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] EURECOM, Sophia Antipolis, France antonio.faonio@eurecom.fr
[2] Universidad Rey Juan Carlos, MACIMTE, Spain mariaisabel.vasco@urjc.es
[3] NEC Laboratories Europe GmbH, Spain, Germany claudio.sorient@neclab.eu

Abstract. Non-repudiation of user messages is a desirable feature in a number of online applications, but it requires digital signatures and certified cryptographic keys. Unfortunately, the adoption of cryptographic keys often results in poor usability, as users must either carry around their private keys (e.g., in a smart-card) or store them in all of their devices. A user-friendly alternative, adopted by several companies and national administrations, is based on so-called "cloud-based PKI certificates". In a nutshell, each user has a certified key-pair 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. However, moving the key-pair from user-private storage to the cloud impairs non-repudiation. In fact, users can always deny having signed a message, by claiming that the signature was produced by the allegedly malicious server without their consent.

In this paper we present Auditable Asymmetric Password Authenticated Public Key Establishment ($\mathsf{A}^2\mathsf{PAKE}$), a cloud-based solution to allow users to manage their signing key-pairs that (i) has the same usability of cloud-based PKI certificates, and (ii) guarantees non-repudiation of signatures. We do so by introducing a new ideal functionality in the Universal Composability framework named $\mathcal{F}_{\mathsf{A}^2\mathsf{PAKE}}$. The functionality is password-based and allows to generate asymmetric key-pairs, where the public key is output to all the parties, but the secret key is the private output of a single one (e.g., the user). Further, the functionality is *auditable*: given a public key output by the functionality, a server can prove to a third party (i.e., a judge) that the corresponding secret key is held by a specific user. Thus, if a user signs messages with the secret key obtained via $\mathsf{A}^2\mathsf{PAKE}$, then signatures are non-repudiable. We provide an efficient instantiation based on distributed oblivious pseudo-random functions for signature schemes based on DLOG. We also develop a prototype implementation of our instantiation and use it to evaluate its performance in realistic settings.

Keywords: Public Key Cryptography, Password Authentication, Server-aided Key Generation, Oblivious Pseudorandom Functions

1 Introduction

Imagine an online application where users sign their messages so that, later on, they can be held accountable. For example, in an online banking application, the banking server

may ask a user to sign her requests to transfer funds to other accounts. Later on, in case of dispute, the server may wish to prove to a third party (i.e., in court) that the user had indeed requested a specific transfer operation.

Most of today's applications, however, do not ask users to sign their messages, mainly because of the usability issues of (certified) cryptographic keys—since users must either carry around their private keys (e.g., in a smart-card) or store them in all of their devices.

The most popular solution to allow users to sign their messages is based on so-called "cloud-based PKI certificates". In a nutshell, each user has a certified key-pair 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. Owing to their usability, cloud-based PKI certificates are used by a number of companies [19,30,35] and national administration authorities [12]. Some of those companies are included as certification authorities—or so called "qualified e-signature service providers"—in the single framework for digital signatures promoted by the European Union eIDAS Regulation [13].

Cloud-based PKI certificates, however, impair non-repudiation of messages signed by a user. In particular, a third party may not be able to tell if signatures issued by the server on behalf of a user, had been actually authorized by that user. Since the server holds a user key-pair and could sign any message on her behalf, that user could deny having signed a message (i.e., by claiming that the signature was produced by the server without her consent).

An ideal solution to enable users to sign their messages and to guarantee non-repudiation would (i) retain the usability of password-only techniques like cloud-based PKI certificates, and (ii) ensure that no server can impersonate or frame a user by issuing a signature on her behalf.

We note that available password-based protocols to derive cryptographic keys are ill-suited for the problem at hand. For example, stand-alone techniques that derive cryptographic keys from low-entropy passwords (e.g., the PBKDF2 key derivation function) are vulnerable to off-line brute-force attacks. In particular, an adversary with access to the user public key could easily enumerate the passwords until it finds the corresponding secret key. Further, online protocols like Password-Authenticated Key Establishment (PAKE) cannot guarantee non-repudiation. In particular, PAKE allows two parties — usually a user and a server — to derive a symmetric key from a common password. As such, if a PAKE-derived key is used for signing messages, one may not ascribe the signature to the user as the same signature may have been produced by the server. Further, by holding the user password, the server can compute the signing key offline and frame the user at will. The same shortcoming holds in case of "asymmetric" PAKE protocols (e.g., [24])—where the server holds a one-way function of the user password: a malicious server could easily (and off-line) brute-force the password and use it to compute the signing key.

1.1 Our Contributions

In this paper we define, instantiate and evaluate a cryptographic protocol named Auditable Asymmetric Password Authenticated Public Key Establishment ($\mathsf{A}^2\mathsf{PAKE}$), that enable users to obtain certified key-pairs. The protocol is "password-only", as users are

only required to remember a (low-entropy) password. At the same time, the protocol ensures non-frameability of honest users. Hence, if keys obtained via $\mathsf{A}^2\mathsf{PAKE}$ are used to sign messages, signatures are non repudiable and users can be held accountable for the messages they sign. Further, the protocol is *auditable*. Namely, a third party with input a public key produced by an execution of $\mathsf{A}^2\mathsf{PAKE}$ and the corresponding protocol transcript, can tell whether the protocol execution went through correctly and produced the alleged public key.

As in threshold PAKE protocols [25,26,27,31], we use multiple servers to ensure that no single server can run an off-line brute-force attack on the user password and obtain her signing key. More in detail, we consider a setting where there are two servers. The "main" server helps users obtain their keys-pairs, whereas the "secondary" server supports its peer in authenticating users and cooperates to produce auditing evidence. As long as one of the servers is honest, (i) the other malicious server cannot run off-line brute-force attacks on user passwords, and (ii) a third party can pinpoint a public key output by the protocol to the user that engaged in that protocol execution. In more detail, we make the following contributions:

New Ideal Functionality. We introduce the $\mathsf{A}^2\mathsf{PAKE}$ ideal functionality. The functionality captures the security requirements of non-repudiation of keys generated by a (possibly malicious) user and non-frameability from a malicious server. $\mathsf{A}^2\mathsf{PAKE}$ allows for the generation of fresh and auditable key-pairs for registered users and provides forward-security, namely, secret keys generated in the past are secure even if the user password is leaked. We defer further discussion to Section 4.

Efficient Universal-Composable Secure Protocol. We provide a protocol that realizes the $\mathsf{A}^2\mathsf{PAKE}$ functionality and prove it secure against static adversaries in the universal composability framework of Canetti [9]. The main ingredient of our protocol is a distributed oblivious pseudo-random function (TOPRF) introduced by Jarecki *et al.* [22]. Roughly speaking, the user inputs their password to the TOPRF protocol to derives a long-term secret key for a signature scheme, which public key was previously certified by the two servers. Once the long-term key is available to the user, we achieve forward-security by running a distributed key-generation protocol so the user obtains a fresh key-pair that she certifies using their long-term key. We defer more details on how we can provide auditability to Section 5.

Implementation. As our last contribution, we provide a prototype implementation written in Python and present the results of an evaluation carried out to assess throughput, latency, and communication overhead. We defer more details to Section 6.

A note on identities. We note that auditing a protocol transcript—i.e., ascribing a public key to an identity—requires establishing and verifying user "identities". In other words, each user registering to the system must prove her identity, so that key-pairs created using $\mathsf{A}^2\mathsf{PAKE}$ can be later attributed to such identity. In our solution, this requirements translate to a registration phase that runs over an authenticated channel; nevertheless, all communication after the registration phase uses unauthenticated channels. We note that many password-based protocols require authenticated channels during user registration [23,31]. Our protocol is, however, agnostic to how the identity of the registering user is established. For example, identities could be bound to an ID by asking the user

to submit a copy of her ID during registration [19,30]. If the registering user holds an eIDs, it can be used to sign the registration transcript with a smart-card reader attached to a PC [12]. 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 her inbox). 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.

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 [32] standard shows how to use passwords to derive symmetric keys to be used for (symmetric) encryption or message authentication. Password-Authenticated Key Exchange (PAKE) [4,5] 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, typically in a threshold manner [25,26,27].

Password-Authenticated Public-Key Encryption (PAPKE) [6] 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 [16] where the signature key is split between the user and a server and the user's share is essentially their password—so that the user can create signatures with the help of the server. We note that in [16] 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 [7], that requires users to carry a personal device such as a smart card. Password-hardening services [14,28,29,34] enable password-based authentication while mitigating the consequences of password database leak. The idea behind these 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 also compromised. PASTA by Agrawal *et al.* [1] and PESTO by Baum *et al.* [3] propose password-based threshold token-based authentication where the role of an identity provider in a protocol such as OAuth [4] is distributed across several parties and the user obtains an authentication token only by authenticat-

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

ing 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 [2,8,20,21,22]. 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 may be used to design a password-based signature scheme (without auditability or forward-security) as follows. During the PPSS registration phase, a user generates a key-pair and secret-shares the signing key across the participating servers. During the PPSS reconstruction phase, that user inputs their password to recover the signing key; the latter could be used to produce signatures that can be verified by whoever holds the corresponding verification key. This design has, however, a number of shortcomings. First, the PPSS functionality [20] does not authenticate users: as shown in [11], lack of user authentication provides a bigger surface for online guessing attacks as the server cannot distinguish adversarial attempts to guess a password from a request to reconstruct the secret by a legitimate user. Similarly, PPSS does not account for auditability. Finally, using PPSS to recover a signing-key does not provide forward-secrecy. That is, if a password is leaked the adversary could forge signatures also for past sessions. Our definition of $\mathsf{A}^2\mathsf{PAKE}$ explicitly allows the main server to authenticate the user, caters for auditability, and ensures forward secrecy.

Looking ahead, we notice that our protocol realizing $\mathsf{A}^2\mathsf{PAKE}$ makes use of an authenticated channel for the registration phase and of a TOPRF, also used in the protocol realizing the PPSS proposed in [21]. Indeed, with some adjustments we could have based our protocol on the PPSS functionality directly, instead of the combination of TOPRF and authenticated channels. However, this might hide an important design choice of our scheme (i.e., the need of an authenticated channel at registration time) without significantly simplifying the proof of security. In our opinion, the protocol presented using OPRF and authenticated channels as primitives is more clear and easy to understand.

3 Preliminaries

Digital Signatures. A signature scheme is a triple of probabilistic polynomial time algorithms $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$. We consider the standard notion of correctness and existential unforgeability under chosen-messages attacks [17].

NIZK 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 $\mathsf{crs} \in \{0,1\}^\lambda$; $\mathsf{P}(\mathsf{crs}, x, w)$, 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*[5], that are NIZKs where P and V additionally

[5] In our protocol the client sends a NIZK proof-of-knowledge of discrete log, to avoid that the adversary re-uses such proofs in different protocol executions we label the NIZKs using the session identifiers of the protocol executions.

take as input a label $L \in \mathcal{L}$ (e.g., a binary string). A NIZK (with labels) is *correct* if for every $\mathsf{crs} \in_{\$} \mathsf{Init}(1^\lambda)$, 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 an oracle that simulates proofs, an adversary cannot produce a valid proof that cannot be extracted. Further we require the NIZK to be adaptive composable zero-knowledge—by now the standard zero-knowledge notion for NIZK, first considered by Groth [18].

Universal Composability. We use the Universal Composability model (Canetti [9]) to define security. As opposed to other game-based definitions, the simulation-based security offered by the UC model allows to formulate security statements that do not need to make any assumptions on the distribution of the passwords (see Canetti *et al.* [10] for further discussion). We review some basic notions of the UC model. In a nutshell, a protocol Π UC-realizes an ideal functionality $\mathcal{F}$ with setup assumption $\mathcal{G}$ if there exists a PPT simulated adversary $\mathcal{B}^*$ (also called the simulator) 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 $\mathcal{B}^*$ 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 [9]. Namely, when a functionality $\mathcal{F}$ *sends a public (resp. private) delayed output* M *to party* $\mathcal{P}_i$ we mean that M is first sent to the simulator (resp. the simulator is notified) and then forwarded to $\mathcal{P}_i$ only after acknowledgement by the simulator. We sometimes say *the ideal functionality* $\mathcal{F}$ *registers the tuple* X *(resp. retrieves* X*)*, in this case we assume that the ideal functionality is stateful and keeps an internal database where stores all the registered tuples.

4 A²PAKE

We start by recalling the settings and high-level goals of our primitive. We assume a number of clients $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, and two servers $\mathcal{S}_1, \mathcal{S}_2$. [6]

The goal is to design a password-only protocol that allows clients to obtain a key-pair that can be used, e.g., to sign messages. Server $\mathcal{S}_1$ is designated as the "main" server. It is the one that learns the client's public key as output by the protocol—so it can verify messages signed by the client. The other server is designated as "support" server and helps the main one to authenticate clients and, most importantly, to produce auditing evidence.

[6] For simplicity, we consider only the two-server scenario, and leave the extension of the ideal functionality to more than two servers as future work.

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

The functionality interacts with clients $\mathcal{C}_1, \ldots, \mathcal{C}_n$, two servers $\mathcal{S}_1$ and $\mathcal{S}_2$, an auditor $\mathcal{A}$, and an adversary $\mathcal{B}$.

Registration: On $(\mathtt{register}, \mathsf{sid}, \mathsf{pw})$ from $\mathcal{C}_j$.
If there is no record of the form $(\mathsf{sid}, C_j, *) \in \mathcal{D}_{\mathsf{pw}}$ then create a fresh record $(\mathsf{sid}, C_j, \mathsf{pw})$ in $\mathcal{D}_{\mathsf{pw}}$.
Send delayed output $(\mathtt{registered}, \mathsf{sid}, C_j)$ to $\mathcal{S}_1, \mathcal{S}_2$.

Init: On $(\mathtt{init}, \mathsf{sid}, j, \mathsf{qid}, \mathsf{pw})$ from a party $\mathcal{P} \in \{\mathcal{C}_j, \mathcal{S}_1, \mathcal{S}_2\}$: //pw is empty if $\mathcal{P} \in \{\mathcal{S}_1, \mathcal{S}_2\}$
Send $(\mathtt{init}, \mathsf{sid}, j, \mathsf{qid}, \mathcal{P})$ to $\mathcal{B}$.
Record that $\mathcal{P}$ initialized the session.
If $\mathcal{C}_j, \mathcal{S}_1$ and $\mathcal{S}_2$ initialized the session, then record the session $(\mathsf{sid}, j, \mathsf{qid})$ is active for $\mathcal{S}_1$ and $\mathcal{C}_j$.
If $\mathcal{P} = \mathcal{C}_j$ and $(\mathsf{sid}, C_j, \mathsf{pw}) \notin \mathcal{D}_{\mathsf{pw}}$ then record the session $(\mathsf{sid}, j, \mathsf{qid})$ is invalid and notify the adversary.
Sample $(\mathsf{pk}, \mathsf{sk}) \in_{\$} \mathsf{KGen}(1^\lambda)$ and register $(\mathsf{sid}, j, \mathsf{qid}, \mathsf{sk}, \mathsf{pk})$.
If $(\mathtt{corrupt}(\mathsf{sid}, j) \vee \mathcal{C}_j \in \mathbb{C})$ then send $(\mathsf{sid}, j, \mathsf{qid}, \mathsf{pk}, \mathsf{sk})$ to the adversary, else send $(\mathsf{sid}, j, \mathsf{qid}, \mathsf{pk})$.

Test Password: On $(\mathtt{test}, \mathsf{sid}, j, \mathsf{qid}, \mathsf{pw}')$ from $\mathcal{B}$.
Assert $\mathcal{S}_1$ and $\mathcal{S}_2$ have initialized the session $(\mathsf{sid}, j, \mathsf{qid})$ or $\mathcal{S}_1, \mathcal{S}_2 \in \mathbb{C}$.
If $(\mathsf{sid}, C_j, \mathsf{pw}') \in \mathcal{D}_{\mathsf{pw}}$ then reply $\mathcal{B}$ with `correct` and set $\mathtt{corrupt}(\mathsf{sid}, j) \leftarrow 1$,
else set the session $(\mathsf{sid}, j, \mathsf{qid})$ as invalid and reply $\mathcal{B}$ with `wrong`.

New Key: On message $(\mathtt{newkey}, \mathsf{sid}, j, \mathsf{qid}, \mathcal{P}, \tilde{\mathsf{pk}}, \tilde{\mathsf{sk}})$ from $\mathcal{B}$.
Assert $\mathcal{P} \in \{\mathcal{C}_j, \mathcal{S}_1\}$ and that session $(\mathsf{sid}, j, \mathsf{qid})$ is active for $\mathcal{P}$;
Mark session $(\mathsf{sid}, j, \mathsf{qid})$ for $\mathcal{P}$ as finalized;
If $\mathcal{P} \in \mathbb{C} \vee \mathtt{corrupt}(\mathsf{sid}, j)$ then set $(\mathsf{pk}, \mathsf{sk}) := (\tilde{\mathsf{pk}}, \tilde{\mathsf{sk}})$ else retrieve $(\mathsf{sid}, j, \mathsf{qid}, \mathsf{pk}, \mathsf{sk})$.
If the session $(\mathsf{sid}, j, \mathsf{qid})$ is invalid set $\mathsf{pk} = \perp$,
if $\mathcal{P} = \mathcal{S}_1$ and session valid then register $(\mathsf{sid}, \mathsf{qid}, C_j, \mathsf{pk})$ in $\mathcal{D}_{\mathsf{pk}}$;
If $\mathcal{P} = \mathcal{S}_1$ then send $(\mathtt{output}, \mathsf{sid}, j, \mathsf{qid}, \mathsf{pk})$ to $\mathcal{S}_1$;
If $\mathcal{P} = \mathcal{C}_j$ then send $(\mathtt{output}, \mathsf{sid}, j, \mathsf{qid}, \mathsf{sk})$ to $\mathcal{C}_j$;

Invalid: On message $(\mathtt{invalid}, \mathsf{sid}, j, \mathsf{qid}, \mathcal{P})$ from $\mathcal{B}$ and $\mathcal{P} \in \{\mathcal{C}_j, \mathcal{S}_1\}$.
Send $(\mathtt{output}, \mathsf{sid}, j, \mathsf{qid}, \perp)$ to $\mathcal{P}$ and mark the session finalized for $\mathcal{P}$.

Audit: On message $(\mathtt{audit}, \mathsf{sid}, \mathsf{qid}, j, \mathsf{pk})$ from party $\mathcal{S}_1$.
NON-FRAMEABILITY. If $\mathcal{C}_j, \mathcal{S}_2 \in \mathbb{H} \wedge \neg\mathtt{corrupt}(\mathsf{sid}, j) \wedge (\mathsf{sid}, \mathsf{qid}, C_j, \mathsf{pk}) \notin \mathcal{D}_{\mathsf{pk}}$ then set $b \leftarrow 0$.
NON-REPUDIABILITY. If $\mathcal{S}_1 \in \mathbb{H} \wedge \mathcal{C}_j \in \mathbb{C} \wedge (\mathsf{sid}, \mathsf{qid}, C_j, \mathsf{pk}) \in \mathcal{D}_{\mathsf{pk}}$ then set $b \leftarrow 1$.
All other cases wait for a bit b' from the adversary $\mathcal{B}$ and set $b \leftarrow b'$.
Send to the auditor $\mathcal{A}$ the message $(\mathtt{audit}, \mathsf{sid}, \mathsf{qid}, j, \mathsf{pk}, b)$.

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

Obtained key-pairs should be forward secure, i.e., leakage of a password should not compromise key-pairs computed before leakage took place. Further, the protocol should be audible. That is, any third party should be able to tell, given a protocol transcript π and the corresponding public key pk, if the protocol execution went through correctly and should be able to ascribe the public key to the user, say u, involved in π. Thus, if signatures verify with respect to the public key pk, then u cannot repudiate signed messages.

The ideal functionality that realizes $\mathsf{A^2PAKE}$, namely $\mathcal{F}_{\mathsf{A^2PAKE}}$, is depicted in Fig. 1. The functionality is parameterised by a security parameter λ and an asymmetric public

key generation algorithm KGen. It receives from the environment the set $\mathbb{C}$ of corrupted parties. Let $\mathbb{H}$ be the set of honest parties (that is, the complement of $\mathbb{C}$). Let $\mathcal{D}_{\mathsf{pk}}$ (resp. $\mathcal{D}_{\mathsf{pw}}$) be the database of the completed sessions (resp. of registered passwords). Both $\mathcal{D}_{\mathsf{pk}}$ and $\mathcal{D}_{\mathsf{pw}}$ are initialized to empty.

The **Registration** interface allows a client to register her passwords pw. Notice that, even if both servers are corrupted, the password is not leaked. Similarly to the notion of strong asymmetric PAKE [24], the only way for an attacker to leak the password is by using the "test password" interface. Note that each password is registered together with a sid and an identifier for the client, which will thereafter thus be used as an identifier for the password that is linked to the client after registration.

Any party can initialize a session via the **Init** interface. In case **Init** is called by a client, it must also submit her password; in case the password has not been registered, the session is marked as invalid. Once a client and the two servers have initialized the same session (identified by qid), the session is marked as active and a fresh key pair is sampled, if either the password or the client are corrupted then the secret key is revealed to the adversary. Note that qid is actually an identifier linked to a concrete signing key generation session; the complete tuple $\mathsf{sid}\|\mathsf{qid}\|j$ will be used later as a global session identifier (for instance, as input to the **Audit** phase.)

The ideal functionality needs inputs from both servers whenever the adversary uses the **Test Password** interface, which captures password guessing attacks. That is, the adversary must wait that the honest server initializes a new session to be able to test a password during such session), 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 out "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 out 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 the adversary. PAKEs based on OPRFs, such as [24], face similar extractability issues. To the best of our knowledge, the random oracle model offers the most natural solution to these issues.

The **New Key** interface accepts inputs from the adversary of the form

$$(\mathtt{newkey}, \mathsf{sid}, j, \mathsf{qid}, \mathcal{P}, \tilde{\mathsf{sk}}, \tilde{\mathsf{pk}}).$$

As a result, a signing key pair will be registered for client j linked to the corresponding identifier $\mathsf{sid}, j, \mathsf{qid}$. This new key generation can be influenced by the adversary in different ways: (1) it can decide when the parties receive the outputs, as we assume that the adversary has complete control over the network, (2) when the client is corrupted or its password is corrupted (resp. the server is corrupted) then the adversary can decide the outputs of the functionality for the client (resp. the server). In this case the output of the client $\mathcal{C}_j$ (resp. the corrupted server) at session $(\mathsf{sid}, \mathsf{qid}, j)$ is $\tilde{\mathsf{sk}}$ (resp. $\tilde{\mathsf{pk}}$). We stress that this is unavoidable when the parties are corrupted, as the adversary has full control of them, and thus of their outputs. When the password is corrupted but the client is honest, for simplicity, we let the adversary decide the output of the client anyway because the

security properties of our functionality cannot be guaranteed, since the adversary could impersonate the client.

Notice that by design, the functionality guarantees a form of forward-secrecy for the generated secret keys: even if a client password is leaked by the adversary, the key-pairs output before password leakage took place are still unknown to the adversary. We elaborate further on this at the end of this section. Also notice that the ideal functionality registers the key-pair in the database of key-pairs only when the server $\mathcal{S}_1$ is honest. Indeed, when the server $\mathcal{S}_1$ 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. Technically, both non-frameability and non-repudiability are enforced by the ideal functionality by maintaining the database $\mathcal{D}_{\mathsf{pk}}$ of the registered public keys. For non-frameability, the ideal functionality makes the auditor $\mathcal{A}$ output 0 whenever the client is honest (and the its password is not leaked) and the alleged public key is not present in the database of the registered public keys. For non-repudiability the ideal functionality makes the auditor output 1 whenever the server is honest and the tuple client/associated public key is present in database (with the corresponding $\mathsf{sid}||\mathsf{qid}$ tags). In all the other cases, the adversary can set the output as it prefers. The reason is that non-frameability is a property aiming to protect an honest client, while non-repudiability cannot hold when a malicious server refutes to engage in the audit protocol.

A note on forward-secrecy. As we mentioned, the functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$ guarantees a form of forward-secrecy, as key-pairs output before password leakage took place are still private. Yet, as our envisioned use of the functionality is to generate keys for a digital signature scheme, forward-secrecy alone would not be enough to prevent an adversary from "back-dating" a signature. In particular, an adversary that has learned the password of a honest client at time t, may still obtain a fresh key-pair, and sign messages with an arbitrary date (e.g, earlier than t). As in any forward-secure signature scheme, we need a mechanism to bound a key-pair to an "epoch", so that only signatures that verify under the public key of the current epoch are considered valid. Such mechanism may be realized using the field qid of the command `init` of the functionality. In particular, some well-defined bits of qid could be used to encode the current epoch—notice that the three parties $\mathcal{C}_j, \mathcal{S}_1$ and $\mathcal{S}_2$ must agree on the qid (thus the epoch) to successfully conclude a key generation and that the generated public key is binded to qid. Finally, the verification procedure of the signature scheme would have to check the association between the public key and the epoch, which we can achieve using the audit interface offered by the functionality.

Our functionality can guarantees forward-secrecy even when both the servers are corrupts. The main reason is that our functionality assumes that only the clients can

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

The functionality is parametrized by a positive integer t and runs with a client $\mathcal{C}$ servers $\mathcal{S}_1, \mathcal{S}_2$, 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 $\mathcal{A}$,
  - mark $\mathcal{S}_i$ active.
- On $(\texttt{Init}, \mathsf{sid}, \mathcal{A}, k)$ from $\mathcal{A}$
  - check that k is unused and $k \neq 0$
  - record $(\mathsf{sid}, \mathcal{A}, k)$
  - 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 $\mathcal{S}_i$ is active
  - send $(\texttt{InitComplete}, \mathsf{sid})$ to S_i
  - 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 $tx(i) := tx(i) + 1$ and send $(\texttt{SndrComplete}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{S}_i$.
- On $(\texttt{RcvComplete}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, p^*)$ for $\mathcal{P} \in \{\mathcal{C}, \mathcal{A}\}$ 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 $tx(1) := tx(1) - 1$ and $tx(2) := tx(2) - 1$.
  - 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.

register themselves. However, looking ahead in our protocol, we need to assume an authenticated channel to enforce such a strong condition. We notice that if we additionally let the adversary register the clients, we would loose forward-secrecy when the servers are both corrupts, since the adversary could re-register a client and could produce a valid key for the attacked client for a given epoch. The only way to avoid this generic attack seems to carefully define forward-secrecy in our context, which we defer for future works.

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 (see the full version of the paper for their formal definitions). The authenticated channel is used only once

by each client at registration time. The key-registration functionality allows to create a PKI between the servers and the auditor. Note that we do not need a *global* functionality for the PKI functionality. Indeed, in our protocol we just need that the messages signed by the second server could be verified by the first server during registration time and by the auditor to achieve non-repudiation. We need a common reference string for the NIZK that we make use of, while we use $\mathcal{F}_{\mathtt{RO}}$ for the coin-tossing part of our protocol.

Additionally and more crucially, we use a threshold oblivious pseudo-random function, formalized by the ideal functionality $\mathcal{F}_{\mathtt{TOPRF}}$. In Fig. 2, we present a simplified version of the functionality of Jarecki et al. in [22] which fits our purpose. The ideal functionality $\mathcal{F}_{\mathtt{TOPRF}}$ produces uniformly random outputs, even in case of adversarial choice of the involved private key and also 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. In particular, note that in Fig. 2 the adversary may initialize servers with a fixed key of his choice through `Init` or without choosing the key himself (using `InitComplete`). The counter is thus controled by him through `SndrComplete`, while the PRF evaluation is completed (and possibly sent to the client) in `RcvComplete`.

Our protocol makes use of the multi-session extension of the ideal functionality $\mathcal{F}_{\mathtt{TOPRF}}$ (that we identify with the *hatted* functionality $\hat{\mathcal{F}}_{\mathtt{TOPRF}}$). When the functionality $\hat{\mathcal{F}}_{\mathtt{TOPRF}}$ is called we thus include a sub-session identifier ssid. Specifically, on input $(\mathsf{sid}, \mathsf{ssid}, m)$ to $\hat{\mathcal{F}}_{\mathtt{TOPRF}}$, the functionality first checks there is a running copy of $\mathcal{F}_{\mathtt{TOPRF}}$ with session identifier ssid and, if so, activates that copy with message m. Otherwise, it invokes a new copy of $\mathcal{F}_{\mathtt{TOPRF}}$ with input (ssid, m), and links to this copy the sub-session identifier ssid. For further details, see [9]. In our concrete usage (see Figs. 3 and 4) there are two layers of executions: the client's index (j) is used as the sub-session identifier when calling $\hat{\mathcal{F}}_{\mathtt{TOPRF}}$ (thus in the protocol each client uses a different instance of $\mathcal{F}_{\mathtt{TOPRF}}$); 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 $\hat{\mathcal{F}}_{\mathtt{TOPRF}}$.

5.2 Generic description of our protocol

An high-level description of our protocol realizing $\mathcal{F}_{\mathsf{A^2PAKE}}$ from the setup assumptions $\mathcal{F}_{\mathtt{RO}}, \mathcal{F}_{\mathtt{AUTH}}, \mathcal{F}_{\mathtt{KRK}}, \mathcal{F}_{\mathtt{TOPRF}}$ and $\mathcal{F}_{\mathtt{CRS}}$ is given in Figs. 3 to 5. The protocol consists of three phases: *registration*, in which the client registers with the two servers, *authentication*, in which the client and the server $\mathcal{S}_1$ 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 $\mathcal{F}_{\mathtt{TOPRF}}$ by calling $\hat{\mathcal{F}}_{\mathtt{TOPRF}}$ with sub-session identifier the index relative to the client. Then, the client and the two servers run the $\hat{\mathcal{F}}_{\mathtt{TOPRF}}$ (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 [7]. The client, using the interfaces provided by $\mathcal{F}_{\mathtt{AUTH}}$, can send an authenticated

[7] 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.

Registration Phase

Protocol for Client $\mathcal{C}_j$:

– On $(\texttt{register}, \mathsf{sid}, \mathsf{pw})$
- send $(\texttt{eval}, \mathsf{sid}, j, \texttt{signup}, \mathsf{pw})$ to $\hat{\mathcal{F}}_{\texttt{TOPRF}}$
- send $(\texttt{CRS}, \mathsf{sid}, \mathcal{C}_j)$ to $\mathcal{F}_{\texttt{CRS}}$
- Wait to receive:
  - $(\texttt{CRS}, \mathsf{sid}, \mathsf{crs})$ from $\mathcal{F}_{\texttt{CRS}}$
  - $(\texttt{eval}, \mathsf{sid}, j, \texttt{signup}, \rho)$ from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$
- set $\mathsf{pk}^* = g^\rho$ and $\mathsf{sk}^* = \rho$
- send $(\texttt{round1} - \texttt{reg}, \mathsf{sid}, j, \mathsf{pk}^*)$ using $\mathcal{F}_{\texttt{AUTH}}$ to $\mathcal{S}_1$ and $\mathcal{S}_2$

Protocol for Server $\mathcal{S}_1$:

– 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 $\mathcal{F}_{\texttt{CRS}}$
- wait to receive:
  - $(\texttt{CRS}, \mathsf{sid}, \mathsf{crs})$ from $\mathcal{F}_{\texttt{CRS}}$
  - $(\texttt{InitComplete}, \mathsf{sid}, j)$ from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$
  - $(\texttt{SndrComplete}, \mathsf{sid}, j, \texttt{signup})$ from $\hat{\mathcal{F}}_{\texttt{TOPRF}}$
  - $(\texttt{round1} - \texttt{reg}, \mathsf{sid}, j, \mathsf{pk}^*)$ from $\mathcal{F}_{\texttt{AUTH}}$
  - $(\texttt{round2} - \texttt{reg}, \sigma_2)$ from $\mathcal{S}_2$
- assert $\mathsf{Vf}(pk_2, \mathsf{sid}\|j\|\mathsf{pk}^*, \sigma_2) = 1$ and store $(\mathsf{sid}, j, \mathsf{pk}^*, \sigma_2)$

Protocol for Server $\mathcal{S}_2$:

– At first activation
- sample $s_2 \in_\$ \mathbb{Z}_q$
- 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 $\hat{\mathcal{F}}_{\texttt{TOPRF}}$
  - $(\texttt{round1} - \texttt{reg}, \mathsf{sid}, j, \mathsf{pk}^*)$ from $\mathcal{F}_{\texttt{AUTH}}$
- compute $\sigma_2 \leftarrow \mathsf{Sign}(\mathsf{sk}_2, \mathsf{sid}\|j\|\mathsf{pk}^*)$
- send $(\texttt{round2} - \texttt{reg}, \sigma_2)$ to $\mathcal{S}_1$.

Fig. 3: Part of the protocol realizing the registration phase of the functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$

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^*, moreover, this is the only step where we use an authenticated channel. Finally, the server $\mathcal{S}_2$ signs the public key pk^* produced and sends such signature to $\mathcal{S}_1$, 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

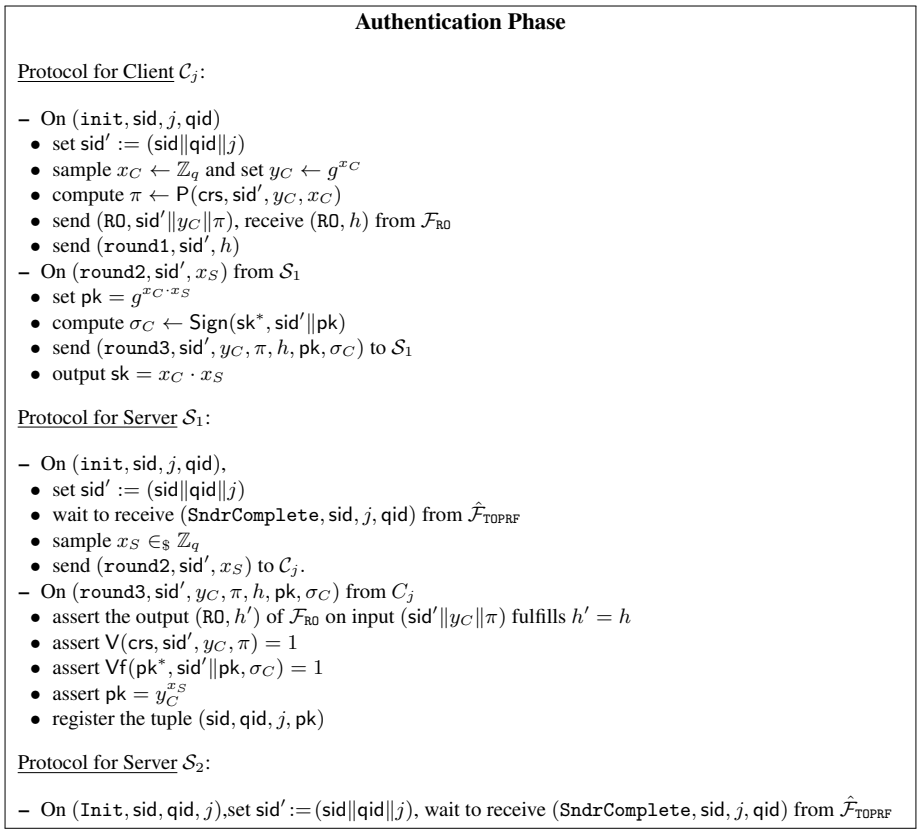


Fig. 4: Part of the protocol realizing the authentication of $\mathcal{F}_{\mathsf{A^2PAKE}}$.

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 $\mathcal{S}_1$ 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 $\mathcal{S}_1$ wants to prove that a public key pk belongs to a client $\mathcal{C}_j$, the server can simply show to the auditor (1) the signature received by $\mathcal{S}_2$ 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 $\mathcal{S}_2$ witnessed the registration of pk^* by client $\mathcal{C}$, and that pk was certified by pk^*. More in detail, the auditor checks that σ_C 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 $\mathcal{S}_2$. If both checks succeed, the auditor concludes that pk belongs to client $\mathcal{C}$.

Theorem 1. *Let* KGen *be the algorithm that upon input the description of a group outputs* $\mathsf{pk} = g^{\mathsf{sk}}$ *and* $\mathsf{sk} \in_{\$} \mathbb{Z}_q$*. The protocol described in Figs. 3 to 5* UC-realizes

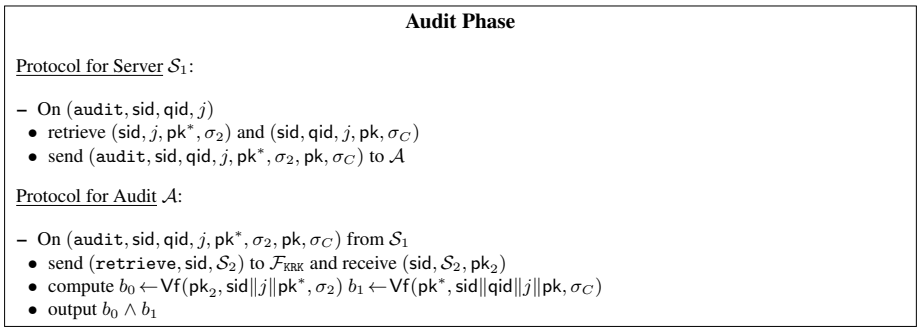


Fig. 5: Part of the protocol realizing the audit of $\mathcal{F}_{\mathsf{A^2PAKE}}$.

the ideal functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$ parametrized by KGen against static adversaries with setup assumptions $\mathcal{F}_{\texttt{RO}}, \mathcal{F}_{\texttt{AUTH}}, \mathcal{F}_{\texttt{KRK}}, \mathcal{F}_{\texttt{CRS}}$ and $\hat{\mathcal{F}}_{\texttt{TOPRF}}$.

We give some intuitions behind the proof the formal proof is in the full version [15]. To prove security we need to show a simulated adversary $\mathcal{B}^*$ that interacts with the ideal functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$ and an environment $\mathcal{Z}$, such that the environment cannot distinguish such interaction with an interaction with the real protocol. First we show how non-frameability and no-repudiability are guaranteed in the real world as they are in the ideal world.

Non-Frameability. Suppose that $\mathcal{Z}$ instructs the corrupted server $\mathcal{S}_1$ to engage the audit phase, $\mathcal{Z}$ feeds the server with an input $(\mathsf{sid}, \mathsf{qid}, C_j, \tilde{\mathsf{pk}})$ and the client $\mathcal{C}_j$ is honest. Moreover, the public key $\tilde{\mathsf{pk}}$ was never produced as the output of an interaction between the client $\mathcal{C}_j$ and the servers. Thus, unless the password of the client $\mathcal{C}_j$ was corrupted, in the ideal world, the auditor will surely output 0. We claim that, unless the password of the client was corrupted, also in the real world, the auditor will output 0 with overwhelming probability. The reason is that, by the security of the $\mathcal{F}_{\texttt{TOPRF}}$ the secret key $\mathsf{sk}^* = \texttt{TOPRF}(\mathsf{pw})$ associated to the client $\mathcal{C}_j$ is known only by the honest client $\mathcal{C}_j$. Notice that, not even the environment knows the secret key since it does not have direct access to the outputs of $\mathcal{F}_{\texttt{TOPRF}}$ during a protocol execution. However, the environment could send an `eval` commands to the OPRF (for example through a man-in-the-middle attack) with the password of the client and obtain the secret key. In this case our simulator sends the same password to the test interface of $\mathcal{F}_{\mathsf{A^2PAKE}}$, and in case of a correct guess reveal the secret key of the signature scheme to the environment. Thus the environment gets to know the secret key only when the password is corrupted. In the case where the password was not corrupted the environment does not know the secret key, the only way to make the auditor accept is to forge a signature, which we can reduce to the existential unforgeability of the signature scheme.

Non-Repudiability. The honest server can make the auditor output 1 in the real world by sending a valid signature on the tuple client/public key. In the ideal world, the auditor outputs 1 if the tuple was registered by the ideal functionality. Thus, the simulator needs

to enforce that a new public key is recorded in the database of the public key of the ideal functionality only when the malicious client sends the message `round3` which contains a valid signature on the client-public-key tuple.

Equivocate, Extract Inputs and Simulate Key Generation. Equivocate and extract the inputs during registration phase is rather straightforward. Indeed, the inputs of the clients are directly sent to the ideal functionality $\mathcal{F}_{\mathtt{TOPRF}}$. The most interesting part is to make sure the key pair output by the ideal functionality $\mathcal{F}_{\mathsf{A^2PAKE}}$ does agree with the transcript generated by the simulator. Recall that in this part of the protocol first the client sends $h = \mathtt{RO}(\mathsf{sid}' \| y_c \| \pi)$, then the server sends x_S and finally the client sends (y_c, π). When the client is honest and the server $\mathcal{S}_1$ is malicious, the simulator chooses the value y_C adaptively once received both the message x_S from the server and the public key from the ideal functionality. Notice it can do so by programming the random oracle and by simulating the NIZK of knowledge of x_C. When the client is malicious and the server $\mathcal{S}_1$ is honest, we can extract the value x_C from the client thanks to the extractability of the NIZK and the observability of the random oracle, and then simulate x_S setting it to be $\mathsf{sk} - x_C$.

5.3 Concrete instantiation

We now describe a concrete instantiation of $\mathcal{F}_{\mathsf{A^2PAKE}}$ that generated DLOG-based key-pairs and that is based on the 2HashDH instantiation of $\mathcal{F}_{\mathtt{TOPRF}}$, 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 prime 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. 6 describes the full protocol. Note that, for clarity, Fig. 6 assumes a direct communication channel between $\mathcal{S}_1$ and $\mathcal{S}_2$. In a real deployment scenario those two parties may not have a long-lasting connection and the client may proxy messages from one server to another; the latter settings is the one we have used in the evaluation of Section 6. We substitute the index j of the client from the generic description of the protocol with a unique username. During registration, client $\mathcal{C}$ and servers $\mathcal{S}_1$, and $\mathcal{S}_2$, run the OPRF protocol. The private input for the client is password pw, while the private input for server $\mathcal{S}_i$ is a freshly sampled key share k_i (for $i \in \{1,2\}$). 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 examples 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 $\mathcal{S}_2$ signs the public key received by the client and provides $\mathcal{S}_1$ with the signature—thereby providing a witness of a correct client registration. At the end of the registration, the user must only remember username $\mathcal{C}$ and password pw; $\mathcal{S}_2$ remembers the client's username and the key-share k_2, whereas $\mathcal{S}_1$ stores the tuple $(\mathcal{C}, k_1, \mathsf{pk}^*, \sigma_2)$. During authentication, the distributed OPRF protocol is run (as during registration) so that the client can recover sk^*. We consider the audit phase as a non-interactive procedure where

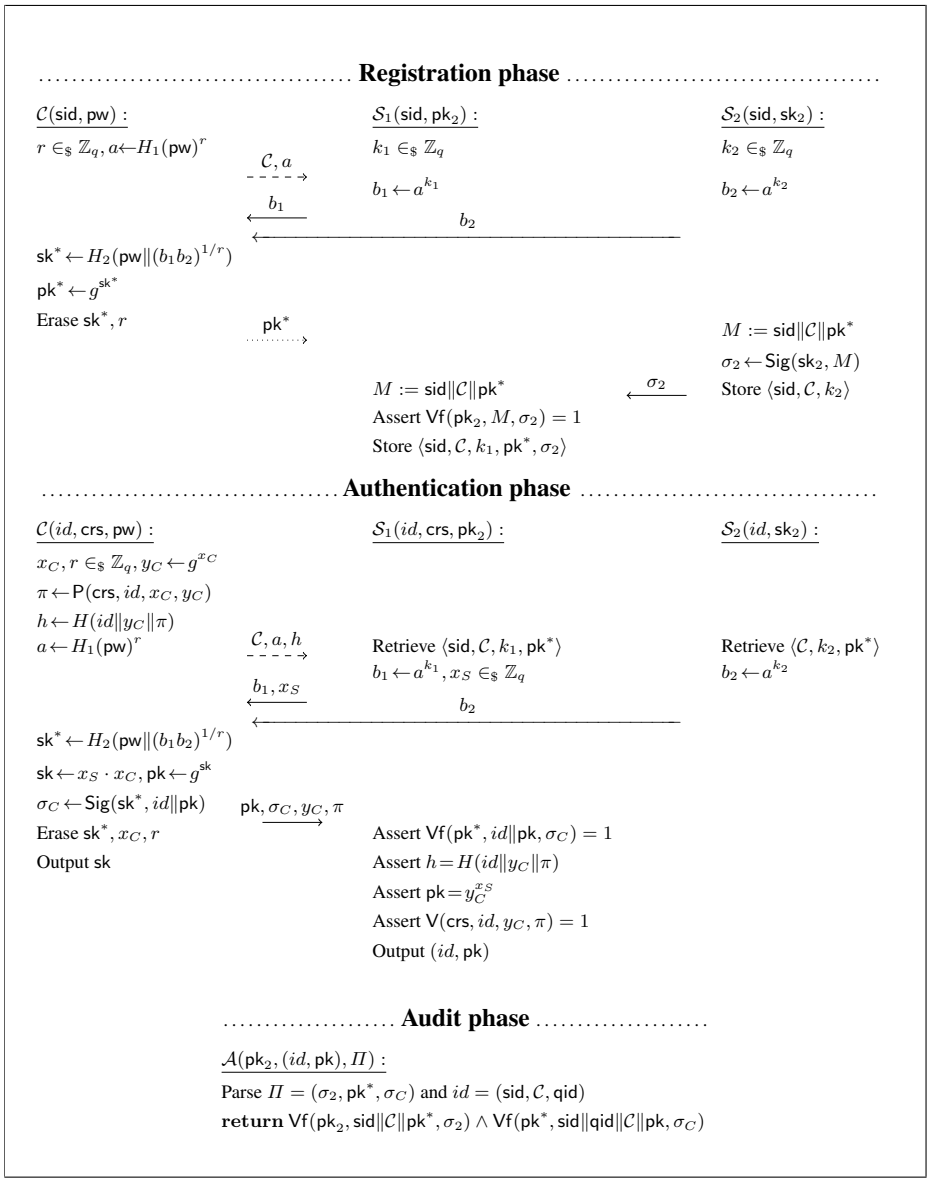
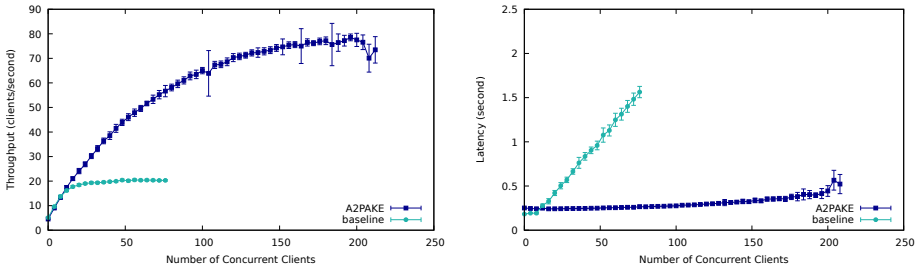


Fig. 6: Concrete instantiation of $\mathsf{A}^2\mathsf{PAKE}$. Dashed arrows depict broadcast messages. The dotted arrow in the registration phase denotes an authenticated channel. In the authentication phase the three parties receive in input an identifier $id = (\mathsf{sid}, \mathcal{C}, \mathsf{qid})$.

the auditor takes in input the public key of the server $\mathcal{S}_2$, the tuple (id, pk) describing the client and its public key and a proof Π that pk is indeed its public key.

A note on forward-secrecy. The protocol of Fig. 6 realizes $\mathcal{F}_{\mathsf{A^2PAKE}}$, thereby guaranteeing forward-secrecy: even if a user password is leaked, key-pairs output by executions of the authentication protocol before password leakage took place are still secure. As in other protocols that ensure forward-secrecy, e.g., [33], forward-secrecy alone is not enough to prevent an adversary from "back-dating" a signature. That is, a notion of "time" is needed to decide whether a signature can be considered valid. For example, in our application scenario, an adversary that has learned the password of a honest client at time t, may still run the authentication protocol on behalf of the client, obtain a fresh key-pair, and sign messages with arbitrary timestamps (e.g, earlier than t). As in any forward-secure signature scheme, we need a mechanism to bound a key-pair $\mathsf{sk}_t, \mathsf{pk}_t$ to an "epoch" t, so that only signatures that verify under the public key of the current epoch are considered valid. This could be achieved in many ways. For example, the client may upload key-pairs to a public and timestamped bulletin board or blockchain. Alternatively, during the authentication protocol when the client uses sk^* to sign $id\|\mathsf{pk}$, it may include a timestamp ts and send the signature to both servers. Hence, $\mathcal{S}_2$ checks that ts is valid, checks that the signature verifies under pk^*, and sends its own signature over $id\|\mathsf{pk}\|ts$ to party $\mathcal{S}_1$, thereby acting as a timestamping authority that binds pk to timestamp ts.

6 Evaluation



(a) Throughput. (b) Latency.

Fig. 7: Comparison of $\mathsf{A^2PAKE}$ versus the baseline protocol over WAN.

We implemented a prototype of the $\mathsf{A^2PAKE}$ instantiation presented in the previous section. The prototype was written in Python with the Charm-crypto[8] 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 2HashDH [22] distributed OPRF. Random oracles were instantiated with SHA256.

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

Theoretical overhead. We now provide a theoretical evaluation of the computation and communication overhead of the protocols in Fig. 6. For elliptic curve operations, we only report the number of point multiplications (mults), including there the computation of multiples of a given point. During registration, the client computes 4 mults and 2 hashes. Apart from a username, the client sends two group elements and a signature. This signature is the one that $\mathcal{S}_2$ sends to $\mathcal{S}_1$ at the end of the registration protocol in Fig. 6; 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 $\mathcal{S}_1$ computes a single exponentiation and verifies the validity of a signature; it sends one group element. During authentication, $\mathcal{S}_1$ computes one exponentiation and one hash. It also verifies a signature and a proof-of-knowledge of discrete log (two mults). $\mathcal{S}_1$ sends one group element and one element in $\mathbb{Z}_q$. During registration, server $\mathcal{S}_2$ computes one exponentiation and one signature; it sends one group element and a signature (to the client that will forward it to $\mathcal{S}_1$). During authentication, $\mathcal{S}_2$ 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 Fig. 6 (e.g., the client sends g^{x_C} for random x_C, the server sends random x_S, the client sets $\mathsf{sk} = x_C \cdot x_S$ 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.

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 baseline 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. Figs. 7a and 7b 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.

Results. 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 little more than 10 concurrent clients. The main server of $\mathsf{A}^2\mathsf{PAKE}$ was able to handle up to 200 clients. 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 Fig. 6. 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.

Acknowledgements

M.I.G. Vasco is supported by research grant PID2019- 109379RB-100 from Spanish MINECO. Antonio Faonio is partially supported by the MESRI-BMBF French-German joint project named PROPOLIS (ANR-20-CYAL-0004-01).

References

1. Agrawal, S., Miao, P., Mohassel, P., Mukherjee, P.: PASTA: PASsword-based threshold authentication. In: ACM CCS 2018. ACM Press (Oct 2018)
2. Bagherzandi, A., Jarecki, S., Saxena, N., Lu, Y.: Password-protected secret sharing. In: ACM CCS 2011. ACM Press (Oct 2011)
3. Baum, C., Frederiksen, T.K., Hesse, J., Lehmann, A., Yanai, A.: PESTO: proactively secure distributed single sign-on, or how to trust a hacked server. In: IEEE EuroS&P. pp. 587–606. IEEE (2020). https://doi.org/10.1109/EuroSP48549.2020.00044, https://doi.org/10.1109/EuroSP48549.2020.00044
4. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: EUROCRYPT 2000. LNCS, Springer, Heidelberg (May 2000)
5. Boyko, V., MacKenzie, P.D., Patel, S.: Provably secure password-authenticated key exchange using Diffie-Hellman. In: EUROCRYPT 2000. LNCS, Springer, Heidelberg (May 2000)
6. Bradley, T., Camenisch, J., Jarecki, S., Lehmann, A., Neven, G., Xu, J.: Password-authenticated public-key encryption. In: ACNS 19. LNCS, Springer, Heidelberg (Jun 2019)
7. Camenisch, J., Lehmann, A., Neven, G., Samelin, K.: Virtual smart cards: How to sign with a password and a server. In: SCN 16. LNCS, Springer, Heidelberg (Aug / Sep 2016)
8. Camenisch, J., Lysyanskaya, A., Neven, G.: Practical yet universally composable two-server password-authenticated secret sharing. In: ACM CCS 2012. ACM Press (Oct 2012)
9. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. IEEE Computer Society Press (Oct 2001)
10. Canetti, R., Halevi, S., Katz, J., Lindell, Y., MacKenzie, P.D.: Universally composable password-based key exchange. In: EUROCRYPT 2005. LNCS, Springer, Heidelberg (May 2005)
11. Das, P., Hesse, J., Lehmann, A.: DPaSE: Distributed password-authenticated symmetric encryption. Cryptology ePrint Archive, Report 2020/1443 (2020), https://eprint.iacr.org/2020/1443

12. para las Administraciones Gobierno de España, I.E.: Clave firma (2019), https://clave.gob.es/clave_Home/dnin.html
13. EU Parliament: eIDAS Regulation (Regulation (EU) N 910/2014) (2014), http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv%3AOJ.L_.2014.257.01.0073.01.ENG
14. Everspaugh, A., Chatterjee, R., Scott, S., Juels, A., Ristenpart, T.: The pythia PRF service. In: USENIX Security 2015. USENIX Association (Aug 2015)
15. Faonio, A., González Vasco, M.I., Soriente, C., Truong, H.T.T.: Auditable asymmetric password authenticated public key establishment. Cryptology ePrint Archive, Report 2020/060 (2020), https://eprint.iacr.org/2020/060
16. Gjøsteen, K., Thuen, Ø.: Password-based signatures. In: 8th European Workshop on Public Key Infrastructures, Services and Applications (EuroPKI). pp. 17–33 (2011)
17. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on Computing (2) (Apr 1988)
18. Groth, J.: Simulation-sound NIZK proofs for a practical language and constant size group signatures. In: ASIACRYPT 2006. LNCS, Springer, Heidelberg (Dec 2006)
19. International, S.O.: Websignatureoffice (2019), https://www.websignatureoffice.com/us/
20. Jarecki, S., Kiayias, A., Krawczyk, H.: Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In: ASIACRYPT 2014, Part II. LNCS, Springer, Heidelberg (Dec 2014)
21. Jarecki, S., Kiayias, A., Krawczyk, H., Xu, J.: Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In: IEEE EuroS&P 2016. pp. 276–291. IEEE (2016)
22. Jarecki, S., Kiayias, A., Krawczyk, H., Xu, J.: TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In: ACNS 17. LNCS, Springer, Heidelberg (Jul 2017)
23. Jarecki, S., Krawczyk, H., Shirvanian, M., Saxena, N.: Device-enhanced password protocols with optimal online-offline protection. In: ASIACCS 16. ACM Press (May / Jun 2016)
24. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In: EUROCRYPT 2018, Part III. LNCS, Springer, Heidelberg (Apr / May 2018)
25. Katz, J., MacKenzie, P.D., Taban, G., Gligor, V.D.: Two-server password-only authenticated key exchange. In: ACNS 05. LNCS, Springer, Heidelberg (Jun 2005)
26. Katz, J., Ostrovsky, R., Yung, M.: Efficient password-authenticated key exchange using human-memorable passwords. In: EUROCRYPT 2001. LNCS, Springer, Heidelberg (May 2001)
27. Kiefer, F., Manulis, M.: Distributed smooth projective hashing and its application to two-server password authenticated key exchange. In: ACNS 14. LNCS, Springer, Heidelberg (Jun 2014)
28. Lai, R.W.F., Egger, C., Reinert, M., Chow, S.S.M., Maffei, M., Schröder, D.: Simple password-hardened encryption services. In: USENIX Security 2018. USENIX Association (Aug 2018)
29. Lai, R.W.F., Egger, C., Schröder, D., Chow, S.S.M.: Phoenix: Rebirth of a cryptographic password-hardening service. In: USENIX Security 2017. USENIX Association (Aug 2017)
30. Limited, A.: Signhub (2019), https://www.signinghub.com/
31. MacKenzie, P.D., Shrimpton, T., Jakobsson, M.: Threshold password-authenticated key exchange. In: CRYPTO 2002. LNCS, Springer, Heidelberg (Aug 2002)
32. Moriarty, K., Kaliski, B., Rusch, A.: Pkcs#5: Password-based cryptography specification version 2.1. Tech. Rep. RFC8010, Internet Engineering Task Force (IETF) (2017), https://tools.ietf.org/html/rfc8018

33. Park, D., Boyd, C., Moon, S.J.: Forward secrecy and its application to future mobile communications security. In: PKC 2000. LNCS, Springer, Heidelberg (Jan 2000)
34. Schneider, J., Fleischhacker, N., Schröder, D., Backes, M.: Efficient cryptographic password hardening services from partially oblivious commitments. In: ACM CCS 2016. ACM Press (Oct 2016)
35. S.p.A., A.: Firma digitale remota (2019), www.aruba.it