the zero-knowledge proof and by recalling that $q^6 < n$, one is ensured that $\mathcal{S}$ can always recover a, b, and c; this range proof is essentially for free since the verifier simply needs to check the bit length of the responses in the zero-knowledge proof; this trick allows us to save on the very expensive operations modulo n^2, which are by far the dominant cost in the proofs.

Finally, we note that for efficiency reasons it does not make sense to use generic multi-party computations protocols to implement Π_{2pass}: the computational complexity of just a single multiplication in the best two-party computation protocol UC-secure against adaptive corruptions is $(90\eta + 200\log_2 n) \cdot \mathsf{exp.n} + (66\eta + 40.5\log_2 n) \cdot \mathsf{exp.n^2}$ [7], corresponding to a computation time of 660 ms at $\eta = 80$, i.e., more than 3.7 times slower than our entire Retrieve protocol.

E.4 Comparison of Computational Complexity for the Random Oracle Model Constructions

In Table 2 we provide an estimate of the computational complexity of our ROM-based protocol, the CLN-protocol [10] and the CLLN-protocol [9] (for $t = 1$, $n = 2$). Again, as we use a different method to count the exponentiations than the CLN and CLLN protocols, our numbers might be different than theirs.

For the ROM-based construction, we use the Fiat-Shamir heuristic [18] to transform our interactive proofs of knowledge into non-interactive ones. For the CCA2-secure encryption, we use ElGamal with Fujisaki-Okamoto padding, as recommended by CLLN. For signatures, we use Schnorr signatures.

		CLN-protocol [10]		Our protocol	
Adaptive corruptions		no		yes, and servers can recover	
Computation	user	$0\log_2 n$exp.p + 32ηexp.n + 40ηexp.$\mathbb{G}$ + 0tprime.2η	8 ms	$(16\ \eta + 0\ \log_2 n)$·exp.n + 92ηexp.$\mathbb{G}$ + $(0\eta + 0\log_2 n)$·exp.n^2	12 ms
time of	$\mathcal{P}$	$2\log_2 n$exp.p + 20ηexp.n + 18ηexp.$\mathbb{G}$ + 2tprime.2η	6 ms	$(30\ \eta + 3\ \log_2 n)$·exp.n + 90ηexp.$\mathbb{G}$ + $(1\eta + 3\log_2 n)$·exp.n^2	39 ms
Setup	$\mathcal{Q}$	$1\log_2 n$exp.p + 14ηexp.n + 18ηexp.$\mathbb{G}$ + 1tprime.2η	5 ms	$(32\ \eta + 3\ \log_2 n)$·exp.n + 90ηexp.$\mathbb{G}$ + $(1\eta + 3\log_2 n)$·exp.n^2	39 ms
Computation	user	$0\log_2 n$exp.p + 32ηexp.n + 36ηexp.$\mathbb{G}$ + 0tprime.2η	8 ms	$(68\ \eta + 6\ \log_2 n)$·exp.n + 188ηexp.$\mathbb{G}$ + $(2\eta + 6\log_2 n)$·exp.n^2	79 ms
time of	$\mathcal{P}$	$2\log_2 n$exp.p + 20ηexp.n + 125ηexp.$\mathbb{G}$ + 2tprime.2η	18 ms	$(26\ \eta + 3\ \log_2 n)$·exp.n + 175ηexp.$\mathbb{G}$ + $(1\eta + 3\log_2 n)$·exp.n^2	48 ms
Retrieve	$\mathcal{Q}$	$2\log_2 n$exp.p + 20ηexp.n + 125ηexp.$\mathbb{G}$ + 2tprime.2η	18 ms	$(26\ \eta + 3\ \log_2 n)$·exp.n + 166ηexp.$\mathbb{G}$ + $(1\eta + 3\log_2 n)$·exp.n^2	47 ms
Comp. time	$\mathcal{P}$	—		$(36\ \eta + 4\ \log_2 n)$·exp.n + 136ηexp.$\mathbb{G}$ + $(0\eta + 4\log_2 n)$·exp.n^2	52 ms
of Refresh	$\mathcal{Q}$	—		$(26\ \eta + 2\ \log_2 n)$·exp.n + 138ηexp.$\mathbb{G}$ + $(2\eta + 2\log_2 n)$·exp.n^2	36 ms

Table 1. Estimate of the computation time per party for the standard-model instantiation of our protocol and the CLN-protocol. The computation time in milliseconds is an estimate for $\eta = 80$ on a standard laptop.

		CLN-protocol [10]		CLLN-protocol [9]		Our protocol	
Adaptive corruptions		no		no		yes, and servers can recover	
Computation	user	44ηexp.$\mathbb{G}$	5 ms	70ηexp.$\mathbb{G}$	8 ms	$(\ 0\ \eta + 0\ \log_2 n)$·exp.n + 52ηexp.$\mathbb{G}$ + $(0\eta + 0\log_2 n)$·exp.n^2	6 ms
time of	$\mathcal{P}$	20ηexp.$\mathbb{G}$	2 ms	40ηexp.$\mathbb{G}$	5 ms	$(30\ \eta + 3\ \log_2 n)$·exp.n + 62ηexp.$\mathbb{G}$ + $(1\eta + 3\log_2 n)$·exp.n^2	35 ms
Setup	$\mathcal{Q}$	18ηexp.$\mathbb{G}$	2 ms	40ηexp.$\mathbb{G}$	5 ms	$(32\ \eta + 3\ \log_2 n)$·exp.n + 62ηexp.$\mathbb{G}$ + $(1\eta + 3\log_2 n)$·exp.n^2	36 ms
Computation	user	40ηexp.$\mathbb{G}$	5 ms	166ηexp.$\mathbb{G}$	19 ms	$(37\ \eta + 4\ \log_2 n)$·exp.n + 100ηexp.$\mathbb{G}$ + $(2\eta + 4\log_2 n)$·exp.n^2	49 ms
time of	$\mathcal{P}$	98ηexp.$\mathbb{G}$	11 ms	88ηexp.$\mathbb{G}$	10 ms	$(26\ \eta + 3\ \log_2 n)$·exp.n + 116ηexp.$\mathbb{G}$ + $(1\eta + 3\log_2 n)$·exp.n^2	41 ms
Retrieve	$\mathcal{Q}$	100ηexp.$\mathbb{G}$	11 ms	88ηexp.$\mathbb{G}$	10 ms	$(26\ \eta + 3\ \log_2 n)$·exp.n + 118ηexp.$\mathbb{G}$ + $(1\eta + 3\log_2 n)$·exp.n^2	41 ms
Comp. time	$\mathcal{P}$	—		—		$(36\ \eta + 4\ \log_2 n)$·exp.n + 98ηexp.$\mathbb{G}$ + $(0\eta + 4\log_2 n)$·exp.n^2	48 ms
of Refresh	$\mathcal{Q}$	—		—		$(26\ \eta + 2\ \log_2 n)$·exp.n + 130ηexp.$\mathbb{G}$ + $(2\eta + 2\log_2 n)$·exp.n^2	35 ms

Table 2. Estimate of the computation time per party for random-oracle-model–instantiations of our protocol, the CLN-protocol, and the 1-out-of-2 CLLN-protocol. The computation time in milliseconds is an estimate for $\eta = 80$ on a standard laptop.