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.

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 $\mathcal{F}_{\mathrm{2pass}}$ for two-server password-authenticated secret sharing, secure against transient corruptions. We provide the formal definition of $\mathcal{F}_{\mathrm{2pass}}$ in the GNUC variant [22] of the UC framework [11] in the full version [6]. $\mathcal{F}_{\mathrm{2pass}}$ 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 the full version [6].

The functionality $\mathcal{F}_{\mathrm{2pass}}$ involves two servers, $\mathcal{P}$ and $\mathcal{Q}$, and a plurality of users. We chose to define $\mathcal{F}_{\mathrm{2pass}}$ for a single user account, specified by the session id sid. Multiple accounts can be realized by multiple instances of $\mathcal{F}_{\mathrm{2pass}}$ or with a multi-session realization of $\mathcal{F}_{\mathrm{2pass}}$. 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 $\mathcal{F}_{\mathrm{2pass}}$. When starting a fresh query, any party can provide input in the role of a user to $\mathcal{F}_{\mathrm{2pass}}$; for subsequent inputs in that query, $\mathcal{F}_{\mathrm{2pass}}$ ensures it comes

$\mathcal{F}_{\text{2pass}}$ 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 the full version [6] for the full formalization.

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 the full version [6] 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.

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 $(out_{\mathcal{U}}; out_{\mathcal{P}}) \stackrel{\$}{\leftarrow} \langle \mathcal{U}.\mathrm{Sub}(in_{\mathcal{U}}), \mathcal{P}.\mathrm{Sub}(in_{\mathcal{P}})\rangle(in_{\mathcal{UP}})$ 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 $\lceil secretData \rceil_{\text{🔒}}$ 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 the full version [6].

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 the full version [6]. 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 the full version [6].

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: $\mathcal{F}_{\mathrm{gzk}}[sid]\{(\nmid\alpha, \beta\ ;\ \exists\gamma) : y = g^{\gamma} \wedge z = g^{\alpha}k^{\beta}h^{\gamma}\}$ 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 $\mathcal{F}_{\mathrm{gzk}}^{2\mathrm{v}}[sid]$ 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 the full version [6]. The proof that the special composition theorem by Camenisch, Krenn, and Shoup [8] holds also for $\mathcal{F}_{\mathrm{gzk}}^{2\mathrm{v}}$ is very similar to the proof that it holds for $\mathcal{F}_{\mathrm{gzk}}$ and is omitted.

Common reference string. Let $\mathcal{F}_{\mathrm{crs}}^{D}[sid]$ 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: $\mathcal{F}_{\mathrm{crs}}^{\mathbb{G}^3}$ provides a uniform CRS over $\mathbb{G}^3$ and $\mathcal{F}_{\mathrm{crs}}^{\mathrm{gzk}}$ provides a CRS as required by Camenisch et al.'s protocol π, the intended realization of $\mathcal{F}_{\mathrm{gzk}}$ [8]. We provide a formal definition in the full version [6].

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 $(pk, sk, kgr) \xleftarrow{\$} \mathrm{Gen}(1^\eta)$, where kgr is the randomness that was used to generate the key pair. We denote the encryption function $(e, er) \xleftarrow{\$} \mathrm{Enc}(pk, pt, l)$ 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 $pt \xleftarrow{\$} \mathrm{Dec}(sk, e, l)$ 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 $\mathcal{F}_{\mathrm{gzk}}$, 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 the full version [6].

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., $\mathcal{F}_{\mathrm{ac}}$, $\mathcal{F}_{\mathrm{osac}}$, $\mathcal{F}_{\mathrm{gzk}}$, and $\mathcal{F}_{\mathrm{gzk}}^{2\mathrm{v}}$: 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., $(c, o) = \mathrm{Com}(i)$ or $(e, er) = \mathrm{Enc}(pk, i, l)$. 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 $\mathcal{F}_{\mathrm{2pass}}$ ideal functionality in the $(\mathcal{F}_{\mathrm{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\mathrm{osac}}, \mathcal{F}_{\mathrm{ac}}, \mathcal{F}_{\mathrm{gzk}}, \mathcal{F}_{\mathrm{gzk}}^{2\mathrm{v}})$-hybrid setting. Our Π_{2pass} protocol further uses a CCA2-secure cryptosystem and an HMT commitment scheme. As for $\mathcal{F}_{\mathrm{2pass}}$, 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 the full version [6]).

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

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

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:

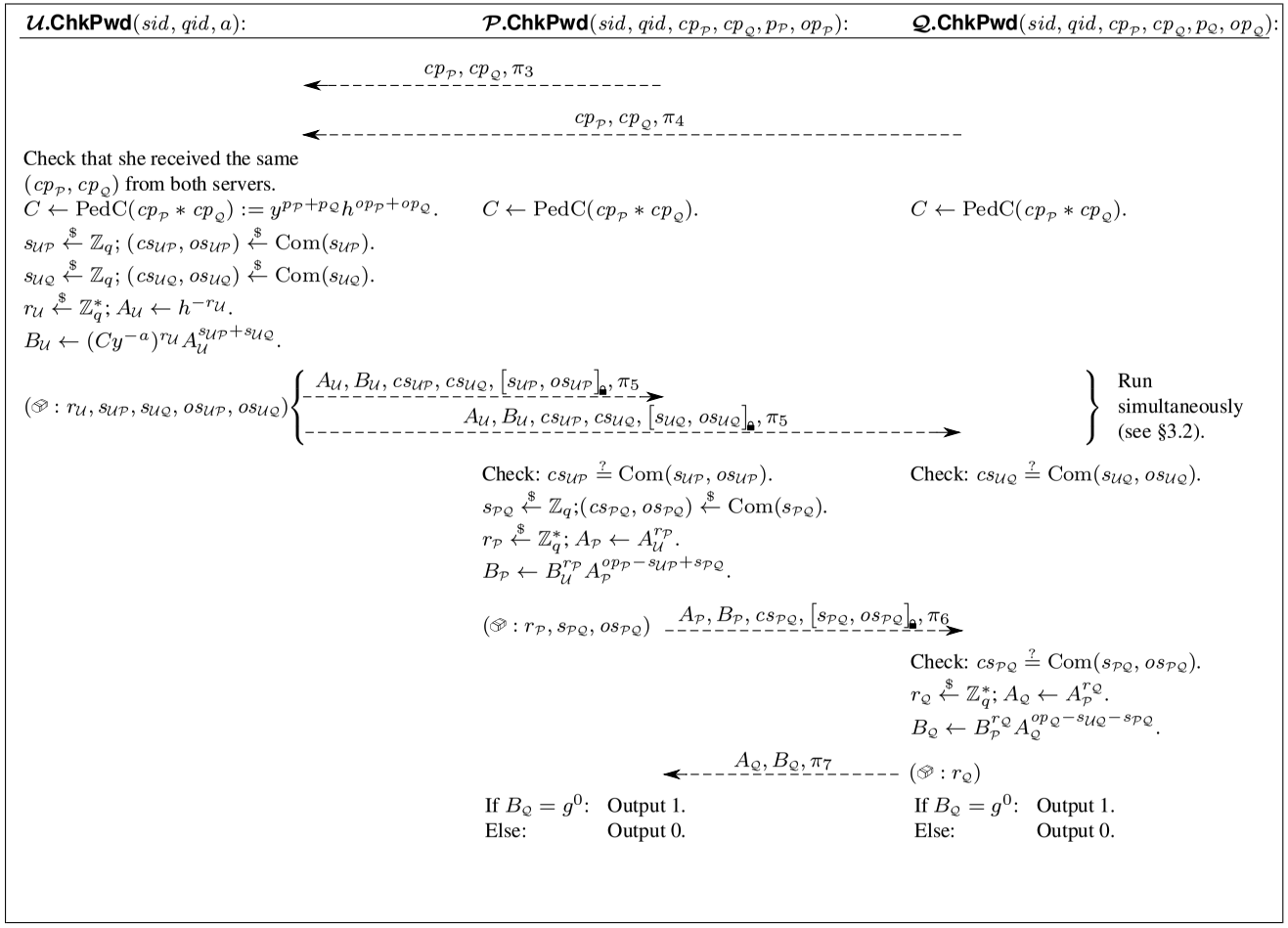


Fig. 2: Subroutine ChkPwd: the servers check if $\mathcal{U}$'s password attempt a is equal to the password $p_\mathcal{P} + p_\mathcal{Q}$. See the next figure for the instantiation of the zero-knowledge proofs.

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

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

$\pi_5 := \mathcal{F}^{2\text{v}}_{\text{gzk}}[sid, qid, cp_{\mathcal{P}}, cp_{\mathcal{Q}}, 5]\{(\gtrdot a, \sigma\ ;\ \exists \rho, \beta) :$
$h = A_{\mathcal{U}}^{\rho} \wedge C = (B_{\mathcal{U}}^{-1})^{\rho} y^{a} h^{\sigma} \wedge (cs_{\mathcal{U}\mathcal{P}} * cs_{\mathcal{U}\mathcal{Q}}) = \text{Com}(\sigma, \beta)$
$\}$, where $\sigma := s_{\mathcal{U}\mathcal{P}} + s_{\mathcal{U}\mathcal{Q}}$, $\rho := -1/r_{\mathcal{U}}$, and $\beta := os_{\mathcal{U}\mathcal{P}} + os_{\mathcal{U}\mathcal{Q}}$.

$\mathcal{U}$ 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).

$\pi_6 := \mathcal{F}_{\text{gzk}}[sid, qid, cp_{\mathcal{P}}, cp_{\mathcal{Q}}, cs_{\mathcal{U}\mathcal{P}}, cs_{\mathcal{U}\mathcal{Q}}, A_{\mathcal{U}}, B_{\mathcal{U}}, 6]\{(\exists p_{\mathcal{P}}, op_{\mathcal{P}}, r_{\mathcal{P}}, \sigma, \beta) :$
$A_{\mathcal{P}} = A_{\mathcal{U}}^{r_{\mathcal{P}}} \wedge A_{\mathcal{P}} \neq g^0 \wedge B_{\mathcal{P}} = B_{\mathcal{U}}^{r_{\mathcal{P}}} A_{\mathcal{P}}^{op_{\mathcal{P}}+\sigma} \wedge$
$cp_{\mathcal{P}} = \text{Com}(p_{\mathcal{P}}, op_{\mathcal{P}}) \wedge (cs_{\mathcal{P}\mathcal{Q}} * cs_{\mathcal{U}\mathcal{P}}^{-1}) = \text{Com}(\sigma, \beta)$
$\}$, where $\sigma := s_{\mathcal{P}\mathcal{Q}} - s_{\mathcal{U}\mathcal{P}}$ and $\beta := os_{\mathcal{P}\mathcal{Q}} - os_{\mathcal{U}\mathcal{P}}$.

$\pi_7 := \mathcal{F}_{\text{gzk}}[sid, qid, cs_{\mathcal{P}\mathcal{Q}}, A_{\mathcal{P}}, B_{\mathcal{P}}, 7]\{(\exists p_{\mathcal{Q}}, op_{\mathcal{Q}}, r_{\mathcal{Q}}, \sigma, \beta) :$
$A_{\mathcal{Q}} = A_{\mathcal{P}}^{r_{\mathcal{Q}}} \wedge A_{\mathcal{Q}} \neq g^0 \wedge B_{\mathcal{Q}} = B_{\mathcal{P}}^{r_{\mathcal{Q}}} A_{\mathcal{Q}}^{op_{\mathcal{Q}}-\sigma} \wedge$
$cp_{\mathcal{Q}} = \text{Com}(p_{\mathcal{Q}}, op_{\mathcal{Q}}) \wedge (cs_{\mathcal{U}\mathcal{Q}} * cs_{\mathcal{P}\mathcal{Q}}) = \text{Com}(\sigma, \beta)$
$\}$, where $\sigma := s_{\mathcal{U}\mathcal{Q}} + s_{\mathcal{P}\mathcal{Q}}$ and $\beta := os_{\mathcal{U}\mathcal{Q}} + os_{\mathcal{P}\mathcal{Q}}$.

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

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

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 $s_{\mathcal{U}\mathcal{P}}$, $s_{\mathcal{U}\mathcal{Q}}$, and $s_{\mathcal{P}\mathcal{Q}}$ as follows, thereby circumventing the problem. $\mathcal{U}$ chooses $s_{\mathcal{U}\mathcal{P}}$ and $s_{\mathcal{U}\mathcal{Q}}$, and sends them to $\mathcal{P}$ and $\mathcal{Q}$, respectively, in a non-committing manner. $\mathcal{U}$ then generates $B_{\mathcal{U}}$ by multiplying $D_{\mathcal{U}}$ with the blinding factor $A_{\mathcal{U}}^{s_{\mathcal{U}\mathcal{P}}+s_{\mathcal{U}\mathcal{Q}}}$, 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 *knowl-*

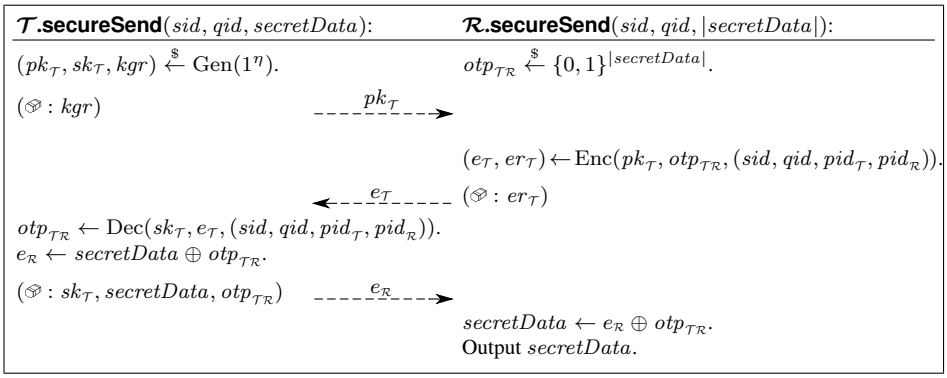


Fig. 4: Subroutine secureSend, the realization of $\overset{[secretData]_{\text{🔒}}}{\dashrightarrow}$: a party $\mathcal{T}$ (user or server) sends $secretData$ to $\mathcal{R}$ (user or server) in a non-committing encrypted form.

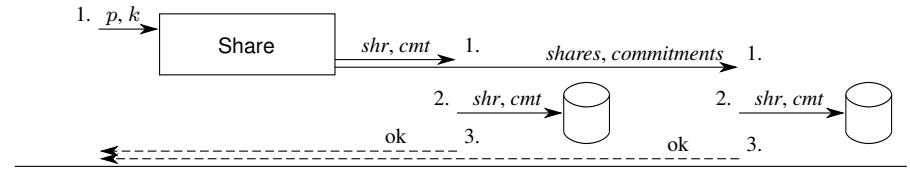
edge in the protocol, thereby massively increasing the computational complexity. Third, we assume that for each query the user establishes a single instance of a one-side-authenticated channel $\mathcal{F}_{\mathrm{osac}}[(sid, qid), \mathcal{P}]$ and $\mathcal{F}_{\mathrm{osac}}[(sid, qid), \mathcal{Q}]$ with each respective server; all communication denoted by arrows: $\dashrightarrow$, and all communication inside the zero-knowledge functionalities $\mathcal{F}_{\mathrm{gzk}}$ and $\mathcal{F}^{2\mathrm{v}}_{\mathrm{gzk}}$ happen through 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 $\langle \text{Setup}, sid = (pid_\mathcal{P}, pid_\mathcal{Q}, (\mathbb{G}, q, g), uacc, ssid), qid = \text{“Setup”}, p, k\rangle$ from the environment $\mathcal{Z}$, she starts a Setup query. Each of the servers starts a Setup query when he receives an input $\langle \text{ReadySetup}, sid, qid\rangle$ 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 $\langle \text{Done}, sid, qid\rangle$ to $\mathcal{Z}$.

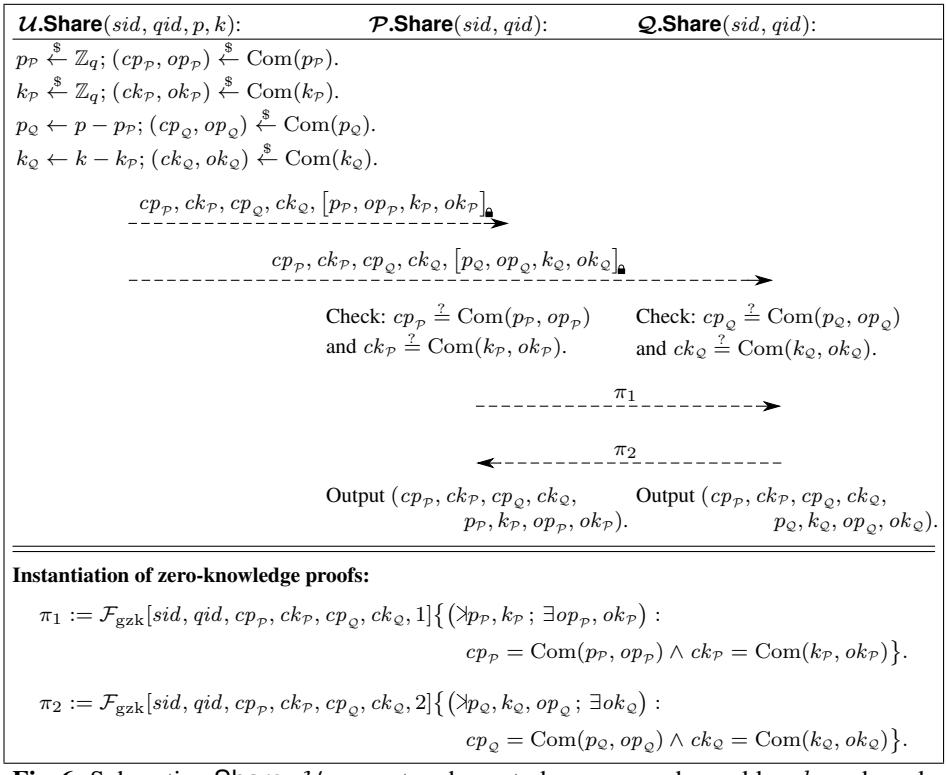
The Share subprotocol Setup uses is depicted in Figure 6. In that subprotocol $\mathcal{U}$ splits her inputs p and k into random additive shares $p_\mathcal{P} + p_\mathcal{Q} := p$ and $k_\mathcal{P} + k_\mathcal{Q} := k$,

[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:
$\big(\varepsilon; (cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{P}}, k_{\mathcal{P}}, op_{\mathcal{P}}, ok_{\mathcal{P}}); (cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{Q}}, k_{\mathcal{Q}}, op_{\mathcal{Q}}, ok_{\mathcal{Q}})\big) \xleftarrow{\$}$
$\langle \mathcal{U}.\mathsf{Share}(p,k), \mathcal{P}.\mathsf{Share}(), \mathcal{Q}.\mathsf{Share}()\rangle(sid, qid)$.
2. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ stores $\big(cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{R}}, k_{\mathcal{R}}, op_{\mathcal{R}}, ok_{\mathcal{R}}\big)$ 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:

$$\pi_1 := \mathcal{F}_{\mathrm{gzk}}[sid, qid, cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, 1]\big\{(\not\exists p_{\mathcal{P}}, k_{\mathcal{P}}\,;\, \exists op_{\mathcal{P}}, ok_{\mathcal{P}}) : cp_{\mathcal{P}} = \mathrm{Com}(p_{\mathcal{P}}, op_{\mathcal{P}}) \wedge ck_{\mathcal{P}} = \mathrm{Com}(k_{\mathcal{P}}, ok_{\mathcal{P}})\big\}.$$

$$\pi_2 := \mathcal{F}_{\mathrm{gzk}}[sid, qid, cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, 2]\big\{(\not\exists p_{\mathcal{Q}}, k_{\mathcal{Q}}, op_{\mathcal{Q}}\,;\, \exists ok_{\mathcal{Q}}) : cp_{\mathcal{Q}} = \mathrm{Com}(p_{\mathcal{Q}}, op_{\mathcal{Q}}) \wedge ck_{\mathcal{Q}} = \mathrm{Com}(k_{\mathcal{Q}}, ok_{\mathcal{Q}})\big\}.$$

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

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 $B_{\mathcal{P}} = (A_{\mathcal{P}})^{s_{\mathcal{U}\mathcal{Q}}+s_{\mathcal{P}\mathcal{Q}}-op_{\mathcal{Q}}}$ in ChkPwd (we note that $\mathcal{S}$ does not need to know the value $op_{\mathcal{P}}$ from π_1 at this point).

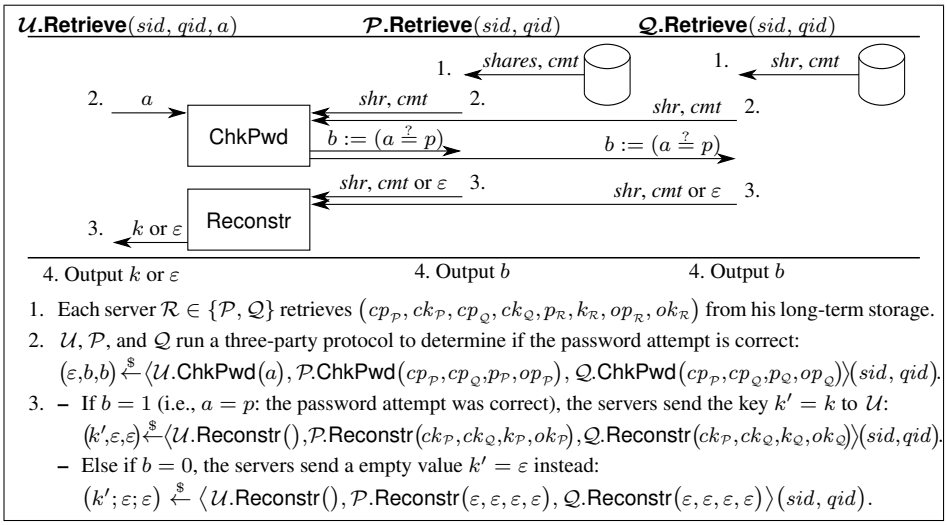
The Retrieve instruction. Recall that the goal of the Retrieve instruction is for a user (not necessarily the same as during Setup) to retrieve the key k, contingent upon her holding a correct password attempt $a \in \mathbb{Z}_q$.

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.



1. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ retrieves $(cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{R}}, k_{\mathcal{R}}, op_{\mathcal{R}}, ok_{\mathcal{R}})$ 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:
   $(\varepsilon,b,b) \stackrel{\$}{\leftarrow} \langle \mathcal{U}.\mathsf{ChkPwd}(a), \mathcal{P}.\mathsf{ChkPwd}(cp_{\mathcal{P}},cp_{\mathcal{Q}},p_{\mathcal{P}},op_{\mathcal{P}}), \mathcal{Q}.\mathsf{ChkPwd}(cp_{\mathcal{P}},cp_{\mathcal{Q}},p_{\mathcal{Q}},op_{\mathcal{Q}})\rangle(sid, qid)$.
3. – If $b = 1$ (i.e., $a = p$: the password attempt was correct), the servers send the key $k' = k$ to $\mathcal{U}$:
   $(k',\varepsilon,\varepsilon) \stackrel{\$}{\leftarrow} \langle \mathcal{U}.\mathsf{Reconstr}(), \mathcal{P}.\mathsf{Reconstr}(ck_{\mathcal{P}},ck_{\mathcal{Q}},k_{\mathcal{P}},ok_{\mathcal{P}}), \mathcal{Q}.\mathsf{Reconstr}(ck_{\mathcal{P}},ck_{\mathcal{Q}},k_{\mathcal{Q}},ok_{\mathcal{Q}})\rangle(sid,qid)$.
   – Else if $b = 0$, the servers send a empty value $k' = \varepsilon$ instead:
   $(k';\varepsilon;\varepsilon) \stackrel{\$}{\leftarrow} \langle \mathcal{U}.\mathsf{Reconstr}(), \mathcal{P}.\mathsf{Reconstr}(\varepsilon,\varepsilon,\varepsilon,\varepsilon), \mathcal{Q}.\mathsf{Reconstr}(\varepsilon,\varepsilon,\varepsilon,\varepsilon)\rangle(sid, qid)$.

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

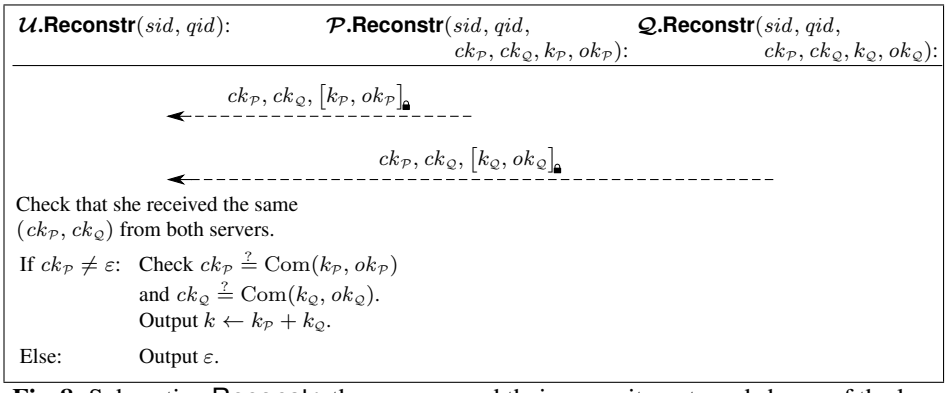
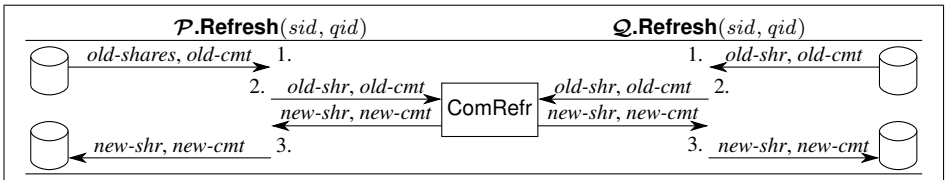


Fig. 8: Subroutine Reconstr: the servers send their commitments and shares of the key to $\mathcal{U}$ so that she may reconstruct her key k.

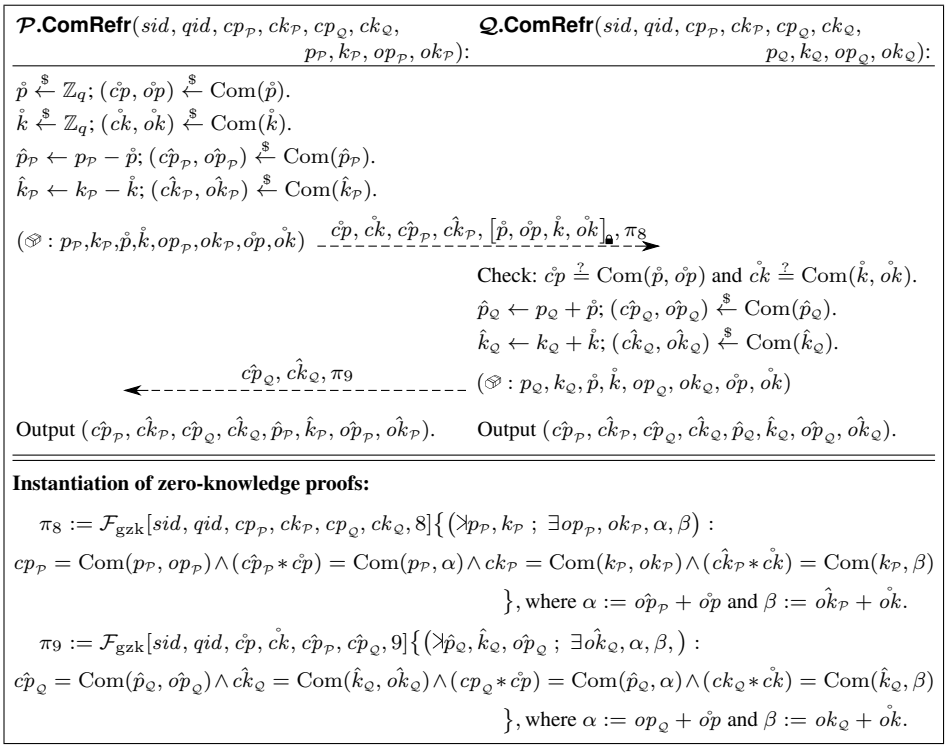
When a server receives an input $\langle \text{Refresh}, sid, qid \rangle$ with the same sid as during Setup from $\mathcal{Z}$, he starts the Refresh instruction. The Refresh protocol runs as follows and is depicted in Figure 9. The servers start by recovering their internal state. The servers then 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 $\langle \text{RefreshDone}, sid, qid \rangle$ 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



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:
   $\big((c\hat{p}_{\mathcal{P}}, c\hat{k}_{\mathcal{P}}, c\hat{p}_{\mathcal{Q}}, c\hat{k}_{\mathcal{Q}}, \hat{p}_{\mathcal{P}}, \hat{k}_{\mathcal{P}}, o\hat{p}_{\mathcal{P}}, o\hat{k}_{\mathcal{P}}); (c\hat{p}_{\mathcal{P}}, c\hat{k}_{\mathcal{P}}, c\hat{p}_{\mathcal{Q}}, c\hat{k}_{\mathcal{Q}}, \hat{p}_{\mathcal{Q}}, \hat{k}_{\mathcal{Q}}, o\hat{p}_{\mathcal{Q}}, o\hat{k}_{\mathcal{Q}})\big) \xleftarrow{\$}$
   $\big\langle \mathcal{P}.\mathsf{ComRefr}\big(p_{\mathcal{P}}, k_{\mathcal{P}}, op_{\mathcal{P}}, ok_{\mathcal{P}}\big), \mathcal{Q}.\mathsf{ComRefr}\big(p_{\mathcal{Q}}, k_{\mathcal{Q}}, op_{\mathcal{Q}}, ok_{\mathcal{Q}}\big)\big\rangle\big(sid, qid, cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}\big)$.
3. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ stores $\big(c\hat{p}_{\mathcal{P}}, c\hat{k}_{\mathcal{P}}, c\hat{p}_{\mathcal{Q}}, c\hat{k}_{\mathcal{Q}}, \hat{p}_{\mathcal{R}}, \hat{k}_{\mathcal{R}}, o\hat{p}_{\mathcal{R}}, o\hat{k}_{\mathcal{R}}\big)$ into his long-term storage.

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



Instantiation of zero-knowledge proofs:

$$\pi_8 := \mathcal{F}_{\mathrm{gzk}}[sid, qid, cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, 8]\big\{\big(\gtrdot p_{\mathcal{P}}, k_{\mathcal{P}}\ ;\ \exists op_{\mathcal{P}}, ok_{\mathcal{P}}, \alpha, \beta\big)\ :$$
$$cp_{\mathcal{P}} = \mathrm{Com}(p_{\mathcal{P}}, op_{\mathcal{P}}) \wedge (c\hat{p}_{\mathcal{P}} * c\mathring{p}) = \mathrm{Com}(p_{\mathcal{P}}, \alpha) \wedge ck_{\mathcal{P}} = \mathrm{Com}(k_{\mathcal{P}}, ok_{\mathcal{P}}) \wedge (c\hat{k}_{\mathcal{P}} * c\mathring{k}) = \mathrm{Com}(k_{\mathcal{P}}, \beta)$$
$$\big\}, \text{where } \alpha := o\hat{p}_{\mathcal{P}} + o\mathring{p} \text{ and } \beta := o\hat{k}_{\mathcal{P}} + o\mathring{k}.$$
$$\pi_9 := \mathcal{F}_{\mathrm{gzk}}[sid, qid, c\mathring{p}, c\mathring{k}, c\hat{p}_{\mathcal{P}}, c\hat{p}_{\mathcal{Q}}, 9]\big\{\big(\gtrdot \hat{p}_{\mathcal{Q}}, \hat{k}_{\mathcal{Q}}, o\hat{p}_{\mathcal{Q}}\ ;\ \exists o\hat{k}_{\mathcal{Q}}, \alpha, \beta, \big)\ :$$
$$c\hat{p}_{\mathcal{Q}} = \mathrm{Com}(\hat{p}_{\mathcal{Q}}, o\hat{p}_{\mathcal{Q}}) \wedge c\hat{k}_{\mathcal{Q}} = \mathrm{Com}(\hat{k}_{\mathcal{Q}}, o\hat{k}_{\mathcal{Q}}) \wedge (cp_{\mathcal{Q}} * c\mathring{p}) = \mathrm{Com}(\hat{p}_{\mathcal{Q}}, \alpha) \wedge (ck_{\mathcal{Q}} * c\mathring{k}) = \mathrm{Com}(\hat{k}_{\mathcal{Q}}, \beta)$$
$$\big\}, \text{where } \alpha := op_{\mathcal{Q}} + o\mathring{p} \text{ and } \beta := ok_{\mathcal{Q}} + o\mathring{k}.$$

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

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 the password (for the same reason as in Share: $\mathcal{S}$ needs $o\hat{p}_{\mathcal{Q}}$ 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 the full version [6] 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 the full version [6] 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 the full version [6] for a detailed analysis.

[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.

5 Proof Sketch

For reasons of space, we provide the security proof in the full version [6] 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 $(\mathcal{F}_{\mathrm{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\mathrm{osac}}, \mathcal{F}_{\mathrm{ac}}, \mathcal{F}_{\mathrm{gzk}}, \mathcal{F}_{\mathrm{gzk}}^{2\mathrm{v}})$-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 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 $\mathcal{F}_{\mathrm{gzk}}$ and $\mathcal{F}_{\mathrm{gzk}}^{2\mathrm{v}}$. The commitments and proofs of *existence* with $\mathcal{F}_{\mathrm{gzk}}$ and $\mathcal{F}_{\mathrm{gzk}}^{2\mathrm{v}}$ 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 $\mathcal{F}_{\mathrm{2pass}}$ whether the password attempt was correct in a Retrieve query, it can generate credible values $B_{\mathcal{U}}$, $B_{\mathcal{P}}$, and $B_{\mathcal{Q}}$ in the ChkPwd subroutine because $\mathcal{S}$ can recover the opening values op from dishonest servers through $\mathcal{F}_{\mathrm{gzk}}$ and $\mathcal{F}_{\mathrm{gzk}}^{2\mathrm{v}}$. 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 $\mathcal{F}_{\mathrm{2pass}}$ 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 $\mathcal{F}_{\mathrm{gzk}}$ and $\mathcal{F}_{\mathrm{gzk}}^{2\mathrm{v}}$ 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. Passwordsˆ12 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.