Auditable Asymmetric Password Authenticated Public Key Establishment

Antonio Faonio[1], Maria Isabel Gonzalez Vasco[2], Claudio Soriente[3], and Hien Thi Thu Truong[3]

[1] IMDEA Software Institute,
[2] Universidad Rey Juan Carlos,
[3] NEC Laboratories Europe GmbH

Abstract. Non-repudiation of messages generated by users is a desirable feature in a number of applications ranging from online banking to IoT scenarios. However, it requires certified public keys and usually results in poor usability as a user must carry around his certificate (e.g., in a smart-card) or must install it in all of his devices. A user-friendly alternative, adopted by several companies and national administrations, is to have a "cloud-based" PKI. In a nutshell, each user has a PKI certificate stored at a server in the cloud; users authenticate to the server—via passwords or one-time codes—and ask it to sign messages on their behalf. As such, there is no way for the server to prove to a third party that a signature on a given message was authorized by a user. As the server holds the user's certified key, it might as well have signed arbitrary messages in an attempt to impersonate that user. In other words, a user could deny having signed a message, by claiming that the signature was produced by the server without his consent. The same holds in case the secret key is derived deterministically from the user's password, for the server, by knowing the password, may still frame the user.
In this paper we provide a "password-only" solution to non-repudiation of user messages by introducing Auditable Asymmetric Password Authenticated Public Key Establishment ($\mathsf{A^2PAKE}$). This is a PAKE-like protocol that generates an asymmetric key-pair where the public key is output to every participant, but the secret key is private output to just one of the parties (e.g., the user). Further, the protocol can be *audited,* i.e., given the public key output by a protocol run with a user, the server can prove to a third party that the corresponding secret key is held by that specific user. Thus, if the user signs messages with that secret key, then signatures are non-repudiable. We provide a universally composable definition of $\mathsf{A^2PAKE}$ and an instantiation based on a distributed oblivious pseudo-random function. We also develop a prototype implementation of our instantiation and use it to evaluate its performance in realistic settings.

1 Introduction

Non-repudiation in online applications is essentially based on digital signatures and PKI certificates. Given that most servers hold a PKI certificate, non-repudiation of messages generated by a server can be easily achieved. The same is not true for messages sent by users, since they often authenticate to servers using only low-entropy passwords.[4] Yet, non repudiation of user messages is a desirable property in a number of scenarios. For example, in an online banking application, the banking server may ask the user to authenticate his requests to, e.g., transfer funds to other accounts; later on, in case of dispute, the server may wish to prove to a third party (e.g., in court) that the user had indeed requested a specific operation. Similarly, in IoT applications the processing server may wish to pinpoint the owner of the device that produced a specific data report.

The most popular solution to the problem at hand leverages "cloud-based PKI". The idea is that each user has a PKI certificate stored at a server in the cloud; users authenticate to the server—via passwords or one-time codes—and ask it to sign messages on their behalf. This design is used by a number of companies [29,19] and national administrations [32]. This design, however, does not allow a third party to tell which signatures were genuinely produced by the user and which were the result of an impersonation attempt by the server. Since the server holds the user's certificate and could sign any message on his behalf, that user could deny having signed a message, e.g., by claiming that the signature was produced by the server without his consent.

This issue arises even if the signing key is derived from the user's password (e.g., by using the PBKDF2 key-derivation function). The server— by knowing the password— may derive the same key and sign messages on behalf

[4] Non-repudiation of users messages would be trivial to achieve if users had PKI client certificates. However, PKI client certificates hinder the user experience—as users can only connect to the server from devices where the certificate is installed.

of the victim user. Similarly, if the server holds a (salted) one-way function of the password, a simple brute-force attack allows it to derive the correct key.

We note that password-authenticated key exchange (PAKE) [5,6] allows two parties to authenticate and jointly establish a strong cryptographic key by relying solely on a common password. However, all PAKE protocols to date deal with symmetric keys and are thus ill-suited for our goals—since knowledge of the symmetric key allows any of the two parties to authenticate messages on behalf of the other.

Solving the problem of non-repudiation for users that only use passwords requires a PAKE-like protocol that allows to generate an asymmetric key-pair where the public key is public output, but the secret key is private output to one of the parties. Further, to achieve non-repudiation, the protocol should be *auditable*. That is, the party getting only the public key, should be able to prove that the corresponding secret key must have been generated by its peer. In particular, given a transcript T of a client-server protocol where the client C gets a key-pair sk, pk and the server S only learns the public key pk, we would like the server to be able to prove to a third party that only C can possibly know the secret key sk matching the public key pk output by T.

Note that auditing a transcript—i.e., ascribing a public key to an identity—requires establishing and verifying user "identities". We are agnostic to how such identities are created and managed. For example, identities could be bound to an ID by asking the user to submit a copy of his ID during registration [29,19]. If the registering user holds an eIDs, it can be used to sign the registration transcript by using a smart-card reader attached to a PC [32]. Later on, during auditing, a public key can be ascribed to the holder of the ID (or eID) used during registration. Another option would be for users to register with an email address: during registration, the user must prove ownership of an email address (e.g., by receiving a one-time code in his inbox) and the new account must be bound to that email address. During auditing, a public key can be attributed to the holder of the email address used during registration. In a similar fashion, identities can be verified by means of mobile phone numbers and one-time codes sent via SMS messages.

Note also that a third party can be convinced that a particular public key was created by a user only if the server cannot frame users by creating key-pairs on their behalf. If key-pairs are generated from passwords, we must ensure that only users know their full passwords; yet we must (i) allow the server to authenticate users, and (ii) prevent the server from mounting offline brute-force attacks. A number of password-based authentication schemes (including some variants of PAKE [25,24,26]) split the password among several servers with the goal of mitigating password leaks due to a server breach. We use the same approach and introduce multiple servers to authenticate users, so that, unless all servers are malicious, they cannot frame a user (e.g., by running an offline brute-force attack to recover his password). In the simple two-server scenario, a "main" server obtains the public key as output of the protocol, and may later on turn to a third party for auditing purposes—i.e., to prove that the public key belongs to a specific user—while a "secondary" server support its peer in authenticating users and cooperates to produce auditing evidence. As long as one of the servers is honest, third parties can pinpoint a public key output by the protocol to the user that engaged in that execution.

In this paper we formally define and instantiate a cryptographic protocol with the above properties for the two server scenario, that we call $\mathsf{A}^2\mathsf{PAKE}$. We give an ideal functionality of $\mathsf{A}^2\mathsf{PAKE}$ that captures the security requirements of non-repudiation of the keys generated by a user and non-frameability from malicious servers. Further, we provide a protocol that realizes the functionality of $\mathsf{A}^2\mathsf{PAKE}$ and prove it secure in the universal composability framework of Canetti [10]. The main ingredient of our protocol is a distributed oblivious pseudo-random function [22]. Finally, we introduce a prototype implementation written in Python and present the results of an evaluation carried out to assess throughput, latency, and communication overhead.

2 Related work

There is a vast literature on password-based cryptography. The basic idea is to design protocols with strong cryptographic guarantees by relying solely on low-entropy passwords.

The popular PKCS#5 [31] standard shows how to use passwords to derive symmetric keys to be used for (symmetric) encryption or message authentication. Password-Authenticated Key Exchange (PAKE) [5,6] enables two parties, holding the same password, to authenticate mutually and to establish a symmetric key. In the client-server settings, compromise of the password database at the server may be mitigated by splitting passwords among multiple

servers [25,24,26]. *Threshold* PAKE [1,30,33] borrows from threshold-based cryptography and distributes the authentication functionality across n servers in a way such that the client authenticates successfully only if it cooperates with at least $t < n$ servers. Passwords are not leaked as long as the adversary compromises $t - 1$ or less servers.

Password-Authenticated Public-Key Encryption (PAPKE) [7] enhances public-key encryption with passwords. In particular, generation of a key-pair is bound to a password and so is encryption. Hence, decryption of a ciphertext reveals the original message only if the password used at encryption time matches the one used when the public key was generated. Thus, PAPKE preserves confidentiality despite a man-in-the-middle that replaces the public key of the receiver (as long as the adversary does not guess the receiver's password when generating its public key).

Password-based signatures were proposed in [15] where the signature key is split between the user and a server and the user's share is essentially his password—so that the user can create signatures with the help of the server. We note that in [15] the server does not authenticate users and that it could recover the full signing key of any user by brute-forcing the password. User authentication and resistance to brute-force attacks for password-based signatures were introduced in [8], that requires users to carry a personal device such as a smart card.

Password-hardening services [28,14,34,27] enable password-based authentication while mitigating the consequences of password database leak. The idea behind password-hardening services is to pair the authentication server with a "cryptographic service" that blindly computes (keyed) hashes of the passwords. The password database at the authentication service stores such hashes so that a leak of the database does not reveal passwords, unless the key of the cryptographic service is compromised.

PASTA by Agrawal *et al.* [2] and PESTO by Baum *et al.* [4] propose password-based threshold token-based authentication where the role of an identity provider in a protocol such as OAuth [5] is distributed across several parties and the user obtains an authentication token only by authenticating to a threshold number of servers; both protocols are based on threshold oblivious pseudo-random functions [22].

To the best of our knowledge, the closest cryptographic primitive to $\mathsf{A}^2\mathsf{PAKE}$ is Password-Protected Secret Sharing [3,9,20,21,22]. PPSS allows users to securely store shares of a secret—e.g., a cryptographic key— on a set of servers while reconstruction is only feasible by using the right password or by corrupting more than a given threshold of servers. In principle, PPSS could be used to realize a functionality similar to the one offered by $\mathsf{A}^2\mathsf{PAKE}$. However, there are a few important differences between $\mathsf{A}^2\mathsf{PAKE}$ and PPSS. First, PPSS does not solve the problem of user authentication. Further, $\mathsf{A}^2\mathsf{PAKE}$ provides forward security—leakage of a password does not compromise past sessions—and outputs uniformly distributed keys despite a malicious client or server. Also, compromise of the PPSS servers immediately reveals the user password; if all $\mathsf{A}^2\mathsf{PAKE}$ servers are compromised, a brute-force attack is still required to learn the password—therefore high-entropy passwords still offer security. Finally, we embed auditability in the $\mathsf{A}^2\mathsf{PAKE}$ functionality whereas it is not clear how to make PPSS auditable by a third-party.

3 Preliminaries

3.1 Digital Signatures

A signature scheme over groups generated by $\mathbb{G}$ is a triple of efficient algorithms $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$. Algorithm KGen outputs a public key pk and a secret key sk. Algorithm Sign takes as input a secret key and a message m in the message space, and outputs a signature σ. Algorithm Vf takes as input a public key pk, a message m and a signature σ, and returns either 1 or 0 (i.e., "accept" or "reject", respectively). The scheme $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ is correct if for every correctly generated key-pair pk, sk, and for every message m in the message space, we have $\mathsf{Vf}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1$.

We consider the standard notion of existential unforgeability under chosen-messages attacks [16].

3.2 Non-Interactive Zero-Knowledge Proof of Knowledge

A non-interactive zero-knowledge (NIZK) proof system for a relation $\mathcal{R}$ is a tuple $\mathcal{NIZK} = (\mathsf{Init}, \mathsf{P}, \mathsf{V})$ of PPT algorithms such that: Init on input the security parameter outputs a (uniformly random) common reference string

[5] `https://oauth.net/`

$\mathsf{crs} \in \{0,1\}^\lambda$; $\mathsf{P}(\mathsf{crs}, x, w)$, given $(x, w) \in \mathcal{R}$, outputs a proof π; $\mathsf{V}(\mathsf{crs}, x, \pi)$, given instance x and proof π outputs 0 (reject) or 1 (accept).

In this paper we consider the notion of *NIZK with labels*, that are NIZKs where P and V additionally take as input a label $L \in \mathcal{L}$ (e.g., a binary string). A NIZK (with labels) is *correct* if for every $\mathsf{crs} \leftarrow_\$ \mathsf{Init}(1^\lambda)$, any label $L \in \mathcal{L}$, and any $(x, w) \in \mathcal{R}$, we have $\mathsf{V}(\mathsf{crs}, L, x, \mathsf{P}(\mathsf{crs}, L, x, w)) = 1$.

We consider a property called *simulation-extractable soundness*. Roughly speaking, the definition of simulation extractable soundness assumes the existence of a Init algorithm that, additionally to the common reference string, outputs a *simulation trapdoor* tp_s that allows to simulate proofs, and a *extraction trapdoor* tp_e that allows to extract the witness from valid (no-simulated) proofs. The security guarantee is that, even in presence of a oracle that simulates proofs, an adversary cannot produce a valid proof that cannot be extracted. The limitation is that an adversary needs to produce a proof on a label-statement tuple that was not previously asked to the oracle. Further we require the NIZK to be adaptive composable zero-knowledge—by now the standard zero-knowledge notion for NIZK, first considered by Groth [17].

Functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$:

The functionality is parameterised by a security parameter λ and an asymmetric public key generation algorithm KGen. The functionality receives from the environment the set $\mathbb{C}$ of corrupted parties. Let $\mathbb{H}$ be the set of honest parties. Let $\mathcal{D}_{\mathsf{pk}}$ (resp. $\mathcal{D}_{\mathsf{pw}}$) be the database of the completed session (resp. of the registered passwords). Both $\mathcal{D}_{\mathsf{pk}}$ and $\mathcal{D}_{\mathsf{pw}}$ are initialized to empty. The functionality has n client $C_1, \ldots, C_n$, two servers S_1 and S_2 and an auditor A:

Registration: On $(\texttt{register}, \mathsf{sid}, \mathsf{pw})$ from C_j. If the request is valid (namely, there is no record of the form $(\mathsf{sid}, C_j, *) \in \mathcal{D}_{\mathsf{pw}}$) create a fresh record $(\mathsf{sid}, C_j, \mathsf{pw})$ in $\mathcal{D}_{\mathsf{pw}}$.
Send $(\texttt{registered}, \mathsf{sid}, C_j)$ to S_1, S_2 and S.

Init: On $(\texttt{init}, \mathsf{sid}, \mathsf{qid}, j)$ from a party $\mathcal{P} \in \{C_j, S_1, S_2\}$. Send $(\texttt{init}, \mathsf{sid}, \mathsf{qid}, j, \mathcal{P})$ to S. Record that $\mathcal{P}$ initialized the session. If C_1, S_1 and S_2 initialized the session, then record that the session $(\mathsf{sid}, \mathsf{qid}, j)$ is active for S_1 and C_j.

Test Password: On $(\texttt{test}, \mathsf{sid}, j, \mathsf{pw}')$ from S. Check whether an entry $(\mathsf{sid}, C_j, \mathsf{pw}) \in \mathcal{D}_{\mathsf{pw}}$ exists. If $\mathsf{pw} = \mathsf{pw}'$ then reply S with $\texttt{correct}$ and set $\texttt{corrupt}(\mathsf{sid}, j) \leftarrow 1$. Notify both S_1 and S_2.

New Key: On message $(\texttt{newkey}, \mathsf{sid}, \mathsf{qid}, j, \mathcal{P})$ from S.
- Assert $(\mathsf{sid}, j, \mathsf{pw}) \in \mathcal{D}_{\mathsf{pw}}$ and $\mathcal{P} \in \{C_j, S_1\}$ and that session $(\mathsf{sid}, \mathsf{qid}, j)$ is active for $\mathcal{P}$;
- Mark session $(\mathsf{sid}, \mathsf{qid}, j)$ for $\mathcal{P}$ as finalized;
- If $\texttt{corrupt}(\mathsf{sid}, j)$ then:
  - if $(\mathsf{sid}, \mathsf{qid}, j, \mathcal{P}, \mathsf{sk}, \mathsf{pk})$ exists in the database retrieve the tuple,
  - else sample $(\mathsf{sk}, \mathsf{pk}) \leftarrow_\$ \mathsf{KGen}(1^\lambda)$ and register $(\mathsf{sid}, \mathsf{qid}, j, \mathcal{P}, \mathsf{sk}, \mathsf{pk})$.

  Else:
  - If $(\mathsf{sid}, \mathsf{qid}, j, *, \mathsf{sk}, \mathsf{pk})$ exists register $(\mathsf{sid}, \mathsf{qid}, j, \mathcal{P}, \mathsf{sk}, \mathsf{pk})$,
  - else sample $(\mathsf{sk}, \mathsf{pk}) \leftarrow_\$ \mathsf{KGen}(1^\lambda)$ and register $(\mathsf{sid}, \mathsf{qid}, j, \mathcal{P}, \mathsf{sk}, \mathsf{pk})$.
- If $S_1 \in \mathbb{H}$ and $\mathcal{P} = S_1$ register $(\mathsf{sid}, \mathsf{qid}, C_j, \mathsf{pk})$ in $\mathcal{D}_{\mathsf{pk}}$;
- If $\mathcal{P} = C_j$ then send a private delayed output $(\texttt{output}, \mathsf{sid}, \mathsf{qid}, j, \mathsf{sk})$ to C_j and the public key to S;
- If $\mathcal{P} = S_1$ then send a public delayed output $(\texttt{output}, \mathsf{sid}, \mathsf{qid}, j, \mathsf{pk})$ to S_1;
- If $\texttt{corrupt}(\mathsf{sid}, j)$ then send sk to S.

Invalid: On message $(\texttt{invalid}, \mathsf{sid}, \mathsf{qid}, j, \mathcal{P})$ from S and $\mathcal{P} \in \{C_1, \ldots, C_n, S_1\}$. If $\mathcal{P} \in \{S_1, C_j\}$ send $(\texttt{output}, \mathsf{sid}, \mathsf{qid}, j, \perp)$ to $\mathcal{P}$ and mark the session finalized for $\mathcal{P}$.

Audit: On message $(\texttt{audit}, \mathsf{sid}, \mathsf{qid}, C_j, \mathsf{pk})$ from party S_1. Set $b \leftarrow \perp$. Next,
- If $\texttt{corrupt}(\mathsf{sid}, j)$ wait for a bit b' from the adversary S and set $b \leftarrow b'$.
- NON-FRAMEABILITY. If $C_j \in \mathbb{H} \wedge \neg\texttt{corrupt}(\mathsf{sid}, j) \wedge (\mathsf{sid}, \mathsf{qid}, C_j, \mathsf{pk}) \notin \mathcal{D}_{\mathsf{pk}}$ then set $b \leftarrow 0$.
- NON-REPUDIABILITY. If $S_1 \in \mathbb{H} \wedge C_j \in \mathbb{C} \wedge (\mathsf{sid}, \mathsf{qid}, C_j, \mathsf{pk}) \in \mathcal{D}_{\mathsf{pk}}$ then set $b \leftarrow 1$.

Send the public delayed output b to A.

Fig. 1: Ideal Functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$

4 A²PAKE

We review some basic notions of the Universal Composability model (Canetti [10]). In a nutshell, a protocol Π UC-realizes an ideal functionality $\mathcal{F}$ with setup assumption $\mathcal{G}$ if there exists a PPT simulator S such that no PPT environment $\mathcal{Z}$ can distinguish an execution of the protocol Π which can interact with the setup assumption $\mathcal{G}$ from a joint execution of the simulator S with the ideal functionality $\mathcal{F}$. The environment $\mathcal{Z}$ provides the inputs to all the parties of the protocols, decides which parties to corrupt (we consider static corruption, where the environment decides the corrupted parties before the protocol starts), and schedules the order of the messages in the networks. When specifying an ideal functionality, we use the "delayed outputs" terminology of Canetti [10]. Namely, when a functionality $\mathcal{F}$ *sends a public (resp. private) delayed output* M *to party* P_i we mean that M is first sent to the simulator (resp. the simulator is notified) and then forwarded to P_i only after acknowledgement by the simulator.

4.1 The Ideal Functionality

We are ready to describe the ideal functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$, depicted in Figure 1. In the following we will use "client" and "user" interchangeably. We start by recalling the settings and high-level goals of our primitive. We assume a number of clients $\{C_1, \ldots, C_n\}$, and two non-colluding servers S_1, S_2. As it will become clear later, we could easily extend the ideal functionality to more than two servers, obtaining security as long as one of the servers is honest; however, we decided to describe our primitive with just two servers for the sake of simplicity.

The server S_1 is designated as the "main" server. It is the one that learns the client's public key—so it can verify messages signed by the client—and that turns to an auditor in case of dispute. The other server is designated as "support" server and helps the main one to authenticate users and, most importantly, to produce auditing evidence.

At registration time, each client can submit its password to the ideal functionality, which registers it. Notice that, even if both servers are corrupted, the password is not leaked. Indeed, the only way for an attacker to leak the password is by using the "test password" interface. The ideal functionality notifies both servers whenever the adversary tests a new password, therefore: (i) if at least one of the server is honest, then only "online" brute-force attacks on the password are possible; (ii) if both servers are corrupted, then the attacker can carry on "off-line" brute-force attack on the password. The latter property requires the simulator of our protocol, playing the role of the ideal-model adversary, to be able to detect off-line password tests made by the real-world adversary. However, when both the servers are corrupted, the adversary can carry on the tests locally, namely without sending any message. Thus, this seems to require a non-black-box assumption which would allow the simulator to extract a password test from adversary. Looking ahead, we will follow a proof strategy from [23] by making use of the random oracle model to obtain such extractability property.

We design the ideal functionality to output a fresh key-pair at every run, so to achieve forward-secrecy—namely, the leakage of a password does not compromise the secret keys output by earlier executions. Moreover, as long as either the client C_j or the servers S_1 are honest and the password was not leaked, i.e., $\texttt{corrupt}(\mathsf{sid}, j) \neq 1$, the ideal functionality guarantees correctness, namely, the public key received by the server corresponds to the secret key received by the client. Also notice that the ideal functionality registers the key-pair in the database of key-pairs only when the server S_1 is honest. Indeed, when the server S_1 is corrupted, it can always deny to have executed the protocol.

Finally, the ideal functionality assures non-frameability and non-repudiablity. For the former, an auditor cannot be convinced that a public key belongs to an honest client if that client did not actually produced the key-pair jointly with the servers. This holds as long as the password of the client is not corrupted. We stress that both servers could be malicious but still cannot frame the client, if, for example, the password of the client has high-entropy. For non-repudiability, an honest server with a transcript of an execution with a (possibly malicious) client, can always convince the auditor that the secret key matching the public key in the transcript belongs to that client.

5 UC-secure protocol

5.1 Setup Assumptions

We leverage functionalities $\mathcal{F}_{\texttt{AUTH}}, \mathcal{F}_{\texttt{KRK}}, \mathcal{F}_{\texttt{RO}}$ and $\mathcal{F}_{\texttt{CRS}}$, which model authenticated channels, key-registration, random oracle, and common reference string, respectively. The authenticated channel is used only once by each client at

Functionality $\mathcal{F}_{\texttt{TOPRF}}$:

The functionality is parametrized by a positive integer t and runs with a client C servers S_1, S_2, auditor A and an adversary $\mathcal{A}$. It maintains a table $T(\cdot,\cdot)$ initialized with null entries and a vector $tx(\cdot)$ initialized to 0.

Initialization:

- On $(\texttt{Init}, \mathsf{sid})$ from S_i, $i \in \{1,2\}$ send $(\texttt{Init}, \mathsf{sid}, S_i)$ to the adversary $\mathcal{A}$ and mark S_i active.
- On $(\texttt{Init}, \mathsf{sid}, \mathcal{A}, k)$ from $\mathcal{A}$ check that k is unused and $k \neq 0$ and record $(\mathsf{sid}, \mathcal{A}, k)$ and return $(\texttt{Init}, \mathsf{sid}, \mathcal{A}, k)$ to the adversary $\mathcal{A}$.
- On $(\texttt{InitComplete}, \mathsf{sid}, S_i)$ for $i \in \{1,2\}$ from the adversary $\mathcal{A}$, if S_i is active send $(\texttt{InitComplete}, \mathsf{sid})$ to S_i and mark S_i as *initialized*.

Evaluation:

- On $(\texttt{eval}, \mathsf{sid}, \mathsf{ssid}, x)$ from $\mathcal{P} \in \{C, \mathcal{A}\}$, if tuple $(\mathsf{ssid}, \mathcal{P}, *)$ already exists, ignore. Else, record $(\mathsf{ssid}, \mathcal{P}, x)$ and send $(\texttt{eval}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P})$ to $\mathcal{A}$.
- On $(\texttt{SndrComplete}, \mathsf{sid}, \mathsf{ssid}, i)$ for $i \in \{1,2\}$ from $\mathcal{A}$ ignore if S_i is not initialized. Else set $tx(i)++$ and send $(\texttt{SndrComplete}, \mathsf{sid}, \mathsf{ssid})$ to S_i.
- On $(\texttt{RcvComplete}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, p^*)$ for $\mathcal{P} \in \{C, \mathcal{A}\}$ from $\mathcal{A}$, retrieve $(\mathsf{ssid}, \mathcal{P}, x)$ if it exists, and ignore this message if there is no such tuple or if any of the following conditions fails:
  (i) if $p^* = 0$ then $tx(1) > 0$ and $tx(2) > 0$,
  (ii) if both servers are honest then $p^* = 0$.
  If $p^* = 0$ then set $tx(1)--$ and $tx(2)--$. If $T(p^*, x)$ is null, pick ρ uniformly at random from $\{0,1\}^t$ and set $T(p^*, x) := \rho$. Send $(\texttt{eval}, \mathsf{sid}, \mathsf{ssid}, T(p^*, x))$ to $\mathcal{P}$.

Fig. 2: Ideal Functionality $\mathcal{F}_{\texttt{TOPRF}}$ (adapted from [22]). Label 0 is reserved for the honest execution.

registration time. The key-registration functionality allows to create a PKI between the servers and the auditor, notice that we do not need a *global* PKI. Indeed, looking ahead, in the protocol we just need that the messages signed by the second server could be verified by the auditor, in order to achieve non-repudiation. The common reference string is necessary for the NIZK proof systems that we make use of, while we make use of $\mathcal{F}_{\texttt{RO}}$ for the coin-tossing part of our protocol.

Further, we leverage the ideal functionality $\mathcal{F}_{\texttt{TOPRF}}$ which models a threshold oblivious pseudo-random function. In Figure 2, we present a simplified version of the functionality presented by Jarecki et al. in [22] which fits our purpose. More in details, the ideal functionality $\mathcal{F}_{\texttt{TOPRF}}$ produces uniformly random outputs, even in case of adversarial choice of the involved private key. The functionality maintains a table $T(\cdot,\cdot)$ storing the PRF evaluations and a counter vector $tx(\cdot)$ for each server, used to ensure the involvement of the 2 servers on each completed evaluation. Our protocol makes use of the multi-session extension of the ideal functionality $\mathcal{F}_{\texttt{TOPRF}}$ (that we identify with the *hatted* functionality $\hat{\mathcal{F}}_{\texttt{TOPRF}}$). When the functionality $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ is called we thus include a sub-session identifier ssid. Specifically, on input $(\mathsf{sid}, \mathsf{ssid}, m)$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$, the functionality first checks there is a running copy of $\mathcal{F}_{\texttt{TOPRF}}$ with session identifier ssid and, if so, activates that copy with message m. Otherwise, it invokes a new copy of $\mathcal{F}_{\texttt{TOPRF}}$ with input (ssid, m), and links to this copy the sub-session identifier ssid. For further details, see [13]. In our concrete usage (see Figure 7) there are two layers of executions: the client's index (j) is used as the sub-session identifier when calling $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ (thus in the protocol each client uses a different instance of $\mathcal{F}_{\texttt{TOPRF}}$); the query identifier qid (used in the command `init` of $\mathcal{F}_{\mathsf{A^2PAKE}}$), is used as the sub-sub-session identifier when calling $\hat{\mathcal{F}}_{\texttt{TOPRF}}$.

We consider the ideal functionality $\mathcal{F}_{\texttt{KRK}}$ described in Figure 3 that realizes a key-registration with knowledge, as defined in [12]. This functionality simply chooses (fairly) a key pair consisting of a secret key (which is forwarded to the corresponding invoking registered party) and a public key (forwarded to all parties).

Furthermore, we consider the ideal functionality $\mathcal{F}_{\texttt{AUTH}}$ depicted in Figure 4, that realizes point-to-point authenticated channels between the parties, adapted from [11].

We also use the functionality realizing the common reference string in our setting (see [13]), described in Figure 5, as well as the $\mathcal{F}_{\texttt{RO}}$ described in Figure 6 that realizes a random oracle, as described in [18].

Functionality $\mathcal{F}_{\texttt{KRK}}$:

Given a deterministic key generation function KGen with security parameter λ the functionality $\mathcal{F}_{\texttt{KRK}}$ runs with servers S_1, S_2, auditor A, and adversary $\mathcal{A}$. $\mathcal{F}_{\texttt{KRK}}$ keeps a list R storing registered parties and corresponding keys, which is initially empty. Further, it operates as follows:

Registration: On input $(\texttt{register}, \mathsf{sid}, r)$ from party $\mathcal{P} \in \{S_1, S_2, A, \mathcal{A}\}$, if the request is valid, i.e., $(\mathcal{P}, \cdot, \cdot)$ is not in the list R compute $(PK, SK) := \mathsf{KGen}(r)$ and add the tuple $(\mathcal{P}, PK, SK)$ to R.

Retrieval:

- On input a request $(\texttt{retrieve}, \mathsf{sid}, \hat{\mathcal{P}})$ from $\mathcal{P} \neq \hat{\mathcal{P}} \in \{S_1, S_2, A, \mathcal{A}\}$, it returns the string $(\mathsf{sid}, \hat{\mathcal{P}}, PK)$ to $\mathcal{P}$ if there is a record $(\hat{\mathcal{P}}, PK, SK) \in R$. Otherwise, it returns $(\mathsf{sid}, \hat{\mathcal{P}}, \perp)$ to $\mathcal{P}$.
- On input a request $(\texttt{retrieve}, \mathsf{sid}, \mathcal{P})$ from $\mathcal{P} \in \{S_1, S_2, A, \mathcal{A}\}$, it returns the string $(\mathsf{sid}, \mathcal{P}, PK, SK)$ to $\mathcal{P}$ if there is a record $(\mathcal{P}, PK, SK) \in R$. Otherwise, it returns $(\mathsf{sid}, \mathcal{P}, \perp)$ to $\mathcal{P}$.

Fig. 3: Functionality $\mathcal{F}_{\texttt{KRK}}$. Note that each instance of this functionality can only be invoked by the parties on a single protocol session (i.e., for a fixed sid.).

Functionality $\mathcal{F}_{\texttt{AUTH}}$:

The functionality runs with clients $C_1, \ldots, C_n$, servers S_1, S_2, auditor A, and adversary $\mathcal{A}$. It receives from the environment the set $\mathbb{C}$ of corrupted parties, and let $\mathbb{H}$ be the set of honest parties.

1. For any party B, upon receiving $(\texttt{send}, \mathsf{sid}, A, B, m)$ from party $A \in \mathbb{H}$, it sends $(\texttt{sent}, \mathsf{sid}, A, B, m)$ to $\mathcal{A}$
2. For any party B', upon receiving $(\texttt{send}, \mathsf{sid}, A, B', m')$ from the adversary $\mathcal{A}$ it does the following:
   - If $A \in \mathbb{C}$, outputs $(\texttt{sent}, \mathsf{sid}, A, m')$ to B'.
   - Else, if $A \in \mathbb{H}$, outputs $(\texttt{sent}, \mathsf{sid}, A, m)$ to B.

Fig. 4: Functionality $\mathcal{F}_{\texttt{AUTH}}$

Functionality $\mathcal{F}_{\texttt{CRS}}$:

The functionality is parametrized by a positive integer t and runs with clients $C_1, \ldots, C_n$, servers S_1, S_2, auditor A, and adversary $\mathcal{A}$.

1. Upon receiving a message $(\texttt{CRS}, \mathsf{sid}, \mathcal{P})$ for $\mathcal{P} \in \{S_1, S_2, C_1, \ldots, C_n\}$, choose uniformly at random a value $\{0,1\}^t$ and send $(\texttt{CRS}, \mathsf{sid}, r)$ to $\mathcal{P}$ and to the adversary $\mathcal{A}$. Next, when receiving $(\texttt{CRS}, \mathsf{sid}, \hat{\mathcal{P}})$ from any $\hat{P} \neq \mathcal{P} \in \{S_1, S_2, C_1, \ldots, C_n\}$, send $(\texttt{CRS}, \mathsf{sid}, r)$ to $\hat{\mathcal{P}}$ and $\mathcal{A}$, and halt.

Fig. 5: Functionality $\mathcal{F}_{\texttt{CRS}}$

Protocol for Client C_j:

- On $(\texttt{register}, \mathsf{sid}, \mathsf{pw})$
  send $(\texttt{eval}, \mathsf{sid}, j, \texttt{signup}, \mathsf{pw})$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$,
  send $(\texttt{CRS}, \mathsf{sid}, C_j, S_1)$ to $\mathcal{F}_{\texttt{CRS}}$,
  receive $(\texttt{eval}, \mathsf{sid}, j, \texttt{signup}, \rho)$, from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$
  receive $(\texttt{CRS}, \mathsf{sid}, \mathsf{crs})$ from $\mathcal{F}_{\texttt{CRS}}$
  set $\mathsf{pk}^* = g^\rho$
  send $(\texttt{round1-reg}, \mathsf{sid}, j, \mathsf{pk}^*)$ using $\mathcal{F}_{\texttt{Auth}}$ to S_1 and S_2.
- On $(\texttt{init}, \mathsf{sid}, j, \mathsf{qid})$
  set $\mathsf{sid}' := (\mathsf{sid} \| \mathsf{qid} \| j)$,
  sample $x_C \leftarrow \mathbb{Z}_q$ and set $y_C \leftarrow g^{x_C}$,
  Compute $\pi \leftarrow \mathsf{P}(\mathsf{crs}, \mathsf{sid}', y_C, x_C)$,
  send $(\texttt{RO}, \mathsf{sid}' \| y_C \| \pi)$ and receive $(\texttt{RO}, h)$ from $\mathcal{F}_{\texttt{RO}}$,
  send $(\texttt{round1}, \mathsf{sid}', y_C, \pi)$
- On $(\texttt{round2}, \mathsf{sid}', x_S)$ from S_1,
  set $\mathsf{pk} = g^{x_C \cdot x_S}$
  compute $\sigma_C \leftarrow \mathsf{Sign}(\rho, \mathsf{sid}' \| \mathsf{pk})$,
  send $(\texttt{round3}, \mathsf{sid}', y_C, \pi, h, \mathsf{pk}, \sigma_C)$ to S_1
  output $\mathsf{sk} = x_C \cdot x_S$.

Protocol for Server S_2:

- At first activation sample $s_2 \leftarrow\!\$\ \mathbb{Z}_q$ and
  send $(\texttt{register}, \mathsf{sid}, s_2)$ to $\mathcal{F}_{\texttt{KRK}}$.
- On $(\texttt{register}, \mathsf{sid}, j)$
  send $(\texttt{init}, \mathsf{sid}, j)$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$,
  Wait to receive:
  - $(\texttt{InitComplete}, \mathsf{sid}, j)$ from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$,
  - $(\texttt{SndrComplete}, \mathsf{sid}, j, \texttt{signup})$ from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ and,
  - $(\texttt{round1-reg}, \mathsf{sid}, j, \mathsf{pk}^*)$ from $\mathcal{F}_{\texttt{Auth}}$.

  compute $\sigma_2 \leftarrow \mathsf{Sign}(\mathsf{sk}_2, \mathsf{sid} \| j \| \mathsf{pk}^*)$
  send $(\texttt{round2-reg}, \sigma_2)$ to S_1.
- On $(\texttt{Init}, \mathsf{sid}, \mathsf{qid}, j)$, set $\mathsf{sid}' := (\mathsf{sid} \| \mathsf{qid} \| j)$,
  Wait to receive
  $(\texttt{SndrComplete}, \mathsf{sid}, j, \mathsf{qid})$ from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$.

Protocol for Server S_1:

- On $(\texttt{register}, \mathsf{sid}, j)$ send $(\texttt{Init}, \mathsf{sid}, j)$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$,
  send $(\texttt{CRS}, \mathsf{sid}, C_j, S_1)$ to $\mathcal{F}_{\texttt{CRS}}$
  receive $(\texttt{InitComplete}, \mathsf{sid}, j)$ from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$.
  receive $(\texttt{CRS}, \mathsf{sid}, \mathsf{crs})$ from $\mathcal{F}_{\texttt{CRS}}$
  Wait to receive:
  - $(\texttt{round1-reg}, \mathsf{sid}, j, \mathsf{pk}^*)$ from $\mathcal{F}_{\texttt{Auth}}$;
  - $(\texttt{round2-reg}, \sigma_2)$ from S_2;
  - $(\texttt{SndrComplete}, \mathsf{sid}, j, \texttt{signup})$ from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$.

  Assert $\mathsf{Vf}(pk_2, \mathsf{sid} \| j \| \mathsf{pk}^*, \sigma_2) = 1$ and store $(\mathsf{sid}, j, \mathsf{pk}^*, \sigma_2)$.
- On $(\texttt{init}, \mathsf{sid}, j, \mathsf{qid})$, set $\mathsf{sid}' := (\mathsf{sid} \| \mathsf{qid} \| j)$.
  Wait to receive:
  - $(\texttt{SndrComplete}, \mathsf{sid}, j, \mathsf{qid})$ from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$;

  Sample $x_S \leftarrow\!\$\ \mathbb{Z}_q$ and send $(\texttt{round2}, \mathsf{sid}', x_S)$ to C_j
- On $(\texttt{round3}, \mathsf{sid}', y_C, \pi, h, \mathsf{pk}, \sigma)$ from C_j,
  assert the output $(\texttt{RO}, h')$ of $\mathcal{F}_{\texttt{RO}}$ on input $(\mathsf{sid}' \| y_C \| \pi)$ fulfills $h' = h$ and
  $\mathsf{V}(\mathsf{crs}, \mathsf{sid}', y_C, \pi) = 1$,
  assert $\mathsf{pk} = y_C^{x_S}$
  register the tuple $(\mathsf{sid}, \mathsf{qid}, j, \mathsf{pk})$.
- On $(\texttt{audit}, \mathsf{sid}, \mathsf{qid}, j)$ retrieve $(\mathsf{sid}, j, \mathsf{pk}^*, \sigma_2)$, $(\mathsf{sid}, \mathsf{qid}, j, \mathsf{pk}, \sigma_C)$
  send $(\texttt{audit}, \mathsf{sid}, \mathsf{qid}, j, \mathsf{pk}^*, \sigma_2, \mathsf{pk}, \sigma_C)$ to A.

Protocol for Auditor A:

- On $(\texttt{audit}, \mathsf{sid}, \mathsf{qid}, j, \mathsf{pk}^*, \sigma_2, \mathsf{pk}, \sigma_C)$ from S_1,
  send $(\texttt{retrieve}, \mathsf{sid}, S_2)$ to $\mathcal{F}_{\texttt{KRK}}$ and receive $(\mathsf{sid}, S_2, \mathsf{pk}_2)$,
  compute $b_0 \leftarrow \mathsf{Vf}(\mathsf{pk}_2, \mathsf{sid} \| j \| \mathsf{pk}^*, \sigma_2)$,
  compute $b_1 \leftarrow \mathsf{Vf}(\mathsf{pk}^*, \mathsf{sid} \| \mathsf{qid} \| j \| \mathsf{pk}, \sigma_C)$,
  output $b_0 \wedge b_1$.

Fig. 7: The protocol realizing $\mathcal{F}_{\mathsf{A^2PAKE}}$ for the dlog key-generation algorithm with setup assumption $\mathcal{F}_{\texttt{RO}}, \mathcal{F}_{\texttt{AUTH}}, \mathcal{F}_{\texttt{KRK}}, \mathcal{F}_{\texttt{CRS}}$ and $\hat{\mathcal{F}}_{\texttt{TOPRF}}$.

Functionality $\mathcal{F}_{\texttt{RO}}$:

The functionality is parametrized by a positive integer t and runs with clients $C_1, \ldots, C_n$, servers S_1, S_2, auditor A, and adversary $\mathcal{A}$.

1. $\mathcal{F}_{\texttt{RO}}$ keeps a list L of bitstring pairs, which is initially empty.
2. On input (sid, x), with $x \in \{0,1\}^*$ from any party $\mathcal{P} \in \{C_1, \ldots, C_n, S_1, S_2, A, \mathcal{A}\}$, do:
   - If there is $(x, \hat{h}) \in L$ for some $\hat{h} \in \{0,1\}^t$, set $h := \hat{h}$;
   - else, choose h uniformly at random from $\{0,1\}^t$ and add the entry (x, h) to L.
   - Send (sid, h) to the activating party $\mathcal{P}$.

Fig. 6: Functionality $\mathcal{F}_{\texttt{RO}}$

5.2 Generic description of our protocol

A high-level description of a protocol realizing $\mathcal{F}_{\mathsf{A^2PAKE}}$ from the setup assumptions $\mathcal{F}_{\texttt{RO}}, \mathcal{F}_{\texttt{AUTH}}, \mathcal{F}_{\texttt{KRK}}, \mathcal{F}_{\texttt{TOPRF}}$ and $\mathcal{F}_{\texttt{CRS}}$ follows in Figure 7.

The protocol consists of three phases: *registration*, in which the client registers with the two servers, *authentication*, in which the client and the server S_1 produce a fresh and authenticated key pair, and *audit*, in which the server can prove to the auditor the relation between clients and public keys.

At registration of a new client, the servers initialize a new fresh instance of $\mathcal{F}_{\texttt{TOPRF}}$ by calling $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ with sub-session identifier the index relative to the client. Then, the client and the two servers run the $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ (on sub-sub-session identifier a special string `signup` used for registration), where the client's private input is the password whereas each server uses its secret key share as private input. The client receives the evaluation of the OPRF that is parsed as a secret key $\mathsf{sk}^* \in \mathbb{Z}_q$ for a DLOG-based signature scheme [6]. Finally, the client, using the interfaces provided by $\mathcal{F}_{\texttt{AUTH}}$, can send an authenticated message to both servers with the public key $\mathsf{pk}^* = g^{\mathsf{sk}^*}$. We notice that using the $\mathcal{F}_{\texttt{AUTH}}$ setup assumption for the last step is necessary, to bind a client identity with public key pk^*. Also, the server S_2 signs the public key pk^* produced, thus witnessing the successful registration of the client.

During authentication, a registered client and the two servers run again the instance of the $\mathcal{F}_{\texttt{TOPRF}}$ associated to the client. Once again, the secret input of the client is a password whereas each server inputs the secret key share picked during registration. Thus, the client recovers the secret key sk^*. Concurrently, client and server run a simple coin-tossing protocol to produce a DLOG key pair. Such protocol ensures randomly generated keys. Additionally, the last message of the client, which defines uniquely the key-pair, is authenticated with a signature under the key pk^*. Server S_1 accepts the public key only if it were correctly generated by the client and the signature on that public key verifies under key pk^*.

At auditing time, if server S_1 wants to prove that a public key pk belongs to a client C_j, the server can simply show to the auditor (1) the signature received by S_2 at registration time on pk^* and the client's identity j, and (2) the signature received by the client at authentication time on pk. In this way the auditor checks that S_2 witnessed the registration of pk^* by client C, and that pk was certified by pk^*. More in details, the auditor checks that σ^* is a valid signature of $\mathsf{sid}\|\mathsf{qid}\|j\|\mathsf{pk}$ under key pk^*, and that σ_2 is a valid signature of the message $\mathsf{sid}\|j\|\mathsf{pk}^*$ under key pk_2—the public key of S_2. If both checks succeed, the auditor concludes that pk belongs to client C.

Theorem 1. *Let* KGen *be the algorithm that upon input the description of a group outputs* $\mathsf{pk} = g^{\mathsf{sk}}$ *and* $\mathsf{sk} \leftarrow_{\$} \mathbb{Z}_q$. *The protocol in Fig 7* UC-realizes *the ideal functionality* $\mathcal{F}_{\mathsf{A^2PAKE}}$ *parametrized by* KGen *with setup assumptions* $\mathcal{F}_{\texttt{RO}}, \mathcal{F}_{\texttt{AUTH}}, \mathcal{F}_{\texttt{KRK}}, \mathcal{F}_{\texttt{CRS}}$ *and* $\hat{\mathcal{F}}_{\texttt{TOPRF}}$.

Proof. We start giving a simulator:
Simulator S:
Simulate setup assumptions:

1. Simulate the setup assumption $\mathcal{F}_{CRS}$ by sampling $\mathsf{crs}, \tau \leftarrow_{\$} \mathsf{Init}(1^\lambda)$.
2. Simulate the setup assumption $\mathcal{F}_{RO}$ storing a lazy-sampled random function. Abort in case of collisions.
3. Simulate the setup assumption $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ by following its logic and maintaining a table $T(j, p, x)$. The table contains extra columns for the coordinates of x labeled with the special symbols $\star_1, \ldots, \star_n$.
4. Simulate the setup assumption $\mathcal{F}_{\texttt{KRK}}$, in particular, if S_2 is corrupt, wait for a tuple $(\texttt{register}, \mathsf{sid}, \tilde{\mathsf{sk}}_2)$ and record it. Else sample a random key $\mathsf{sk}_2 \leftarrow_{\$} \mathbb{Z}_q$ and reply with $\mathsf{pk}_2 \leftarrow g^{\mathsf{sk}_2}$ to the `retrieve` queries of the adversary.

Registration:

5. Upon message $(\texttt{register}, \mathsf{sid}, j)$ from $\mathcal{F}_{\mathsf{A^2PAKE}}$ (the client C_j is honest). Simulate the registration phase of the protocol for the client. In particular, simulate perfectly all the messages sent by $C_j, \mathcal{F}_{\texttt{TOPRF}}$, the honest servers and to the adversary $\mathcal{A}$. The adversary $\mathcal{A}$ eventually sends the message
$$(\texttt{RcvComplete}, \mathsf{sid}, j, \mathsf{qid}, C_j, p_j^*).$$
(In particular, if both servers are honest we assume that $p_j^* = 0$.) Simulate perfectly the ideal functionality $\mathcal{F}_{\texttt{Auth}}$ and send to the adversary the message $(\texttt{round1-reg}, \mathsf{sid}, j, \mathsf{pk}^*)$, where $\mathsf{pk}^* \leftarrow g^{T(j, p_j^*, \star_j)}$.
6. If party $C_j \in \mathbb{C}$ sends $(\texttt{eval}, \mathsf{sid}, j, \texttt{signup}, \mathsf{pw})$, simulate the protocol with the malicious client and send the message
$$(\texttt{register}, \mathsf{sid}, \mathsf{pw})$$
to the ideal functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$. Send to C_j the message $(\texttt{eval}, \mathsf{sid}, j, \texttt{signup}, T(j, p^*, \mathsf{pw}))$.
Receive the message $(\texttt{round1-reg}, \mathsf{sid}, j, \mathsf{pk}^*)$ from C_j.

[6] The choice of the signature scheme is arbitrary and taken for the sake of simplicity. Indeed, with minor modifications to the protocol we could use any EUF-CMA secure signature scheme.

7. If the adversary $\mathcal{A}$ instructs $\mathcal{F}_{\texttt{AUTH}}$ to forward the message $(\texttt{round1-reg}, \mathsf{sid}, j, \mathsf{pk}^*)$ to S_2 and $S_2 \in \mathbb{H}$ then compute $\sigma_2 \leftarrow \mathsf{Sign}(\mathsf{sk}_2, \mathsf{sid}\|j\|\mathsf{pk}^*)$ and send the message

$$(\texttt{round2-reg}, \sigma_2)$$

to S_1.

Test Password:

8. On $(\texttt{eval}, \mathsf{sid}, j, \tilde{\mathsf{pw}})$ from the adversary, send $(\texttt{test}, \mathsf{sid}, \mathsf{qid}, j, \tilde{\mathsf{pw}})$ to $\mathcal{F}_{\mathsf{A^2PAKE}}$. If the answer from the ideal functionality is correct for any p such that $T(j, p, \star_j)$ is defined, set $T(j, p, \tilde{\mathsf{pw}}) := T(j, p, \star_j)$. (Simulate the logic of $\hat{\mathcal{F}}_{\texttt{TOPRF}}$, in particular, notify the server S_b using $\mathcal{F}_{\mathsf{A^2PAKE}}$, when the adversary $\mathcal{A}$ sends $(\texttt{SndComplete}, \mathsf{sid}, j, \mathsf{qid}, S_b)$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$.)

Invalid:

9. On $(\texttt{RcvComplete}, \mathsf{sid}, j, \mathsf{qid}, C_j, p^*)$ if $p^* \neq p_j^*$ (recall p_j^* is the label used at registration time) then mark the session $(\mathsf{sid}, \mathsf{qid}, j)$ as invalid.

Corrupted Client and Honest Server 1: The following commands are executed in the scenario where the client C_j is corrupted and the server S_1 is honest. The server S_2 might be honest or corrupted.

10. On $(\texttt{round1}, \mathsf{sid}', \tilde{y})$ from C_j to S_1, look up in the random oracle table for the value $(\mathsf{sid}, \mathsf{sid}'\|\tilde{y}_C\|\tilde{\pi}, \tilde{h})$, check validity of $\tilde{\pi}$, namely, check that $\mathsf{V}(\mathsf{crs}, \mathsf{sid}', \tilde{y}_C, \tilde{\pi}) = 1$, and if so extract $\tilde{\pi}$ and obtain $\tilde{x}_C$, abort if $\tilde{y}_C \neq g^{\tilde{x}_C}$, else store $(\mathsf{sid}', \tilde{y}_C, \tilde{x}_C)$.
11. On $(\texttt{SndComplete}, \mathsf{sid}, j, \mathsf{qid})$ from $\mathcal{A}$ to $\mathcal{F}_{\texttt{TOPRF}}$, send $(\texttt{newkey}, \mathsf{sid}, \mathsf{qid}, j, C_j)$ to $\mathcal{F}_{\mathsf{A^2PAKE}}$ and obtain sk from the ideal functionality. Retrieve $(\mathsf{sid}', \tilde{y}_C, \tilde{x}_C)$ and set $x_S \leftarrow \mathsf{sk}/\tilde{x}_C$ and send $(\texttt{round2}, x_S)$ to C_j.
12. On $(\texttt{round3}, \mathsf{sid}', \tilde{y}_C, \tilde{\pi}, \tilde{\sigma}_C)$ from C_j, simulate by following the procedure described in the protocol. (If all the checks pass then send $(\texttt{newkey}, \mathsf{sid}, \mathsf{qid}, j, S_1)$ to $\mathcal{F}_{\mathsf{A^2PAKE}}$, so that S_1 can obtain its output, else send $(\texttt{invalid}, \mathsf{sid}, \mathsf{qid}, j, S_1)$.
13. On $(\texttt{audit}, \mathsf{sid}, \mathsf{qid}, C_j, \mathsf{pk})$ from the ideal functionality, simulate by following the procedure described in the protocol.

Honest Client and Corrupted Server 1: The following commands are executed in the scenario where the client C_j is honest and the server S_1 is corrupted. The server S_2 might be honest or corrupted.

14. On $(\texttt{init}, \mathsf{sid}, \mathsf{qid}, j, \mathcal{P})$ from $\mathcal{F}_{\mathsf{A^2PAKE}}$ set $\mathsf{sid}' := (\mathsf{sid}\|\mathsf{qid}\|j)$, sample random $\bar{h}$ and store the value $(\mathsf{sid}, \mathsf{sid}'\|\perp\|\perp, \bar{h})$ in the random oracle table. Send $(\texttt{round1}, \mathsf{sid}', \bar{h})$ to S_1. Send $(\texttt{eval}, \mathsf{sid}, j, \mathsf{qid}, C_j)$ to the adversary $\mathcal{A}$ (simulating the setup assumption $\hat{\mathcal{F}}_{\texttt{TOPRF}}$).
15. On $(\texttt{round2}, \tilde{x}_S)$ to C_j from S_1, send $(\texttt{newkey}, \mathsf{sid}, \mathsf{qid}, j, C_j)$ to $\mathcal{F}_{\mathsf{A^2PAKE}}$ and obtain pk from the ideal functionality. Set $\bar{y}_C \leftarrow \mathsf{pk}^{1/\tilde{x}_S}$ and $\bar{\pi} \leftarrow \mathsf{Sim}(\tau, \mathsf{sid}', \bar{y}_C)$. Program the random oracle with the tuple $(\mathsf{sid}, (\mathsf{sid}'\|\bar{y}_C\|\bar{\pi}), \bar{h})$. Instruct $\mathcal{F}_{\mathsf{A^2PAKE}}$ to send the output to C_j.
16. On $(\texttt{RcvComplete}, \mathsf{sid}, j, \mathsf{qid}, C_j, p^*)$ from the adversary (and $(\texttt{round2}, \tilde{x}_S)$ of the previous step arrived), send $(\texttt{round3}, \mathsf{sid}', \bar{y}_C, \bar{\pi}, \bar{\sigma}_C)$, where $\bar{\sigma}_C \leftarrow \mathsf{Sign}(T(j, p^*, \star_j), sid'\|\mathsf{pk})$.
17. On $(\texttt{audit}, \mathsf{sid}, \mathsf{qid}, j, \tilde{\mathsf{pk}}^*, \tilde{\sigma}_2, \tilde{\mathsf{pk}}, \tilde{\sigma}_C)$ from $\mathcal{A}$ to A, send $(\texttt{audit}, \mathsf{sid}, \mathsf{qid}, j, \tilde{\mathsf{pk}})$ to the ideal functionality.
If $\texttt{corrupt}(\mathsf{sid}, j) = 1$ then execute the same as described in the protocol for A (thus obtaining a bit b') and send to $\mathcal{F}_{\mathsf{A^2PAKE}}$ the bit b'.
Else receive the bit b^* from the ideal functionality. Abort if one of the two cases hold:
   - $b^* = 1$ and $\mathsf{Vf}(\tilde{\mathsf{pk}}^*, \mathsf{sid}'\|\tilde{\mathsf{pk}}, \sigma_C) = 0$ or $\mathsf{Vf}(\mathsf{pk}_2, \mathsf{sid}\|j\|\mathsf{pk}^*, \tilde{\sigma}_2) = 0$.
   - C_j not registered and $S_2 \in \mathbb{H}$ and $\mathsf{Vf}(\mathsf{pk}_2, \mathsf{sid}\|j\|\mathsf{pk}^*, \tilde{\sigma}_2) = 1$.

Honest Client and Honest Server 1: The following commands are executed in the scenario where the client C_j is honest and the server S_1 is honest. The server S_2 might or might not be honest.

18. On $(\texttt{init}, \mathsf{sid}, \mathsf{qid}, j, C_j)$ from $\mathcal{F}_{\mathsf{A^2PAKE}}$ simulate $\bar{h}$ and the setup assumption $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ as in Step 14,
19. On $(\texttt{init}, \mathsf{sid}, \mathsf{qid}, j, S_1)$ from $\mathcal{F}_{\mathsf{A^2PAKE}}$, wait for $(\texttt{SndComplete}, \mathsf{sid}, j, \mathsf{qid}, S_1)$, and for $(\texttt{round1}, \mathsf{sid}', \tilde{h})$ from $\mathcal{A}$.
    As in Step 10, compute the tuple $(\mathsf{sid}', \tilde{y}_C, \tilde{x}_C)$.
    If the password of the client is corrupted then send $(\texttt{newkey}, \mathsf{sid}, \mathsf{qid}, j, S_1)$ to $\mathcal{F}_{\mathsf{A^2PAKE}}$ and receive both sk' and pk' (in particular, $\mathsf{sk}' \neq \mathsf{sk}$ w.h.p. from step 17) , retrieve $\tilde{x}_C$ and set $x_S \leftarrow \mathsf{sk}'/\tilde{x}_C$; else sample $x_S \leftarrow_\$ \mathbb{Z}_q$. Send the message $(\texttt{round2}, \mathsf{sid}', x_S)$.
20. On $(\texttt{round2}, \tilde{x}_S)$ to C_j from $\mathcal{A}$, send $(\texttt{newkey}, \mathsf{sid}, \mathsf{qid}, j, C_j)$ to $\mathcal{F}_{\mathsf{A^2PAKE}}$ and obtain pk from the ideal functionality. Proceed as in Step 15. Send the output to C_j.
21. On $(\texttt{round3}, \mathsf{sid}', \tilde{y}_C, \tilde{\pi}, \tilde{\sigma}_C)$, if the session $(\mathsf{sid}, \mathsf{qid}, j)$ is marked as invalid then send $(\texttt{invalid}, \mathsf{sid}, \mathsf{qid}, j, S_1)$. Else execute the same check of S_1 in the protocol. Additionally, set $\tilde{\mathsf{pk}} := \tilde{y}_C^{x_S}$. Let pk be the output of $\mathcal{F}_{\mathsf{A^2PAKE}}$ for S_1, abort if the following condition holds:
    - $\mathsf{pk} \neq \mathsf{pk}$ and $\texttt{corrupt}(\mathsf{sid}, j) = 0$ and $\mathsf{Vf}(\mathsf{pk}^*, \mathsf{sid}' \| \tilde{\mathsf{pk}}, \tilde{\sigma}_C) = 1$

    If the checks pass, send $(\texttt{newkey}, \mathsf{sid}, \mathsf{qid}, j, S_1)$ and send the output to S_1.
22. On $(\texttt{audit}, \mathsf{sid}, \mathsf{qid}, C_j, \mathsf{pk})$ from the ideal functionality, simulate by following the procedure described in the protocol.

Now we proceed with a series of hybrid experiments. The first hybrid $\mathbf{H}_0$ is the ideal world where $\mathcal{Z}$ interacts with the simulator described above, which interacts with the ideal functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$. We let $\epsilon_i(\lambda) := \Pr\left[\mathbf{H}_i(1^\lambda) = 1\right]$. As it is clear from the context, we will simply write ϵ_i, omitting the argument.

Hybrid 1. The hybrid $\mathbf{H}_1$ is the same as $\mathbf{H}_0$ all the inputs/outputs from the environment $\mathcal{Z}$ are recorded by the hybrid. The hybrid $\mathbf{H}_1$ is equivalent to $\mathbf{H}_0$, the changes are only syntactical.

Hybrid 2. The hybrid $\mathbf{H}_2$ is the same as $\mathbf{H}_1$ but:

- Step 9 is not executed anymore.
- In Step 21, send the message $(\texttt{invalid}, \mathsf{sid}, \mathsf{qid}, j, S_1)$ if the checks $\mathsf{Vf}(\mathsf{pk}^*, \mathsf{sid} \| \mathsf{qid} \| j \| \mathsf{pk}, \sigma_C) = 1$ does not hold (instead of checking that the session is invalid).

Lemma 1. $|\epsilon_1 - \epsilon_2| \in \mathsf{negl}(\lambda)$.

Proof. Let $\mathbf{Bad}$ be the event that exists an execution of step 21 in $\mathbf{H}_1$ where the session $(\mathsf{sid}, \mathsf{qid}, j)$ is marked as invalid but $\mathsf{Vf}(\mathsf{pk}^*, \mathsf{sid} \| \mathsf{qid} \| j \| \mathsf{pk}, \sigma_C) = 1$. We have that $|\epsilon_1 - \epsilon_2| \leq \Pr[\mathbf{Bad}]$.

We reduce to the unforgeability of the signature scheme. Consider the following adversary against unforgeability:

Adversary $\mathsf{B}(\mathsf{pk}^*)^{\mathsf{Sign}(\mathsf{sk}^*, \cdot)}$

1. Sample random index $j^* \leftarrow_\$ [n]$, if $C_{j^*} \in \mathbb{C}$ then abort.
2. Run the hybrid $\mathbf{H}_2(1^\lambda)$ but:
    (a) At registration of the client C_{j^*}, namely at step 5, simulate the message of the client by sending $(\texttt{round1-reg}, \mathsf{sid}, j^*, \mathsf{pk}^*, \pi)$ (where pk^* is the public key received by B as input).
    (b) At execution of the protocol for the client C_{j^*}, specifically at step 15, if $p^* = p_j^*$ (the session is valid) then query the signature oracle on message $\mathsf{sid}' \| \mathsf{pk}$, else execute the same as described in the hybrid.
3. When the event $\mathbf{Bad}$ happens, let
$$(\texttt{round3}, \tilde{\mathsf{sid}} \| \tilde{\mathsf{qid}} \| \tilde{j}, \tilde{y}_C, \tilde{\pi}, \tilde{\sigma}_C)$$
be the message received. If $\tilde{j} \neq j^*$ then abort, else set $\tilde{\mathsf{pk}} = \tilde{y}_C^{x_S}$ and output the forgery $\tilde{M} := (\tilde{\mathsf{sid}} \| \tilde{\mathsf{qid}} \| \tilde{j} \| \tilde{\mathsf{pk}}), \tilde{\sigma}_C$.
4. If the hybrid $\mathbf{H}_2$ terminates and the event $\mathbf{Bad}$ did not happen then abort.

From the description of the adversary B, we have that B does not abort with probability at least $\Pr[\mathbf{Bad}]/n$, in fact, the value j^* is independent from the execution of $\mathbf{H}_2$ (as all the public keys for honest clients are uniformly random). Moreover, by the definition of the event $\mathbf{Bad}$, the forgery output by B does verify.

So we need only to show that the forged message was never queried to the signature oracle. If $\tilde{M}$ is equal to one of the messages signed at step15 using the signature oracle then it means that $p^* = p_j^*$, however, this in contradiction with the fact that the session is invalid.

This proves that the event $\mathbf{Bad}$ can happen only with negligible probability.

Hybrid 3. The hybrid $\mathbf{H}_3$ is the same as $\mathbf{H}_2$, but now the output of A in 17 is computed executing the real protocol for A, instead of being the output of the ideal functionality. (We still execute the code of the simulator, so it could still abort.)

Lemma 2. $\epsilon_2 = \epsilon_3$.

Proof. First we notice that if the abort condition of step 17 does hold then two hybrids are equivalent. Thus we can assume that it does not hold. In this case, if $\texttt{corrupt}(\mathsf{sid}, j) = 1$ then the two hybrids behave exactly the same, because the output of the ideal functionality is computed by the simulator, which will follow the specification of the algorithm follow by A. Else, $\texttt{corrut}(\mathsf{sid}, j) = 0$, by definition of the abort condition, in particular by the condition that $b^* = 1$ and $\mathsf{Vf}(\tilde{\mathsf{pk}}^*, \mathsf{sid}' \| \tilde{\mathsf{pk}}, \sigma_C) = 0$ or $\mathsf{Vf}(\mathsf{pk}_2, \mathsf{sid} \| j \| \mathsf{pk}^*, \tilde{\sigma}_2) = 0$, the output to A of the ideal functionality and the output of A in the real protocol must be the same.

Hybrid 4. The hybrid $\mathbf{H}_4$ is the same as $\mathbf{H}_3$ but we remove the abort condition of step 17.

Lemma 3. $|\epsilon_2 - \epsilon_3| \in \mathsf{negl}(\lambda)$.

Proof. By the previous hybrid step, the two hybrids are equivalent unless one of the abort conditions from step 17 occurs. In fact, we can assume that the client C_j is honest, because when the client is corrupted, by step 13, the hybrids behave exactly the same way. Also we can assume that the flag $\texttt{corrupt}(\mathsf{sid}, j) = 0$, in fact, by the definition of step 17, in the case $\texttt{corrupt}(\mathsf{sid}, j) = 1$, the hybrid $\mathbf{H}_3$ would not abort.

Recall that in step 17 the ideal functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$ sends a bit b^*. Specifically, the two hybrids differ if and only:

(1) Either $\mathsf{Vf}(\mathsf{pk}^*, \mathsf{sid}' \| \tilde{\mathsf{pk}}, \sigma_C) = 1$,
(2) or C_j is not registered and $S_2 \in \mathbb{H}$ and $\mathsf{Vf}(\mathsf{pk}_2, \mathsf{sid} \| j \| \tilde{\mathsf{pk}}^*, \tilde{\sigma}_2) = 1$.

Let $\mathbf{Bad}_1$ (resp. $\mathbf{Bad}_2$) be the event that the condition (1) (resp. condition (2)) happens. Therefore we have:

$$|\epsilon_2 - \epsilon_3| \leq \Pr[\mathbf{Bad}_1] + \Pr[\mathbf{Bad}_2].$$

Similarly to Lemma 1 above, we can show $\Pr[\mathbf{Bad}_1] \in \mathsf{negl}(\lambda)$. Let us further argue that $\Pr[\mathbf{Bad}_2]$ is also negligible. We reduce to the unforgeability of the signature scheme used by S_2. Notice we can assume that $S_2 \in \mathbb{H}$, as otherwise $\Pr[\mathbf{Bad}_2] = 0$. Let B be an adversary against the unforgeability of SS defined as follows:

Adversary $\mathsf{B}(\mathsf{pk}_2)^{\mathsf{Sign}(\mathsf{sk}_2, \cdot)}$

1. Run the hybrid $\mathbf{H}_3(1^\lambda)$ but when executing the protocol for the server S_2, specifically at step 7, if the session is valid, query the signature oracle on message $\mathsf{sid} \| j' \| \mathsf{pk}^*$, i.e., proceed as described in the hybrid.
2. When the event $\mathbf{Bad}_2$ happens output $(\mathsf{sid} \| j \| \mathsf{pk}^*), \tilde{\sigma}_2$.
3. If the hybrid $\mathbf{H}_3$ terminates and the event $\mathbf{Bad}_2$ did not happen then abort.

B does not abort with probability $\Pr[\mathbf{Bad}_2]$, also for any j a message of the form $(* \| j \| *)$ is queried by B if and only if C_j gets registered, thus the output of B is a valid forgery.

Hybrid 5. The hybrid $\mathbf{H}_5$ is the same as $\mathbf{H}_4$ but the output of S_1 in step 21 is computed executing the real protocol for S_1 instead of being the output of the ideal functionality. (We execute the code of the simulator, so the simulator, at this step, could still abort.)

Lemma 4. $\epsilon_5 = \epsilon_4$

Proof. If the abort condition of step 21 holds, then the two hybrids are the same. Thus we can assume that the condition does not hold. In this, by the definition of the condition, we have that $\tilde{\mathsf{pk}} = \tilde{y}_S^{x_S}$, namely the output of the server S_1, is equal to pk, the output of the ideal functionality.

Hybrid 6. The hybrid $\mathbf{H}_6$ is the same as $\mathbf{H}_5$ but we remove the abort condition of step 21.

Lemma 5. $|\epsilon_6 - \epsilon_5| \in \mathsf{negl}(\lambda)$.

Proof (Sketch). We can reduce to the unforegeability of the signature scheme. In particular, we can define an adversary B which, similar to Lemma 1, samples at random an honest client, and uses the signature oracle to simulate its signatures, then when the distinguish event happens, it outputs its forgery. Notice that in this case the distinguish event is that, at step 21, the hybrid $\mathbf{H}_5$ would abort. Thus, we have that $\tilde{\mathsf{pk}} \neq \mathsf{pk}$ and $\texttt{corrupt}(\mathsf{sid}, j) = 0$ and $\mathsf{Vf}(\mathsf{pk}^*, \mathsf{sid}' \| \tilde{\mathsf{pk}}, \tilde{\sigma}_C) = 1$. The $\texttt{corrupt}(\mathsf{sid}, j) = 0$ assures that we can simulate correctly the hybrid, while the other two assures that the forgery is indeed valid and for a new message. In fact, the reduction would only sign $\mathsf{sid}' \| \mathsf{pk} \neq \mathsf{sid}' \| \tilde{\mathsf{pk}}$ and messages $\mathsf{sid}'' \| \mathsf{pk}''$ where $\mathsf{sid}'' \neq \mathsf{sid}'$.

Hybrid 7. The hybrid $\mathbf{H}_7$ is the same as $\mathbf{H}_6$ but in step 5 we set $\mathsf{pk}^* \leftarrow T(j, p_j^*, \mathsf{pw})$, instead of $\mathsf{pk}^* \leftarrow T(j, p_j^*, \star_j)$.

Lemma 6. $\epsilon_7 = \epsilon_6$.

Proof. Note that until $\mathcal{A}$ sends the message $(\texttt{eval}, \mathsf{sid}, j, \mathsf{pw})$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ the value $T(j, p_j^*, \mathsf{pw})$ is undefined. Thus, in the simulation, following the logic of $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ (see step 3), this entry will be set by selecting an element in $\mathbb{Z}_p$ uniformly at random. Further, given how the answer to adversarial queries of the form $(\texttt{eval}, \mathsf{sid}, j, \mathsf{pw})$ is simulated (see step 8), indeed once $\mathcal{A}$ sends the message $(\texttt{eval}, \mathsf{sid}, j, \mathsf{pw})$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ $T(j, p_j^*, \mathsf{pw})$ will be set as $T(j, p_j^*, \mathsf{pw}) = T(j, p_j^*, \star_j)$. As a result, the two hybrids are perfectly indistinguishable.

Hybrid 8. The hybrid $\mathbf{H}_8$ is the same as $\mathbf{H}_7$ but, in step 8, it emulates correctly the setup assumption $\hat{\mathcal{F}}_{\texttt{TOPRF}}$. In particular, it does no query the $\mathcal{F}_{\mathsf{A^2PAKE}}$: on receiving a message $(\texttt{eval}, \mathsf{sid}, j, \tilde{\mathsf{pw}})$ from the adversary, it simulates a query $(\texttt{eval}, \mathsf{sid}, \mathsf{qid}, j, \tilde{\mathsf{pw}})$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$, an in turn notifies the server S_b when the adversary $\mathcal{A}$ sends $(\texttt{SndComplete}, \mathsf{sid}, j, \mathsf{qid}, S_b)$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$.)

By the changes introduced in the $\mathbf{H}_7$ and $\mathbf{H}_8$, the functionality $\hat{\mathcal{F}}_{\texttt{TOPRF}}$ is perfectly emulated by now.

Hybrid 9. The hybrid $\mathbf{H}_9$ is the same as $\mathbf{H}_8$ but the output of S_1 in step 12 is computed following the real protocol.

Lemma 7. $\epsilon_9 = \epsilon_8$.

Proof. We first analyze the case when the simulator sends $(\texttt{invalid}, \mathsf{sid}, \mathsf{qid}, j, S_1)$. This case is the simplest, indeed, the simulator performs the same checks that S_1 would do in the real protocol, and if they don't hold send the message.

Secondly, we analyzes the case when the simulator sends $(\texttt{newkey}, \mathsf{sid}, \mathsf{qid}, j, S_1)$. We notice that two hybrids behave the same if at step 10 the simulator aborts. Also, if it does not abort, then we extracted a valid $\tilde{x}_C$. Moreover, in step 11 the simulator sets x_S to be equal to $\mathsf{sk}/\tilde{x}_C$, thus the output of the ideal functionality sent to S_1 is pk, while the output, as computed by the procedure described by S_1, is $\tilde{y}_0^{x_S} = g^{\tilde{x}_C \cdot \mathsf{sk}/\tilde{x}_C} = \mathsf{pk}$.

Hybrid 10. The hybrid $\mathbf{H}_{10}$ is the same as $\mathbf{H}_9$ but, in step 11, we compute $x_S \leftarrow_{\$} \mathbb{Z}_q$.

Lemma 8. $\epsilon_{10} = \epsilon_9$.

Proof. In hybrid $\mathbf{H}_9$ the value x_S is set to be $\mathsf{sk}/\tilde{x}_C$, where $\tilde{x}_C$ is set by the adversary before seeing the value sk (we additionally are assuming that the adversary cannot find collision in the random oracle), and sk is uniformly random. While in $\mathbf{H}_{10}$ the value is sampled uniformly at random. The two distributions are equivalent.

Hybrid 11. The hybrid $\mathbf{H}_{11}$ is the same as $\mathbf{H}_{10}$ but it does not abort in step 10 and 17 if $\tilde{y}_C \neq g^{\tilde{x}_C}$.

We can straight-forwardly reduce to the simulation extractable soundness of the NIZK. Thus:

$$|\epsilon_{11} - \epsilon_{10}| \in \mathsf{negl}(\lambda).$$

Hybrid 12. The hybrid $\mathbf{H}_{12}$ is the same as $\mathbf{H}_{11}$ but, at step 14 (and at step 18) the hybrid computes $y_C \leftarrow g^{\mathsf{sk}/\tilde{x}_S}$. (Recall that by $\mathbf{H}_1$, the hybrid knows all the inputs and outputs of the ideal functionality.)This step is only syntactic and preparatory for the next hybrid step.

Hybrid 13. The hybrid $\mathbf{H}_{13}$ is the same as $\mathbf{H}_{12}$ but at step 14 (and at step 18) the hybrid computes $y_C \leftarrow g^{x_C}$ for $x_C \leftarrow_{\$} \mathbb{Z}_q$. Moreover, at step 20 it sets the output of C_j being $x_C \cdot \tilde{x}_S$, as in the description of the protocol.

Lemma 9. $\epsilon_{13} = \epsilon_{12}$.

Proof. Notice that $\tilde{x}_S$ is sent by the adversary before it can see the outputs of the ideal functionality, thus, y_C set $g^{\mathsf{sk}/\tilde{x}_S}$ is uniformly random.

Also notice that once we starting sampling y_C in this way, the output of y_C is independent of the output provided by the ideal functionality. Moreover, in the previous hybrid steps, we set the output of the server S_1 being computed following the real protocol, thus independent of the outputs provided by the ideal functionality.

Hybrid 14. The hybrid $\mathbf{H}_{14}$ is the same as $\mathbf{H}_{13}$ but it does not program the random oracle in step 15, instead it samples directly y_C (as described in the previous hybrid) and simulate $\bar{\pi}$ in this step.

Notice the simulator simulates the random oracle in a way that there cannot be collisions. Also the value y_C, by the previous hybrid step, it is now computed when the honest client sends the first-round message. Thus the two hybrids are equivalent:

$$\epsilon_{14} = \epsilon_{13}.$$

Hybrid 15. The hybrid $\mathbf{H}_{15}$ is the same as $\mathbf{H}_{14}$ but, instead of the simulator simulating the random oracle, we use directly the ideal functionality $\mathcal{F}_{\texttt{RO}}$.

The two hybrids diverge only when we find a collision in the random oracle $\mathcal{F}_{\texttt{RO}}$. Thus:

$$|\epsilon_{15} - \epsilon_{14}| \in \mathsf{negl}(\lambda)$$

Hybrid 16. The hybrid $\mathbf{H}_{16}$ is the same as $\mathbf{H}_{15}$ but, in step 19, it samples $x_S \leftarrow_{\$} \mathbb{Z}_q$, independently of the variable $\texttt{corrupt}(\mathsf{sid}, j)$.

Lemma 10. $\epsilon_{16} = \epsilon_{15}$.

Proof. By the change introduced in $\mathbf{H}_3$, the output of S_1 is computed exactly as in the real protocol. Moreover, when $\texttt{corrupted}(\mathsf{sid}, j) = 0$, we already have that in both hybrids the value x_S is sampled uniformly at random. On the other hand $\mathbf{H}_{15}$ sets x_S as $\mathsf{sk}'/\tilde{x}_C$ (where $\mathsf{sk}' \neq \mathsf{sk}$). Notice that $\tilde{x}_C$ is set by the adversary before that the value sk' is sampled by the ideal functionality, thus $\tilde{x}_C$ is uniformly random.

Hybrid 17. The hybrid $\mathbf{H}_{17}$ is the same as $\mathbf{H}_{16}$ but all the simulated proofs are computed honestly and the common reference string for the NIZK is sampled in sound mode.

This last step straight-forwardly reduces to the composable zero-knowledge property of the NIZK. Moreover, notice that at this point all the witnesses for the NIZK of the honest clients are known. Thus:

$$|\epsilon_{17} - \epsilon_{16}| \in \mathsf{negl}(\lambda)$$

Finally, we notice that at this point all the steps are executed as in the real protocol, also the simulator does not simulate anymore neither $\mathcal{F}_{\texttt{RO}}$ nor $\hat{\mathcal{F}}_{\texttt{TOPRF}}$. Moreover, the other setup assumption where already simulated perfectly by the simulator (so they can be interchanged). Therefore, the last hybrid is equivalent to an execution of the protocol in the $(\mathcal{F}_{\texttt{RO}}, \mathcal{F}_{\texttt{AUTH}}, \mathcal{F}_{\texttt{KRK}}, \hat{\mathcal{F}}_{\texttt{TOPRF}}, \mathcal{F}_{\texttt{CRS}})$-hybrid world.

5.3 Concrete instantiation

We now describe a concrete instantiation of $\mathcal{F}_{\mathsf{A^2PAKE}}$ that, in turn, leverages the 2HashDH instantiation of $\mathcal{F}_{\texttt{TOPRF}}$, presented in [22]. For concreteness, we use the same cyclic group to generate key-pairs as output by $\mathcal{F}_{\mathsf{A^2PAKE}}$ and to instantiate the underlying distributed OPRF.

Let G be a cyclic group of primer order p with generator g. Also let H, H_1, and H_2 be three hash functions ranging over $\{0,1\}^\ell$, G, and $\mathbb{Z}_q$, respectively. Given an input x and a key k from $\mathbb{Z}_q$, function $f_k(x)$ is defined as $H_2(x, H_1(x)^k)$ (where key k is shared among the servers).

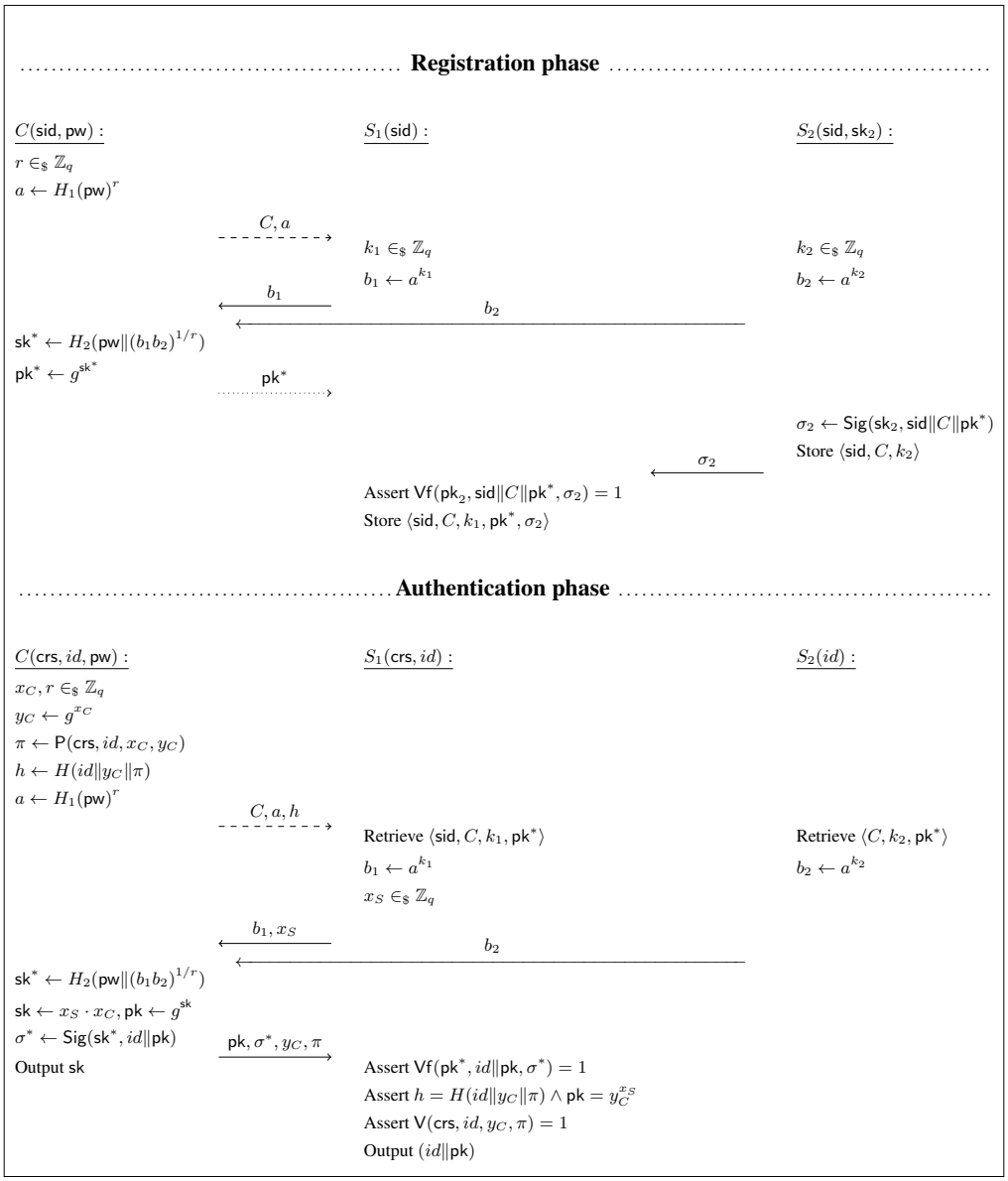
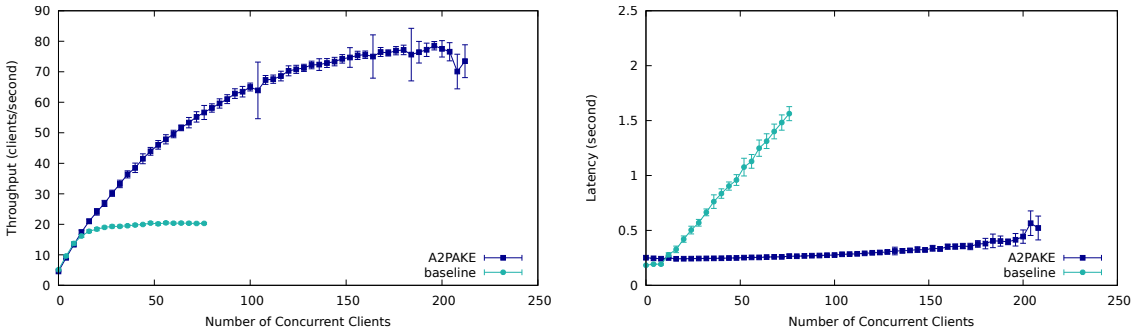


Fig. 8: Concrete instantiation of $\mathsf{A}^2\mathsf{PAKE}$. Dashed arrows depict broadcast messages. The dotted arrow in the registration phase denotes an authenticated channel. In the authentication phase the three parties receive in input an identifier $id = (\mathsf{sid}, C, \mathsf{qid})$.

Figure 8 describes both the registration protocol and the authentication protocol. We substitute the index j of the client from the generic description of the protocol with an unique username. In the registration protocol the client C, and the two servers S_1, and S_2, run the OPRF protocol with private input the password pw, and a freshly sampled key share k_1, and a freshly sampled key share k_2, respectively. Note that k_1 and k_2 are chosen within the protocol for that specific client. The private client's output is set as its secret key sk^* with corresponding public key pk^*. The public key is sent by the client to both servers via an authenticated channel so that pk^* can be bound to a client identity. We do not specify how this channel should be implemented and gave a few example in Section 1. One option is for the client to sign pk^* with its digital ID so to bound the public key to an ID number. Server S_2 signs the public key received by the client and provides S_1 with the signature—thereby providing a witness of a correct client registration. At the end



(a) Throughput.

(b) Latency.

Fig. 9: Comparison of $\mathsf{A}^2\mathsf{PAKE}$ versus the baseline protocol over WAN.

of the registration, the user must only remember his username C and password; S_2 remembers the client's username and the key-share k_2, whereas S_1 stores the tuple $(C, k_1, \mathsf{pk}^*, \sigma)$. In the authentication protocol the distributed OPRF protocol is run (as during registration) so that the client can recover sk^*.

6 Evaluation

We implemented a prototype of the $\mathsf{A}^2\mathsf{PAKE}$ instantiation presented in the previous section. The prototype was written in Python with the Charm-crypto[7] library for cryptographic operations. We instantiated ECDSA over elliptic curve `prime192v1` as the digital signature scheme; thus signature generation and verification take one and two point multiplications, respectively. Further each signature amounts to two elements in $\mathbb{Z}_q$. We used the same curve to instantiate the $\mathsf{2HashDH}$ [22] distributed OPRF. Random oracles were instantiated with SHA256.

Theoretical overhead. We now provide a theoretical evaluation of the computation and communication overhead of the protocols in Figure 8. For elliptic curve operations, we only report the number of point multiplications (mults).

During registration, the client computes 2 mults and 2 hashes; apart from a username, the client sends two group elements and a signature. This signature is the one that S_2 sends to S_1 at the end of the registration protocol in Figure 8; we decide to use the client as a proxy between the two servers because in a real deployment the two servers will not likely have a long-lasting connection (and may not even know their end-points). During authentication, the client computes 6 mults, one signature and 3 hashes; apart from a username, the client sends 3 group elements, a signature, a proof of knowledge of discrete log (one group element and one element in $\mathbb{Z}_q$), and one hash.

At registration time, server S_1 computes a single exponentiation and verifies the validity of a signature; it sends one group element. During authentication, S_1 computes one exponentiation and one hash. It also verifies a signature and a proof-of-knowledge of discrete log (two mults). S_1 sends one group element and one element in $\mathbb{Z}_q$.

During registration, server S_2 computes one exponentiation and one signature; it sends one group element and a signature (to the client that will forward it to S_1). During authentication, S_2 computes one exponentiation and sends one group element.

Baseline comparison. As a baseline comparison for $\mathsf{A}^2\mathsf{PAKE}$, we implement a simple client-server protocol that allows the server to authenticate the client and both of them to generate an asymmetric key pair—where the secret key is only learned by the client. In particular, client and server run a distributed coin-tossing protocol similar to the one of Figure 8 (e.g., the client sends $H(g^{x_C})$ for random x_C, the server sends random x_S, the client sets $\mathsf{sk} = x_C \cdot x_S$ and sends g^{x_C} and $\mathsf{pk} = g^{\mathsf{sk}}$). Further, the client derives a MAC key from the password (by using PBKDF2) and authenticates the public key pk. The server uses the password to derive the same MAC key and accepts pk only if the MAC sent by the client is valid. Note that such protocol does not ensure non-repudiation (since either party could have created and MACed a key-pair), but it is a straightforward example of how to authenticate clients and create fresh key-pairs by using passwords.

[7] `http://charm-crypto.io/`

Experiments. We setup the two servers on two Amazon EC2 machines (t3a.2xlarge instances) and use four laptops (Intel i7-6500U, 16GB RAM) to instantiate clients. With this setup at hand, we measure latency and throughput for $\mathsf{A}^2\mathsf{PAKE}$ and for the basedline protocol. In particular, we send a number of concurrent client requests from the laptops and measure the end-to-end time for a client to complete; we keep increasing the number of requests until the aggregated throughput saturates. Figure 9a and Figure 9b provide average and standard deviations for throughput and latency, respectively. In all of the figures, one data-point is the average resulting from measuring 30 runs.

As expected, latency of $\mathsf{A}^2\mathsf{PAKE}$ (from 0.25s with few clients up to 0.56s right before the main server saturates) is slightly higher than the one of the baseline protocol (from 0.18s with few clients up to 0.50s right before the server saturates). However, the baseline protocol saturates the server with a smaller number of clients compared to $\mathsf{A}^2\mathsf{PAKE}$. Indeed the server of the baseline protocol saturated with roughly 20 concurrent clients. It took more than 70 clients before the main server of $\mathsf{A}^2\mathsf{PAKE}$ saturated.

A closer look at the timings showed that most of the overhead in the baseline protocol is due to the PBKDF2 key derivation function. Indeed, this function is designed to slow-down the computation, in order to discourage brute-force attacks. Replacing this function with, say SHA256, would definitely improve the performance, but would pose passwords at greater risk in case of compromise of the password database, and is discouraged according to PKCS series. Using a two-server settings—as in $\mathsf{A}^2\mathsf{PAKE}$—allows us to use faster hashes like SHA256 because compromise of a single server does make brute-force attacks any easier.

We also measured the time it takes to run the registration protocol of Figure 8. Since this procedure is executed once per client, we are not interested in throughput but just in latency. We use one client laptop and two servers and measure 100 executions. Registration takes 0.14s on average (standard deviation 0.004s); most of the time is taken by network latency, especially because the client must proxy a signature from the secondary server to the main one.

7 Conclusions

Auditable Asymmetric Password Authenticated Public Key Establishment $\mathsf{A}^2\mathsf{PAKE}$ is the only "password-only" protocol that allows users to authenticate to a server and generate a certified asymmetric key pair. The public key is public output while the secret key is private output of only one party (e.g., the user). Further, $\mathsf{A}^2\mathsf{PAKE}$ provides *auditability*, i.e., given a protocol transcript, the key-pair can be ascribed to the identity of the user that obtained the secret key. We have provided a UC-based definition of $\mathsf{A}^2\mathsf{PAKE}$ and a concrete instantiation based on distributed oblivious pseudorandom functions. We have also presented a prototype and its evaluation in realistic settings.

References

1. Michel Abdalla, Olivier Chevassut, Pierre-Alain Fouque, and David Pointcheval. A simple threshold authenticated key exchange from short secrets. In *Asiacrypt*, pages 566–584, 2005.
2. Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. PASTA: password-based threshold authentication. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2042–2059, 2018.
3. Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *ACM Conference on Computer and Communications Security (CCS),*, pages 433–444, 2011.
4. Carsten Baum, Tore K. Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. Pesto: Proactively secure distributed single sign-on, or how to trust a hacked server. Cryptology ePrint Archive, Report 2019/1470, 2019. `https://eprint.iacr.org/2019/1470`.
5. Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated Key Exchange Secure Against Dictionary Attacks. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 139–155. Springer, 2000.
6. Victor Boyko, Philip MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using diffie-hellman. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings*, volume 1807 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2000.
7. Tatiana Bradley, Jan Camenisch, Stanislaw Jarecki, Anja Lehmann, Gregory Neven, and Jiayu Xu. Password-authenticated public-key encryption. In *Applied Cryptography and Network Security - 17th International Conference, (ACNS)*, pages 442–462, 2019.

8. Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. Virtual smart cards: How to sign with a password and a server. In *10th International Conference on Security and Cryptography for Networks (SCN)*, pages 353–371, 2016.
9. Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In *ACM CCS 12*, 2012.
10. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, 2001.
11. Ran Canetti. Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239, 2003. `https://eprint.iacr.org/2003/239`.
12. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85. Springer, 2007.
13. Ran Canetti and Tal Rabin. Universal composition with joint state. Cryptology ePrint Archive, Report 2002/047, 2002. `https://eprint.iacr.org/2002/047`.
14. Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In *24th USENIX Security Symposium*, pages 547–562, 2015.
15. Kristian Gjøsteen and Øystein Thuen. Password-based signatures. In *8th European Workshop on Public Key Infrastructures, Services and Applications (EuroPKI)*, pages 17–33, 2011.
16. Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, (2), 1988.
17. Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In *ASIACRYPT 2006*, 2006.
18. Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In Moni Naor, editor, *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings*, volume 2951 of *Lecture Notes in Computer Science*, pages 58–76. Springer, 2004.
19. Step Over International. Websignatureoffice, 2019. `https://www.websignatureoffice.com/us/`.
20. Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *Asiacrypt*, pages 233–253, 2014.
21. Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing or: How to protect your bitcoin wallet online). In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 276–291, 2016.
22. Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: cost-minimal password-protected secret sharing based on threshold OPRF. In *15th International Conference on Applied Cryptography and Network Security (ACNS)*, pages 39–58, 2017.
23. Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT 2018, Part III*, 2018.
24. Jonathan Katz, Philip D. MacKenzie, Gelareh Taban, and Virgil D. Gligor. Two-server password-only authenticated key exchange. In *Applied Cryptography and Network Security, (ACNS)*, pages 1–16, 2005.
25. Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient Password-Authenticated Key Exchange Using Human-Memorable Passwords. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 475–494. Springer, 2001.
26. Franziskus Kiefer and Mark Manulis. Distributed smooth projective hashing and its application to two-server password authenticated key exchange. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, volume 8479 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2014.
27. Russell W. F. Lai, Christoph Egger, Manuel Reinert, Sherman S. M. Chow, Matteo Maffei, and Dominique Schröder. Simple password-hardened encryption services. In *27th USENIX Security Symposium*, pages 1405–1421, 2018.
28. Russell W. F. Lai, Christoph Egger, Dominique Schröder, and Sherman S. M. Chow. Phoenix: Rebirth of a cryptographic password-hardening service. In *26th USENIX Security Symposium*, pages 899–916, 2017.
29. Ascertia Limited. Signhub, 2019. `https://www.signinghub.com/`.
30. Philip D. MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. *J. Cryptology*, 19(1):27–66, 2006.
31. K. Moriarty, B. Kaliski, and A. Rusch. Pkcs#5: Password-based cryptography specification version 2.1. Technical Report RFC8010, Internet Engineering Task Force (IETF), 2017. https://tools.ietf.org/html/rfc8018.
32. Identidad Electrónica para las Administraciones Gobierno de España. Clave firma, 2019. `https://clave.gob.es/clave_Home/dnin.html`.
33. Mario Di Raimondo and Rosario Gennaro. Provably secure threshold password-authenticated key exchange. *J. Comput. Syst. Sci.*, 72(6):978–1001, 2006.

34. Jonas Schneider, Nils Fleischhacker, Dominique Schröder, and Michael Backes. Efficient cryptographic password hardening services from partially oblivious commitments. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1192–1203, 2016.