Password-Protected Threshold Signatures*

Stefan Dziembowski[1,2], Stanislaw Jarecki[3], Pawel Kedzior[1], Hugo Krawczyk[4], Chan Nam Ngo[5], and Jiayu Xu[6]

[1] University of Warsaw, Poland, {s.dziembowski, p.kedzior}@mimuw.edu.pl
[2] IDEAS NCBR, Poland
[3] University of California Irvine, USA, sjarecki@uci.com
[4] Amazon Web Services, USA, hugokraw@gmail.com
[5] Privacy + Scaling Explorations, Vietnam namncc@pse.dev
[6] Oregon State University, USA, xujiay@oregonstate.edu

Abstract. We witness an increase in applications like cryptocurrency wallets, which involve users issuing signatures using private keys. To protect these keys from loss or compromise, users commonly outsource them to a custodial server. This creates a new point of failure, because compromise of such a server leaks the user's key, and if user authentication is implemented with a password then this password becomes open to an offline dictionary attack (ODA). A better solution is to secret-share the key among a set of servers, possibly including user's own device(s), and implement password authentication and signature computation using threshold cryptography.

We propose a notion of *augmented password-protected threshold signature* (aptSIG) scheme which captures the best possible security level for this setting. Using standard threshold cryptography techniques, i.e. threshold password authentication and threshold signatures, one can guarantee that compromising up to t out of n servers reveals no information on either the key or the password. However, we extend this with a novel property, namely that compromising *even all n servers* also does not leak any information, except via an unavoidable ODA attack, which reveals the key (and the password) only if the attacker guesses the password.

We define aptSIG in the Universally Composable (UC) framework and show that it can be constructed very efficiently, using a black-box composition of any UC threshold signature [13] and a UC *augmented Password-Protected Secret Sharing (aPPSS)*, which we define as an extension of prior notion of PPSS [30]. As concrete instantiations we obtain secure aptSIG schemes for ECDSA (in the case of $t = n-1$) and BLS signatures with very small overhead over the respective threshold signature.

1 Introduction

Threshold signatures have been studied for over 30 years [19]. Recently, their practical applicability increased significantly due to the use of signatures in

* This is an extended version of the paper which appeared in [21].

blockchains and cryptocurrencies, especially for transaction authorization on behalf of users. In particular, multiple schemes have been developed for threshold ECDSA given the wide use of ECDSA in blockchains, e.g. [14,20,24,34]. Recall that in a (t, n)-threshold signature the private signing key is shared between a set of n servers, and $t + 1$ of them must collaborate to produce a signature; security requires that breaking into any t servers does not allow an attacker to forge signatures. Users that utilize a signature to authorize electronic transactions, e.g., the transfer of monies between accounts, but want to protect their keys from loss or compromise, can outsource signature generation to a trusted service that implements a threshold signature scheme. Yet, this setting raises the question of how a user can authorize the servers to sign on her behalf. An attacker who impersonates the user in this authorization process can request signatures on messages of its choice. On the other hand, if this authentication requires a user-held cryptographic key then we have a chicken-and-egg problem: we outsourced one user's key but we still require the user to hold another.

We can break this loop if we consider a setting where the authorization depends on a user's *password*. However, this presents another conundrum: Asking the user to pick an independent password for each server requires too much memorization (without secure storage), but using the same password with each server would create n points of failure, because an attacker who manages to break any one of the n servers would be able to run an offline dictionary attack against the user's password, and then use the password to authorize all other servers to sign any message.

Augmented Password-protected Threshold Signatures. Our goal is a threshold signature scheme where all the user needs to authorize messages to be signed is *a single password*. The break of any t servers should leak no information that allows to attack either the signature scheme or the password, and the security should not rely on any secret or public keys stored or carried by the user. We refer to this notion as *Password-protected Threshold Signature* (ptSIG).

But we want more: We want that *even after the compromise of more than t servers (and possibly all n servers)*, the only information the attacker can gain requires finding the right password via an *exhaustive offline dictionary attack* (ODA). (Note that if a password triggers correct signature generation then an ODA on all-servers compromise is unavoidable.) In other words, the password should not only authenticate the user to the servers, but even if all servers are compromised they cannot produce signatures unless the attacker guesses the password. In particular, a solution that simply secret-shares the signing key among the servers would not work. In summary, we seek solutions that offer the following guarantees:

1. Each protocol execution, either by the user or by the servers, allows the attacker an online password test for only one password guess.
2. Compromising up to t servers results in no security loss, i.e. the attacker learns no information on either the signature key or the password.

3. Compromising $t+1$ or more servers (even all n) does not give the attacker any information either, without the attacker first succeeding in an exhaustive offline dictionary attack (ODA) against the user's password.

Properties 1 and 2 can be achieved by a composition of threshold Password-Authenticated Key Exchange (tPAKE) [36] and threshold signature scheme (tSIG) [18]. However, property 3 is not implied by such composition, and indeed does not seem easy to achieve using any tPAKE and tSIG schemes alone.

Support for Server-side Security Mechanisms. We add one further requirement, and we refer to a notion which satisfies all requirements 1–4 as *Augmented Password-protected Threshold Signatures* (aptSIG):

4. An attacker who knows the password, can sign only one message per each interaction with $t+1$ servers, and only if these servers agree to sign it. In particular, if the attacker compromised $t' \leq t$ servers, it can sign only one message per each interaction with $(t+1) - t'$ uncompromised servers.

Property 4 implies that the scheme cannot reveal the signing key to the user even if they hold the right password, as this would allow an attacker who compromises the password to sign messages without further server involvement. In contrast, an aptSIG scheme can limit such attacker by several mechanisms, such as *rate-limiting*, i.e. allowing only a limited number of signatures per time interval; implementing *multi-factor authentication*, which the attacker would need to bypass even if it learns the password; and signing messages only if they are compliant with an *application policy*, i.e. only messages with application-compliant semantics (e.g. including the correct current date). Note that property 4 also protects the user in case of a break into the client machine: Such break might leak the password, but it cannot leak the signing key.

Augmented Password-Protected Secret Sharing (aPPSS). We introduce a protocol tool that plays an essential role in our aptSIG construction. Recall the notion of *Password-Protected Secret Sharing* (PPSS) [5]. A (t, n)-PPSS scheme allows user U to share a secret s among n servers and "protect" this sharing by a password pw, in the sense that PPSS reconstruction will recover s if and only if the user interacts with $t+1$ servers using the same password pw. (No extra user storage or authentication infrastructure such as PKI is assumed except during user registration.) PPSS security requires that compromising any t servers leaks no information on either the secret s or the password pw. However, for the purpose of building an aptSIG scheme, we need a stronger notion of PPSS with the following additional property: a compromise of more than t servers (even all n of them) still does not leak s and pw immediately, but only allows the attacker to stage an offline dictionary attack on the password, and this offline attack will leak s only if the attacker finds pw. We formalize this notion in the Universally Composable (UC) model [11] and refer to it as *augmented PPSS* (aPPSS), and we show that an existing PPSS scheme of [30] sufficiently realizes this stronger notion.

From aPPSS to aptSIG. Armed with the aPPSS tool, we build an aptSIG as follows. We start with a threshold signature scheme (tSIG) which relies on n servers and an additional entity U, called the user, where breaking the tSIG scheme requires breaking into $t+1$ servers *plus* compromising U. A tSIG scheme for this "1+threshold" access structure can be obtained from regular (t, n)-threshold signature by e.g. providing multiple shares to the user, but many threshold signatures can be adapted to this access structure more efficiently, as we exemplify by the BLS-based construction of Section 2.1.

At a high level, our aptSIG scheme works as follows:

- At initialization, which we assume runs over authenticated channels, e.g. using PKI for server authentication[1], the tSIG scheme is initialized so that the servers and the user get the information needed to later run the signing protocol. Let ts_U denote the state that U needs to store to run tSIG signature protocol (this would include the share of the signature key, but also possibly the keys needed to authenticate/encrypt tSIG protocol messages). In addition, servers and U initialize an aPPSS instance under the user's password which produces a random secret sk learned by U. The user authenticates-and-encrypts the state ts_U under key sk to obtain an authenticated encryption ciphertext aec_U, and sends aec_U to all servers who store it. U then erases all information and only remembers its password.
- To sign message m, party U and the servers run aPPSS reconstruction by which U, using its password, retrieves sk. The servers send aec_U back to U who authenticates-and-decrypts it under sk to learn its tSIG state ts_U. Finally, now that U holds its tSIG state, U and the servers run the tSIG scheme to sign m.

Definitions, generic construction, efficient aptSIG instantiations. Regarding the security of our construction, all of our constructions are defined in the UC model, which is essential for security under arbitrary composition: First, we frame the new notions of aPPSS and aptSIG as UC functionalities; second, we generalize the UC tSIG notion of Canetti et al. [13], which was defined only for the n-out-of-n setting, to arbitrary (t, n)-threshold and 1+threshold access structures.

Next, we show how to efficiently realize our UC aptSIG notion: the schematic outline above provides a generic design of UC aptSIG scheme from any UC aPPSS and UC tSIG that supports the 1+threshold access structure. In this construction, the only overhead incurred while compiling a tSIG to an aptSIG is the cost of the aPPSS scheme, which can be instantiated efficiently: our UC aPPSS scheme, which is essentially identical to the PPSS of [30], is a generic construction from any UC Oblivious PRF (OPRF), and using the 2HashDH OPRF of [30] it requires only two communication flows and its computational cost is 1 exponentiation for each server and $t + 2$ for the user.

[1] Authenticated channels between user and servers are needed at initialization in order for the user to identify the servers it is communicating with, but such channels, or PKI, are not needed for later signature generation.

At first glance, it seems that this generic construction leads to a UC-secure aptSIG implementation of ECDSA based on the UC ECDSA scheme of [13] adapted to the 1+threshold access structure. However, that scheme was shown secure only for the additive n-out-of-n sharing, so the result in [13] only implies an aptSIG with $t = n - 1$. In the general case, one would have to carefully verify whether the generalization of ECDSA of [13] to the (t, n)-threshold and 1+threshold settings realizes the UC tSIG functionality for these access structures. Moreover, that scheme requires several rounds of interaction.

For the general case, we instead present a concrete round-minimal and highly practical aptSIG scheme (see Fig. 8 in Section 5) based on a threshold BLS signature [8,7]. It requires only 2 communication flows in signing, 3 flows in initialization, uses no server-to-server communication, and takes $O(1)$ exponentiations per server and $O(n)$ exponentiations and bilinear maps for the user. We prove that this BLS-based scheme realizes the UC tSIG functionality for the 1+threshold access structure for any $t \leq n$ s.t. $\binom{n}{t}$ is polynomial in the security parameter; this probably can be extended to any parameters n, t using the results of Bacho and Loss [4] and Das and Ren [16] (see Section 2.1).

Extensions to Password-Protected MPC. While this paper develops definitions and mechanisms specific to the case of aptSIG, our approach and techniques can be generalized to provide "password-protection" of other cryptographic functions. For example, in the case of encryption, a user may want to decrypt encrypted data only in collaboration with a threshold of servers conditioned on knowledge of a password, and with additional assurances similar to those in our aptSIG treatment (e.g., enforcing a decryption policy by the servers, allowing for rate limits, etc.). In another example, one can consider a variant of aptSIG where the keyed function is a *blind* signature scheme, to keep messages signed hidden from the servers. In general, one can use this approach to password-protect multi-party computation of *arbitrary functions*, with security guarantees as in items 1–4 above, but with signatures replaced by an arbitrary keyed function. We leave such extensions and generalizations as subjects for future work.

MPC for Obfuscated Point Function. Finally, observe that aPPSS can be seen as a distributed computation of the point function

$$PF_{\mathsf{pw},s}(x) = \begin{cases} s & \text{if } x = \mathsf{pw} \\ \perp & \text{otherwise} \end{cases} .$$

The aPPSS protocol computes $PF_{\mathsf{pw},s}(\cdot)$ in a distributed setting, by user U holding input x and the servers holding the secret-sharing of the function description $\langle \mathsf{pw}, s \rangle$, with U computing the output $y = PF_{\mathsf{pw},s}(x)$. Moreover, the aPPSS property that even a compromise of all servers allows for recovery of s (and pw) only via an offline dictionary attack, implies that the server-held shares reconstruct an *obfuscated* representation of point function $PF_{\mathsf{pw},s}$, i.e. a software black-box which allows evaluation of $PF_{\mathsf{pw},s}(\cdot)$ on any input (e.g. password guess), but it leaks no information on (pw, s) unless one queries it on input $x = \mathsf{pw}$. Thus, an

efficient aPPSS scheme implies an efficient evaluation of a *secret-shared obfuscated point function*, and as such it can find other applications.[2]

Applications to blockchain wallets. Some very attractive applications for threshold cryptography come from the blockchain domain. Recall that cryptocurrency coins are signature keys, spending a coin is implemented as a signing operation, and that storage of these signature keys is one of the most sensitive parts of the entire blockchain ecosystem. This problem is addressed by the use of so-called *hardware wallets* (see, e.g., [2]), *threshold wallets* (see, e.g., [15]), or *MPC wallets* (see, e.g., [3]). Our solution provides a stronger, practical, and flexible alternative to these methods. Our solution implements a threshold wallet, enabling storing cryptocurrencies in a threshold way, but it simultaneously protects them with a password in two ways: One way, which is standard, is that the user must use a correct password to access their cryptocurrency stored in a threshold wallet. The second way, which is novel, is that the shares stored by the threshold wallet parties are effectively encrypted under the password, so even corruption of all the threshold wallets parties does not leak the cryptocurrency keys in the clear. Instead, a corruption of all threshold wallet parties reveals an obfuscated "output-a-key-only-if-input-is-a-correct-password" black-box, which allows only offline dictionary attacks against a password, and leaks the cryptocurrency keys only if the adversary finds the correct password.

1.1 Further Related Works

Threshold signatures. Threshold signatures were formalized by Desmedt and Frankel in [19] with precursors including [9,17,18]. Since then countless papers have studied threshold signatures for a variety of signature schemes. More recent work in the area has been motivated by cryptocurrency applications with particular focus on Threshold ECDSA, e.g. [14,20,24,34] as a prevalent signature scheme used in these applications. Among these works, our paper adopts the UC formalism for threshold signatures from Canetti et al. [13] who present a threshold ECDSA scheme that realizes this formalism.

Server-aided signatures. Using passwords in the context of threshold signatures has been studied in the setting of server-aided signatures and their variants [10,23,27,35,40]. These papers address the case of a user with access to a dedicated device that stores a strong signing key but requires user's password to generate signatures. The password prevents an attacker that gets hold of the device from producing signatures at will, but an attacker can run an offline dictionary attack by entering password guesses to the device. To prevent such dictionary attacks these works add a remote server with whom the device shares the signing key and whose participation is required for producing signatures. The user typically enters its password on the device, but the interaction with the remote server limits the number of password attempts an adversary can try

[2] A similar idea of using an OPRF to evaluate a (non-secret-shared) obfuscated point function was first noticed in [37].

once it controls the user's device. Some of the schemes also support hiding the message being signed from the remote server. Most schemes in the literature consider a single remote server but e.g. the work of [40] includes distributing the remote server into a group of servers using a threshold signature scheme.

However, in all these cases, the user depends on its own device for generating signatures. In particular, the device stores strong cryptographic keys. Our setting is different. We assume users that carry with them nothing but their memorized passwords; they do not even carry high-entropy public values (such as servers' public keys), let alone dedicated devices. In particular, in our solution, a user can trigger signatures by logging in from an arbitrary device.

Password-authenticated threshold signatures. A different line of work that shares similarities with our paper, but targets a different application and has different security properties, is [1,6]. These papers deal with a single sign-on setting where an identity provider (e.g., Google) authenticates users using passwords, and upon authentication provides users with signed tokens (which authenticates a user to some 3rd-party service). These works distribute the identity provider operation over a set of servers and use threshold cryptography in two ways: First, they use threshold password authentication (tPAKE) to authenticate users to the servers that implement a distributed identity provider; second, the servers use a threshold signature (tSIG) to sign the requested token.

However, in this application the signing key is the provider's key, which is used to sign messages for *all* users, and it can be reconstructed if $t + 1$ servers are compromised. By contrast, in our case each user shares its own private key across a set of servers, and neither this key nor the user's password is leaked, except via offline dictionary attack, even if all servers collude. Indeed, none of the above cited works models or claims the "augmented" property we introduce in the aptSIG notion, namely that the break of the system requires not only that the attacker breaks into a sufficient threshold of servers, but that it also succeeds in subsequent exhaustive offline attack against the user's password.

Augmented threshold PAKE and proactive security. In a concurrent work, Gu et al. [28] define the notion of *augmented* threshold PAKE (atPAKE), where the term "augmented" denotes the same security property as in our augmented PPSS and augmented Password-protected Threshold Signatures. As the standard notion of tPAKE [36], a (t, n)-threshold atPAKE allows the user to authenticate using a password to a set of servers who secret-share password-related information, and the scheme leaks nothing if up to t out of n servers are compromised. However, if $t+1$ or more servers are compromised, the password still doesn't leak in the clear unless the attacker succeeds in an offline dictionary attack (ODA). Intuitively, in atPAKE servers must secret-share a (salted) *hash* of the user's password, rather than the password itself.

Apart from the fact that the work of [28] tackles a similar augmented property in the context of a different threshold cryptosystem (threshold PAKE rather than threshold password-protected signatures), their work also defines and constructs a UC threshold OPRF (tOPRF), and we believe that the tOPRF-to-PPSS compiler of [32] offers an alternative implementation of UC aPPSS. One reason this

alternative aPPSS implementation is interesting is that all building blocks here can be made *proactively* secure: the tOPRF of [28] can be proactively secure, which leads to a proactively secure aPPSS, which (combined with a practively secure threshold signature) in turn would result in a *proactively secure aptSIG*.

Paper organization. Section 2 defines UC threshold signature (tSIG) for arbitrary access structures, and exemplifies it with a threshold BLS signature scheme. Section 3 defines Augmented Password-Protected Secret Sharing (aPPSS) and shows that the PPSS scheme of [30] realizes this notion. Section 4 defines Augmented Password-protected Threshold Signature (aptSIG), and shows a generic construction of secure aptSIG from aPPSS and tSIG schemes. Finally, in Section 5 we exemplify this generic compiler with an efficient and practical scheme based on threshold BLS.

We defer some material to the appendices. Specifically, in the appendices we include the proof of security for the threshold BLS scheme (Appendix B), we define the adaptive UC OPRF functionality $\mathcal{F}_{\text{OPRF}}$ and the 2HashDH protocol that realizes it (Appendix C), we include the security proof for our aPPSS scheme (Appendix D), we compare our UC aPPSS model with prior PPSS definitions (Appendix E), we include the security proof for our aptSIG scheme (Appendix F), we introduce versions of our aptSIG model and the aptSIG protocol that add the property of *Perfect Forward Security* (PFS) to the basic model (Appendix G), and we show a concrete BLS-based instantiation of the PFS-aptSIG scheme (Appendix H).

Acknowledgments. *Stefan Dziembowski, Pawel Kedzior, Chan Nam Ngo:* This work is part of a project that received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grants PROCONTRA-885666). *Stefan Dziembowski* was also partly supported by the Polish NCN Grant 2019/35/B/ST6/04138 and the Nicolaus Copernicus Polish-German Research Award 2020 COP/01/2020. *Stanislaw Jarecki:* This work was supported by NSF SaTC TTP award 2030575. *Hugo Krawczyk:* This work was done while the author was at the Algorand Foundation. *Chan Nam Ngo:* The majority of this work was done while the author was with the University of Warsaw, Poland.

2 Threshold Signatures

Figure 1 shows a generalization of the ideal functionality for threshold signature $\mathcal{F}_{\text{tSIG}}$ of Canetti et al. [13] to an arbitrary access structure $\mathbb{S}$. The UC threshold signature model of [13] extends the formalization of standard (i.e. non-threshold) signatures as a UC functionality [12] (for prior and related work on UC signatures see references therein) to the *distributed* setting where the signing key is secret-shared among n servers. However, the UC formalization of [13] defined it solely for the case of an n-out-of-n secret-sharing, where the signature is unforgeable if the adversary corrupts up to $n-1$ servers, but all servers have to participate to issue a valid signature. Here we extend the definition of [13] to arbitrary

Notation: We assume $sid = (\ldots, \mathbf{P})$ where $\mathbf{P}$ is a list of parties, and we let $\mathbf{P}_{sid}$ denote set $\mathbf{P}$ specified by string sid. $\mathbb{S}_{sid}$ denotes an access structure $\mathbb{S}$ applied to set $\mathbf{P}_{sid}$, i.e. signatures for sid can be created only by a set A of parties s.t. $A \in \mathbb{S}_{sid}$. The functionality interacts with a set of parties $\mathcal{P}$ and an adversary $\mathcal{A}^*$. **Corr** is initialized to the initial set of corrupted parties.

Key Generation:

[K.P] On $(\mathsf{tsig.keygen}, sid)$ from party P (or $(\mathsf{tsig.keygen}, sid, \mathsf{P})$ from $\mathcal{A}^*$ if $\mathsf{P} \in \mathbf{Corr}$), record and send to $\mathcal{A}^*$ tuple $(\mathsf{tsig.keygen}, sid, \mathsf{P})$.

[K.V] On $(\mathsf{tsig.publickey}, sid, \mathsf{V})$ from $\mathcal{A}^*$, if $(\mathsf{tsig.keygen}, sid, \mathsf{P})$ is recorded for all $\mathsf{P} \in \mathbf{P}_{sid}$ then record (sid, V).

[K.F] On $(\mathsf{tsig.keygencomplete}, sid, \mathsf{P})$ from $\mathcal{A}^*$, if $\exists$ record (sid, V) then send $(\mathsf{tsig.publickey}, sid, \mathsf{V})$ to P.

Signing:

[S.P] On $(\mathsf{tsig.sign}, sid, \mathsf{m})$ from P (or $(\mathsf{tsig.sign}, sid, \mathsf{P}, \mathsf{m})$ from $\mathcal{A}^*$ if $\mathsf{P} \in \mathbf{Corr}$), if $\exists$ record (sid, V) then record and send to $\mathcal{A}^*$ tuple $(\mathsf{tsig.sign}, sid, \mathsf{m}, \mathsf{P})$.

[S.S] On $(\mathsf{tsig.signature}, sid, \mathsf{m}, S, \sigma)$ from $\mathcal{A}^*$, if $S \in \mathbb{S}_{sid}$ and tuple $(\mathsf{tsig.sign}, sid, \mathsf{m}, \mathsf{P})$ is recorded for all $\mathsf{P} \in S$ then do the following:

- [S.S.1] If $\exists$ record $(sid, \mathsf{m}, \sigma, 0)$ then ignore this message;
- [S.S.2] Else, if $\mathsf{V}(\mathsf{m}, \sigma) = 1$ then record tuple $(sid, \mathsf{m}, \sigma, 1)$;
- [S.S.3] If $\mathsf{V}(\mathsf{m}, \sigma) = 0$ then ignore this message.

[S.F] On $(\mathsf{tsig.signcomplete}, sid, \mathsf{m}, \mathsf{P})$ from $\mathcal{A}^*$, if $\exists$ record $(sid, \mathsf{m}, \sigma, 1)$ then send $(\mathsf{tsig.signature}, sid, \mathsf{m}, \sigma)$ to P.

Verification:

[V.V] On $(\mathsf{tsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$ from P, send $(\mathsf{tsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$ to $\mathcal{A}^*$ and:

- [V.1] If $\exists$ records (sid, V) and $(sid, \mathsf{m}, \sigma, \beta')$ then set $\beta := \beta'$;
- [V.2] Else, if $\exists$ record (sid, V) but no record $(sid, \mathsf{m}, \sigma', 1)$ for any σ' then set $\beta := 0$;
- [V.3] Else set $\beta := \mathsf{V}(\mathsf{m}, \sigma)$.

[V.F] Record $(sid, \mathsf{m}, \sigma, \beta)$ and send $(\mathsf{tsig.verified}, sid, \mathsf{m}, \sigma, \beta)$ to P.

Party Compromise: (*This query requires permission from the environment.*)

[PC] On $(\mathsf{tsig.compromise}, sid, \mathsf{P})$ from $\mathcal{A}^*$, set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$.

Fig. 1: Threshold signature functionality $\mathcal{F}_{\text{tSIG}}$ for arbitrary access structure $\mathbb{S}$

access structures, including the (t, n)-threshold access structure the specialized "1+threshold" access structure we use in our aptSIG application.

The threshold signature functionality $\mathcal{F}_{\text{tSIG}}$ consists of three parts, Key Generation, Signing and Verification. In contrast to [13], our functionality omits Key-Refresh, but both versions support adaptive party compromise. Following [13], w.l.o.g. we identify a public key V with an arbitrary deterministic algorithm, i.e. signature σ on message m is valid iff $\mathsf{V}(\mathsf{m}, \sigma) = 1$. Also following [13], we assume that if party P participates in key generation, then P runs on an instance identifier *sid* of a form $\sigma = (\ldots, \mathbf{P})$ where $\mathbf{P}$ is a set of parties, including P, which P intends to involve in this instance. We denote the unique set $\mathbf{P}$ specified by *sid* as $\mathbf{P}_{sid}$.

We use $\mathbb{S}_{sid}$ to denote access structure $\mathbb{S}$ instantiated over set $\mathbf{P}_{sid}$. For example, if $\mathbf{P}_{sid} = \{\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3\}$ and $\mathbb{S}$ is a 1-out-of-3 threshold access structure then $\mathbb{S}_{sid} = \{\{\mathsf{P}_1\}, \{\mathsf{P}_2\}, \{\mathsf{P}_3\}\}$. Our aptSIG scheme in Section 4 relies on a threshold signature for a specialized "1+threshold" access structure $\mathbb{S}$, where $\mathbf{P}_{sid}$ is a sequence of $n+1$ parties $(\mathsf{P}_0, \mathsf{P}_1, ..., \mathsf{P}_n)$, P_0 has a special status, and $\mathbb{S}$ consists of all subsets $S \subseteq \mathbf{P}_{sid}$ s.t. (1) $\mathsf{P}_0 \in S$ and (2) $|S \cap \{\mathsf{P}_1, ..., \mathsf{P}_n\}| \geq t+1$. In other words, a valid subset S must contain the special party P_0 and at least $t+1$ of parties $\mathsf{P}_1, ..., \mathsf{P}_n$. (Looking ahead, in our aptSIG implementation servers will play the role of parties $\mathsf{P}_1, ..., \mathsf{P}_n$, and P_0 will be the user.)

Threshold Signature Functionality: Discussion. To simplify notation in the key generation phase we assume that a signature scheme instance invoked with identifier *sid* generates a public key V, and a sharing of the corresponding private key, only if *all* parties in set $\mathbf{P}_{sid}$ participate in the key generation using the same identifier *sid*. However, once the key generation succeeds, then a signature valid under the generated public key can be issued as long as it is requested by *any subset* $S \subseteq \mathbf{P}_{sid}$ of parties s.t. $S \in \mathbb{S}_{sid}$.

Functionality $\mathcal{F}_{\text{tSIG}}$ of Figure 1 simplifies the one in [13] by omitting the option that lets all parties agree on a unique misbehaving party in each protocol phase. Supporting this option seems to require reliable authenticated broadcast, and since other protocols we use neither support a corresponding feature nor require reliable broadcast, we omit it here. Following [13], our functionality $\mathcal{F}_{\text{tSIG}}$ does not support *ssid*'s in the signing phase and uses the message as an index of a signing protocol instance. Functionality $\mathcal{F}_{\text{tSIG}}$ can be extended so every signer has additional input *ssid*, and signature is output only if for some subset $S \in \mathbb{S}_{sid}$ all signers $\mathsf{P} \in S$ run on the same $(ssid, m)$. However, a cost-minimal protocol like the Threshold BLS scheme in Fig 2 does not enforce such *ssid*-uniformity, so we opt for a simplified version of a signature functionality which, like the functionality of [13], doesn't enforce that either.

2.1 Threshold BLS Signature

The UC threshold signature functionality $\mathcal{F}_{\text{tSIG}}$ can be implemented for BLS signature using the well-known protocol of Boldyreva [7]. Recall that a BLS signature [8] assumes a group G of prime order p with a bilinear map e :

$G \times G \to G_T$, and defines σ as a signature on m under public key $\mathsf{V} = g^s$ if $e(g, \sigma) = e(\mathsf{V}, H(m))$, where g generates G and H is a hash onto G. BLS signature is CMA-unforgeable in ROM under the *Gap DH* assumption, i.e. if the computational Diffie-Hellman is hard in G even on access to a DDH oracle [8].

Notation: $G = \langle g \rangle$ is a group of prime order p with a bilinear map $e : G \times G \to G_T$; $\mathsf{H} : \{0,1\}^* \to G$ is an RO hash; $\mathbb{S}$ is a "1+threshold" access structure for any $t < n$.

Key Generation: (*assuming honest* P_0 *and secure point-to-point channels*)

1. Party P_0 on input $(\mathsf{tsig.keygen}, sid)$ s.t. $\mathbf{P}_{sid} = (\mathsf{P}_0, \mathsf{P}_1, ...\mathsf{P}_n)$, picks $s_0 \leftarrow_\$ \mathbb{Z}_p$, picks random t-degree polynomial f over $\mathbb{Z}_p$, sets $\{s_i := f(i) \bmod p\}_{i=1,..,n}$, $s := s_0 + s' \bmod p$, $\mathsf{V} := g^s$, $\{\mathsf{V}_i := g^{s_i}\}_{i=0,...,n}$, and $\vec{\mathsf{V}} := (\mathsf{V}_0, ..., \mathsf{V}_n)$.
   Then, for each $i = 1, ..., n$, party P_0 sends $\mathsf{sec}_{\mathsf{P}_0 \to \mathsf{P}_i}\{(sid||i), s_i, \mathsf{V}, \vec{\mathsf{V}}\}$ to P_i. Finally, P_0 saves $(0, s_0, \mathsf{V}, \vec{\mathsf{V}})$ and outputs $(\mathsf{tsig.publickey}, sid, \mathsf{V})$.
2. Party P_i on input $(\mathsf{tsig.keygen}, sid)$ s.t. $\mathbf{P}_{sid} = (\mathsf{P}_0, \mathsf{P}_1, ...\mathsf{P}_n)$ and $i > 0$, waits for message $\mathsf{sec}_{\mathsf{P}_0 \to \mathsf{P}_i}\{(sid||i), s_i, \mathsf{V}, \vec{\mathsf{V}}\}$ from P_0, and once such message is received then P_i saves $(i, s_i, \mathsf{V}, \vec{\mathsf{V}})$ and outputs $(\mathsf{tsig.publickey}, sid, \mathsf{V})$.

Signing:

1. On input $(\mathsf{tsig.sign}, sid, \mathsf{m})$, party P_i retrieves $(i, s_i, \mathsf{V}, (\mathsf{V}_0, ..., \mathsf{V}_n))$ and sends (i, σ_i) for $\sigma_i := \mathsf{H}(\mathsf{m})^{s_i}$ to all other parties.
   Once P_i receives (j, σ_j) s.t. $e(g, \sigma_j) = e(\mathsf{V}_j, \mathsf{H}(\mathsf{m}))$ from a set of parties whose union with $\{\mathsf{P}_i\}$ is $S \in \mathbb{S}_{sid}$ (i.e., P_i can "complete" the set by adding itself to it), party P_i outputs $(\mathsf{tsig.signature}, sid, \mathsf{m}, \sigma)$ for

$$\sigma := \sigma_0 \cdot \prod_{\mathsf{P}_j \in S^-} (\sigma_j)^{\lambda_j}$$

   where $S^- = S \setminus \{\mathsf{P}_0\}$ and λ_j's are Lagrange interpolation coefficients corresponding to set S^-. (Note that if $\mathbf{P}_{sid} = (\mathsf{P}_0, \mathsf{P}_1, ..., \mathsf{P}_n)$ and $S \in \mathbb{S}_{sid}$, then $S = \{\mathsf{P}_0\} \cup S^-$ where S^- is some subset of $t+1$ parties in $\{\mathsf{P}_1, ..., \mathsf{P}_n\}$.)

Verification:

1. On $(\mathsf{tsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$, party P_i sets $\beta := 1$ if $e(g, \sigma) = e(\mathsf{V}, \mathsf{H}(\mathsf{m}))$ and $\beta := 0$ otherwise, and outputs $(\mathsf{tsig.verified}, sid, \mathsf{m}, \sigma, \beta)$.

Fig. 2: Threshold BLS scheme for the "1+threshold" access structure

Figure 2 shows a threshold BLS signature scheme that realizes functionality $\mathcal{F}_{\text{tSIG}}$ for the "1+threshold" access structure, for any threshold $t < n$. We support this access structure by combining a 2-out-of-2 sharing with a stan-

dard threshold sharing. Namely, sharing $\vec{s} = (s_0, s_1, ..., s_n)$ is formed by picking $s_0 \leftarrow_{\$} \mathbb{Z}_p$, setting $(s_1, ..., s_n)$ as a (t, n)-threshold secret-sharing of random s' in $\mathbb{Z}_p$, and setting the shared secret as $s = s_0 + s' \bmod p$. This way for any set S consisting of P_0 and some $t+1$ parties in $\{\mathsf{P}_1, ..., \mathsf{P}_n\}$, secret s can be reconstructed as $s = s_0 + \sum_{\mathsf{P}_i \in S^-} \lambda_i \cdot s_i \bmod p$ where $S^- = S \setminus \{\mathsf{P}_0\}$ and λ_i's are Lagrange interpolation coefficients corresponding to set S^-.

Standard threshold access structure. Note that setting $s_0 = 0$ and removing P_0 from signing transforms the protocol in Figure 2 to a tSIG scheme which supports the standard (t, n)-threshold access structure. Moving in the other direction, we believe that most threshold signature schemes based on Shamir secret-sharing which realize $\mathcal{F}_{\text{tSIG}}$ for the (t, n)-threshold access structure, can be transformed to support the "1+threshold" access structure using the above approach, but unfortunately it is not a black-box transformation and must be verified case by case.

Distributed key generation. The protocol in Figure 2 realizes $\mathcal{F}_{\text{tSIG}}$ in the presence of secure point-to-point channels in the Key Generation phase, and assuming that party P_0 in list $\mathbf{P}_{sid} = \{\mathsf{P}_0, ..., \mathsf{P}_n\}$ is honest in that phase. The assumption on authenticated channels in key generation is unavoidable because $\mathcal{F}_{\text{tSIG}}$ enforces that a shared key is generated only if all parties in $\mathbf{P}_{sid}$ execute $(\mathsf{tsig.keygen}, sid)$, and using arbitrary key exchange protocol allows the participants to upgrade authenticated channels to secure point-to-point channels. As for the assumption on one honest party in key generation, this suffices for our aptSIG application, but this assumption can be easily eliminated by using any Distributed Key Generation (DKG) protocol for a discrete-log-based cryptosystem, e.g. [25,39]. The analysis of the protocol in Figure 2, presented in Appendix B, can be upgraded to this more general setting, e.g., by modeling the DKG subprotocol using the UC DKG functionality $\mathcal{F}_{\text{DKG}}$ of Wikstrom [39], adapted to the 1+threshold access structure.

Theorem 1 *If BLS signature is CMA-unforgeable then the threshold signature scheme in Fig. 2 realizes functionality $\mathcal{F}_{\text{tSIG}}$ for the "1+threshold" access structure for parameters t, n s.t. $\binom{n}{t}$ is polynomial in the security parameter, assuming secure point-to-point channels and honest party P_0 in the Key Generation phase.*

We defer the proof of Theorem 1 to Appendix B:

Security for arbitrary t, n parameters. First, as sketched above, the scheme of Figure 2 can be strengthened by replacing honest P_0 with a secure DKG protocol. Moreover, Theorem 1 can be extended to arbitrary (t, n) values if the environment is restricted to *static corruptions*, i.e. all corruptions are made at the outset. This can be easily verified by inspecting the proof of Theorem 1 in Appendix B: the current reduction needs to guess a subset of corrupted parties, causing it to fail except with $1/\binom{n}{t}$ probability; however, in the static corruption setting, the reduction no longer has to make such a guess.

Furthermore, Theorem 1 can be extended to arbitrary (t, n) values while allowing adaptive corruptions, following the analysis of threshold BLS by Bacho

and Loss [4] in the Algebraic Group Model (AGM) [22], under the One-More Discrete Logarithm (OMDL) assumption. The analysis of [4] was done for the standard (t, n)-threshold BLS but we believe that it can be extended to BLS which supports the 1+threshold access structure. The result of [4] also applies to several instantiations of a DKG protocol, including Pedersen's JF-DKG [38] and New-DKG by Gennaro et al. [25]. In a recent work Das and Ren [16] showed a (t, n)-threshold BLS protocol which they show adaptively secure in the standard model, without AGM, and this protocol can also be extended to the 1+threshold setting. We note that the analysis of both [4] and [16] was arguing tSIG security defined via a game-based notion, so one also has to verify that they extend to the UC notion of tSIG captured by functionality $\mathcal{F}_{\text{tSIG}}$.

3 Augmented Password-Protected Secret Sharing

Augmented Password-Protected Secret Sharing (aPPSS) is a main component in our aptSIG scheme construction. Here we follow the informal description of aPPSS in the introduction with a formalization of this notion in the UC model. We then show how to instantiate this primitive with the PPSS construction of [29]. The latter masks shares of a threshold secret-sharing with outputs of Oblivious Pseudorandom Functions (OPRF) computed on the password. Since UC OPRF can be realized very inexpensively with protocol 2HashDH (see Appendix C.1), this OPRF-based scheme leads to aPPSS with a retrieval cost of only 1 exponentiation per server and $t + 2$ exponentiations per user. Concrete instantiation of aPPSS is shown in Fig. 8 as part of aptSIG protocol.

3.1 Modeling Augmented Password-Protected Secret Sharing

The augmented PPSS functionality $\mathcal{F}_{\text{aPPSS}}$ presented in Fig. 3 has four phases. In the *initialization* phase, user U can use command $\mathsf{ppss.uinit}$ on input a password pw ([I.U]), to initialize a PPSS instance with a set of n servers whose identities $\mathbf{P}_{sid} = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ are assumed to be encoded in the session identifier, i.e. $sid = (sid', \mathbf{P}_{sid})$. The servers in $\mathbf{P}_{sid}$ join this initialization using command $\mathsf{ppss.sinit}$ for matching sid and U ([I.S]). Finally, command $\mathsf{ppss.fininit}$ from the ideal adversary $\mathcal{A}^*$ corresponds to successful initialization, which allows U to output a secret random key sk which will be protected using this aPPSS instance ([I.F]). (Observe that this random key sk can be used to authenticate-and-encrypt arbitrary data, and indeed this is how we use it in the aptSIG protocol of Section 4.)

The *reconstruction* command $\mathsf{ppss.urec}$ represents a user U' at a potentially different network entity, attempting to recover the secret key sk using password pw', which may or may not be equal to pw used in initialization ([R.U]). The reconstruction operation is directed to a set of $t+1$ servers $\mathbf{S}$. We emphasize that the user maintains no state between the initialization and the reconstruction operations except for memorizing password pw and its username sid (although we also model the user forgetting pw and causing a failure during reconstruction —

Notation: We assume strings sid of form $sid = (\ldots, \mathbf{P})$ where where $\mathbf{P} = (\mathsf{P}_1, \ldots, \mathsf{P}_n)$. $\mathbf{P}_{sid}$ denotes set $\mathbf{P}$ specified by string sid. The functionality interacts with a set of parties and an adversary $\mathcal{A}^*$. Let **Corr** be the initial set of corrupted parties. Values t, n, λ are parameters. Functionality initializes $\mathsf{ppss.pwtested}(\mathsf{pw}) := \emptyset$ for all pw, and $\mathsf{tx}(\mathsf{P}_i) := 0$ for all P_i.
(The functionality code handles only one instance, tagged by a unique string sid.)

Initialization:

[I.U] On $(\mathsf{ppss.uinit}, sid, \mathsf{pw}, \mathsf{sk}^*)$ from party U s.t. $|\mathbf{P}_{sid}| = n$: Send $(\mathsf{ppss.uinit}, sid, \mathsf{U})$ to $\mathcal{A}^*$. If U is honest then set $\mathsf{sk} \leftarrow_{\$} \{0,1\}^{\lambda}$, else set $\mathsf{sk} := \mathsf{sk}^*$. Save $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$. Ignore future $\mathsf{ppss.uinit}$ calls for same sid.
[I.S] On $(\mathsf{ppss.sinit}, sid, i, \mathsf{U})$ from party S, or $(\mathsf{ppss.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{Corr}$, send $(\mathsf{ppss.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ to $\mathcal{A}^*$, save $(\mathsf{ppss.sinit}, sid, \mathsf{U}, \mathsf{S}, i)$.
[I.A] If $\exists$ rec. $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$ and $(\mathsf{ppss.sinit}, sid, \mathsf{U}, \mathsf{S}, i)$ s.t. $\mathsf{S}{=}\mathbf{P}_{sid}[i]$, mark S as ACTIVE.
[I.F] On $(\mathsf{ppss.fininit}, sid)$ from $\mathcal{A}^*$, if $\exists$ rec. $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$ and all parties in list $\mathbf{P}_{sid}$ are marked ACTIVE, send $(\mathsf{ppss.fininit}, sid, \mathsf{sk})$ to U.

Server Compromise: (*This query requires permission from the environment.*)

[SC] On $(\mathsf{ppss.compromise}, sid, \mathsf{P})$ from $\mathcal{A}^*$, set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$.

Reconstruction:

[R.U] On $(\mathsf{ppss.urec}, sid, ssid, \mathbf{S}, \mathsf{pw}')$ from party U' or from $\mathsf{U}' = \mathcal{A}^*$, send $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \mathbf{S})$ to $\mathcal{A}^*$. If $\exists$ record $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$ then create record $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \mathsf{pw}, \mathsf{pw}', \mathsf{sk})$, else create record $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \perp, \mathsf{pw}', \perp)$. Ignore future $\mathsf{ppss.urec}$ calls for same $ssid$.
[R.S] On $(\mathsf{ppss.srec}, sid, ssid, \mathsf{U}')$ from party S or $(\mathsf{ppss.srec}, sid, ssid, \mathsf{S}, \mathsf{U}')$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{Corr}$, send $(\mathsf{ppss.srec}, sid, ssid, \mathsf{S}, \mathsf{U}')$ to $\mathcal{A}^*$. If S is marked ACTIVE then increment $\mathsf{tx}(\mathsf{S})$ by 1.
[R.F] On $(\mathsf{ppss.finrec}, sid, ssid, \mathbf{C}, \mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*)$ from $\mathcal{A}^*$, if $\exists$ rec. $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \mathsf{pw}, \mathsf{pw}', \mathsf{sk})$ then erase it and send $(\mathsf{ppss.finrec}, sid, ssid, \mathsf{sk}')$ to U' s.t.
  [R.F.1] if $\mathsf{flag} = 1$, $|\mathbf{C}| = t+1$, and $\forall_{\mathsf{S} \in \mathbf{C}}(\mathsf{tx}(\mathsf{S}) > 0)$ then set $\mathsf{tx}(\mathsf{S})$−− for all $\mathsf{S} \in \mathbf{C}$, and if $\mathsf{pw} = \mathsf{pw}'$ then set $\mathsf{sk}' := \mathsf{sk}$ else set $\mathsf{sk}' := \perp$;
  [R.F.2] if $\mathsf{flag} = 2$ and $\mathsf{pw}^* = \mathsf{pw}'$ then set $\mathsf{sk}' := \mathsf{sk}^*$;
  [R.F.3] otherwise set $\mathsf{sk}' := \perp$.

Password Test:

[PT] On $(\mathsf{ppss.testpw}, sid, \mathsf{S}, \mathsf{pw}^*)$ from $\mathcal{A}^*$, retrieve $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$. If $\mathsf{tx}(\mathsf{S}){>}0$ then add S to set $\mathsf{ppss.pwtested}(\mathsf{pw}^*)$ and set $\mathsf{tx}(\mathsf{S})$−−. If $|\mathsf{ppss.pwtested}(\mathsf{pw}^*)| = t{+}1$ then return sk to $\mathcal{A}^*$ if $\mathsf{pw}^* = \mathsf{pw}$, else return $\perp$.

Fig. 3: Augmented PPSS functionality $\mathcal{F}_{\text{aPPSS}}$

see below). In particular, the user might connect to a different set of servers in initialization and in reconstruction. Hence, for example, if a user executes the reconstruction protocol with a set of corrupted servers $\mathbf{S}$, the $\mathcal{F}_{\text{aPPSS}}$ functionality guarantees that even in this case, the adversary can *only* perform an inevitable on-line guessing attack — which we explain below.

Similar to the ppss.uinit and ppss.sinit commands in the initialization phase, the ppss.urec and ppss.srec queries control resp. user and server entering into the reconstruction subprotocol. The crucial rule enforced by $\mathcal{F}_{\text{aPPSS}}$ is that each server $\mathsf{S} \in \mathbf{P}_{sid}$ which joined the initialization is associated with a ticket counter $\mathsf{tx}(\mathsf{S})$, and this ticket counter is incremented only if S enters into the aPPSS reconstruction instance. (Which in particular means that corrupt S can increase these tickets at will, see below.) Since we do not assume authenticated links, U' session can be "routed" by the adversary to arbitrary servers; hence in the ppss.finrec command, $\mathcal{A}^*$ specifies a set $\mathbf{C}$ of servers of its choice for participation in this reconstruction ([R.F]). The protocol finalization command ppss.fininit can result in three possible outcomes:

- In a successful reconstruction session ([R.F.1]), U' outputs key sk created in the initialization, which can happen only if (I) $\mathsf{pw}' = \mathsf{pw}$, i.e., U' runs on the correct password, (II) $\mathsf{tx}(\mathsf{S}) > 0$ for all $\mathsf{S} \in \mathbf{C}$, i.e., an adversary connected U' to servers who participated in the initialization and these servers engaged in PPSS reconstruction (note that each of these ticket is decremented at ppss.fininit, hence each PPSS reconstruction can be "used" only once), and (III) the adversary allowed all these reconstructions to proceed without interference, which is modeled by setting $\mathsf{flag} = 1$.
- The adversary can connect U' only to corrupt servers ([R.F.2]), which offers $\mathcal{A}^*$ an ability to perform an on-line guessing attack on the user, because w.l.o.g. the adversary could execute the reconstruction protocol on behalf of corrupt servers on password pw^* and secret sk^* of its choice, and if $\mathsf{pw}^* = \mathsf{pw}$ this would cause U' to reconstruct the adversarially chosen value sk^*. An on-line guessing attack is modeled by $\mathcal{A}^*$ setting $\mathsf{flag} = 2$.
- In all other cases the reconstruction fails and U' outputs $\perp$ ([R.F.3]).

Adaptive Compromise and Password Tests. Command ppss.compromise allows $\mathcal{A}^*$ to adaptively compromise any party P ([SC]). The only effect this has is if $\mathsf{P} = \mathsf{S}$ for some $\mathsf{S} \in \mathbf{P}_{sid}$, i.e. if $\mathcal{A}^*$ compromises one of the servers participating in the initialization. Moreover, the effect of such compromise is not a leakage of any data (password pw or secret sk), but an ability for $\mathcal{A}^*$ to create unlimited "tickets" for $\mathcal{A}^*$, i.e. to increment $\mathsf{tx}(\mathcal{A}^*)$ at will. Such tickets can be used in the *test password* command ppss.testpw ([PT]): This query lets $\mathcal{A}^*$ specify a password guess pw^* and a server S, and $\mathcal{F}_{\text{aPPSS}}$ adds S to the set of servers for which $\mathcal{A}^*$ tests pw^*, but each such action "costs" one ticket because $\mathcal{F}_{\text{aPPSS}}$ decrements $\mathsf{tx}(\mathsf{S})$. If the adversary tests the same pw^* on $t+1$ servers then if $\mathsf{pw}^* \neq \mathsf{pw}$, $\mathcal{F}_{\text{aPPSS}}$ responds $\perp$, but if $\mathsf{pw}^* = \mathsf{pw}$ then $\mathcal{F}_{\text{aPPSS}}$ leaks the aPPSS-protected secret sk. Note that the ticket-counting mechanism of $\mathcal{F}_{\text{aPPSS}}$ enforces that any aPPSS instance completed by a server can be used either for a

single instance of the honest user reconstructing a secret, or for a single instance of an adversary who uses ppss.testpw to attempt to reconstruct sk using a guessed password pw^*.

On authenticated channels. Functionality $\mathcal{F}_{\text{aPPSS}}$ assumes authenticated channels during *initialization*: When user U specifies, via command ppss.uinit, a set $\mathbf{P}_{sid}$ of servers to initialize a secret-sharing instance, the adversary can only decide whether or not to allow this protocol to complete. This means that the adversary can block any party from communicating with the user, but it cannot divert this initialization to a different set of parties. In particular, only the corruption of parties in $\mathbf{P}_{sid}$ may have an effect on the security of the protocol with consequences as described above. To enforce these conditions, U needs the means to authenticate each $\mathsf{P} \in \mathbf{P}_{sid}$ during initialization which is modeled via the authenticated channel functionality $\mathcal{F}_{\text{AUTH}}$. Importantly, we do not assume authenticated channels in the reconstruction phase of $\mathcal{F}_{\text{aPPSS}}$.

3.2 aPPSS Protocol

In Fig. 4 we show a UC aPPSS scheme, denoted Π_{aPPSS}, based on the PPSS scheme of Jarecki et al. [31]. Protocol Π_{aPPSS} uses UC OPRF, modeled by functionality $\mathcal{F}_{\text{OPRF}}$, and it assumes authenticated channels, modeled by functionality $\mathcal{F}_{\text{AUTH}}$, but it uses the latter only in the Initialization phase. At a high level the protocol proceeds as follows:

Initialization: User U asks for an OPRF evaluation ρ from each server S's $\mathcal{F}_{\text{OPRF}}$ using its password pw, and uses those evaluations as encryption keys for encrypting the threshold shares $\{s_i\}$ generated with the Shamir's secret sharing scheme from a random secret s. Together with the user's password pw, and the encrypted shares $\mathbf{e} = \{e_i\}$, U creates a cryptographic commitment $[C||\mathsf{sk}] = \mathsf{H}(\mathsf{pw}, \mathbf{e}, s)$ and uses sk as the secret key. The ciphertexts $\mathbf{e} = \{e_i\}$ and C are then sent via the authenticated channel (via $\mathcal{F}_{\text{AUTH}}$) and kept at the servers. The user keeps nothing besides remembering the password pw.

Reconstruction: To reconstruct, user U starts with asking for the OPRF evaluation ρ from each server S's $\mathcal{F}_{\text{OPRF}}$ using its password pw along with the ciphertexts $\mathbf{e} = \{e_i\}$ and commitment C. The OPRF evaluations $\{\rho_i\}$ are used to decrypt the ciphertexts to Shamir's shares $\{s_i\}$ which can be used to reconstruct the secret s via interpolation. Finally the user U can recreate $[C||\mathsf{sk}] = \mathsf{H}(\mathsf{pw}, \mathbf{e}, s)$ and obtain sk, after checking that C matches the ones sent by the servers.

Protocol Π_{aPPSS} in Fig. 4 is, up to some small differences (e.g., using a global OPRF functionality) the same as the PPSS of Jarecki et al. [30], except that we replace generic non-malleable commitment used in [30] with a specific RO-based implementation H. However, the novelty here with respect to the PPSS protocol of [30] is its analysis as an *augmented* PPSS.

Public parameters: Security parameter λ, threshold parameters $t, n \in \mathbb{N}$ with $t \leq n$, field $\mathbb{F} := \mathbb{GF}(2^\lambda)$, hash function H with range $\{0,1\}^{2\lambda}$.

Initialization for user U:

1. On input $(\mathsf{ppss.uinit}, sid, \mathsf{pw})$ s.t. $|\mathbf{P}_{sid}| = n$, send $(\mathsf{oprf.eval}, [sid||i||0], \mathsf{pw}, \mathbf{P}_{sid}[i])$ to $\mathcal{F}_{\text{OPRF}}$ for each $i \in [n]$.
2. Wait for messages $(\mathsf{oprf.eval}, [sid||i||0], \rho_i, \mathsf{tr}_i)$ from $\mathcal{F}_{\text{OPRF}}$ and $(\mathsf{sent}, [sid||i||0], \mathbf{P}_{sid}[i], \mathsf{U}, \mathsf{tr}'_i)$ from $\mathcal{F}_{\text{AUTH}}$, for all $i \in [n]$. Abort if $\exists i \in [n]$ s.t. $\mathsf{tr}'_i \neq \mathsf{tr}_i$.
3. Pick $s \leftarrow_\$ \mathbb{F}$, set $(s_1, ..., s_n)$ as a (t, n) Shamir secret sharing of s over $\mathbb{F}$.
4. Set $e_i := s_i \oplus \rho_i$ for $i \in [n]$, set $\mathbf{e} := (e_1, ..., e_n)$, set $[C||\mathsf{sk}] := \mathsf{H}(\mathsf{pw}, \mathbf{e}, s)$ s.t. $|C| = |\mathsf{sk}| = \lambda$. Set $\omega := (\mathbf{e}, C)$.
5. Send $(\mathsf{send}, [sid||i||1], \mathbf{P}_{sid}[i], \omega)$ to $\mathcal{F}_{\text{AUTH}}$ for each $i \in [n]$ and output $(\mathsf{ppss.fininit}, sid, \mathsf{sk})$.

Initialization for server S:

1. On input $(\mathsf{ppss.sinit}, sid, i, \mathsf{U})$, send $(\mathsf{oprf.init}, [\mathsf{S}||sid])$ and $(\mathsf{oprf.sndrcomplete}, [\mathsf{S}||sid], 0)$ to $\mathcal{F}_{\text{OPRF}}$.
2. Given response $(\mathsf{oprf.sndrtrans}, [\mathsf{S}||sid], 0, \mathsf{tr}_\mathsf{S})$ from $\mathcal{F}_{\text{OPRF}}$, send $(\mathsf{send}, [sid||i||0], \mathsf{U}, \mathsf{tr}_\mathsf{S})$ to $\mathcal{F}_{\text{AUTH}}$.
3. On $(\mathsf{sent}, [sid||i||1], \mathsf{U}, \mathsf{P}_i, \omega)$ from $\mathcal{F}_{\text{AUTH}}$, save (sid, i, ω).

Reconstruction for user U:

1. On input $(\mathsf{ppss.urec}, sid, ssid, \mathbf{S}, \mathsf{pw}')$ s.t. $|\mathbf{S}| = t{+}1$, send $(\mathsf{oprf.eval}, [sid||j||ssid], \mathbf{S}[j], \mathsf{pw}')$ to $\mathcal{F}_{\text{OPRF}}$ for $j \in [t{+}1]$.
2. Wait for messages $(\mathsf{oprf.eval}, [sid||j||ssid], \phi_j, \mathsf{tr}_j)$ from $\mathcal{F}_{\text{OPRF}}$ and messages (i_j, ω_j) from $\mathbf{S}[j]$, for all $j \in [t{+}1]$. If $\exists\ j_1 \neq j_2$ s.t. $i_{j_1} = i_{j_2}$ or $\omega_{j_1} \neq \omega_{j_2}$ or $\exists\ j$ s.t. $i_j \notin [n]$ (i.e., if i_j's are not all distinct, or ω_j's are not all the same, or some i_j is out of range $[n]$), output $(\mathsf{ppss.urec}, sid, ssid, \perp)$ and halt. Otherwise set $\rho'_{i_j} := \phi_j$ for $j \in [t{+}1]$ and $I := \{i_j \mid j \in [t{+}1]\}$.
3. Parse any ω_j as $(\mathbf{e}', C')$, parse $\mathbf{e}'$ as $(e'_1, ..., e'_n)$, set $s'_i := e'_i \oplus \rho'_i$ for all $i \in I$.
4. Interpolate $\{(i, s'_i)\}_{i \in I}$ to recover secret s' and shares $\{s'_i\}_{i \notin I}$.
5. Set $[C''||\mathsf{sk}'] := \mathsf{H}(\mathsf{pw}', \mathbf{e}', s')$. If $C' \neq C''$ then reset $\mathsf{sk}' := \perp$.
6. Output $(\mathsf{ppss.finrec}, sid, ssid, \mathsf{sk}')$.

Reconstruction for server S:

1. On input $(\mathsf{ppss.srec}, sid, ssid, \mathsf{U})$, retrieve record (sid, i, ω) (if no such record then abort), send $(\mathsf{oprf.sndrcomplete}, [\mathsf{S}||sid], ssid)$ to $\mathcal{F}_{\text{OPRF}}$ and (i, ω) to U.

Fig. 4: Protocol Π_{aPPSS} which realizes $\mathcal{F}_{\text{aPPSS}}$ in $(\mathcal{F}_{\text{OPRF}}, \mathcal{F}_{\text{AUTH}})$-hybrid world

Theorem 2 *If* H *is a random oracle, then the protocol in Fig. 4 UC-realizes the* $\mathcal{F}_{\text{aPPSS}}$ *functionality assuming access to the OPRF functionality* $\mathcal{F}_{\text{OPRF}}$ *and the message authentication functionality* $\mathcal{F}_{\text{AUTH}}$.

Proof of Theorem 2 is deferred to Appendix D.

4 Augmented Password-Protected Threshold Signature

We introduce our model for Augmented Password-protected Threshold Signature (aptSIG), and we show a secure construction of aptSIG scheme by generic composition of aPPSS and a Threshold Signature (tSIG).

4.1 Modeling Augmented Password-protected Threshold Signature

We model Augmented Password-protected Threshold Signature (aptSIG) using an ideal functionality $\mathcal{F}_{\text{aptSIG}}$, shown in Figure 5 and Figure 6. A (t, n)-threshold aptSIG involves $n + 1$ parties, a *user* U and n *server* $\mathsf{S}_1, ..., \mathsf{S}_n$, and it supports two distributed protocols, *initialization* and *signing*. An initialization protocol generates a public key for a signature scheme and protects the corresponding private key by secret-sharing it and protecting this sharing using user's password pw s.t. the sharing can be reconstructed only using this password. The signing protocol allows the user and the servers to sign any message m as long as (a) the user and at least $t + 1$ of the servers agree to sign it, and (b) the user provides a matching password $\mathsf{pw}' = \mathsf{pw}$ into the signing protocol. Therefore, aptSIG scheme functions as an *outsourced signature service* for party U, where U's secret key is *distributed and password-protected* by the servers, but using the right password lets U obtain signatures as long as $t + 1$ servers agree to sign.

Corruption of up to t out of n servers gives no information to the attacker, while corruption of $t + 1$ or more servers allows the attacker to reconstruct only password-protected data. In particular, the data collected from all servers allows the attacker an *offline dictionary attack* against the password, but that is *all that it allows*. If the attacker finds the password via this offline search then security is gone, and in our scheme the attacker reconstructs the signature private key, but if the password is chosen with high-enough entropy and the dictionary attack fails then the attacker gets no information about the signature key even if it corrupts all n servers. We stress that in a secure aptSIG scheme the signing key can never be reconstructed in one place. In particular, if the password leaks but the adversary compromises fewer than $t + 1$ servers then signatures can only be created via the on-line signing protocol. Consequently, servers S_i can function as *rate limiters* or *policy limiters*, i.e. they can apply whatever policy the environment specifies regarding the messages they can sign.

Ideal Functionality $\mathcal{F}_{\text{aptSIG}}$**.** In what follows we explain the security properties imposed by the ideal functionality $\mathcal{F}_{\text{aptSIG}}$ of Figure 5 and Figure 6. Since we show that our aptSIG protocol of Section 4.2 securely realizes this functionality, this will in particular imply the security properties of that aptSIG scheme.

Notation: *(This figure uses the same notation as in $\mathcal{F}_{\text{aPPSS}}$, see Figure 3)*

Initialization:

[I.U] On $(\mathsf{ptsig.uinit}, sid, \mathsf{pw})$ from party U for $sid = (..., \mathbf{P}_{sid})$ s.t. $|\mathbf{P}_{sid}| = n$, send $(\mathsf{ptsig.uinit}, sid, \mathsf{U})$ to $\mathcal{A}^*$, save $(sid, \mathsf{U}, \mathbf{P}_{sid}, \mathsf{pw})$ and set flag $\mathsf{flag}_{sid} = 0$. Ignore further $\mathsf{ptsig.uinit}$ calls for same sid.

[I.S] On $(\mathsf{ptsig.sinit}, sid, i, \mathsf{U})$ from party S, or $(\mathsf{ptsig.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{Corr}$, send $(\mathsf{ptsig.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ to $\mathcal{A}^*$, save $(sid, \mathsf{U}, \mathsf{S}, i)$.

[I.F] On $(\mathsf{ptsig.uinit}, sid, \mathsf{V})$ from $\mathcal{A}^*$, if $\exists$ record $(sid, \mathsf{U}, \mathbf{P}_{sid}, \mathsf{pw})$ and records $(sid, \mathsf{U}, \mathsf{S}, i)$ for each $\mathsf{S} \in \mathbf{P}_{sid}$, then create record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ and send $(\mathsf{ptsig.verificationkey}, sid, \mathsf{V})$ to U.

Signing:

[S.U] On $(\mathsf{ptsig.usign}, sid, ssid, \mathbf{S}, \mathsf{pw}', \mathsf{m})$ from party U' or from $\mathsf{U}' = \mathcal{A}^*$, send $(\mathsf{ptsig.usign}, sid, ssid, \mathsf{U}', \mathbf{S}, \mathsf{m})$ to $\mathcal{A}^*$. If $\exists$ record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ then save $(sid, ssid, \mathsf{U}', \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{pw}', \mathsf{V}, \mathsf{m})$, else save $(sid, ssid, \mathsf{U}', \perp, \perp, \mathsf{pw}', \perp, \mathsf{m})$. Ignore further $\mathsf{ptsig.usign}$ calls for same $ssid$.

[S.S] On $(\mathsf{ptsig.ssign}, sid, ssid, \mathsf{U}', \mathsf{m})$ from party S or $(\mathsf{ptsig.ssign}, sid, ssid, \mathsf{S}, \mathsf{U}', \mathsf{m}, b)$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{Corr}$, if $\exists$ record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ s.t. $\mathsf{S} \in \mathbf{P}_{sid}$ then send $(\mathsf{ptsig.ssign}, sid, ssid, \mathsf{S}, \mathsf{U}', \mathsf{m})$ to $\mathcal{A}^*$, save $(sid, \mathsf{m}, \mathsf{S})$, set $\mathsf{tx}(\mathsf{S})$++ if S is honest or $b = 1$.

[S.P] On $(\mathsf{ptsig.pretest}, sid, ssid, \mathbf{C}, \mathsf{flag}, \mathsf{pw}^*)$ from $\mathcal{A}^*$, if $\exists$ $\mathsf{rec} = (sid, ssid, \mathsf{U}', \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{pw}', \cdot, \cdot)$ not marked as $\mathsf{pretested}(c)$ for any c then:

[S.P.1] if $\mathsf{flag} = 1$, $|\mathbf{C}| = t{+}1$, and $\forall_{\mathsf{S} \in \mathbf{C}}(\mathsf{tx}(\mathsf{S}){>}0)$, then set $\mathsf{tx}(\mathsf{S})--$ for all $\mathsf{S} \in \mathbf{C}$, set $b := (\mathsf{pw}' == \mathsf{pw})$, send b to $\mathcal{A}^*$ and mark rec as $\mathsf{pretested}(b)$;

[S.P.1*] moreover, if $b = 1$ and $\mathsf{U}' \in \mathbf{Corr} \cup \{\mathcal{A}^*\}$ set $\mathsf{flag}_{sid} = 1$;

[S.P.2] if $\mathsf{flag} = 2$ then set $b := (\mathsf{pw}' == \mathsf{pw}^*)$, send b to $\mathcal{A}^*$, and if $b = 1$ then mark rec as $\mathsf{pretested}(2)$ else mark rec as $\mathsf{pretested}(0)$;

[S.F] On $(\mathsf{ptsig.finsign}, sid, ssid, \mathbf{C}', \mathsf{flag}, \sigma^*, \mathsf{m}^*)$ from $\mathcal{A}^*$, retrieve $\mathsf{rec} = (sid, ssid, \mathsf{U}', \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{pw}', \mathsf{V}, \mathsf{m})$ and do:

[S.F.0] if $\mathsf{m} = \perp$ and $\mathsf{U}' \in \mathbf{Corr} \cup \{\mathcal{A}^*\}$ reset $\mathsf{m} := \mathsf{m}^*$;

[S.F.1] if $\mathsf{flag}_{sid} = 1$, rec is marked $\mathsf{pretested}(0)$, and $\mathsf{U}' \in \mathbf{Corr} \cup \{\mathcal{A}^*\}$, then change rec mark to $\mathsf{pretested}(1)$;

[S.F.F] send $(\mathsf{ptsig.finsign}, sid, ssid, \mathsf{m}, \sigma)$ to U' s.t.

[S.F.F.1] if $\mathsf{flag} = 1$, rec is marked $\mathsf{pretested}(1)$, $|\mathbf{C}'| = t{+}1$, $\mathbf{C}' \subseteq \mathbf{P}_{sid}$, $\exists$ record $(sid, \mathsf{m}, \mathsf{S})$ for all $\mathsf{S} \in \mathbf{C}'$, $\mathsf{V}(\mathsf{m}, \sigma^*) = 1$, and there is no saved record $(sid, \mathsf{m}, \sigma^*, 0)$, then save record $(sid, \mathsf{m}, \sigma^*, 1)$ and set $\sigma := \sigma^*$;

[S.F.F.2] if $\mathsf{flag} = 2$ and rec is marked $\mathsf{pretested}(2)$ then set $\sigma := \sigma^*$;

[S.F.F.3] if neither of the above two cases is met set $\sigma := \perp$.

Verification:

On $(\mathsf{ptsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$ from Q, send $(\mathsf{ptsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$ to $\mathcal{A}^*$ and do:

[V.1] if $\exists$ records $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ and $(sid, \mathsf{m}, \sigma, \beta')$ then set $\beta := \beta'$;

[V.2] else, if $\exists$ record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ but no $(sid, \mathsf{m}, \sigma, 1)$ for any σ then set $\beta := 0$;

[V.3] else set $\beta := \mathsf{V}(\mathsf{m}, \sigma)$.

[V.V] Record $(sid, \mathsf{m}, \sigma, \beta)$ and send $(\mathsf{ptsig.verified}, sid, \mathsf{m}, \sigma, \beta)$ to Q.

Fig. 5: $\mathcal{F}_{\text{aptSIG}}$: Ideal Functionality for Password-Protected Threshold Signature

Notation: *(This figure uses the same notation as in $\mathcal{F}_{\text{aPPSS}}$, see Figure 3)*

Server Compromise: (*This query requires permission from the environment.*)
[SC] On $(\mathsf{ptsig.corrupt}, sid, \mathsf{P})$ from $\mathcal{A}^*$, set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$.

Password Test:
[PT] On $(\mathsf{ptsig.testpw}, sid, \mathsf{S}, \mathsf{pw}^*)$ from $\mathcal{A}^*$, retrieve record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$. If $\mathsf{tx}(\mathsf{S}) > 0$ then add S to set $\mathsf{ppss.pwtested}(\mathsf{pw}^*)$ and set $\mathsf{tx}(\mathsf{S})$−−. If $|\mathsf{ppss.pwtested}(\mathsf{pw}^*)| = t + 1$ then return bit $b = (\mathsf{pw}^* == \mathsf{pw})$ to $\mathcal{A}^*$. If $b = 1$ set $\mathsf{flag}_{sid} = 1$.

Fig. 6: Adversarial Interfaces of $\mathcal{F}_{\text{aptSIG}}$

(1) $\mathcal{F}_{\text{aptSIG}}$: honest party operation. Query $(\mathsf{ptsig.uinit}, sid, \mathsf{pw})$ from U models user U starting initialization on a password pw with n servers specified in identifier sid. (Using the convention of aPPSS, we assume $sid = (sid', \mathbf{P}_{sid})$ for $\mathbf{P}_{sid} = (\mathsf{S}_1, ..., \mathsf{S}_n)$.) Query $(\mathsf{ptsig.uinit}, sid, i, \mathsf{U})$ from $\mathsf{S} \in \mathbf{P}_{sid}$ models server S entering into an initialization protocol, as an i-th server in list $\mathbf{P}_{sid}$, with U as an intended "owner" of this password-protected signature instance. Query $(\mathsf{ptsig.uinit}, sid, \mathsf{V})$ from $\mathcal{A}^*$ models the ideal-world adversary allowing an initialization instance identified by sid to complete, and U to output the public key V. Note that all parties input the identities of all participants into the protocol, and $\mathcal{F}_{\text{aptSIG}}$ reacts to query $\mathsf{ptsig.uinit}$ only if all intended parties participate in the initialization. This is realizable if U and each S_i can authenticate each other, and our aptSIG protocol indeed relies on authenticated channels in initialization. The public key V is associated with initialization identifier sid in the sense that sid serves as a handle to the *password-protected secret-sharing (ppss)* of a private signing key corresponding to V. (Functionality $\mathcal{F}_{\text{aptSIG}}$ does not ensure that this sharing is successfully established when U outputs V, but $\mathcal{F}_{\text{aptSIG}}$ allows U to verify it, e.g. if U invokes the signing protocol on a test message.)

Once key V is created, query $(\mathsf{ptsig.usign}, sid, ssid, \mathbf{S}, \mathsf{pw}', \mathsf{m})$ from U' models user U' (possibly using a different platform than U, hence a different name tag U') who holds password pw' (which might or might not equal to pw) starting a signing protocol instance on message m and a ppss-protected key identified by sid. Identifier $ssid$ is a handle of U' on that instance, and $\mathbf{S}$ is a subset of $t + 1$ servers with whom U intends to communicate. However, $\mathcal{F}_{\text{aptSIG}}$ doesn't enforce authentication in signing, and the signing instance record it creates, $(sid, ssid, \mathsf{U}', \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{pw}', \mathsf{V}, \mathsf{m})$ ignores field $\mathbf{S}$. Query $(\mathsf{ptsig.ssign}, sid, ssid, \mathsf{U}', \mathsf{m})$ from S models S agreeing to sign m using the ppss-protected key identified by sid. Field U' is a counterparty address, $ssid$ is S's local instance handle, but they play no security roles and $\mathcal{F}_{\text{aptSIG}}$ ignores them. In particular, $\mathcal{F}_{\text{aptSIG}}$ does not enforce equality of $ssid$ or U' tags used by the participants in signing.

(2) $\mathcal{F}_{\text{aptSIG}}$: signature completion. Signing protocol output is controlled by two queries by an ideal-world adversary $\mathcal{A}^*$: $\mathsf{ptsig.pretest}$ and $\mathsf{ptsig.finsign}$.

$\mathcal{F}_{\text{aptSIG}}$ associates servers $\mathsf{S} \in \mathbf{P}_{sid}$ with ticket counters $\mathsf{tx}(\mathsf{S})$, as in the aPPSS functionality $\mathcal{F}_{\text{aPPSS}}$ of Section 3, and each S can trigger $\mathcal{F}_{\text{aptSIG}}$ to record $(sid, \mathsf{m}, \mathsf{S})$ which stands for S agreeing to sign m, as in the tSIG functionality $\mathcal{F}_{\text{tSIG}}$ of Section 2. When S issues a query $(\mathsf{ptsig.ssign}, ..., \mathsf{m})$ then $\mathcal{F}_{\text{aptSIG}}$ increments $\mathsf{tx}(\mathsf{S})$ and records $(sid, \mathsf{m}, \mathsf{S})$ at the same time.

Queries ptsig.pretest and ptsig.finsign serve two purposes: The first one, denoted by $\mathcal{A}^*$ using $\mathsf{flag} = 1$, is a passive completion of the signing instance. First, $\mathcal{A}^*$ can use ptsig.pretest with $\mathsf{flag} = 1$ to "pre-complete" that instance and learn if party U' runs the protocol on the correct password $\mathsf{pw}' = \mathsf{pw}$. This is akin to TestAbort query in the UC aPAKE model [26]: A protocol can make it detectable whether U' runs on the correct password, e.g. because otherwise U' aborts, in which case the adversary learns if $\mathsf{pw}' = \mathsf{pw}$ by observing the protocol. In this test, $\mathcal{A}^*$ must specify a subset $\mathbf{C}$ of $t+1$ servers with non-zero ticket counters (which $\mathcal{F}_{\text{aptSIG}}$ decrements), which enforces that U' finalization requires $t+1$ participating servers. Note that these servers can run on different messages than U', i.e. $\mathcal{A}^*$ can mix and match S sessions in completing ptsig.pretest.

If $\mathsf{pw}' = \mathsf{pw}$ then $\mathcal{A}^*$ can follow up the $(\mathsf{ptsig.pretest}, ..., \mathsf{flag} = 1, ...)$ query with $(\mathsf{ptsig.finsign}, ..., \mathbf{C}', \mathsf{flag} = 1, \sigma^*, \perp)$, which corresponds to finalizing the signing instance on message m with signature σ^*. Indeed, if U' runs on the correct password and the attacker is passive then U' can output a signature. $\mathcal{F}_{\text{aptSIG}}$ processes this query in the same way as the threshold signature functionality $\mathcal{F}_{\text{tSIG}}$ of Section 2, i.e. it checks that $t+1$ servers in subset $\mathbf{C}'$ agreed to sign m, that σ^* was not previously recorded as a faulty signature, and that $\mathsf{V}(\mathsf{m}, \sigma^*) = 1$, and if all conditions are met then it outputs σ^* to U' and declares σ^* as a valid signature on m by recording a "signature" tuple $(sid, \mathsf{m}, \sigma^*, 1)$. These tuples control the outputs of a signature verification query ptsig.verify, and $\mathcal{F}_{\text{aptSIG}}$ handles that exactly as $\mathcal{F}_{\text{tSIG}}$, i.e. if there is no recorded tuple $(sid, \mathsf{m}, \sigma^*, 1)$ then $(\mathsf{ptsig.verify}, ..., \mathsf{m}, \sigma^*, \mathsf{V})$ query should return 0.

We note that $\mathcal{F}_{\text{aptSIG}}$ does not enforce that $\mathbf{C}' = \mathbf{C}$, i.e. the adversary is allowed to mix-and-match servers and use a different subset $\mathbf{C}$ of server instances to "pre-complete" a signature session via the ptsig.pretest query, and a different subset $\mathbf{C}'$ to complete the session via the ptsig.finsign query. Moreover, the second set of servers must be signing m, but the first one might not. We allow this "disconnection" in $\mathcal{F}_{\text{aptSIG}}$ to enable an efficient aptSIG protocol of Section 4.2, which does not enforce $\mathbf{C}' = \mathbf{C}$. However, the practical import of adversary replacing part of m-signing server session with parts taken from some m'-signing server session seems innocuous, given that in the end a signature on m cannot be created unless a pw-holding user and $t+1$ servers all agree to it.

(3) $\mathcal{F}_{\text{aptSIG}}$: active attacks. The first type of active attack is an on-line password guessing attack against honest servers, where $\mathcal{A}^*$ poses as a user, or employs a corrupt user U', and runs a signing protocol via interface ptsig.usign on some password pw' ([S.U]), followed by ptsig.pretest and ptsig.finsign with $\mathsf{flag} = 1$ ([S.P.1]). The same logic as above will apply to this sequence, except since the adversary contributed pw' in ptsig.usign, the same interface will reveal if $\mathsf{pw}' = \mathsf{pw}$ (in [S.P.1] the functionality sends this bit to the adversary). Moreover, each pt-

SIG instance *sid* is associated with a flag flag_{sid} which switches from 0 to 1 if it ever happens that the adversary found password pw' in this way ([S.P.1*]) (or via offline attacks, see below). The consequence of $\mathsf{flag}_{sid} = 1$ is that any adversarial signing instance, even one that starts with an incorrect password pw', and consequently its reconstruction record rec would be marked $\mathsf{pretested}(0)$ in ptsig.pretest, is effectively treated in ptsig.finsign as if it was marked $\mathsf{pretested}(1)$, which means that the functionality will "sign" message m^* in this signing session (as long as $t+1$ servers also agree to sign it) ([S.F.1]). In other words, if the adversary guesses the right password on some ptSIG session, then we allow him to "late switch" any incorrect password to the correct one on all his other signing sessions.

The second type of active attack is an on-line password guessing attack against an honest user. This is modeled via ptsig.pretest ([S.P.2]) and ptsig.finsign queries with $\mathsf{flag} = 2$ ([S.F.F.2]). Here $\mathcal{A}^*$ can set $\mathbf{C} = \perp$, but must enter a password guess pw^*, and in ptsig.pretest it will learn if $\mathsf{pw}' = \mathsf{pw}^*$ where pw' is a password used by an honest user U' ([S.P.2]). If not then U' can subsequently only abort, but if so then subsequent ptsig.finsign makes U' output as signature an arbitrary value σ^* chosen by $\mathcal{A}^*$ ([S.F.F.2]). This reflects the fact that the only security hedge which U' enters into signing is its password pw', so if an online attacker guesses pw', the attacker can wlog. run aptSIG initialization on pw' and then run the aptSIG signing on the resulting values, thus making U' output e.g. a signature on m but issued by an adversarial key. However, this attack does not imply signature forgery, because $\mathcal{F}_{\text{aptSIG}}$ does not add tuple $(sid, \mathsf{m}, \sigma^*, 1)$ to its records. In particular, a user could run signature verification $(\mathsf{ptsig.verify}, sid, \mathsf{m}, \sigma^*, \mathsf{V})$ on its aptSIG output, and in case of the above attack she would learn that σ^* is not a valid signature **and** that she was subject of an active attack by someone who learned her password pw'. [3]

(4) $\mathcal{F}_{\text{aptSIG}}$: adaptive server corruptions and ODA. Adversary $\mathcal{A}^*$ can adaptively corrupt any server S ([SC]), which allows $\mathcal{A}^*$ to (1) freely issue tickets for S, using ptsig.ssign with $b = 1$, and (2) freely issue S's "partial signatures" on arbitrary messages m, using ptsig.ssign with $\mathsf{m} \neq \perp$ ([S.S]). The latter actions can result in signatures if U using the correct password $\mathsf{pw}' = \mathsf{pw}$ wants to sign the same m ([S.F.F.1]), or if the attacker learns pw and invokes user-side on that pw and m. The former actions allow the attacker to test passwords via command ptsig.testpw, which lets $\mathcal{A}^*$ exchange $t+1$ tickets from some $t+1$ servers for an off-line test of *one* password guess pw^* specified by $\mathcal{A}^*$. Note that corrupt S_i's these tickets are "free" to $\mathcal{A}^*$ so after corrupting $t+1$ servers these tests can be done fully offline, but if $\mathcal{A}^*$ needs to add the tickets from honest servers to this mix then only one such ticket is created in each signing instance S_i runs, i.e. if adversary corrupts $t' \leq t+1$ servers then it can test q passwords only by on-line interactions with $q * (t+1-t')$ servers ([PT]).

[3] $\mathcal{F}_{\text{aptSIG}}$ lets $\mathcal{A}^*$ set the user instance's message m to arbitrary m^* in the finalization of the signing protocol, but only for adversarial user instances, i.e. we allow adversarial signing instances to "late-commit" to their messages.

Crucially, *even if all servers are corrupted*, attacker $\mathcal{A}^*$ has no avenue to forge message signatures unless $\mathcal{A}^*$ finds out user's password pw and runs $\mathsf{ptsig.usign}$ (e.g. as corrupt U') on pw. (Moreover, if fewer than $t+1$ servers are corrupt than even knowing pw lets $\mathcal{A}^*$ sign only messages which some uncorrupted servers agree to sign.) Moreover, the only avenues to finding password pw ([PT]) consist of (1) *online* guessing attacks against either the servers or the user as long as $\mathcal{A}^*$ corrupts fewer than $t+1$ servers, and (2) (fully) *offline* dictionary attacks (ODA), as explained above, enabled once $\mathcal{A}^*$ corrupts $t+1$ servers.

User/message authentication and Perfect Forward Secrecy. In the aptSIG ideal model $\mathcal{F}_{\text{aptSIG}}$, when servers sign they take input m from the environment, and they do not know if their counterparty holds the right password, and even if they do then whether they authorize signing this message. A model which assures both properties extends the aptSIG model to capture perfect forward security (PFS), because it would imply that if no password-holding entity wants to sign some message at a given time, then the adversary who might capture the password in the future, cannot "redo" these signature instances, and can only use the compromised password on new signature sessions.

The PFS property can be added in black-box way by running two instances of aptSIG: Consider a modified signing protocol which executes two instances of aptSIG, first one on the message m concatenated with nonce *ssid*, and only if this one creates a valid signature on the $\mathsf{m}, ssid$, then the proper aptSIG instance would execute on just m. The first aptSIG instance accomplishes the above requirements, because only a correct password could have caused this aptSIG instance to issue a valid signature on the $\mathsf{m}, ssid$ pair.

In Appendix G we define a PFS version of the aptSIG ideal model, denoted $\mathcal{F}_{\text{aptSIG-PFS}}$, and we show that the efficient aptSIG scheme which we show in the next subsection, can be adapted more efficiently to implement the PFS property. The idea is very similar to the one above except that the first instance of aptSIG is replaced by a standard signature made on pair $\mathsf{m}, ssid$ by the user U. Indeed, efficiency-wise the PFS protocol variant shown in Appendix G adds only the cost of issuing a single standard signature for user U and a signature verification for each server S. See Appendix G for more details.

4.2 Generic aptSIG Protocol

In Figure 7 we show a generic construction of an augmented password-protected threshold signature (aptSIG), using an augmented Password-Protected Secret Sharing (aPPSS) and a Threshold Signature (tSIG). The protocol in addition relies on functionality $\mathcal{F}_{\text{AUTH}}$ but it is used only in initialization. The protocol also relies on an Equivocable Authenticated Encryption scheme, denoted AE.

Threshold Signature Protocol Π_{tSIG}. In the description of protocol Π_{tSIG} in Figure 7, we don't use the threshold signature functionality $\mathcal{F}_{\text{tSIG}}$, but use the tSIG protocol directly. We choose this way of describing the aptSIG scheme because whereas the server parties $\mathsf{P}_i \in \mathbf{P}$ can store secret state between tSIG

Public parameters: Security parameter λ, threshold parameters t, n s.t. $t \leq n$.
Let $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ be an Equivocable Authenticated Encryption, and let $\mathsf{tSIG} = (\Pi_{\mathsf{TKeyGen}}, \Pi_{\mathsf{TSign}}, \Pi_{\mathsf{TVerify}})$ be a Threshold Signature scheme realizing functionality $\mathcal{F}_{\text{tSIG}}$ (see text). $\mathsf{add}(sid, \mathsf{U})$ parses $sid = (sid', \mathbf{P}_{sid})$ and outputs $sid^+ = (sid', \mathbf{P}^+_{sid})$ s.t. if $\mathbf{P}_{sid} = (\mathsf{P}_1, ..., \mathsf{P}_n)$ then $\mathbf{P}^+_{sid} = (\mathsf{U}, \mathsf{P}_1, ..., \mathsf{P}_n)$.

Initialization for user U:

1. On input $(\mathsf{ptsig.uinit}, sid, \mathsf{pw})$, send $(\mathsf{ppss.uinit}, sid, \mathsf{pw}, \perp)$ to $\mathcal{F}_{\text{aPPSS}}$, and let sk denote $\mathcal{F}_{\text{aPPSS}}$'s output.
2. Run $\mathsf{tSIG}.\Pi_{\mathsf{TKeyGen}^+}$ on input $sid^+ = \mathsf{add}(sid, \mathsf{U})$. Let $(\mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U})$ and V be resp. U's local output and the generated public key.
3. Set $\mathsf{aec}_\mathsf{U} := \mathsf{AE.AuthEnc}_{\mathsf{sk}}(\mathsf{U}, \mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U})$, send $(\mathsf{send}, sid, \mathsf{P}_i, \mathsf{aec}_\mathsf{U})$ to $\mathcal{F}_{\text{AUTH}}$ for all $\mathsf{P}_i \in \mathbf{P}_{sid}$, output $(\mathsf{ptsig.verificationkey}, sid, \mathsf{V})$.

Initialization for server S:

1. On input $(\mathsf{ptsig.sinit}, sid, i, \mathsf{U})$ send $(\mathsf{ppss.sinit}, sid, i, \mathsf{U})$ to $\mathcal{F}_{\text{aPPSS}}$ and run $\mathsf{tSIG}.\Pi_{\mathsf{TKeyGen}^+}$ on $sid^+ = \mathsf{add}(sid, \mathsf{U})$. Let $(\mathsf{ts}_i, \mathsf{tcs}_i)$ be S's local output.
2. On message $(\mathsf{sent}, sid, \mathsf{U}, \mathsf{S}, \mathsf{aec}_\mathsf{U})$ from $\mathcal{F}_{\text{AUTH}}$, save $(sid, sid^+, \mathsf{ts}_i, \mathsf{tcs}_i, \mathsf{aec}_\mathsf{U})$.

Signing for user U'

1. On input $(\mathsf{ptsig.usign}, sid, ssid, \mathbf{S}, \mathsf{pw}', \mathsf{m})$ for $|\mathbf{S}| \geq t{+}1$ from U', send $(\mathsf{ppss.urec}, sid, ssid, \mathbf{S}, \mathsf{pw}')$ to $\mathcal{F}_{\text{aPPSS}}$, and wait to receive $(\mathsf{ppss.urec}, sid, ssid, \mathsf{sk})$ from $\mathcal{F}_{\text{aPPSS}}$ and message $(sid, \mathsf{aec}_\mathsf{U})$ from all $\mathsf{S} \in \mathbf{S}$.
2. Output $(\mathsf{ptsig.usign}, sid, ssid, \mathsf{m}, \perp)$ and abort if either (1) $\mathsf{sk} = \perp$, or (2) it is not the case that all $\mathsf{S} \in \mathbf{S}$ send the same message $(sid, \mathsf{aec}_\mathsf{U})$, or (3) $\mathsf{AE.AuthDec}_{\mathsf{sk}}(\mathsf{aec}_\mathsf{U})$ returns $\perp$.
3. Otherwise, let $(\mathsf{U}, \mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U}) = \mathsf{AE.AuthDec}_{\mathsf{sk}}(\mathsf{aec}_\mathsf{U})$, set $sid^+ = \mathsf{add}(sid, \mathsf{U})$, run protocol $\mathsf{tSIG}.\Pi_{\mathsf{TSign}^+}$ on input $(sid^+, \mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U}, \mathsf{m})$, and when this protocol outputs σ, output $(\mathsf{ptsig.finsign}, sid, ssid, \mathsf{m}, \sigma)$.

Signing for server S

1. On input $(\mathsf{ptsig.ssign}, sid, ssid, \mathsf{U}', \mathsf{m})$ from S, retrieve stored tuple $(sid, sid^+, \mathsf{ts}_i, \mathsf{tcs}_i, \mathsf{aec}_\mathsf{U})$, send $(\mathsf{ppss.srec}, sid, ssid, \mathsf{U}')$ to $\mathcal{F}_{\text{aPPSS}}$, send $(sid, \mathsf{aec}_\mathsf{U})$ to U', and run $\mathsf{tSIG}.\Pi_{\mathsf{TSign}^+}$ on input $(sid^+, \mathsf{ts}_i, \mathsf{tcs}_i, \mathsf{m})$.

Verification for Q

1. On input $(\mathsf{ptsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$ from Q, runs $\beta = \mathsf{tSIG}.\Pi_{\mathsf{TVerify}}(\mathsf{V}, \mathsf{m}, \sigma)$, and output $(\mathsf{ptsig.verified}, sid, \mathsf{m}, \sigma, \beta)$

Fig. 7: Protocol Π_{aptSIG} which realizes $\mathcal{F}_{\text{aptSIG}}$ in $(\mathcal{F}_{\text{aPPSS}}, \mathcal{F}_{\text{AUTH}})$-hybrid world

initialization and signature phases, the user party U is assumed to have no secure storage (except for memorizing the password), hence it is in particular incapable of locally storing the secret share generated in key generation of tSIG. Indeed, we use the aPPSS scheme together with the authenticated encryption AE to "securely transmit" this user's tSIG state between initialization and signature phase, but since this secure transmission can fail, i.e., in case of successful password-guessing attack on aPPSS, an honest user may execute tSIG on adversarially chosen inputs. In essence, our aptSIG protocol runs the real-world tSIG protocol rather than an ideal functionality $\mathcal{F}_{\text{tSIG}}$, because functionality $\mathcal{F}_{\text{tSIG}}$ does not support a party running the signing protocol on the inputs which do not correspond to the state created by the key generation for this party. Note that this proof technique was used in the analysis of the OPAQUE protocol [33], for the same reason that a UC-secure protocol tool, UC AKE in OPAQUE and UC tSIG here, is used within a protocol on keys which might not match the ones prescribed by the protocol.

tSIG Functionality and Communication Setting. We assume that the tSIG scheme consists of (1) protocol Π_{TKeyGen}, which implements $\mathcal{F}_{\text{tSIG}}$ command $(\mathsf{tsig.keygen}, sid')$ for $sid' = (sid, \mathbf{P}^+)$; (2) protocol Π_{TSign}, which implements $\mathcal{F}_{\text{tSIG}}$ command $(\mathsf{tsig.sign}, sid, \mathsf{m})$; and (3) algorithm $\Pi_{\mathsf{TVerify}}(\mathsf{V}, \mathsf{m}, \sigma)$ which implements $(\mathsf{tsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$, which simply returns $\mathsf{V}(\mathsf{m}, \sigma)$. Note that set $\mathbf{P}^+$ is a list of $n+1$ tSIG participants, and we form it by prepending the user party identifier U to the list of server identifiers $\mathbf{P} = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$.

We use ts_i to denote the state created for player P_i by the distributed key generation protocol Π_{TKeyGen}, including $i = \mathsf{U}$. (In the following we will use P_U and U interchangeably.) However, many threshold signature schemes assume that protocol parties have access to some additional secure communication channels, in the very least secure point-to-point channels and often also a reliable authenticated broadcast channel. (These are the communication assumptions of most work on threshold cryptosystems, including e.g. the UC threshold ECDSA of [13] and the threshold BLS scheme in Section 2, albeit the latter only in the initialization phase.) Whereas aptSIG servers can be connected by such channels, we cannot assume this for the user. Indeed, in aptSIG initialization we assume user U has only point-to-point authenticated channels with each server S_i, and in aptSIG signing we assume a plain network. If threshold signature protocols Π_{TKeyGen} and/or Π_{TSign} make such communication assumptions, in the initialization phase our aptSIG prepends protocol Π_{TKeyGen} with a subprotocol which adds U to this assumed communication setting.

For the above communication setting, this subprotocol could work as follows: Since in aptSIG initialization U and each S_i have pairwise authenticated channels, these can be upgraded to secure channels using key exchange, e.g. Diffie-Hellman, executed between U and each S_i. As for authenticated broadcast, it is typically implemented using PKI (e.g. assuming partial synchrony and reliable message delivery between uncorrupted parties), in which case U can generate a signing key, deliver it over authenticated channels to each S_i, and S_i's can agree on it using their authenticated broadcast channels. Likewise, each S_i

can send the list of all servers' public keys to U, and U can reject unless all the lists are the same. We denote this extended Π_{TKeyGen} protocol as Π_{TKeyGen^+}, and we use tcs_i to denote the secure communication tokens each P_i retains from it in subsequent Π_{TKeyGen} and Π_{TSign} executions. Whereas each server S_i can update its pre-existing communication tokens with tcs_i's output by Π_{TKeyGen^+}, user U cannot retain state between executions. However, we solve this by adding the communication tokens tcs_U to the threshold signature state ts_U created by Π_{TKeyGen}, and we encrypt both using the aPPSS-protected key.

Equivocable Authenticated Encryption. Protocol Π_{aptSIG} uses symmetric Authenticated Encryption scheme $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ to encrypt the local state of U output by Π_{TKeyGen^+}. We denote this state as $(\mathsf{U}, \mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U})$, where $\mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U}$ are explained above, and identity U needs to be retained as well because tSIG assumes that its identifier sid^+ includes the identities of all tSIG participants, i.e. $\mathbf{P}^+ = (\mathsf{U}, \mathsf{P}_1, ..., \mathsf{P}_n)$, and aptSIG should allow the user to retrieve signatures using the password only, i.e. it should not rely on the user remembering the identifier U used in the initialization.

We need the authenticated encryption to have *ciphertext integrity* under a single chosen message attack. This is a relaxation of standard ciphertext integrity security notion for authenticated encryption, included in Appendix C. We also require the scheme AE to be *equivocable*, i.e. in the scenario where the adversary gets a ciphertext followed by the key, there is a simulator that can create an indistinguishable ciphertext with no information about the plaintext except for its length, and then create the key to decrypt this ciphertext to any given plaintext. Formally, we call an (authenticated) encryption AE *equivocable* if there is an efficient simulator SIM s.t. for any efficient algorithm $\mathcal{A}$, the distinguishing advantage $\mathsf{Adv}^{\mathrm{eqv},\mathsf{ae}}_{\mathcal{A}}(\lambda) = |p^0_{\mathcal{A}} - p^1_{\mathcal{A}}|$ is a negligible function of λ, where $p^b_{\mathcal{A}} = \Pr[1 \leftarrow \mathcal{A}(\mathsf{st}_\mathcal{A}, \mathsf{aec}, \mathsf{sk}) \mid (\mathsf{st}_\mathcal{A}, \mathsf{aec}, \mathsf{sk}) \leftarrow \mathsf{Exp}^b]$, and

$$\begin{aligned}\mathsf{Exp}^0 &: (\mathsf{m}, \mathsf{st}_\mathcal{A}) \leftarrow \mathcal{A}(\lambda), \mathsf{sk} \leftarrow\!\!\$\ \{0,1\}^\lambda, \mathsf{aec} \leftarrow \mathsf{AuthEnc}_\mathsf{sk}(\mathsf{m})\\ \mathsf{Exp}^1 &: (\mathsf{m}, \mathsf{st}_\mathcal{A}) \leftarrow \mathcal{A}(\lambda), (\mathsf{aec}, \mathsf{st}_\mathsf{S}) \leftarrow \mathsf{SIM}(|\mathsf{m}|), \mathsf{sk} \leftarrow \mathsf{SIM}(\mathsf{st}_\mathsf{S}, \mathsf{m})\end{aligned}$$

Note that equivocability implies standard semantic security of encryption. In the following we will use the term *Equivocable Authenticated Encryption* if encryption is (1) equivocable and (2) has ciphertext integrity. These properties are easy to achieve in an idealized model like ROM [33], e.g. $\mathsf{E}(\mathsf{sk}, \mathsf{m}) = \mathsf{m} \oplus G(\mathsf{sk})$ is equivocable if G is a random oracle. If an equivocable encryption is extended to authenticated encryption, e.g. by computing a MAC on the ciphertext, this achieves ciphertext integrity but does not effect equivocation because the authentication mechanism is computed over the ciphertext.

4.3 Security of the aptSIG Protocol

Simulation Overview. We construct a simulator SIM which will show that no environment $\mathcal{Z}$ can distinguish the ideal-world and real-world interactions. Since protocol Π_{aptSIG} relies on the UC security of three components, aPPSS, tSIG,

and AUTH, we first overview how the real world and the ideal world interactions involve the protocols, functionalities, or simulators of these components. Figure 15 in appendix F captures the top-level view of these interactions in the real-world and ideal-world executions. Without loss of generality we assume a "dummy" adversary $\mathcal{A}^*$ that is an adversary who merely passes all its messages and computations to the environment $\mathcal{Z}$. Our proof assumes that the real execution happens in the $(\mathcal{F}_{\text{aPPSS}}, \mathcal{F}_{\text{AUTH}})$-hybrid world, and below we omit the details of interactions with the adversary where in the ideal world SIM will emulate $\mathcal{F}_{\text{aPPSS}}$ and $\mathcal{F}_{\text{AUTH}}$, because that part of the simulation is trivial: SIM gains all the information needed from $\mathcal{A}^*$'s interfaces to these functionalities, and simply follows the code of $\mathcal{F}_{\text{aPPSS}}$ and $\mathcal{F}_{\text{AUTH}}$.

Simulator SIM interacts with the ideal functionality $\mathcal{F}_{\text{aptSIG}}$, which in turn interacts with the environment $\mathcal{Z}$ via "dummy" honest parties playing the role of either user U and server(s) S. The environment $\mathcal{Z}$ can also instruct $\mathcal{A}^*$ to send malicious inputs to SIM on behalf of corrupt or compromised parties, e.g. compromised servers. There are three types of SIM-$\mathcal{A}^*$ interactions, corresponding to three difference interfaces $\mathcal{A}^*$ has in the real world. First, there is the *network* interface, i.e. messages which protocol Π_{aptSIG} sends over plain channels. This interface is used solely for sending $\mathsf{aec}_{\mathsf{U}}$ in the signing protocol. Second, $\mathcal{A}^*$ can communicate with functionalities $\mathcal{F}_{\text{AUTH}}$ and $\mathcal{F}_{\text{aPPSS}}$, which SIM will emulate in the ideal-world. Third, since protocol Π_{aptSIG} runs the real-world protocol Π_{tSIG} instead of using $\mathcal{F}_{\text{tSIG}}$ as a black-box (see also the explanation above), $\mathcal{A}^*$ expects to interact with Π_{tSIG} instances. In the ideal-world, SIM will not execute the real-world protocol Π_{tSIG}, and instead it will delegate this to a simulator SIM_{tSIG} (the simulator whose existence is implied by the assumption that protocol Π_{tSIG} UC-realizes functionality $\mathcal{F}_{\text{tSIG}}$), which SIM will execute as a black-box. Simulator SIM_{tSIG} can emulate execution of Π_{tSIG} instances to $\mathcal{A}^*$ if SIM_{tSIG} interacts with the ideal functionality $\mathcal{F}_{\text{tSIG}}$. Therefore, SIM will implement an "$\mathcal{F}_{\text{tSIG}}$" interface (just like the "$\mathcal{F}_{\text{AUTH}}$" and "$\mathcal{F}_{\text{aPPSS}}$" interfaces described above) on which it will talk not to $\mathcal{A}^*$ but to SIM_{tSIG}. Note that from SIM's perspective SIM_{tSIG} can be thought of as an extension of adversary $\mathcal{A}^*$ (indeed SIM treats SIM_{tSIG} as a black box, just like it treats $\mathcal{A}^*$), at which point SIM's goals is just to correctly emulate the "$\mathcal{F}_{\text{tSIG}}$" interface with SIM_{tSIG}.

As discussed above, there is one further unusual aspect of the simulation: In one special case, which corresponds to an honest party U recovering wrong tSIG shares because of a successful online active attack against U's password in the aPPSS subprotocol, the real-world execution in this case involves U running the tSIG signing subprotocol on *adversarial inputs* rather than the inputs prescribed for U in the tSIG key generation. Such honest party's execution is not supported by functionality $\mathcal{F}_{\text{tSIG}}$, so the simulator cannot send any messages on the "$\mathcal{F}_{\text{tSIG}}$" interface to SIM_{tSIG} to emulate such tSIG signing protocol instances on behalf of U. Instead, SIM will simply execute itself the tSIG instance on behalf of U on such adversarial inputs. (Note that SIM learns from the "$\mathcal{F}_{\text{aPPSS}}$" interface the adversarial inputs which the real-world U would use, because the adversary sends them to the real-world U via functionality $\mathcal{F}_{\text{aPPSS}}$) This U instance can be

thought of as another extension of the adversary, and SIM will inform SIM_{tSIG} (and pass to $\mathcal{A}^*$) whatever this instance sends e.g. to honest tSIG servers, which are emulated by SIM_{tSIG}.

Theorem 3 *If* $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ *is an Equivocable Authenticated Encryption, and* $\mathsf{tSIG} = (\Pi_{\mathsf{TKeyGen}}, \Pi_{\mathsf{TSign}}, \Pi_{\mathsf{TVerify}})$ *is a Threshold Signature scheme which UC-realizes functionality* $\mathcal{F}_{\text{tSIG}}$*, then protocol* Π_{aptSIG} *in Fig. 7 UC-realizes functionality* $\mathcal{F}_{\text{aptSIG}}$ *in Fig. 5 in the* ($\mathcal{F}_{\text{aPPSS}}$,$\mathcal{F}_{\text{AUTH}}$)*-hybrid model.*

The detailed specification of the simulator SIM, as well as the rest of the proof of Theorem 3, are presented in Appendix F.

5 Concrete Instantiation of the aptSIG Protocol

In Figure 8 we show a concrete instantiation of the generic Π_{aptSIG} protocol from Figure 7, called *aptSIG-BLS*. This instantiation uses tSIG implemented using threshold BLS as shown in Figure 2 in Section 2.1, and the aPPSS shown in Figure 4 in Section 3. Finally, the latter is instantiated with a specific OPRF protocol, 2HashDH [29], included in Appendix C.1, Figure 11. This concrete aptSIG protocol relies on authenticated channels between user U and each server S_i in initialization, an assumption we take throughout the paper. In addition, the initialization relies on a secure channel for U-to-S_i communication, but secure channels can be implemented on top of authenticated channels using key exchange. Moreover, a typical application would use TLS to implement authenticated channels, which provides secure channels without any additional cost.

Notation and parameters. Figure 8 assumes the following notation for public parameters: Security parameter l, threshold parameters t, n, $t \leq n$, field $\mathbb{F} = GF(2^l)$, cyclic group G of prime order p, bilinear map group $\hat{G}$ of prime order $\hat{p}$ and generator $\hat{g}$; hash functions H_1, H_2, H_3, H_4 with ranges G, $\{0,1\}^l$, $\{0,1\}^{2l}$, $\hat{G}$. Let $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ be an Equivocable Authenticated Encryption. $\mathsf{auth}_{A \to B}\{m\}$ and $\mathsf{sec}_{A \to B}\{m\}$ stand for A sending message m to B via resp. authenticated and secure $A \to B$ channel.

Performance. Our concrete aptSIG protocol is very practical: The initialization protocol takes 3 flows (after receiving OPRF replies the user can send all the remaining messages in one flow), and the signing protocol takes only 2 flows. Each server performs 2 exponentiations in both initialization and signing (one in a standard group, one in a group with a bilinear map), while the user performs O(n) exponentiations and one bilinear map. The protocol involves no server-to-server communication, and the bandwidth between user and each server is $O(n)$, but the only $O(n)$-sized message is a ciphertext vector $\mathbf{e}$, which can be stored more efficiently using error correction instead of replicating it on all servers, which reduces bandwidth to $O(1)$ for $t = O(n)$. In Appendix A we show a simplified rendering of this protocol which highlights its simplicity and efficiency.

Adding robustness to aptSIG-BLS. In the protocol in Figure 8, user U chooses $t + 1$ servers to interact with, and it aborts if any server misbehaves.

Parameters: The notation and parameters are defined in text, on page 28.

Initialization for user U on input $(sid, \mathsf{S}_1, ..., \mathsf{S}_n, \mathsf{pw})$:

1. Pick $\alpha \leftarrow_\$ \mathbb{Z}_p$, set $a = (\mathsf{H}_1(\mathsf{pw}))^\alpha$, and send $((sid||i||0), a)$ to S_i for $i \in [n]$.
2. Receive $\mathsf{auth}_{\mathsf{S}\to\mathsf{U}}\{a_i, b_i(= a^{k_i})\}$ for each S_i, abort if $(a_i \neq a)$ for any i.
3. Pick $s \leftarrow_\$ \mathbb{F}$, generate shares $(s_1, ..., s_n)$ as a (t, n)-secret-sharing of s over $\mathbb{F}$. Set $\rho_i = \mathsf{H}_2(\mathsf{pw}, b_i^{1/\alpha})$ and $e_i = s_i \oplus \rho_i$ for all $i \in [n]$.
4. Set $\mathbf{e} := (e_1, ..., e_n)$, $(C||\mathsf{sk}) := \mathsf{H}_3(\mathsf{pw}, \mathbf{e}, s)$, and $\omega := (\mathbf{e}, C)$.
5. Send $\mathsf{auth}_{\mathsf{U}\to\mathsf{S}_i}\{(sid||i||1), \omega\}$ for all $i \in [n]$.
6. Pick $v', v_0 \leftarrow_\$ \mathbb{Z}_{\hat{p}}$, set $v = v_0 + v' \bmod \hat{p}$, generate shares $(v_1, ..., v_n)$ as (t, n)-sharing of v' over $\mathbb{Z}_{\hat{p}}$. Send $\mathsf{sec}_{\mathsf{U}\to\mathsf{S}_i}\{(sid||i), v_i\}$ for all $i \in [n]$.
7. Set $\mathsf{V} = \hat{g}^v$ and $\vec{\mathsf{V}} = (\mathsf{V}_1, ..., \mathsf{V}_n)$ where $\mathsf{V}_i = \hat{g}^{v_i}$ for every $i \in [n]$. Set $\mathsf{aec}_\mathsf{U} := \mathsf{AE.AuthEnc}_{\mathsf{sk}}(\mathsf{U}, \mathsf{V}, \vec{\mathsf{V}}, v_0)$, send $\mathsf{auth}_{\mathsf{U}\to\mathsf{S}_i}\{(sid, \mathsf{aec}_\mathsf{U})\}$ for all $i \in [n]$. Output $(\mathsf{ptsig.verificationkey}, sid, \mathsf{V})$.

Initialization for server S on input (sid, i, U):

1. Set $k \leftarrow_\$ \mathbb{Z}_p$, on $((sid||i||0), a)$ from U, abort if $a \notin G$, else send $\mathsf{auth}_{\mathsf{S}\to\mathsf{U}}\{a, a^k\}$.
2. On message $\mathsf{auth}_{\mathsf{U}\to\mathsf{S}}\{((sid||i||1), \omega\}$ from U, store (sid, i, ω, k).
3. Receive $\mathsf{sec}_{\mathsf{U}\to\mathsf{S}}\{(sid||i), v_i\}$, abort if $v_i \notin \mathbb{Z}_{\hat{p}}$. Save (sid, v_i).
4. On $\mathsf{auth}_{\mathsf{U}\to\mathsf{S}}\{sid, \mathsf{aec}_\mathsf{U}\}$ save $(sid, \mathsf{aec}_\mathsf{U})$.

Signing for user U on input $(sid, ssid, \mathbf{S} = \{\mathsf{S}_1, ..., \mathsf{S}_{t+1}\}, \mathsf{pw}, \mathsf{m})$:

1. Pick $\alpha \leftarrow_\$ \mathbb{Z}_p$, set $a = (H_1(\mathsf{pw}))^\alpha$, send $((sid, ssid, j), a)$ to $\mathsf{S}_j \in \mathbf{S}$.
2. Given $((sid, ssid, j), (b_j, i_j, \omega_j))$ and $(sid, \mathsf{aec}_{\mathsf{U}j})$ from each S_j, set $\phi_j = \mathsf{H}_2(\mathsf{pw}, b_j^{1/\alpha})$ for $j \in [t+1]$. Abort if $i_{j_1} = i_{j_2}$ or $\omega_{j_1} \neq \omega_{j_2}$ for any $j_1 \neq j_2$. Otherwise set $\rho_{i_j} := \phi_j$ for all $j \in [t+1]$ and $I := \{i_j | j \in [t+1]\}$.
3. Parse any ω_j as $(\mathbf{e}, C)$ and $\mathbf{e}$ as $(e_1, ..., e_n)$. Set $s_i := e_i \oplus \rho_i$ for each $i \in I$.
4. Recover s and the shares s_i for $i \notin I$ by interpolating points (i, s_i) for $i \in I$.
5. Parse $\mathsf{H}_3(\mathsf{pw}, \mathbf{e}, s)$ as $(C'||\mathsf{sk})$. Abort if $C' \neq C$.
6. Abort if $\mathsf{aec}_{\mathsf{U}j_1} \neq \mathsf{aec}_{\mathsf{U}j_2}$ for any $j_1, j_2 \in [t+1]$, else set aec_U to any $\mathsf{aec}_{\mathsf{U}j}$. Abort if $\mathsf{AE.AuthDec}_{\mathsf{sk}}(\mathsf{aec}_\mathsf{U}) = \perp$, else parse $\mathsf{AE.AuthDec}_{\mathsf{sk}}(\mathsf{aec}_\mathsf{U})$ as $(\mathsf{U}, \mathsf{V}, \vec{\mathsf{V}}, v_0)$.
7. On messages (j, σ_j) from each $\mathsf{S}_j \in \mathbf{S}$ if $e(g, \sigma_j) \neq e(\mathsf{V}_j, \mathsf{H}_4(\mathsf{m}))$ for any $j \in [t+1]$, output $(\mathsf{ptsig.finsign}, sid, ssid, m, \perp)$. Else compute $\sigma := \sigma_0 \cdot (\prod_{j\in S}(\sigma_i)^{\lambda_i})$, where $\sigma_0 = \mathsf{H}_4(\mathsf{m})^{v_0}$ and λ_i's are Lagrange interpolation coefficients corresponding to the set of indexes in S corresponding to $\mathbf{S}$.
8. Output $(\mathsf{ptsig.finsign}, sid, ssid, \mathsf{m}, \sigma)$.

Signing for server S on input $(sid, ssid, \mathsf{U}, \mathsf{m})$:

1. Given $((sid, ssid, j), a)$ from U, abort if $a \notin G$ or S does not hold records (sid, i, ω, k), (sid, v_i) and $(sid, \mathsf{aec}_\mathsf{U})$ with the matching sid.
2. Otherwise set $b := a^k$ and send $((sid, ssid, j), (b, i, \omega))$, $(sid, \mathsf{aec}_\mathsf{U})$ to U.
3. Send (i, σ) to U, where $\sigma := \mathsf{H}_4(m)^{v_i}$.

Fig. 8: *aptSIG-BLS*: an aptSIG protocol instantiated with T-BLS and aPPSS of Fig. 4 with *2HashDH* OPRF. The aPPSS sub-protocol is marked in gray.

Consequently, there is no guarantee that the protocol outputs a correct signature. To achieve guaranteed output, one needs to enhance the OPRF function with a *verifiable* OPRF [29], namely, where each server has a public OPRF verification key that is provided to the user at initialization and included in the vector ω. In particular, the OPRF construction from Fig. 11 (Appendix C.1) can be made verifiable (see [29]) by setting each server's public key to g^k where k is the server's OPRF key and where verification is implemented via a non-interactive ZK proof of equality of dlog. In this case, U can run the aPPSS protocol with any subset of $t+1$ or more servers that sent the same ω value. If reconstruction succeeds, the user has correct keying material, including the public keys to verify individual BLS signatures by the servers (and discard invalid signatures). If reconstruction fails, a new (disjoint) set of $t+1$ or more servers with same value ω is chosen by U and the process is repeated. It is guaranteed that if U has undisturbed connectivity to $t+1$ honest servers, the correct signature σ on message m will be produced. The above process repeats for at most $\lfloor n/(t+1) \rfloor$ times, hence it is efficient even with dishonest majority.

Adding PFS security to aptSIG-BLS. In the protocol in Figure 8, server S_i in step 3 of the signing phase sends its partial signature σ_i without a proof that U knows the correct password pw and wants to sign m. This enables the adversary to gather partial signatures on a message m without prior knowledge of pw, and then complete these to the full signature if it compromises password pw in the future. However, we can prevent this attack and guarantee Perfect Forward Secrecy (PFS). This extension is sketched at the end of Section 4.1, and is fully described in Appendix G. The PFS-version of the fully instantiated protocol aptSIG-BLS is deferred to Figure 8 in Appendix H.

References

1. Agrawal, S., Miao, P., Mohassel, P., Mukherjee, P.: PASTA: PASsword-based threshold authentication. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 2042–2059. ACM Press (Oct 2018)
2. Arapinis, M., Gkaniatsou, A., Karakostas, D., Kiayias, A.: A formal treatment of hardware wallets. In: Goldberg, I., Moore, T. (eds.) Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11598, pp. 426–445. Springer (2019). https://doi.org/10.1007/978-3-030-32101-7_26, `https://doi.org/10.1007/978-3-030-32101-7_26`
3. Aumasson, J., Hamelink, A., Shlomovits, O.: A survey of ECDSA threshold signing. IACR Cryptol. ePrint Arch. p. 1390 (2020), `https://eprint.iacr.org/2020/1390`
4. Bacho, R., Loss, J.: On the adaptive security of the threshold BLS signature scheme. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 193–207. ACM Press (Nov 2022). https://doi.org/10.1145/3548606.3560656
5. Bagherzandi, A., Jarecki, S., Saxena, N., Lu, Y.: Password-protected secret sharing. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) ACM CCS 2011. pp. 433–444. ACM Press (Oct 2011)

6. Baum, C., Frederiksen, T., Hesse, J., Lehmann, A., Yanai, A.: Pesto: Proactively secure distributed single sign-on, or how to trust a hacked server. In: 2020 IEEE European Symposium on Security and Privacy (EuroSP). pp. 587–606 (2020)
7. Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In: Desmedt, Y. (ed.) Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2567, pp. 31–46. Springer (2003). https://doi.org/10.1007/3-540-36288-6_3, https://doi.org/10.1007/3-540-36288-6_3
8. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. J. Cryptol. **17**(4), 297–319 (2004). https://doi.org/10.1007/s00145-004-0314-9, https://doi.org/10.1007/s00145-004-0314-9
9. Boyd, C.: Digital multisignatures. Cryptography and Coding (1986)
10. Camenisch, J., Lehmann, A., Neven, G., Samelin, K.: Virtual smart cards: How to sign with a password and a server. In: Zikas, V., De Prisco, R. (eds.) SCN 16. LNCS, vol. 9841, pp. 353–371 (Aug / Sep 2016)
11. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA. pp. 136–145. IEEE Computer Society (2001). https://doi.org/10.1109/SFCS.2001.959888, https://doi.org/10.1109/SFCS.2001.959888
12. Canetti, R.: Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239 (2003), https://eprint.iacr.org/2003/239
13. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1769–1787. ACM Press (Nov 2020)
14. Castagnos, G., Catalano, D., Laguillaumie, F., Savasta, F., Tucker, I.: Bandwidth-efficient threshold EC-DSA. In: Kiayias, A., Kohlweiss, M., Wallden, P., Zikas, V. (eds.) PKC 2020, Part II. LNCS, vol. 12111 (May 2020)
15. Das, P., Erwig, A., Faust, S., Loss, J., Riahi, S.: Bip32-compatible threshold wallets. IACR Cryptol. ePrint Arch. p. 312 (2023), https://eprint.iacr.org/2023/312
16. Das, S., Ren, L.: Adaptively secure BLS threshold signatures from DDH and co-CDH. In: Reyzin, L., Stebila, D. (eds.) CRYPTO 2024, Part VII. LNCS, vol. 14926, pp. 251–284. Springer, Cham (Aug 2024)
17. Desmedt, Y.: Society and group oriented cryptography: A new concept. In: Pomerance, C. (ed.) CRYPTO'87. LNCS, vol. 293 (Aug 1988)
18. Desmedt, Y., Frankel, Y.: Threshold cryptosystems. In: Brassard, G. (ed.) CRYPTO'89. LNCS, vol. 435 (Aug 1990)
19. Desmedt, Y., Frankel, Y.: Shared generation of authenticators and signatures (extended abstract). In: Feigenbaum, J. (ed.) CRYPTO'91. LNCS, vol. 576 (Aug 1992)
20. Doerner, J., Kondi, Y., Lee, E., shelat, a.: Threshold ECDSA from ECDSA assumptions: The multiparty case. In: 2019 IEEE Symposium on Security and Privacy. IEEE Computer Society Press (May 2019)
21. Dziembowski, S., Jarecki, S., Kedzior, P., Krawczyk, H., Ngo, C.N., Xu, J.: Password-protected threshold signatures. In: Advances in Cryptology – ASIACRYPT 2024 (2024)

22. Fuchsbauer, G., Kiltz, E., Loss, J.: The algebraic group model and its applications. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 33–62 (Aug 2018). https://doi.org/10.1007/978-3-319-96881-0_2
23. Ganesan, R.: Yaksha: augmenting kerberos with public key cryptography. In: Proceedings of the Symposium on Network and Distributed System Security. pp. 132–143 (1995)
24. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. ACM Press (Oct 2018)
25. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. J. Cryptol. **20**(1), 51–83 (2007). https://doi.org/10.1007/s00145-006-0347-3, https://doi.org/10.1007/s00145-006-0347-3
26. Gentry, C., MacKenzie, P.D., Ramzan, Z.: A method for making password-based key exchange resilient to server compromise. In: Dwork, C. (ed.) Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4117, pp. 142–159. Springer (2006). https://doi.org/10.1007/11818175_9, https://doi.org/10.1007/11818175_9
27. Gjøsteen, K., Thuen, Ø.: Password-based signatures. In: Petkova-Nikova, S., Pashalidis, A., Pernul, G. (eds.) Public Key Infrastructures, Services and Applications. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
28. Gu, Y., Jarecki, S., Kedzior, P., Nazarian, P., Xu, J.: Threshold PAKE with security against compromise of all servers. In: Advances in Cryptology – ASIACRYPT 2024 (2024)
29. Jarecki, S., Kiayias, A., Krawczyk, H.: Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 233–253. Springer (2014). https://doi.org/10.1007/978-3-662-45608-8_13, https://doi.org/10.1007/978-3-662-45608-8_13
30. 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: 2016 IEEE European Symposium on Security and Privacy (EuroSP). pp. 276–291 (2016). https://doi.org/10.1109/EuroSP.2016.30
31. 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: 2016 IEEE European Symposium on Security and Privacy (EuroSP). pp. 276–291 (2016). https://doi.org/10.1109/EuroSP.2016.30
32. Jarecki, S., Kiayias, A., Krawczyk, H., Xu, J.: TOPPSS: cost-minimal password-protected secret sharing based on threshold OPRF. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10355, pp. 39–58. Springer (2017). https://doi.org/10.1007/978-3-319-61204-1_3, https://doi.org/10.1007/978-3-319-61204-1_3
33. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In: Nielsen, J.B., Rijmen, V. (eds.) Ad-

vances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III. Lecture Notes in Computer Science, vol. 10822, pp. 456–486. Springer (2018). https://doi.org/10.1007/978-3-319-78372-7_15, https://doi.org/10.1007/978-3-319-78372-7_15
34. Lindell, Y., Nof, A.: Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. ACM Press (Oct 2018)
35. MacKenzie, P.D., Reiter, M.K.: Networked cryptographic devices resilient to capture. In: 2001 IEEE Symposium on Security and Privacy. pp. 12–25. IEEE Computer Society Press (May 2001)
36. MacKenzie, P.D., Shrimpton, T., Jakobsson, M.: Threshold password-authenticated key exchange. J. Cryptol. **19**(1), 27–66 (2006). https://doi.org/10.1007/s00145-005-0232-5, https://doi.org/10.1007/s00145-005-0232-5
37. McQuoid, I., Rosulek, M., Xu, J.: How to obfuscate MPC inputs. In: Kiltz, E., Vaikuntanathan, V. (eds.) TCC 2022, Part II. LNCS, vol. 13748, pp. 151–180. Springer, Cham (Nov 2022)
38. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO'91. LNCS, vol. 576, pp. 129–140 (Aug 1992)
39. Wikström, D.: Universally composable DKG with linear number of exponentiations. In: Blundo, C., Cimato, S. (eds.) Security in Communication Networks, 4th International Conference, SCN 2004, Amalfi, Italy, September 8-10, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3352, pp. 263–277. Springer (2004). https://doi.org/10.1007/978-3-540-30598-9_19, https://doi.org/10.1007/978-3-540-30598-9_19
40. Xu, S., Sandhu, R.S.: Two efficient and provably secure schemes for server-assisted threshold signatures. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 355–372 (Apr 2003)

A Simplified rendering of $\Pi_{\mathbf{aptSIG}}$ protocol from Section 5

Figure 9 shows a simplified rendering of the signing phase of protocol Π_{aptSIG} shown in Figure 8 in Section 5. To increase readability, Figure 9 omits accounting details like *sid*'s, message indexing, and details of polynomial interpolation. We focus on the signing phase to show that it is very efficient and requires only two flows of communication between U and S_i's. In particular, everything the user needs to reconstruct a signature is sent by the server in one message. The initialization phase has similar computational cost and takes only takes three communication flows, see Section 5.

B Security of threshold BLS: Proof of Theorem 1

Proof. For any adversary $\mathcal{A}$, we construct a simulator SIM. Without loss of generality, we may assume that $\mathcal{A}$ is a "dummy" adversary that merely passes all its messages and computations to the environment $\mathcal{Z}$. The simulator SIM

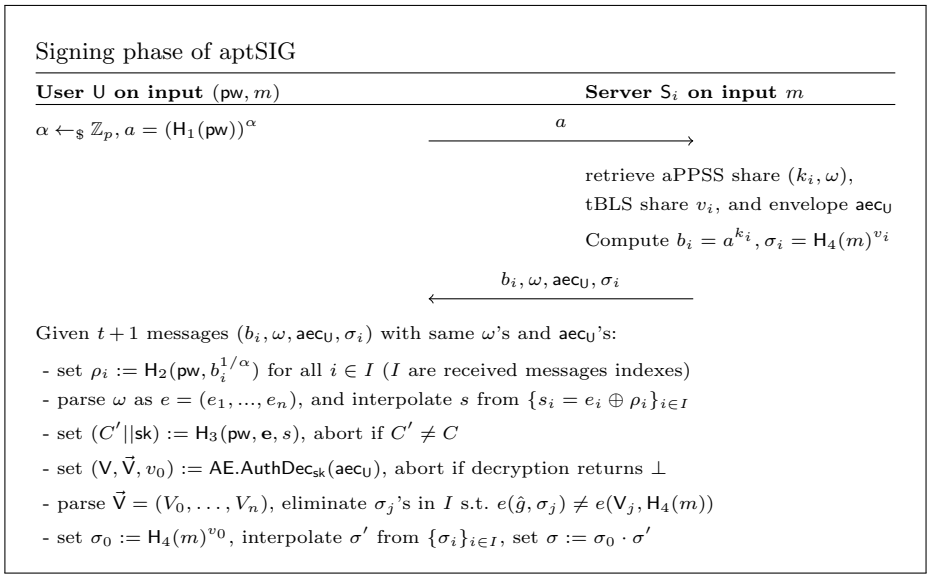


Fig. 9: Rendering of the signing phase of protocol Π_{aptSIG} from Section 5

is straightforward so we only provide a sketch. On $(\mathsf{tsig.keygen}, sid, \mathsf{P}_0)$ from $\mathcal{F}_{\text{tSIG}}$, SIM runs the Key Generation code of P_0 as in the protocol, obtains shares $(s_0, ..., s_n)$ and public values $\mathsf{V}, \vec{\mathsf{V}}$, emulates sending $s_i, \mathsf{V}, \vec{\mathsf{V}}$ to each P_i over a secure channel $\mathsf{sec}_{\mathsf{P}_0 \to \mathsf{P}_i}\{\cdot\}$, and outputs $(\mathsf{tsig.publickey}, sid, \mathsf{V})$ followed by $(\mathsf{tsig.keygencomplete}, sid, \mathsf{P}_0)$ to $\mathcal{F}_{\text{tSIG}}$. Likewise, on $(\mathsf{tsig.keygen}, sid, \mathsf{P}_i)$ from $\mathcal{F}_{\text{tSIG}}$ for any $i > 0$, SIM waits for transmission of the emulated secure channel message which produced for the $\mathsf{sec}_{\mathsf{P}_0 \to \mathsf{P}_i}\{\cdot\}$ channel, and if the adversary delivers this message then SIM sends $(\mathsf{tsig.keygencomplete}, sid, \mathsf{P}_i)$ to $\mathcal{F}_{\text{tSIG}}$.

On $(\mathsf{tsig.sign}, sid, \mathsf{m}, \mathsf{P}_i)$ from $\mathcal{F}_{\text{tSIG}}$, SIM runs P_i's signing code on s_i created above, and when P_i outputs σ because it obtains valid partial signatures (including its own) from all parties $\mathsf{P}_j \in S$ for some set $S \in \mathbb{S}_{sid}$, then SIM sends $(\mathsf{tsig.signature}, sid, \mathsf{m}, S, \sigma)$ followed by $(\mathsf{tsig.signcomplete}, sid, \mathsf{m}, \mathsf{P}_i)$ to $\mathcal{F}_{\text{tSIG}}$. Finally, if the environment allows P_i to be compromised, SIM sends s_i to the real-world adversary and sends $(\mathsf{tsig.compromise}, sid, \mathsf{P}_i)$ to $\mathcal{F}_{\text{tSIG}}$.

It is not hard to see that $\mathcal{Z}$'s real-world view and ideal-world view (simulated by SIM) are identical, except if (1) $\mathcal{Z}$ sends $(\mathsf{tsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$ to some party P such that $e(g, \sigma) \neq e(\mathsf{V}, \mathsf{H}(\mathsf{m}))$, (2) the set of compromised parties **Corr** is not in $\mathbb{S}_{sid}$ (hence the adversary cannot forge messages on its own), and (3) σ was not created (by SIM) in response to $(\mathsf{tsig.sign}, sid, \mathsf{m}, \mathsf{P}_i)$ messages sent from $\mathcal{F}_{\text{tSIG}}$ for some sufficiently large set of honest parties (i.e. any set S s.t. $S \cup \mathbf{Corr} \in \mathbb{S}_{sid}$). In this case the real-world party P will output $(\mathsf{tsig.verified}, sid, \mathsf{m}, \sigma, 1)$, i.e. signature verification passes, but in the ideal world P would output $(\mathsf{tsig.verified}, sid, \mathsf{m}, \sigma, 0)$, because $\mathcal{F}_{\text{tSIG}}$ never received message $(\mathsf{tsig.signature}, sid, \mathsf{m}, \cdot, \sigma)$ from SIM.

Call the above event Forge. We construct a reduction $\mathcal{R}_{BLS}$ that aims to break the CMA-unforgability of the BLS signature scheme if Forge occurs. Note that this event can occur only before $\mathcal{Z}$ compromises some set $S \in \mathbb{S}_{sid}$, i.e. before $\mathcal{Z}$ compromises P_0 and some subset of $t+1$ parties in $(\mathsf{P}_1, ..., \mathsf{P}_n)$. $\mathcal{R}_{BLS}$ is given BLS public key V, and runs the environment $\mathcal{Z}$ against the threshold BLS signature scheme. $\mathcal{R}_{BLS}$ sets $S = S^- \cup \{0\}$ where S^- is a randomly chosen t-element subset of $\{1, ..., n\}$. Set S will be the $\mathcal{R}_{BLS}$'s guess of the parties which $\mathcal{Z}$ compromises before event Forge.

In the Key Generation phase, $\mathcal{R}_{BLS}$ on input V picks shares $\{s_i\}_{i \in S}$ at random, sets $\mathsf{V}_i = g^{s_i}$ for each $i \in S$, and uses "interpolation in exponent" on values V/V_0 and $\{\mathsf{V}_i\}_{i \in S^-}$, to compute all remaining values V_j for $j \notin S$. On $\mathcal{Z}$'s message $(\mathsf{tsig.sign}, sid, \mathsf{m})$ to any honest P_i, $\mathcal{R}_{BLS}$ simulates P_i's signature protocol by computing $\sigma_i := \mathsf{H}(\mathsf{m})^{s_i}$ for all $i \in S$ (here $\mathcal{R}_{BLS}$ uses the shares $\{s_i\}_{i \in S}$ which $\mathcal{R}_{BLS}$ chose above), then querying the signing oracle to obtain $\sigma = \mathsf{H}(\mathsf{m})^s$, and finally using interpolation in exponent to interpolate the signatures for uncorrupted parties σ_i for $i \notin S$ (similarly to how V_j's for $j \notin S$ were computed above). If the adversary delivers sufficient number of valid partial signatures to P_i then $\mathcal{R}_{BLS}$ sends $(\mathsf{tsig.signature}, sid, \mathsf{m}, \mathbf{P}, \sigma)$ to P_i.

If $\mathcal{Z}$ compromises any party not in set S before event Forge occurs, then $\mathcal{R}_{BLS}$ aborts. When $\mathcal{Z}$ compromises any P_i s.t. $i \in S$ then SIM sends s_i to $\mathcal{Z}$. Finally, on $(\mathsf{tsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$ from party P, $\mathcal{R}_{BLS}$ checks if Forge happens, and if so, it outputs (m, σ) as its forgery.

We now analyze the probability that $\mathcal{R}_{BLS}$ breaks the CMA-unforgability of the BLS signature scheme. The probability that $\mathcal{R}_{BLS}$'s guess is correct is $1/\binom{n}{t}$. However, if $\mathcal{R}_{BLS}$'s guess of set S is correct then $\mathcal{Z}$'s view of the game simulated by $\mathcal{R}_{BLS}$ is identical to its view of the real-world protocol. We conclude that

$$\mathsf{Adv}^{\mathrm{bls}}_{\mathcal{R}_{BLS}}(\lambda) \geq \frac{1}{\binom{n}{t}} \cdot \Pr[\mathsf{Forge}];$$

and assuming that the BLS signature scheme is CMA-unforgeable, and $\binom{n}{t}$ is a polynomial function of λ, it follows that $\Pr[\mathsf{Forge}]$ must be a negligible function of λ, which completes the proof.

C Protocol Tools and Building Blocks

We provide definitions of some cryptographic tools used in the paper, including adaptively secure OPRF, authenticated encryption, signatures, and an authenticated channel functionality.

C.1 Adaptive Oblivious Pseudorandom Function (OPRF)

For reference we include in Figure 10 the UC OPRF functionality modeled on [33]. However, we make two cosmetic changes: (1) we adapt the functionality to the *multi-session* model, whereas the original presentation of this functionality in

[33] was done in a single-session model, and (2) each party can output a *transcript* of the protocol interaction to the environment. The functionality additionally enforces that transcript equality implies that two parties are passively connected, so if U and S outputs the same transcripts then U evaluates the OPRF on the key of S.

Efficient adaptive OPRF protocol. In Figure 11 we include the 2HashDH protocol, which was shown to realize the OPRF functionality in [33], adjusted to the syntax of the $\mathcal{F}_{\text{OPRF}}$ functionality in Figure 10, and so that the protocol transcript is included in both parties' outputs.

C.2 Ciphertext Integrity of Authenticated Encryption

We say that an authentication encryption scheme AE has *ciphertext integrity*, if for any efficient algorithm $\mathcal{A}$, the adversarial advantage $\mathsf{Adv}^{\mathsf{ci,ae}}_{\mathcal{A}}(\lambda)$ is a negligible function of λ, where

$$\mathsf{Adv}^{\mathsf{ci,ae}}_{\mathcal{A}}(\lambda) = \Pr[\ \mathsf{AuthDec}_{\mathsf{sk}}(\mathsf{aec_U}^*) \neq \perp \wedge \mathsf{aec_U}^* \neq \mathsf{aec_U} \mid \mathsf{sk} \leftarrow_{\$} \{0,1\}^{\lambda}, \\ \mathsf{m} \leftarrow \mathcal{A}(1,\lambda), \mathsf{aec_U} \leftarrow \mathsf{AuthEnc}_{\mathsf{sk}}(\mathsf{m}), \mathsf{aec_U}^* \leftarrow \mathcal{A}(2,\mathsf{aec_U})\ ]$$

C.3 Signatures

A Signature Scheme is a tuple of algorithms $\mathsf{Sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, s.t.

Key Generation KeyGen takes as input the security parameter λ and outputs a key pair $(\mathsf{sk}, \mathsf{pk})$;

$$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^{\lambda})$$

Signature Generation Sign takes as input the signing key sk and a message m and outputs σ;

$$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$$

Signature Verification Verify takes as input the verification key pk and the signature σ on message m, and outputs 0 or 1 upon rejection or acceptance of σ on m.

$$\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{pk}, \sigma, m)$$

The standard security requirement on a signature is EUF-CMA (existiential unforgeability under adaptive chosen message attack). Formally, scheme Sig is EUF-CMA if the adversarial advantage $\mathsf{Adv}^{\mathrm{euf-cma,sig}}_{\mathcal{A}}(\lambda)$ is a negligible function of λ, where

$$\mathsf{Adv}^{\mathrm{euf-cma,sig}}_{\mathcal{A}}(\lambda) = \Pr[\ \mathsf{Verify}_{\mathsf{pk}}(\sigma^*, m^*) = 1 \mid \\ (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^{\lambda}), (\sigma^*, m^*) \leftarrow \mathcal{A}(\mathsf{Sign}(\mathsf{sk}, \cdot), \mathsf{pk}))]$$

where $\mathcal{A}$ has access to the signing oracle $\mathsf{Sign}(\mathsf{sk}, \cdot)$ but m^* has never been queried to the signing oracle $\mathsf{Sign}(\mathsf{sk}, \cdot)$.

Public Parameters: PRF output-length l, polynomial in security parameter λ. This functionality is tagged by a **global** session identifier sid, but since all calls should include that globally fixed identifier we omit it from the syntax.
$\mathcal{F}_{\text{OPRF}}$ interacts with arbitrary parties and an adversary $\mathcal{A}^*$. Let $\mathcal{P}$ denote the set of all parties (potentially dynamically changing), and let $\mathcal{P}^* = \mathcal{P} \cup \{\mathcal{A}^*\}$.
Convention: For every function identifier id and argument x, value $F_{id}(x)$ is initially undefined. Whenever an undefined value $F_{id}(x)$ is referenced then $\mathcal{F}_{\text{OPRF}}$ assigns $F_{id}(x) \leftarrow_{\$} \{0,1\}^l$.

Initialization

- On $(\mathsf{oprf.init}, id)$ from $\mathsf{S} \in \mathcal{P}^*$, reject this query if $id \neq [\mathsf{S}||id']$ for some id' or if $\exists$ prior record $(\mathsf{oprf.init}, id, *)$. Otherwise, record $(\mathsf{oprf.init}, id, \mathsf{S})$, set $\mathsf{tx}[id] = 0$, send $(\mathsf{oprf.init}, id, \mathsf{S})$ to $\mathcal{A}^*$. If S is corrupted or $\mathsf{S} = \mathcal{A}^*$ then declare id $\mathsf{compromised}$.

Server Compromise (*Query* $\mathsf{oprf.compromise}$ *requires permission from the environment.*)

- On $(\mathsf{oprf.compromise}, id, \mathsf{S})$ from $\mathcal{A}^*$, if $\exists$ record $(\mathsf{oprf.init}, id, \mathsf{S})$, declare id $\mathsf{compromised}$.

Offline Evaluation

- On $(\mathsf{oprf.offeval}, id, x)$ from $\mathsf{P} \in \mathcal{P}^*$, if $\exists$ rec. $(\mathsf{oprf.init}, id, \mathsf{S})$ s.t. either (i) $\mathsf{P} = \mathsf{S}$ or (ii) $\mathsf{P} = \mathcal{A}^*$ and id is $\mathsf{compromised}$, send $(\mathsf{oprf.offeval}, id, x, F_{id}(x))$ to P.

Online Evaluation

- On $(\mathsf{oprf.eval}, ssid, x, \mathsf{S})$ from $\mathsf{U} \in \mathcal{P}^*$, reject this query if $\exists$ prior record $(\mathsf{oprf.eval}, ssid, \mathsf{U}, *)$. Otherwise send $(\mathsf{oprf.eval}, ssid, \mathsf{U}, \mathsf{S})$ to $\mathcal{A}^*$, record $(\mathsf{oprf.eval}, ssid, \mathsf{U}, x)$ marked fresh.
- On $(\mathsf{oprf.sndrcomplete}, id, ssid')$ from $\mathsf{P} \in \mathcal{P}^*$, if $\exists$ record $(\mathsf{oprf.init}, id, \mathsf{S})$ and either (i) $\mathsf{P} = \mathsf{S}$ or (ii) $\mathsf{P} = \mathcal{A}^*$ and id is $\mathsf{compromised}$, send $(\mathsf{oprf.sndrcomplete}, id, ssid')$ to $\mathcal{A}^*$ and on reply tr_{S}, record $(\mathsf{oprf.sndrtrans}, id, \mathsf{tr}_{\mathsf{S}})$ send $(\mathsf{oprf.sndrtrans}, id, ssid', \mathsf{tr}_{\mathsf{S}})$ to S, set $\mathsf{tx}[id]{+}{+}$.
- On $(\mathsf{oprf.rcvcomplete}, id^*, ssid, \mathsf{U}, \mathsf{tr}_{\mathsf{U}})$ from $\mathcal{A}^*$, if $\mathsf{tx}[id^*] > 0$ and $\exists$ record $(\mathsf{oprf.eval}, ssid, \mathsf{U}, x)$ marked fresh, then:
  1. If $\exists$ record $(\mathsf{oprf.sndrtrans}, id, \mathsf{tr}_{\mathsf{U}})$ then abort if $id^* \neq id$.
  2. Send $(\mathsf{oprf.eval}, ssid, F_{id^*}(x), \mathsf{tr}_{\mathsf{U}})$ to U, set $\mathsf{tx}[id^*]{-}{-}$, mark $(\mathsf{oprf.eval}, ssid, \mathsf{U}, x)$ $\mathsf{completed}$.

Fig. 10: Global OPRF functionality $\mathcal{F}_{\text{OPRF}}$ with Adaptive Compromise

Components: Hash functions $H(\cdot,\cdot)$, $H'(\cdot)$ with ranges $\{0,1\}^\ell$ and $\mathcal{G}$, respectively. Functions H, H' are specific to the $OPRF$ instance initialized for a unique session ID sid, and they should be implemented by folding sid into their inputs.

Initialization:

1. On input $(\mathsf{oprf.init}, id)$, S picks $k \leftarrow\!\!\$\ \mathbb{Z}_q$ and stores (id, k).

Server Compromise:

1. On $(\mathsf{oprf.compromise}, id, \mathsf{S})$ from the adversary, reveal key k.

Offline Evaluation:

1. On input $(\mathsf{oprf.offeval}, id, \mathsf{S}, x)$ for id matching record (id, k), S outputs $(\mathsf{oprf.offeval}, id, y)$ for $y = H(x, (H'(x))^k)$.

Online Evaluation:

- On input $(\mathsf{oprf.eval}, ssid, \mathsf{S}', x)$, U picks $r \leftarrow\!\!\$\ \mathbb{Z}_q$, records $(ssid, r)$, and sends $(ssid, a)$ for $a = H'(x)^r$ to S'
- On input $(\mathsf{oprf.sndrcomplete}, id, ssid')$ and message $(ssid, a)$ from U s.t. $a \in \mathcal{G}$, S retrieves pair (id, k) with matching id, aborts if such pair is not found, else sends $(ssid, b)$ for $b = a^k$ to U and outputs $(\mathsf{oprf.sndrtrans}, ssid', (a, b))$.
- On message $(ssid, b)$ s.t. $b \in \mathcal{G}$, U retrieves tuple $(ssid, r)$, aborts if tuple not found, else outputs $(\mathsf{oprf.eval}, ssid, y, (a, b))$ for $y = H(x, b^{1/r})$.

Fig. 11: Adaptive OPRF Protocol 2HashDH

C.4 Authenticated Channel Model

In Figure 12 we recall the ideal functionality $\mathcal{F}_{\text{AUTH}}$ for message authentication from [11]. This functionality can be realized using signatures if party names like P_S can be securely mapped to a public key, e.g. using PKI.

Ideal Functionality $\mathcal{F}_{\text{AUTH}}$

- On input $(\mathsf{send}, sid, \mathsf{P}_R, \mathsf{m})$ from P_S, record tuple $(\mathsf{send}, sid, \mathsf{P}_S, \mathsf{P}_R, \mathsf{m})$ and send it to adversary $\mathcal{A}^*$.
- On message $(\mathsf{sent}, sid, \mathsf{P}_S, \mathsf{P}_R, \mathsf{m})$ from $\mathcal{A}^*$, if this tuple is stored then send it to P_R and delete it from the storage.

Fig. 12: Ideal Functionality $\mathcal{F}_{\text{AUTH}}$

D Proof for aPPSS security

In this section we provide the missing proof of Theorem 2.

Proof. For any adversary $\mathcal{A}^*$, we construct a simulator SIM as shown in Fig. 13 and 14. Without loss of generality, we may assume that $\mathcal{A}^*$ is a "dummy" adversary that merely passes all its messages and computations to the environment $\mathcal{Z}$. We omit all interactions with corrupted U and S where SIM acts as $\mathcal{F}_{\text{OPRF}}$, since the simulation is trivial (SIM gains all information needed and simply follows the code of $\mathcal{F}_{\text{OPRF}}$). Following [33], we assume that for any server S, the adversary $\mathcal{A}$ always sends $(\mathsf{oprf.compromise}, sid, \mathsf{S})$ and $(\mathsf{ppss.compromise}, sid, \mathsf{S})$ messages together, since both messages correspond to the real-world action of compromising S. We now show that the distinguishing advantage of $\mathcal{Z}$ between the real world and the simulated world is negligible. The argument uses a sequence of games, starting from the real world and ending at the simulated world; for any two adjacent games G_i and G_{i+1}, let $Dist_{\mathcal{Z}}^{G_i,G_{i+1}}$ denote the distinguishing advantage of $\mathcal{Z}$ between them, i.e., $Dist_{\mathcal{Z}}^{G_i,G_{i+1}} = |\Pr[\mathcal{Z} \text{ outputs 1 in } G_i] - \Pr[\mathcal{Z} \text{ outputs 1 in } G_{i+1}]|$. ($Dist_{\mathcal{Z}}^{G_i,G_{i+1}}$ is a function of the security parameter λ, but we omit λ below.)

In our analysis we will use following convention: if value is picked or calculated during Initialization, then it will be denoted in pure form e.g. s, pw. In the Reconstruction phase values have added apostrophe to denote the fact, that they were sent by $\mathcal{A}^*$ or calculated using other values sent by $\mathcal{A}^*$. Let H_L denotes first λ bits of output of H. Respectively H_R denotes last λ bits of H. Define *Succesful Password Test (SPT)* as an event, in which $\mathcal{A}^*$ receives $t+1$ values $F_{id}(\mathsf{pw})$, where id are function identifier used by different servers in the User Initialization and pw is password by the User in that phase.

Game G_1: Collision in H. At the start of G_1 set $\mathsf{H}(x) = \perp$ for all x. When U or $\mathcal{A}^*$ queries H on some input x, if $\mathsf{H}(x) \neq \perp$, then return $\mathsf{H}(x)$. Otherwise, if $\mathsf{H}(x) = \perp$, pick $y_1 \leftarrow\!\!\$\ \{0,1\}^\lambda$ and $y_2 \leftarrow\!\!\$\ \{0,1\}^\lambda$. If there exists x' such that $\mathsf{H}_L(x') = y_1$, then abort. Otherwise set $\mathsf{H}(x) = [y_1||y_2]$ and output $[y_1||y_2]$. Probability of finding a collision on first λ bit in G_0 when querying H on two inputs is equal to $\frac{1}{2^\lambda}$. In G_1 there is no collisions in H. In case of no collisions on H both games outputs random value in $\{0,1\}^{2\lambda}$. $\mathcal{A}^*$ can query H up to q_H times, which results in probability of collision being equal bounded by $\frac{q_H^2}{2^\lambda}$.

$$Dist_{\mathcal{Z}}^{G_0,G_1} \leq \frac{q_H^2}{2^\lambda}$$

Game G_2: In step 2 of User Reconstruction, U gets $(\mathsf{oprf.eval}, [sid||j||ssid], \phi_j, \mathsf{tr}_j)$ from $\mathcal{F}_{\text{OPRF}}$ and messages (i_j, ω_j) from $\mathbf{S}[j]$, for all $j \in [t{+}1]$. Values σ_j are calculated as a value of a random function $F_{id^*}(\mathsf{pw}')$, where id^* was chosen by $\mathcal{A}^*$.

Set $\mathsf{H}(x) := \perp$, $x_H := \emptyset$, $F_{id}(\mathsf{pw}) := \perp, \mathsf{tested}(\mathsf{pw}) := \emptyset, \mathsf{pointer}(i) := \perp, \mathsf{pointer}(ssid, j) := \perp$, for all values $x, id, \mathsf{pw}, i, ssid, j$. Assume that for any x if $\mathsf{H}(x)$ is referenced for $\mathsf{H}(x) = \perp$ then SIM assigns $\mathsf{H}(x) \leftarrow_\$ \{0,1\}^{2\lambda}$ and $x_H := x \cup x_H$. Analogously for any id, x if $F_{id}(x)$ is referenced for $F_{id}(x) = \perp$ then SIM assigns $F_{id}(x) \leftarrow_\$ \{0,1\}^{\lambda}$.

1. On $(\mathsf{ppss.uinit}, sid, \mathsf{U})$ from $\mathcal{F}_{\text{aPPSS}}$ for honest U, parse $sid = (sid' || \mathbf{P}_{sid})$, send $(\mathsf{oprf.eval}, [sid||i||0], \mathsf{U}, \mathbf{P}_{sid}[i])$ to $\mathcal{A}^*$ for $i \in [n]$.
2. On $(\mathsf{ppss.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ from $\mathcal{F}_{\text{aPPSS}}$ for honest S, send $(\mathsf{oprf.init}, [\mathsf{S}||sid], \mathsf{S})$ and $(\mathsf{oprf.sndrcomplete}, [\mathsf{S}||sid], 0)$ to $\mathcal{A}^*$, and on reply $\mathsf{tr_S}$, save $(\mathsf{oprf.sndrtrans}, [\mathsf{S}||sid], \mathsf{tr_S})$, set $\mathsf{tx}[\mathsf{S}||sid]{+}{+}$, and send $(\mathsf{send}, [sid||i||0], \mathsf{S}, \mathsf{U}, \mathsf{tr_S})$ to $\mathcal{A}^*$.
3. On $(\mathsf{oprf.rcvcomplete}, id^*, [sid||i||0], \mathsf{U}, \mathsf{tr_U})$ and $(\mathsf{sent}, [sid||i||0], \mathsf{S}, \mathsf{U}, \mathsf{tr}^*_{\mathsf{S}})$ from $\mathcal{A}^*$ for U which was initialized via $\mathsf{ppss.uinit}$ in step 1 (otherwise ignore this message), proceed only if: (a) $\mathsf{S} = \mathbf{P}[i]$, (b) if S is honest then there exists record $(\mathsf{oprf.sndrtrans}, [\mathsf{S}||sid], \mathsf{tr}^*_{\mathsf{S}})$, (c) if $\exists$ record $(\mathsf{oprf.sndrtrans}, id, \mathsf{tr_U})$ then $id^* = id$, (d) $\mathsf{tx}[id^*] > 0$, (e) $\mathsf{tr_U} = \mathsf{tr}^*_{\mathsf{S}}$.
   If all conditions are met then do:
   - set $\mathsf{tx}[id^*]{-}{-}$ and $\mathsf{pointer}(i) := id^*$,
   - if S is corrupt send $(\mathsf{ppss.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ to $\mathcal{F}_{\text{aPPSS}}$,
   - if $\mathsf{pointer}(i) \neq \perp$ for all $i \in [n]$, pick $\mathbf{e} \leftarrow\!\!\$\ \mathbb{F}^n$ and $C \leftarrow\!\!\$\ \{0,1\}^{\lambda}$, set $\omega := (\mathbf{e}, C)$, and send $(\mathsf{ppss.fininit}, sid)$ to $\mathcal{F}_{\text{aPPSS}}$ and $\{(\mathsf{send}, [sid||i||1], \mathsf{U}, \mathbf{P}[i], \omega)\}_{i \in [n]}$ to $\mathcal{A}^*$.
4. On $(\mathsf{sent}, [sid||i||1], \mathsf{U}, \mathsf{S}, \omega_i)$ from $\mathcal{A}^*$ for S which was initialized via $\mathsf{ppss.sinit}$ in step 2 with matching inputs (sid, i, U) (otherwise ignore this message):
   If U is honest then abort unless $\omega_i = \omega$. Otherwise, save (sid, i, ω).
5. *Emulation of adversarial OPRF calls, server compromise, and off-line password tests:*
   (a) On $(\mathsf{oprf.init}, id)$ for $id = [\mathsf{P}||id']$ and new id' from either corrupt party P or from $\mathsf{P} = \mathcal{A}^*$, set $\mathsf{tx}[id] := 0$, mark id $\mathsf{compromised}$.
   (b) If the environment lets $\mathcal{A}^*$ send $(\mathsf{oprf.compromise}, id, \mathsf{S})$ for $id = [\mathsf{S} \,||\, sid]$ for honest S (for which SIM executed $\mathsf{ppss.sinit}$), declare id $\mathsf{compromised}$, and if $\exists\, i$ s.t. $id = \mathsf{pointer}(i)$ then send $(\mathsf{ppss.compromise}, sid, \mathsf{S})$ to $\mathcal{F}_{\text{aPPSS}}$.
   (c) On $(\mathsf{oprf.sndrcomplete}, id, ssid)$ and $\mathsf{tr_S}$ from $\mathcal{A}^*$ for $\mathsf{compromised}$ id and any $ssid$, save $(\mathsf{oprf.sndrtrans}, id, \mathsf{tr_S})$ and set $\mathsf{tx}[id]{+}{+}$ and if $\exists\, i$ s.t. $id = \mathsf{pointer}(i)$ then send $(\mathsf{ppss.srec}, sid, ssid, \mathbf{P}[i], \perp)$ to $\mathcal{F}_{\text{aPPSS}}$.
   (d) On $(\mathsf{oprf.eval}, ssid, x, \mathsf{S})$ from P followed by $(\mathsf{oprf.rcvcomplete}, id, ssid, \mathsf{P}, \mathsf{tr_U})$ by $\mathcal{A}^*$ where P is either corrupt or $\mathsf{P} = \mathcal{A}^*$ (w.l.o.g. assume $\mathcal{A}^*$ sends $\mathsf{tr_U}$ that allows evaluation of F_{id}):
   If $\mathsf{tx}[id] > 0$ then set $\mathsf{tx}[id]{-}{-}$, execute (5f), and send $(\mathsf{oprf.eval}, ssid, F_{id}(x), \mathsf{tr_U})$ to P.
   (e) On $(\mathsf{oprf.offeval}, id, x)$ from $\mathcal{A}^*$ for $\mathsf{compromised}$ id, do: If $\exists\, i$ s.t. $id = \mathsf{pointer}(i)$ then send $(\mathsf{ppss.srec}, sid, \perp, \mathbf{P}[i], \perp)$ to $\mathcal{F}_{\text{aPPSS}}$; Execute (5f) and send $(\mathsf{oprf.offeval}, id, x, F_{id}(x))$ to $\mathcal{A}^*$.
   (f) Before SIM in line (5d) or (5e) sends $F_{id}(x)$ to $\mathsf{P}/\mathcal{A}^*$ for $id = \mathsf{pointer}(i)$, add i to $\mathsf{tested}(x)$, send $(\mathsf{ppss.testpw}, sid, \mathbf{P}[i], x)$ to $\mathcal{F}_{\text{aPPSS}}$, and if $\mathcal{F}_{\text{aPPSS}}$ replies $\mathsf{sk} \neq \perp$ then set $s_i = e_i \oplus F_{\mathsf{pointer}(i)}(x)$ for all $i \in \mathsf{tested}(x)$, interpolate $(s, \{s_i\}_{i \notin \mathsf{tested}(x)})$ from $\{(i, s_i)\}_{i \in \mathsf{tested}(x)}$, set $\mathsf{H}(x, \mathbf{e}, s) := [C||\mathsf{sk}]$ (abort simulation if $\mathsf{H}(x, \mathbf{e}, s)$ already defined), and set $F_{\mathsf{pointer}(i)}(x) := s_i \oplus e_i$ for all $i \in [n] \setminus \mathsf{tested}(x)$.

Fig. 13: Simulator SIM for protocol Π_{aPPSS} (init., OPRF, off-line attacks)

6. On $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \mathbf{S})$ from $\mathcal{F}_{\text{aPPSS}}$, send $\{(\mathsf{oprf.eval}, [sid||j|ssid], \mathsf{U}', \mathbf{S}[j])\}_{j\in[t+1]}$ to $\mathcal{A}^*$.
7. On $(\mathsf{ppss.srec}, sid, ssid, \mathsf{S}, \mathsf{U})$ from $\mathcal{F}_{\text{aPPSS}}$, if SIM saved (sid, i, ω') for S in step 4 (otherwise ignore this message), send $(\mathsf{oprf.sndrcomplete}, [\mathsf{S}||sid], ssid)$ to $\mathcal{A}^*$, and on reply tr_S, save $(\mathsf{oprf.sndrtrans}, [\mathsf{S}||sid], \mathsf{tr}_\mathsf{S})$, set $\mathsf{tx}[\mathsf{S}||sid]$++, and send (i, ω') to $\mathcal{A}^*$.
8. On $(\mathsf{oprf.rcvcomplete}, id_j^*, [sid||j||ssid], \mathsf{U}', \mathsf{tr}_{\mathsf{U}_j})$ and (i_j, ω_j') for any $j\in[t+1]$ from $\mathcal{A}^*$, for U' which ran $\mathsf{ppss.urec}$ on $(sid, ssid)$ in step 6 (otherwise ignore this message), proceed only if: (a) $\mathsf{tx}[id_j^*] > 0$, (b) if $\exists$ record $(\mathsf{oprf.sndrtrans}, id, \mathsf{S}, \mathsf{tr}_{\mathsf{U}_j})$ then it must hold that $id_j^* = id$. If all conditions are met then set $\mathsf{tx}[id_j^*]$−− and set $\mathsf{pointer}(ssid, i_j) := id_j^*$.

   Moreover, if $\mathsf{pointer}(ssid, i_j) \neq \perp$ for all $j \in [t+1]$ then send $(\mathsf{ppss.urec}, sid, ssid, \mathbf{C}, \mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*)$ to $\mathcal{F}_{\text{aPPSS}}$ for $(\mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*)$ s.t.:
   (a) If some i_j's are repeated, i.e. $\exists\, j_1 \neq j_2$ s.t. $i_{j_1} = i_{j_2}$, or if some ω_j''s are not the same, i.e. $\exists\, j_1, j_2$ s.t. $\omega_{j_1}' \neq \omega_{j_2}'$, then $(\mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*) := (0, \perp, \perp)$
   (b) Otherwise set $\omega' := \omega_1'$. If $\omega' = \omega$ and $\mathsf{pointer}(ssid, i_j) = \mathsf{pointer}(i_j)$ $\forall\, j\in[t+1]$ then $(\mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*) := (1, \perp, \perp)$.
   (c) Otherwise, parse $\omega' = (\mathbf{e}', C')$. Check if there exists $x \in x_H$ such that $\mathsf{H}(x) = [C'||\mathsf{sk}']$ for any $\mathsf{sk}' \in \{0,1\}^\lambda$. If there exists such x then parse $x = (\mathsf{pw}'', \mathbf{e}'', s'')$, set $I := \{i_j \,|\, j \in [t+1]\}$. If $F_{\mathsf{pointer}(ssid,i)}(\mathsf{pw}'') \neq \perp$ for all $i \in I$, then set $s_i' := e_i' \oplus F_{\mathsf{pointer}(ssid,i)}(\mathsf{pw}'')$ for all $i \in I$, interpolate s' from $\{(i, s_i')\}_{i\in I}$). If $s'' = s'$ and $\mathbf{e}'' = \mathbf{e}'$, then set $(\mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*) := (2, \mathsf{pw}'', \mathsf{sk}'')$.
   (d) If $(\mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*)$ were not set above, then $(\mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*) := (0, \perp, \perp)$.

Fig. 14: Simulator SIM for protocol Π_{aPPSS} (reconstruction and on-line attacks)

If $\exists\, j_1 \neq j_2$ s.t. $i_{j_1} = i_{j_2}$ or $\omega_{j_1} \neq \omega_{j_2}$ or $\exists\, j$ s.t. $i_j \notin [n]$, output $\perp$ and halt. Otherwise set $\rho'_{i_j} := \phi_j$ for $j \in [t+1]$ and $I := \{i_j \,|\, j \in [t+1]\}$ and $\omega' := \omega_1$. G_2 modifies G_1 if $\omega' = \omega$ and $id_j^* = id_j$ for all $j \in [t+1]$: in this case G_2 outputs sk, if $\mathsf{pw}' = \mathsf{pw}$. If $\mathsf{pw}' \neq \mathsf{pw}$, then G_2 outputs $\perp$. Observe, that if $\mathsf{pw}' \neq \mathsf{pw}$, then U in G_1 calculates $\mathsf{H}(\mathsf{pw}', \mathbf{e}, s') := [C'||\mathsf{sk}']$. From the definition of H we obtain $C' \neq C$ and G_1 outputs $\perp$. On the other hand, if $\mathsf{pw}' = \mathsf{pw}$, then both games output sk.

$$Dist_{\mathcal{Z}}^{G_1,G_2} = 0$$

Game G_3: In step 2 of User Reconstruction, U gets $(\mathsf{oprf.eval}, [sid||j||ssid], \phi_j, \mathsf{tr}_j)$ from $\mathcal{F}_{\text{OPRF}}$ and messages (i_j, ω_j) from $\mathbf{S}[j]$, for all $j \in [t+1]$. Values ϕ_j are calculated as a value of a random function $F_{id^*}(\mathsf{pw}')$, where id^* was chosen by $\mathcal{A}^*$.

If $\exists\, j_1 \neq j_2$ s.t. $i_{j_1} = i_{j_2}$ or $\omega_{j_1} \neq \omega_{j_2}$ or $\exists\, j$ s.t. $i_j \notin [n]$, output $\perp$ and halt. Otherwise set $\rho'_{i_j} := \phi_j$ for $j \in [t+1]$ and $I := \{i_j \,|\, j \in [t+1]\}$ and $\omega' := \omega_1$.

If $\omega' \neq \omega$ or there exists $j \in [t{+}1]$ such that $id_j^* \neq id_j$ G_3 modifies G_2 in the following way:
In steps 3. and 4. of User Reconstruction U reconstructs s'. If there exists x such that $\mathsf{H}(x) = [C'||\mathsf{sk}^*]$ for some sk^*, then parse $x = (\mathsf{pw}^*, \mathbf{e}^*, s^*)$, else set $(\mathsf{pw}^*, \mathbf{e}^*, s^*) = (\perp, \perp, \perp)$. If $\mathbf{e}^* = \mathbf{e}'$ and $s^* = s'$, then check if $\mathsf{pw}^* = \mathsf{pw}'$. If all statements are true, then output sk^*. Otherwise output $\perp$.

Observe, that if G_3 outputs $\mathsf{sk}^* \neq \perp$, then G_2 will output the same sk^*. In both games U reconstructs the same s' and there $\exists x = (\mathsf{pw}', \mathbf{e}', s')$ s.t. $\mathsf{H}_L(x) = C'$; both games output $\mathsf{H}_R(x) = \mathsf{sk}^*$. On the other hand, if G_2 outputs $\perp$, then G_3 also outputs $\perp$. It is so, because G_2 outputs $\perp$ when $\mathsf{H}_L(\mathsf{pw}', \mathbf{e}', s') \neq C'$ and if G_3 outputs $\mathsf{sk} \neq \perp$, then there exists x such that $\mathsf{H}_L(x) = C'$ and $x = (\mathsf{pw}^*, \mathbf{e}', s')$. However, $\mathsf{pw}^* = \mathsf{pw}'$, which leads to a contradiction.

In the rest of the analysis, we show, that if G_2 outputs $\mathsf{sk}^* \neq \perp$, then G_3 also outputs sk^*. We can differentiate 3 cases:

1. Case 1: "$C' \neq C$".
   G_2 outputs $\mathsf{sk}' \neq \perp$, if $\mathsf{H}(\mathsf{pw}', \mathbf{e}', s') = [C'||\mathsf{sk}']$. There exists at most one value $(\mathsf{pw}', \mathbf{e}', s')$ which, when input to H outputs C' as first λ bits.
   If $\mathcal{A}^*$ has queried H on $(\mathsf{pw}', \mathbf{e}', s')$ before, then G_3 outputs sk'. Otherwise, if $\mathcal{A}^*$ has not queried H on $(\mathsf{pw}', \mathbf{e}', s')$ before, then G_3 outputs $\perp$ and $\mathsf{H}_L(\mathsf{pw}', \mathbf{e}', s')$ is picked at random from $\{0,1\}^\lambda$. Probability, that $\mathsf{H}_L(\mathsf{pw}', \mathbf{e}', s') = [C']$ is equal to $\frac{1}{2^\lambda}$.
2. Case 2: "$C' = C, \mathbf{e}' \neq \mathbf{e}$".
   There exists exactly one value $(\mathsf{pw}, \mathbf{e}, s)$ which when input to H outputs C as first λ bits. In this case $\mathsf{H}(\mathsf{pw}', \mathbf{e}', s') \neq [C||\mathsf{sk}^*]$ for every sk^*, which results in both games G_2 and G_3 outputting $\perp$.
3. Case 3: "$C' = C, \mathbf{e}' = \mathbf{e}, \exists i : id_i^* \neq id_i$".
   In both games G_2 and G_3 user U reconstructs s' from extrapolating values $s_i' = e_i' \oplus \rho_i'$. If $s_i' \neq s_i$ for at least one $i \in [t{+}1]$, then the probability of reconstructing $s' = s$ is at most $\frac{1}{2^\lambda}$. There exists only one $s' = s$ such that $\mathsf{H}_L(\mathsf{pw}', \mathbf{e}', s') = C'$, for which G_2 outputs $\mathsf{sk}^* \neq \perp$.
   On the other hand, $F_{id_i^*}(\mathsf{pw}')$ and $F_{id_i}(\mathsf{pw}')$ are both random strings in $\{0,1\}^\lambda$, so the probability that $F_{id_i^*}(\mathsf{pw}') = F_{id_i}(\mathsf{pw}')$ is equal to $\frac{1}{2^\lambda}$. In this case G_2 outputs sk, if $\mathsf{pw}' = \mathsf{pw}$ and G_3 always outputs $\perp$.

$$Dist_{\mathcal{Z}}^{G_2, G_3} \leq \frac{1}{2^\lambda}$$

Game G_4: If $\mathcal{A}^*$ has queried $\mathsf{H}(\mathsf{pw}, \mathbf{e}, s)$ before step 4 of User Initialization, then abort. $\mathcal{A}^*$ can query H at most q_H times and s is chosen at random from $\mathbb{F}$ in step 3 of User Initialization, where $|\mathbb{F}| = 2^\lambda$. If $\mathcal{A}^*$ did not query $\mathsf{H}(\mathsf{pw}, \mathbf{e}, s)$, then $[C||\mathsf{sk}]$ in both games are indistinguishable, because $[C||\mathsf{sk}]$ is s independently random of everything else.

$$Dist_{\mathcal{Z}}^{G_3, G_4} \leq \frac{q_H}{2^\lambda}$$

Game G_5: User U picks s at random as in $G43$ in step 3 of User Initialization, but in step 4 picks $\mathbf{e} = (e_1, ..., e_n)$ at random and sets $F_{id_i}(\mathsf{pw}) := e_i \oplus s_i$ for

$i \in [n]$. By definition of $\mathcal{F}_{\text{OPRF}}$ values of $F_{id_i}(\mathsf{pw})$ in G_4 are chosen at random from $\{0,1\}^\lambda$. In G_5 they are calculated as *xor* of a random values e_i and s_i, which also results in $F_{id_i}(\mathsf{pw})$ being a random value. By the same logic, in G_4 values e_i's are calculated as *xor* of a random value $F_{id_i}(\mathsf{pw})$ and s_i, which results in a random value, just like in G_5.

$$Dist_{\mathcal{Z}}^{G_4,G_5} = 0$$

Game G_6: Leave values of $F_{id_i}(\mathsf{pw})$ undefined until $\mathcal{A}^*$ queries them. Obviously,

$$Dist_{\mathcal{Z}}^{G_5,G_6} = 0$$

Game G_7: Do not pick secret-sharing of s in step 3 of User Initialization, but leave s_i undefined until $F_{id_i}(\mathsf{pw})$ is queried by $\mathcal{A}^*$. Then pick s_i at random and set $F_{id_i}(\mathsf{pw}) := e_i \oplus s_i$, unless $\mathcal{A}^*$ queried before t servers on pw. If $\mathcal{A}^*$ has queried t servers, extrapolate $(t+1)$-st to n-th shares from s and previously sent s_i's. Note that until the SPT event occurs, $\mathbf{s}$ is independently random from any other value in G_6. Obviously:

$$Dist_{\mathcal{Z}}^{G_6,G_7} = 0$$

Game G_8: is the simulated world. We can see that the change from G_7 to G_8 is merely conceptual, with the game challenger split into the ideal functionality $\mathcal{F}_{\text{aPPSS}}$ and the simulator SIM. We have that

$$Dist_{\mathcal{Z}}^{G_7,G_8} = 0$$

Summing up all results above, we conclude that $\mathcal{Z}$'s distinguishing advantage between the real world and the simulated world is a negligible function of λ . This completes the proof.

E Changes between UC aPPSS and UC PPSS of [30]

We list the changes between the augmented (and adaptively secure) PPSS functionality $\mathcal{F}_{\text{aPPSS}}$ of Figure 3 and the PPSS functionality $\mathcal{F}_{\text{PPSS}}$ of [30]. The main differences which make the functionality augmented and adaptively secure are as follows:

1. $\mathcal{F}_{\text{aPPSS}}$ allows adaptive corruptions of secret-sharing parties via interface $\mathsf{ppss.compromise}$, which did not exist in $\mathcal{F}_{\text{PPSS}}$. The effect of $\mathsf{ppss.compromise}$ is adding a party to the corrupted parties list **Corr**.
2. In the initialization stage $\mathcal{F}_{\text{PPSS}}$ reveals secret sk to the adversary if at least $t+1$ parties in **P** are in **Corr**. In $\mathcal{F}_{\text{aPPSS}}$ this leakage is not automatic, and requires the adversary to stage an offline dictionary attack, as explained in the next item.

3. If $\mathsf{P}_i \in \mathbf{Corr}$ then $\mathcal{F}_{\text{aPPSS}}$ allows $\mathcal{A}^*$ to execute the PPSS reconstruction protocol on behalf of P_i, via command ppss.srec. The effect of P_i executing the PPSS reconstruction protocol is modeled by incrementing P_i's ticket counter $\mathsf{tx}(\mathsf{P}_i)$. Such ticket increase can be then utilized for *either* completing PPSS reconstruction by some user U' via query ppss.urec, *or* for an adversarial offline password test via query ppss.testpw.
4. In the processing of offline password test command ppss.testpw functionality $\mathcal{F}_{\text{PPSS}}$ allows the adversary to test a password guess pw^* by utilizing the tickets from $(t+1) - |\mathbf{P} \cap \mathbf{Corr}|$ servers. In other words, the adversary must explicitly perform the test using only uncorrupted servers, whereas the contributions from corrupt servers come "for free". By contrast, $\mathcal{F}_{\text{aPPSS}}$ requires that the adversary utilizes the tickets for some $t+1$ serves for each password test. Such tickets can be obtained if the environment asks P_i to run ppss.srec (see item 6 below) or, if P_i is corrupted, then the adversary $\mathcal{A}^*$ can execute PPSS reconstruction on behalf of P_i as well (see item 3 above).

In addition to the above essential changes we made several changes which include adjusting the functionality to the setting of unauthenticated channels in the reconstruction stage, simplifying some syntax, and adjusting the functionality to either strengthen the model by giving more power to the environment, or to weaken the model to allow for its efficient realization. We stress that the latter modifications still create a useful primitive, because, as we show, any protocol that realizes $\mathcal{F}_{\text{aPPSS}}$ can be used in black-box way to implement a password protected signature.

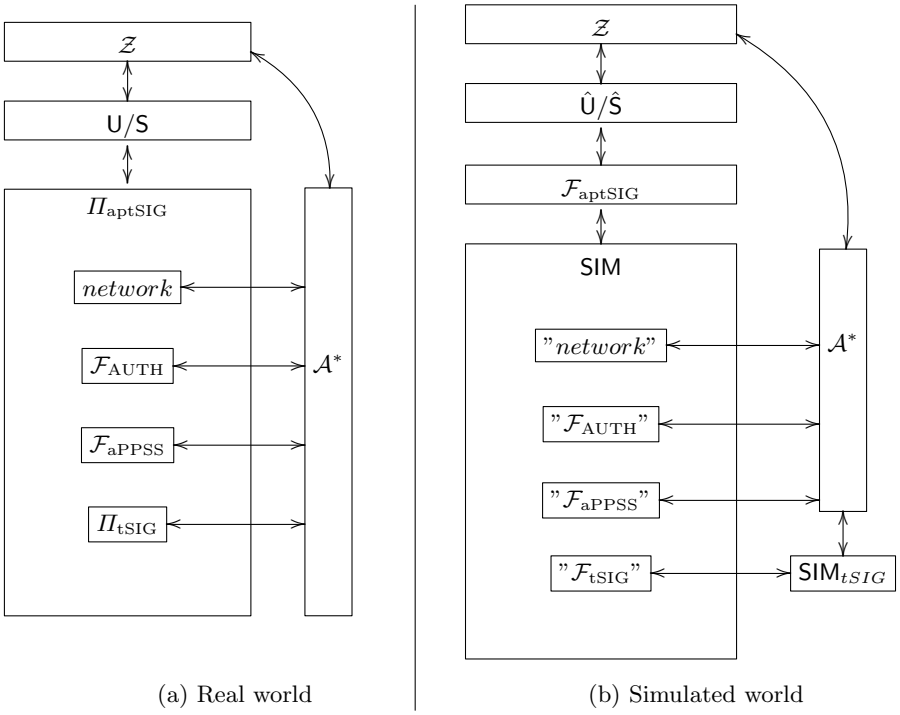
5. Functionality $\mathcal{F}_{\text{PPSS}}$ assumes authenticated channels in both initialization and reconstruction phases while $\mathcal{F}_{\text{aPPSS}}$ assumes it only in initialization. This is reflected by the reconstruction query ppss.urec not including the set $\mathbf{R}$ of parties which the user intends to participate in the reconstruction instance. (See also discussion below.) The effect of this change is that $\mathcal{F}_{\text{aPPSS}}$ allows $\mathcal{A}^*$ to specify in ppss.urec any set $\mathbf{C}$ of parties with positive ticket counters as the participants in the reconstruction, whereas $\mathcal{F}_{\text{PPSS}}$ required that set $\mathbf{C}$ is identical to set $\mathbf{R}$ except for parties in $\mathbf{R} \cap \mathbf{Corr}$.
6. Functionality $\mathcal{F}_{\text{PPSS}}$ gave $\mathcal{A}^*$ the power to decide on the participation of party P_i in both the PPSS initialization and in the PPSS reconstruction instances. This gives too much power to the adversary, esp. regarding reconstruction. We tighten up this processing in $\mathcal{F}_{\text{aPPSS}}$ so that queries ppss.sinit and ppss.srec are both made by P_i, and the adversary $\mathcal{A}^*$ is merely informed about it. This change allows us also to model offline password tests correctly, see item 4 above, because offline password tests require utilizing ticket counters of $t+1$ active servers, and this change assures that the adversary cannot increase servers' ticket counters at will, except in the case of corrupted servers.
7. Since P_i can be marked ACTIVE only if $\mathsf{P}_i \in \mathbf{P}$, for set $\mathbf{P}$ specified in ppss.uinit, and ticket counter $\mathsf{tx}(\mathsf{P}_i)$ can be increased only if P_i is marked ACTIVE, functionality $\mathcal{F}_{\text{aPPSS}}$ foregoes on checking membership of P_i in $\mathbf{P}$ in the processing of ppss.urec and ppss.testpw.

8. Functionality $\mathcal{F}_{\text{aPPSS}}$ drops the sub-session identifier *ssid* input from the server reconstruction query ppss.srec. This input was present in $\mathcal{F}_{\text{PPSS}}$ but it played no security role, and it looked as if the functionality enforces consistency of *ssid* identifiers used by U' and P_i in the reconstruction, which is not the case.

F Proof for aptSIG scheme security

In this section we provide the missing proof of Theorem 3 of Section 4.3.

Simulation Overview. We start by showing in Figure 15 the top-level view of the interactions between different components, including protocols, functionalities, and simulators, which define the real-world and ideal-world executions. For a more detailed overview of these interactions please see Section 4.3.



(a) Real world (b) Simulated world

Fig. 15: Real-world vs. Ideal-World interactions

Simulation of honest parties in Signing. Most of the cases are trivial as SIM can determine the respective case using flag and ptsig.pretest and handle

On $(\mathsf{ptsig.uinit}, sid, \mathsf{U})$ from $\mathcal{F}_{\mathrm{aptSIG}}$ for honest U:

1. Send $(\mathsf{ppss.uinit}, sid, \mathsf{U})$ to $\mathcal{A}^*$
2. Wait to receive:
   (a) $(\mathsf{ptsig.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})$ from $\mathcal{F}_{\mathrm{aptSIG}}$ for all $\mathsf{P}_i \in \mathbf{P}_{sid} \setminus \mathbf{Corr}$
   (b) $(\mathsf{ppss.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})$ from $\mathcal{A}^*$ for all $\mathsf{P}_i \in \mathbf{P}_{sid} \cap \mathbf{Corr}$
   (c) $(\mathsf{ppss.fininit}, sid)$ from $\mathcal{A}^*$
3. Set $sid^+ := \mathsf{add}(sid, \mathsf{U})$ and send $(\mathsf{tsig.keygen}, sid^+, \mathsf{U})$ to SIM_{tSIG}
4. Wait to receive
   (d) $(\mathsf{tsig.keygen}, sid^+, \mathsf{P}_i)$ from SIM_{tSIG} for all $\mathsf{P}_i \in \mathbf{P}_{sid} \cap \mathbf{Corr}$
   (e) $(\mathsf{tsig.publickey}, sid^+, \mathsf{V})$ from SIM_{tSIG}
   (f) $(\mathsf{tsig.keygencomplete}, sid^+, \mathsf{U})$ from SIM_{tSIG}
5. Set $\mathsf{aec}_\mathsf{U} := \mathsf{SIM}_{AE}(\mathsf{len}_\mathsf{m})$, send $\{(\mathsf{send}, sid, \mathsf{U}, \mathsf{P}_i, \mathsf{aec}_\mathsf{U})\}_{\mathsf{P}_i \in \mathbf{P}_{sid}}$ to $\mathcal{A}^*$, send $\{(\mathsf{ptsig.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})\}_{\mathsf{P}_i \in \mathbf{P}_{sid} \cap \mathbf{Corr}}$ and $(\mathsf{ptsig.uinit}, sid, \mathsf{V})$ to $\mathcal{F}_{\mathrm{aptSIG}}$.

On $(\mathsf{ptsig.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ from $\mathcal{F}_{\mathrm{aptSIG}}$ for honest $\mathsf{S} \in \mathbf{P}_{sid}$ (assuming honest U):

1. Send $(\mathsf{ppss.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ to $\mathcal{A}^*$
2. Set $sid^+ := \mathsf{add}(sid, \mathsf{U})$ and send $(\mathsf{tsig.keygen}, sid^+, \mathsf{S})$ to SIM_{tSIG}
3. Wait to receive
   (a) $(\mathsf{ptsig.uinit}, sid, \mathsf{U})$ from $\mathcal{F}_{\mathrm{aptSIG}}$
   (b) $(\mathsf{ptsig.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})$ from $\mathcal{F}_{\mathrm{aptSIG}}$ for all $\mathsf{P}_i \in \mathbf{P}_{sid} \setminus (\mathbf{Corr} \cup \{\mathsf{S}\})$
   (c) $(\mathsf{ppss.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})$ from $\mathcal{A}^*$ for all $\mathsf{P}_i \in \mathbf{P} \cap \mathbf{Corr}$
   (d) $(\mathsf{tsig.keygen}, sid^+, \mathsf{P}_i)$ from SIM_{tSIG} for all $\mathsf{P}_i \in \mathbf{P} \cap \mathbf{Corr}$
   (e) $(\mathsf{tsig.publickey}, sid^+, \mathsf{V})$ from SIM_{tSIG}
   (f) $(\mathsf{tsig.keygencomplete}, sid^+, \mathsf{U})$ from SIM_{tSIG}
   (g) $(\mathsf{sent}, sid, \mathsf{U}, \mathsf{S}, \mathsf{aec}_\mathsf{U})$ from $\mathcal{A}^*$
4. If all messages received, save record $(sid, sid^+, \mathsf{aec}_\mathsf{U})$, mark S as *initialized*.

Fig. 16: Simulator for Π_{aptSIG} (1): Initialization for honest parties

accordingly as the code of $\mathcal{F}_{\mathrm{aPPSS}}$ and $\mathcal{F}_{\mathrm{tSIG}}$, except for the case where $\mathcal{A}^*$ sends $\mathsf{flag} = 2$ and correctly guess the password ($\mathsf{pw} = \mathsf{pw}'$ and $\mathsf{ptsig.pretest}$ returns a bit $b = 1$) and a shifted ciphertext $\mathsf{aec}_\mathsf{U}{}^* \leftarrow \mathsf{AuthDec}_{\mathsf{sk}^*}(\mathsf{U}, \mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U})$. In this case, SIM has to treat U as $\mathsf{attacked}$ and run $\mathsf{tSIG}.\Pi_{\mathsf{TSign}^+}$ with the adversarial inputs $(\mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U})$.

Simulation of corrupt parties in Signing. Most of cases can be handled trivially as described in the Fig. 18 except for the case where $\mathcal{A}^*$ specifies $\mathsf{flag} = 1$ in $\mathsf{ppss.finrec}$ message with a correct password $\mathsf{pw} = \mathsf{pw}'$ (where $\mathsf{ptsig.pretest}$ returns a bit $b = 1$). Playing the role of $\mathcal{F}_{\mathrm{aPPSS}}$ the simulator SIM would need to return the secret key sk to the $\mathcal{A}^*$. In this case, SIM would need to go ahead and compromise the U in the tSIG protocol to extract the local state $(\mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U})$ from SIM_{tSIG} and invoke SIM_{AE} to equivocate the ciphertext aec_U to decrypt to $(\mathsf{U}, \mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U})$ under the key sk and then send sk to $\mathcal{A}^*$.

Proof. Let $\mathsf{Adv}^{\mathsf{ci},\mathsf{ae}}_\mathsf{D}(\lambda)$, $\mathsf{Adv}^{\mathsf{eqv},\mathsf{ae}}_\mathsf{D}(\lambda)$ denote an advantage of the algorithm D in the ciphertext integrity and the equivocable game of the authenticated encryption scheme AE.

On $(\mathsf{ptsig.usign}, sid, ssid, \mathsf{U}', \mathbf{S}, \mathsf{m})$ from $\mathcal{F}_{\text{aptSIG}}$ for honest U':

1. Send $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \mathbf{S})$ to $\mathcal{A}^*$
2. When $\mathcal{A}^*$ sends message $(\mathsf{ppss.finrec}, sid, ssid, \mathbf{C}, \mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*)$ and $\mathcal{A}^*$ sends $(sid, \mathsf{aec_U}^*)$ as a message from all $\mathsf{S} \in \mathbf{S}$ to U' (else go to step 2d) then:
   (a) If ($\mathsf{flag} = 1$ and $\mathsf{aec_U}^* = \mathsf{aec_U}$), then send $(\mathsf{ptsig.pretest}, sid, ssid, \mathbf{C}, \mathsf{flag}, \cdot)$ to $\mathcal{F}_{\text{aptSIG}}$, if $\mathcal{F}_{\text{aptSIG}}$'s replies $b = 0$ go to step 2d, else do:
      i. set $sid^+ := \mathsf{add}(sid, \mathsf{U})$ and send $(\mathsf{tsig.sign}, sid^+, \mathsf{U}', \mathsf{m})$ to SIM_{tSIG};
      ii. wait for $(\mathsf{tsig.signature}, sid^+, \mathsf{m}, \mathbf{P}, \sigma, \mathsf{P}^*)$ from SIM_{tSIG};
      iii. if $\sigma = \perp$ go to step 2d, else wait for the following messages:
         – $(\mathsf{ptsig.ssign}, sid, \cdot, \mathsf{S}, \cdot, \mathsf{m})$ from $\mathcal{F}_{\text{aptSIG}}$ for all $\mathsf{S} \in \mathbf{P} \setminus \mathbf{Corr}$
         – $(\mathsf{tsig.sign}, sid^+, \mathsf{m}, \mathsf{S})$ from SIM_{tSIG} for all $\mathsf{S} \in \mathbf{P} \cap \mathbf{Corr}$;
      iv. wait for $(\mathsf{tsig.signcomplete}, sid^+, \mathsf{U}')$ from SIM_{tSIG};
      v. send $(\mathsf{ptsig.finsign}, sid, ssid, \mathbf{P}, \mathsf{flag}, \sigma, \perp)$ to $\mathcal{F}_{\text{aptSIG}}$;
   (b) If ($\mathsf{flag} = 1$ and $\mathsf{aec_U}^* \neq \mathsf{aec_U}$ and U is marked $\mathsf{attacked}$), then reset $\mathsf{flag} := 2$, $\mathsf{sk}^* := \mathsf{sk_U}$, and $\mathsf{pw}^* := \mathsf{pw_U}$, and go to step 2c.
   (c) If $\mathsf{flag} = 2$, let b be $\mathcal{F}_{\text{aptSIG}}$'s reply to $(\mathsf{ptsig.pretest}, sid, ssid, \perp, \mathsf{flag}, \mathsf{pw}^*)$:
      i. If $b = 0$ or $\mathsf{AE.AuthDec}_{\mathsf{sk}^*}(\mathsf{aec_U}^*) = \perp$ then go to step 2d;
      ii. Else set $(\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U}) = \mathsf{AE.AuthDec}_{\mathsf{sk}^*}(\mathsf{aec_U}^*)$, $sid^+ = \mathsf{add}(sid, \mathsf{U})$, run $\mathsf{tSIG}.\Pi_{\mathsf{TSign}^+}$ on input $(sid^+, \mathsf{ts_U}, \mathsf{tcs_U}, \mathsf{m})$ on behalf of a virtual party U^*, interacting with SIM_{tSIG} as an adversary. If U^* completes the protocol with local output σ^* then send $(\mathsf{ptsig.finsign}, sid, ssid, \perp, \mathsf{flag}{=}2, \sigma^*, \perp)$ to $\mathcal{F}_{\text{aptSIG}}$;
   (d) Else send $(\mathsf{ptsig.finsign}, sid, ssid, \perp, \mathsf{flag}{=}0, \perp, \perp)$ to $\mathcal{F}_{\text{aptSIG}}$ and abort;

On $(\mathsf{ptsig.ssign}, sid, ssid, i, \mathsf{S}, \mathsf{U}', \mathsf{m})$ from $\mathcal{F}_{\text{aptSIG}}$ for honest S marked *initialized*:

1. Send $(\mathsf{ppss.srec}, sid, ssid, \mathsf{S}, \mathsf{U}')$ and $(sid, \mathsf{aec_U})$ to $\mathcal{A}^*$;
2. Recover record $(sid, sid^+, \mathsf{aec_U})$ and send $(\mathsf{tsig.sign}, sid^+, \mathsf{m})$ to $\mathsf{SIM}_{tSIG}+$.

Fig. 17: Simulator for Π_{aptSIG} (2): Signing for honest parties

We now show that the distinguishing advantage of $\mathcal{Z}$ between the real world and the simulated world is negligible. The argument uses a sequence of games, starting from the real world and ending at the simulated world; for any two adjacent games G_i and G_{i+1}, let $Dist_{\mathcal{Z}}^{G_i, G_{i+1}}$ denote the distinguishing advantage of $\mathcal{Z}$ between them, i.e., $Dist_{\mathcal{Z}}^{G_i, G_{i+1}} = |\Pr[\mathcal{Z} \text{ outputs 1 in } G_i] - \Pr[\mathcal{Z} \text{ outputs 1 in } G_{i+1}]|$. ($Dist_{\mathcal{Z}}^{G_i, G_{i+1}}$ is a function of the security parameter λ, but we omit λ below.)
In our analysis we will use following convention: if value is picked or calculated during Initialization, then it will be denoted in pure form e.g. pw, sk, $\mathsf{aec_U}$, $\mathsf{ts_U}$, $\mathsf{tcs_U}$. In Signing phase values have added apostrophe to denote the fact, that they were sent by $\mathcal{A}^*$ or calculated using other values sent by $\mathcal{A}^*$.

Game G_0: This is the real world execution.

Game G_1: User not marked $\mathsf{attacked}$ aborts if $\mathsf{pw}' = \mathsf{pw}$, $\mathcal{A}^*$ finalizes aPPSS with $\mathsf{flag} = 1$, but sends $\mathsf{aec_U}^* \neq \mathsf{aec_U}$. Note that G_1 is the same

On $(\mathsf{ppss.urec}, sid, ssid, \mathbf{S}, \mathsf{pw}')$ from $\mathcal{A}^*$ for corrupt U':

1. Send $(\mathsf{ptsig.usign}, sid, ssid, \mathbf{S}, \mathsf{pw}', \perp)$ to $\mathcal{F}_{\text{aptSIG}}$
2. If $\mathcal{A}^*$ sends $(\mathsf{ppss.finrec}, sid, ssid, \mathbf{C}, \mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*)$ then:
   (a) If $\mathsf{flag} = 0$ send $(\mathsf{ppss.finrec}, sid, ssid, \perp)$ to $\mathcal{A}^*$.
   (b) If $\mathsf{flag} = 1$ then add $ssid$ to Set_{ssid}:
      i. Send $(\mathsf{ptsig.pretest}, sid, ssid, \mathbf{C}, \mathsf{flag}{=}1, \perp)$ to $\mathcal{F}_{\text{aptSIG}}$ and if $\mathcal{F}_{\text{aptSIG}}$ sends $b = 0$ then send $(\mathsf{ppss.finrec}, sid, ssid, \perp)$ to $\mathcal{A}^*$;
      ii. Otherwise if $\mathcal{F}_{\text{aptSIG}}$ sends $b = 1$ then :
         A. If U is not yet marked $\mathsf{attacked}$ then mark U $\mathsf{attacked}$ and:
            – send $(\mathsf{tsig.compromise}, sid, \mathsf{U})$ to SIM_{tSIG} to obtain $(\mathsf{ts_U}, \mathsf{tcs_U})$
            – set $\mathsf{sk_U} \leftarrow \mathsf{SIM}_{AE}(\mathsf{aec_U}, (\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U}))$
         B. send $(\mathsf{ppss.finrec}, sid, ssid, \mathsf{sk_U})$ to $\mathcal{A}^*$
   (c) If $\mathsf{flag} = 2$ then send $(\mathsf{ppss.finrec}, sid, ssid, \mathsf{sk}^*)$ to $\mathcal{A}^*$ if $\mathsf{pw}^* = \mathsf{pw}'$, otherwise send $(\mathsf{ppss.finrec}, sid, ssid, \perp)$.

On $(\mathsf{tsig.sign}, sid^+, \mathsf{U}, \mathsf{m})$ from SIM_{tSIG} for U marked $\mathsf{attacked}$:

1. Wait to receive $(\mathsf{tsig.signature}, sid^+, \mathsf{m}, \mathbf{P}, \sigma^*, \mathsf{P}^*)$ from SIM_{tSIG}. If there were at least $t{+}1$ messages received of either of the following two types:
   (a) $(\mathsf{ptsig.ssign}, sid, \cdot, \mathsf{S}, \cdot, \mathsf{m})$ from $\mathcal{F}_{\text{aptSIG}}$ for $\mathsf{S} \in \mathbf{P}_{sid} \setminus \mathbf{Corr}$
   (b) $(\mathsf{tsig.sign}, sid^+, \mathsf{S}, \mathsf{m})$ from SIM_{tSIG} for $\mathsf{S} \in \mathbf{P}_{sid} \cap \mathbf{Corr}$
   Then remove one $ssid$ from Set_{ssid} and send $(\mathsf{ptsig.finsign}, sid, ssid, \mathbf{P}\backslash\{\mathsf{U}\}, \mathsf{flag}{=}1, \sigma^*, \mathsf{m})$ to $\mathcal{F}_{\text{aptSIG}}$.

On $(\mathsf{ppss.srec}, sid, \cdot, \mathsf{S}, \cdot)$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{P}_{sid} \cap \mathbf{Corr}$:

1. Send $(\mathsf{ptsig.ssign}, sid, ssid{=}\perp, \mathsf{S}, \mathsf{U}'{=}\perp, \mathsf{m}{=}\perp, 1)$ to $\mathcal{F}_{\text{aptSIG}}$.

On $(\mathsf{tsig.sign}, sid^+, \mathsf{S}, \mathsf{m})$ from SIM_{tSIG} for $\mathsf{S} \in \mathbf{P}_{sid} \cap \mathbf{Corr}$:

1. Send $(\mathsf{ptsig.ssign}, sid, ssid{=}\perp, \mathsf{S}, \mathsf{U}'{=}\perp, \mathsf{m}, 0)$ to $\mathcal{F}_{\text{aptSIG}}$.

Fig. 18: Simulator for Π_{aptSIG} (3): Signing for corrupt parties

as G_0 except possibly in the case where the user is not marked $\mathsf{attacked}$, $\mathcal{A}^*$ finalizes aPPSS with $\mathsf{flag} = 1$ (which denotes passive completion of the aPPSS reconstruction protocol), the user runs on the correct password $\mathsf{pw}' = \mathsf{pw}$, but the user receives $\mathsf{aec_U}^* \neq \mathsf{aec_U}$ from $\mathcal{A}^*$.

In this case, in the ideal world, the user would abort following sending $(\mathsf{ptsig.finsign}.sid, ssid, \mathsf{flag} = 0, \perp, .)$ to $\mathcal{F}_{\text{aptSIG}}$. In the real world (game G_0), the user would also abort unless it can decrypt $\mathsf{aec_U}^*$ using sk. This implies that G_0 and G_1 are identical unless $\perp \neq \mathsf{AuthDec_{sk}}(\mathsf{aec_U}^*)$. We can construct a reduction $\mathcal{R}_{CI}$ to the ciphertext authenticity property of AE where sk is the challenge AE key and $\mathsf{aec_U}$ is the challenge ciphertext: $\mathcal{R}_{CI}$ runs the code of G_0 except that it "outsources" values sk and $\mathsf{aec_U}$ created in the initialization to the CI challenge oracle, and then submits $\mathsf{aec_U}^*$ sent by $\mathcal{A}^*$ as the CI forgery. We have that

$$Dist_{\mathcal{Z}}^{G_0,G_1} \leq \mathsf{Adv}^{\mathsf{ci,ae}}_{\mathcal{R}_{CI}}(\lambda)$$

which is a negligible function of λ.

Note: We assume $\mathcal{A}^*$ compromises all components of S at the same time.

On $(\mathsf{ppss.testpw}, sid, \mathsf{S}, \mathsf{pw}^*)$ from $\mathcal{A}^*$, send $(\mathsf{ptsig.testpw}, sid, \mathsf{S}, \mathsf{pw}^*)$ to $\mathcal{F}_{\text{aptSIG}}$, and let b be $\mathcal{F}_{\text{aptSIG}}$'s response. If $b = 0$ then sent $\perp$ to $\mathcal{A}^*$, else do:

1. If U is already marked $\mathsf{attacked}$ then send $\mathsf{sk_U}$ to $\mathcal{A}^*$ and abort;
2. Otherwise mark U $\mathsf{attacked}$ and:
   - send $(\mathsf{tsig.compromise}, sid, \mathsf{U})$ to SIM_{tSIG} to obtain $(\mathsf{ts_U}, \mathsf{tcs_U})$
   - set $\mathsf{sk_U} \leftarrow \mathsf{SIM}_{AE}(\mathsf{aec_U}, (\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U}))$
   - send $\mathsf{sk_U}$ to $\mathcal{A}^*$

Fig. 19: Simulator for Π_{aptSIG} (4): Test Password

Game G_2: creating $\mathsf{aec_U}$ via encryption equivocability simulator SIM_{AE}. At the beginning of the game, let SIM_{AE} simulate $\mathsf{aec_U}$ and leave sk undefined, then let SIM_{AE} simulate sk when $\mathcal{A}^*$ computes it, i.e.

1. At the beginning of the game, set $\mathsf{aec_U} \leftarrow \mathsf{SIM}_{AE}(\mathsf{len_m})$
2. When $\mathcal{A}^*$ obtains the correct password pw through either the online or offline password test avenue ($\mathsf{ppss.testpw}$ with enough tickets $\mathsf{tx(S)}$). In such an event SIM would go ahead and compromise U in the tSIG scheme and obtain the simulated local state of the user U $(\mathsf{ts_U}, \mathsf{tcs_U})$ from SIM_{tSIG}. Then SIM asks SIM_{AE} to equivocate $\mathsf{sk} \leftarrow \mathsf{SIM}_{AE}(\mathsf{aec_U}, (\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U}))$.

Observe that in G_1, $\mathcal{Z}$ sees $\mathsf{aec_U} \leftarrow \mathsf{AuthEnc_{sk}}(\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U})$, and unless and until $\mathcal{A}^*$ obtains the correct password pw and learns sk, sk is not used by any party except in generating $\mathsf{aec_U}$, hence is a random string in $\{0,1\}^\lambda$ and independent of everything else (except for $\mathsf{aec_U}$) in $\mathcal{Z}$'s view. , in G_1, all processing is based on whether $\mathsf{flag} = 1$, $\mathsf{pw} = \mathsf{pw}'$, and $\mathsf{aec_U}^* \neq \mathsf{aec_U}$. Therefore, in G_5 $\mathsf{aec_U}$ followed by sk are formed as in the real game in the equivocability game of AE, where $\mathcal{A}^*$ sees the encryption $\mathsf{aec_U}$ of $(\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U})$ under key sk. On the other hand, in G_2, the ciphertext $\mathsf{aec_U}$ followed by key sk are formed as in the ideal game in the equivocability game of AE. We can thus conduct a reduction $\mathcal{R}_{EQV}$ to the equivocability of $\mathsf{AuthEnc}$: $\mathcal{R}_{EQV}$ runs the code of G_1 except that it uses input as $\mathsf{aec_U}$ and sk, and copies $\mathcal{Z}$'s output. We have that

$$Dist_{\mathcal{Z}}^{G_1,G_2} \leq \mathsf{Adv}^{\text{eqv},\mathsf{ae}}_{\mathcal{R}_{EQV}}(\lambda)$$

which is a negligible function of λ.

Game G_3: In this game we outsource the interaction with tSIG to functionality $\mathcal{F}_{\text{aptSIG}}$ and the simulator SIM_{tSIG}, whose existence is implied by the fact that tSIG UC-realizes functionality $\mathcal{F}_{\text{aptSIG}}$. Note that in game G_2 the uncompromised parties run tSIG key generation followed by tSIG signing on the created shares, and the rest of the game can be thought of as an environment for tSIG. The only apparent exception is when U runs tSIG singing not on shares create in key generation but on adversarial shares which it outputs from the aPPSS subprotocol.

That concretely is the case where $\mathcal{A}^*$ sends $\mathsf{flag} = 2$ and correctly guess the password ($\mathsf{pw} = \mathsf{pw}'$ and $\mathsf{ptsig.pretest}$ returns a bit $b = 1$) and a shifted ciphertext $\mathsf{aec_U}^* \leftarrow \mathsf{AuthDec}_{\mathsf{sk}^*}(\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U})$. In this case, SIM has to treat U as $\mathsf{attacked}$ and run $\mathsf{tSIG}.\Pi_{\mathsf{TSign}^+}$ with the adversarial inputs $(\mathsf{ts_U}, \mathsf{tcs_U})$. This is depicted in the Fig. 15 as $\mathsf{tSIG}+$, we treat such U execution separately, as an extension of $\mathcal{A}^*$, and after this "externalization" the rest of the game can be seen as an environment for tSIG.

Therefore, by the fact that tSIG UC-realizes $\mathcal{F}_{\text{aptSIG}}$, this environment cannot distinguish (except for negligible probability) an interaction with the tSIG implementation (as in G_2), from an interaction with functionality $\mathcal{F}_{\text{tSIG}}$ and simulator SIM_{tSIG}. Let $\mathcal{Z}'$ be an environment formed by $\mathcal{Z}$ together with the rest of game G_2. It follows that

$$Dist_Z^{G_2,G_3} \leq \mathsf{Adv}^{\mathsf{uc},\mathsf{tsig}}_{\mathcal{Z}'}(\lambda)$$

where $\mathsf{Adv}^{\mathsf{uc},\mathsf{tsig}}_{\mathcal{Z}'}(\lambda)$ is an upper-bound on the advantage of environment $\mathcal{Z}'$ in distinguishing between the real-world protocol tSIG and an ideal-world interaction of $\mathcal{F}_{\text{tSIG}}$ and simulator SIM_{tSIG}.

Game G_4: This is the ideal-world interaction where honest parties interact with $\mathcal{F}_{\text{aptSIG}}$ which in turn interacts with SIM as shown in Figures 16-19. We can see that the change from G_3 to G_4 is merely notational, with the game challenger split into the ideal functionality $\mathcal{F}_{\text{aptSIG}}$ and the simulator SIM which internally emulates functionalities $\mathcal{F}_{\text{aPPSS}}$ and $\mathcal{F}_{\text{AUTH}}$. It follows that

$$Dist_Z^{G_3,G_4} = 0$$

Summing up all inequalities above, we conclude that $\mathcal{Z}$'s distinguishing advantage between the real world and the simulated world is a negligible function of λ, which completes the proof.

G Password-Protected Signatures with PFS Security

In this section we present a variant of the aptSIG functionality $\mathcal{F}_{\text{aptSIG}}$ shown in Figure 5 in Section 4.1. The variant defined strengthens the notion by capturing the property of Perfect Forward Security (PFS). This functionality, denoted $\mathcal{F}_{\text{aptSIG}-\text{PFS}}$, is shown in Figure 20. The modifications are small, and they are marked in grey. Next, we present a protocol $\Pi_{\text{aptSIG}-\text{PFS}}$, shown in Figure 21, which modifies the Π_{aptSIG} protocol shown in Figure 7 in Section 4.2. Finally, we sketch why protocol $\Pi_{\text{aptSIG}-\text{PFS}}$ realizes functionality $\mathcal{F}_{\text{aptSIG}-\text{PFS}}$.

PFS version of aptSIG functionality. The difference between the aptSIG functionality $\mathcal{F}_{\text{aptSIG}-\text{PFS}}$ which captures the PFS property, and the basic funcionality $\mathcal{F}_{\text{aptSIG}}$, is very small, and it is all contained in how the functionality responds to the server S's signing query $\mathsf{ptsig.ssign}$. Namely, instead of "blindly" incrementing tickets $\mathsf{tx}(\mathsf{S})$ and issuing a S-willing-to-sign-m record $(sid, \mathsf{m}, \mathsf{S})$, the modified functionality still increments tickets $\mathsf{tx}(\mathsf{S})$, but it only issues record

Initialization:

1. On $(\mathsf{ptsig.uinit}, sid, \mathsf{pw})$ from party U for $sid = (..., \mathbf{P}_{sid})$ s.t. $|\mathbf{P}_{sid}| = n$, send $(\mathsf{ptsig.uinit}, sid, \mathsf{U})$ to $\mathcal{A}^*$, save $(sid, \mathsf{U}, \mathbf{P}_{sid}, \mathsf{pw})$ and set flag $\mathsf{flag}_{sid} = 0$. Ignore further $\mathsf{ptsig.uinit}$ calls for same sid.
2. On $(\mathsf{ptsig.sinit}, sid, i, \mathsf{U})$ from party S, or $(\mathsf{ptsig.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{Corr}$, send $(\mathsf{ptsig.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ to $\mathcal{A}^*$, save $(sid, \mathsf{U}, \mathsf{S}, i)$.
3. On $(\mathsf{ptsig.uinit}, sid, \mathsf{V})$ from $\mathcal{A}^*$, if $\exists$ record $(sid, \mathsf{U}, \mathbf{P}_{sid}, \mathsf{pw})$ and records $(sid, \mathsf{U}, \mathsf{S}, i)$ for each $\mathsf{S} \in \mathbf{P}_{sid}$, then create record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ and send $(\mathsf{ptsig.verificationkey}, sid, \mathsf{V})$ to U.

Server Compromise: (*This query requires permission from the environment.*) On $(\mathsf{ptsig.corrupt}, sid, \mathsf{P})$ from $\mathcal{A}^*$, set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$.

Signing:

1. On $(\mathsf{ptsig.usign}, sid, ssid, \mathbf{S}, \mathsf{pw}', \mathsf{m})$ from party U' or from $\mathsf{U}' = \mathcal{A}^*$, send $(\mathsf{ptsig.usign}, sid, ssid, \mathsf{U}', \mathbf{S}, \mathsf{m})$ to $\mathcal{A}^*$. If $\exists$ record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ then save $(sid, ssid, \mathsf{U}', \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{pw}', \mathsf{V}, \mathsf{m})$, else save $(sid, ssid, \mathsf{U}', \perp, \perp, \mathsf{pw}', \perp, \mathsf{m})$. Ignore future $\mathsf{ptsig.usign}$ calls for same $ssid$.
2. On $(\mathsf{ptsig.ssign}, sid, ssid, \mathsf{U}', \mathsf{m})$ from party S or $(\mathsf{ptsig.ssign}, sid, ssid, \mathsf{S}, \mathsf{U}', \mathsf{m}, b)$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{Corr}$, if $\exists$ record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ s.t. $\mathsf{S} \in \mathbf{P}_{sid}$ then send $(\mathsf{ptsig.ssign}, sid, ssid, \mathsf{S}, \mathsf{U}', \mathsf{m})$ to $\mathcal{A}^*$ and do:
   (a) in every case except ($\mathsf{S} \in \mathbf{Corr}$ and $b = 0$) set $\mathsf{tx}(\mathsf{S})$++;
   (b) if $\mathsf{S} \in \mathbf{Corr}$ or ($\mathsf{S} \notin \mathbf{Corr}$ and $\exists$ record $(sid, ssid, *, *, \mathsf{pw}, \mathsf{pw}', *, \mathsf{m})$ s.t. $\mathsf{pw} = \mathsf{pw}'$) then save $(sid, \mathsf{m}, \mathsf{S})$.
3. On $(\mathsf{ptsig.pretest}, sid, ssid, \mathbf{C}, \mathsf{flag}, \mathsf{pw}^*)$ from $\mathcal{A}^*$, if $\exists$ $\mathsf{rec} = (sid, ssid, \mathsf{U}', \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{pw}', \cdot, \cdot)$ not marked as $\mathsf{pretested}(c)$ for any c then:
   (a) if $\mathsf{flag} = 1$, $|\mathbf{C}| = t{+}1$, and $\forall_{\mathsf{S} \in \mathbf{C}}(\mathsf{tx}(\mathsf{S}){>}0)$, then set $\mathsf{tx}(\mathsf{S})--$ for all $\mathsf{S} \in \mathbf{C}$, set $b := (\mathsf{pw}' == \mathsf{pw})$, send b to $\mathcal{A}^*$ and mark rec as $\mathsf{pretested}(b)$; Moreover, if $b = 1$ and U' is corrupted or $\mathsf{U}' = \mathcal{A}^*$ then set $\mathsf{flag}_{sid} = 1$;
   (b) if $\mathsf{flag} = 2$ then set $b := (\mathsf{pw}' == \mathsf{pw}^*)$, send b to $\mathcal{A}^*$, and if $b = 1$ then mark rec as $\mathsf{pretested}(2)$ else mark rec as $\mathsf{pretested}(0)$.
4. On $(\mathsf{ptsig.finsign}, sid, ssid, \mathbf{C}', \mathsf{flag}, \sigma^*, \mathsf{m}^*)$ from $\mathcal{A}^*$, if $\exists$ $\mathsf{rec} = (sid, ssid, \mathsf{U}', \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{pw}', \mathsf{V}, \mathsf{m})$ then:
   (a) if $\mathsf{m} = \perp$ and U' is corrupted (or $\mathsf{U}' = \mathcal{A}^*$) then reset $\mathsf{m} := \mathsf{m}^*$;
   (b) if $\mathsf{flag}_{sid} = 1$, rec is marked $\mathsf{pretested}(0)$, and U' is corrupted or $\mathsf{U}' = \mathcal{A}^*$ then change rec mark to $\mathsf{pretested}(1)$;
   (c) send $(\mathsf{ptsig.finsign}, sid, ssid, \sigma)$ to U' s.t.
      i. if $\mathsf{flag} = 1$, rec is marked $\mathsf{pretested}(1)$, $|\mathbf{C}'| = t{+}1$, $\mathbf{C}' \subseteq \mathbf{P}_{sid}$, $\exists$ record $(sid, \mathsf{m}, \mathsf{S})$ for all $\mathsf{S} \in \mathbf{C}'$, $\mathsf{V}(\mathsf{m}, \sigma^*) = 1$, and there is no saved record $(sid, \mathsf{m}, \sigma^*, 0)$, then save record $(sid, \mathsf{m}, \sigma^*, 1)$ and set $\sigma := \sigma^*$;
      ii. if $\mathsf{flag} = 2$ and rec is marked $\mathsf{pretested}(2)$ then set $\sigma := \sigma^*$;
      iii. if neither of the above two cases is met set $\sigma := \perp$.

Verification:
On $(\mathsf{ptsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$ from Q, send $(\mathsf{ptsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$ to $\mathcal{A}^*$ and do:
- if $\exists$ records $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ and $(sid, \mathsf{m}, \sigma, \beta')$ then set $\beta := \beta'$;
- else, if $\exists$ record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$ but no $(sid, \mathsf{m}, \sigma, 1)$ for any σ then set $\beta := 0$;
- else set $\beta := \mathsf{V}(\mathsf{m}, \sigma)$.

Record $(sid, \mathsf{m}, \sigma, \beta)$ and send $(\mathsf{tsig.verified}, sid, \mathsf{m}, \sigma, \beta)$ to Q.

Password Test:
On $(\mathsf{ptsig.testpw}, sid, \mathsf{S}, \mathsf{pw}^*)$ from $\mathcal{A}^*$, retrieve record $(sid, \mathbf{P}_{sid}, \mathsf{pw}, \mathsf{V})$. If $\mathsf{tx}(\mathsf{S}) > 0$ then add S to set $\mathsf{ppss.pwtested}(\mathsf{pw}^*)$ and set $\mathsf{tx}(\mathsf{S})--$. If $|\mathsf{ppss.pwtested}(\mathsf{pw}^*)| = t + 1$ then return bit $b = (\mathsf{pw}^* == \mathsf{pw})$ to $\mathcal{A}^*$.

Fig. 20: $\mathcal{F}_{\text{aptSIG}-\text{PFS}}$: Functionality for Password-Protected Threshold Signature with PFS. Changes from $\mathcal{F}_{\text{aptSIG}}$ marked in gray.

Public parameters: Security parameter λ, threshold parameters t, n s.t. $t \leq n$. Let $\overline{\mathsf{AE}} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ be an Equivocable Authenticated Encryption scheme, let $\mathsf{tSIG} = (\Pi_{\mathsf{TKeyGen}}, \Pi_{\mathsf{TSign}}, \Pi_{\mathsf{TVerify}})$ be a Threshold Signature scheme realizing functionality $\mathcal{F}_{\mathrm{tSIG}}$ (see text), let $\mathsf{Sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a standard Signature scheme. Procedure $\mathsf{add}(sid, \mathsf{U})$ parses $sid = (sid', \mathbf{P}_{sid})$ and outputs $sid^+ = (sid', \mathbf{P}^+_{sid})$ s.t. if $\mathbf{P}_{sid} = (\mathsf{P}_1, ..., \mathsf{P}_n)$ then $\mathbf{P}^+_{sid} = (\mathsf{U}, \mathsf{P}_1, ..., \mathsf{P}_n)$.

Initialization for user U:

1. On input $(\mathsf{ptsig.uinit}, sid, \mathsf{pw})$, run $\mathsf{tSIG}.\Pi_{\mathsf{TKeyGen}^+}$ on input $sid^+ = \mathsf{add}(sid, \mathsf{U})$, let $(\mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U})$ and V be resp. U's local output and the generated public key.
2. Generate $(\mathsf{sks}_\mathsf{U}, \mathsf{pk}_\mathsf{U}) \leftarrow_\$ \mathsf{KeyGen}$.
3. Send $(\mathsf{ppss.uinit}, sid, \mathsf{pw}, \perp)$ to $\mathcal{F}_{\mathrm{aPPSS}}$, let sk be $\mathcal{F}_{\mathrm{aPPSS}}$'s output.
4. Set $\mathsf{aec}_\mathsf{U} := \mathsf{AE.AuthEnc}_\mathsf{sk}(\mathsf{U}, \mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U}, \mathsf{sks}_\mathsf{U})$, send $(\mathsf{send}, sid, \mathsf{P}_i, (\mathsf{aec}_\mathsf{U}, \mathsf{pk}_\mathsf{U}))$ to $\mathcal{F}_{\mathrm{AUTH}}$ for all $\mathsf{P}_i \in \mathbf{P}_{sid}$, output $(\mathsf{ptsig.verificationkey}, sid, \mathsf{V})$.

Initialization for server S:

1. On input $(\mathsf{ptsig.sinit}, sid, i, \mathsf{U})$, run $\mathsf{tSIG}.\Pi_{\mathsf{TKeyGen}^+}$ on $sid^+ = \mathsf{add}(sid, \mathsf{U})$ and send $(\mathsf{ppss.sinit}, sid, i, \mathsf{U})$ to $\mathcal{F}_{\mathrm{aPPSS}}$. Let $(\mathsf{ts}_i, \mathsf{tcs}_i)$ be S's local output from $\mathsf{tSIG}.\Pi_{\mathsf{TKeyGen}^+}$.
2. On $(\mathsf{sent}, sid, \mathsf{U}, \mathsf{S}, (\mathsf{aec}_\mathsf{U}, \mathsf{pk}_\mathsf{U}))$ from $\mathcal{F}_{\mathrm{AUTH}}$, save $(sid, sid^+, \mathsf{ts}_i, \mathsf{tcs}_i, \mathsf{aec}_\mathsf{U}, \mathsf{pk}_\mathsf{U})$.

Signing for user U′

1. On input $(\mathsf{ptsig.usign}, sid, ssid, \mathbf{S}, \mathsf{pw}', \mathsf{m})$ for $|\mathbf{S}| \geq t{+}1$ from U', send $(\mathsf{ppss.urec}, sid, ssid, \mathbf{S}, \mathsf{pw}')$ to $\mathcal{F}_{\mathrm{aPPSS}}$, and wait to receive $(\mathsf{ppss.urec}, sid, ssid, \mathsf{sk})$ from $\mathcal{F}_{\mathrm{aPPSS}}$ and messages aec_U for all $\mathsf{S} \in \mathbf{S}$.
2. If $\mathsf{sk} \neq \perp$, all $\mathsf{S} \in \mathbf{S}$ send the same value aec_U, and $\mathsf{AE.AuthDec}_\mathsf{sk}(\mathsf{aec}_\mathsf{U})$ output parses as $(\mathsf{U}, \mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U}, \mathsf{sks}_\mathsf{U})$, then send $\sigma_\mathsf{U} \leftarrow \mathsf{Sign}(\mathsf{sks}_\mathsf{U}, (\mathsf{m}, ssid))$ to each $\mathsf{S} \in \mathbf{S}$, but if any conditions fail then output $(\mathsf{ptsig.usign}, sid, ssid, \perp)$ and abort.
3. Run protocol $\mathsf{tSIG}.\Pi_{\mathsf{TSign}^+}$ on input $(sid^+, \mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U}, \mathsf{m})$ for $sid^+ = \mathsf{add}(sid, \mathsf{U})$, and when this protocol outputs σ, output $(\mathsf{ptsig.finsign}, sid, ssid, \sigma)$.

Signing for server S

1. On input $(\mathsf{ptsig.ssign}, sid, ssid, \mathsf{U}', \mathsf{m})$, retrieve stored tuple $(sid, sid^+, \mathsf{ts}_i, \mathsf{tcs}_i, \mathsf{aec}_\mathsf{U})$, send $(\mathsf{ppss.srec}, sid, ssid, \mathsf{U}')$ to $\mathcal{F}_{\mathrm{aPPSS}}$ and send aec_U to U'.
2. On received message σ_U, if $\mathsf{Verify}(\mathsf{pk}_\mathsf{U}, (\mathsf{m}, ssid), \sigma_\mathsf{U}) = 1$ then run $\mathsf{tSIG}.\Pi_{\mathsf{TSign}^+}$ on input $(sid^+, \mathsf{ts}_i, \mathsf{tcs}_i, \mathsf{m})$, otherwise abort.

Verification for Q

1. On input $(\mathsf{ptsig.verify}, sid, \mathsf{m}, \sigma, \mathsf{V})$, compute $\beta = \mathsf{tSIG}.\Pi_{\mathsf{TVerify}}(\mathsf{V}, \mathsf{m}, \sigma)$ and output $(\mathsf{ptsig.verified}, sid, \mathsf{m}, \sigma, \beta)$

Fig. 21: Protocol $\Pi_{\mathrm{aptSIG-PFS}}$ which realizes $\mathcal{F}_{\mathrm{aptSIG-PFS}}$ in $(\mathcal{F}_{\mathrm{aPPSS}}, \mathcal{F}_{\mathrm{AUTH}})$-hybrid world (shadow text omitted for Π_{aptSIG})

$(sid, \mathsf{m}, \mathsf{S})$ if there is a user-side signing record $(sid, ssid, *, *, \mathsf{pw}, \mathsf{pw}', *, \mathsf{m})$ s.t. $\mathsf{pw} = \mathsf{pw}'$, i.e. that some party (an honest or adversarial) was (1) requesting to sign m, (2) they did so while holding the right password, $\mathsf{pw}' = \mathsf{pw}$, and (3) they did so explicitly for the signature subsession identified as $ssid$, and this identifier matches the one that S used in the signing query ptsig.ssign. Only when all these conditions are satisfied then the record $(sid, \mathsf{m}, \mathsf{S})$ is created. (And if there are $t + 1$ such records for the same m then the user signing session will be allowed to generate a signature.)

The three above imply that any server sessions which terminated in the past, and they used $ssid$'s for which the user party (which includes an adversary) did not hold a correct password, such sessions did not create any willing-to-sign records which the adversary could "complete" some time in the future, when it captured the password. Indeed, the only way to create signatures using a password is if the servers agree to run ptsig.ssign with fresh subsession identifiers $ssid$, which the adversary will be able to complete by runnig ptsig.usign with the new $ssid$ and the compromised password.

PFS version of Π_{aptSIG} protocol. Protocol $\Pi_{\text{aptSIG}-\text{PFS}}$, shown in Figure 21, achieves PFS security in a simple way: First, we extend initialization so U creates a standard signature key pair $(\mathsf{sks}_\mathsf{U}, \mathsf{pk}_\mathsf{U})$, encrypts sks_U along with their share $\mathsf{ts}_\mathsf{U}, \mathsf{tcs}_\mathsf{U}$ in the threshold signature scheme in envelope aec_U, and registers pk_U with the servers along with aec_U. Then in the signing protocol U recovers sks_U along with their threshold signature share, and uses it to explicitly authorize issuing of a signature on m for subsession identifier $ssid$ (and to authenticate itself as owner of pw) by sending signature σ_U on $(\mathsf{m}, ssid)$ under key sks_U. Finally, each server S waits to receive such signature, and verifies it, before engaging in the threshold signature protocol on m.

Theorem 4 *If* $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ *is an Equivocable Authenticated Encryption,* $\mathsf{Sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *is an EUF-CMA Digital Signature Scheme, and* $\mathsf{tSIG} = (\Pi_{\mathsf{TKeyGen}}, \Pi_{\mathsf{TSign}}, \Pi_{\mathsf{TVerify}})$ *is a Threshold Signature scheme which UC-realizes functionality* $\mathcal{F}_{\text{tSIG}}$*, then protocol* Π_{aptSIG} *in Fig. 21 UC-realizes functionality* $\mathcal{F}_{\text{aptSIG}}$ *in Fig. 20 in the (*$\mathcal{F}_{\text{aPPSS}}$*,*$\mathcal{F}_{\text{AUTH}}$*)-hybrid model with Perfect Forward Secrecy.*

Proof. We show how to modify the simulator in Fig. 22-24. The series of games will include one more game G_{1b} that is in between G_1 and G_2 and we show the transition from G_1 to G_{1b} and from G_{1b} to G_2. We also make slight modification to G_2 by adding sks_U into the plaintext of aec_U but transition from G_2 to G_3 stays the same as it is not affected by the modification.

Game G_{1b}: creating pk_U and sks_U inside SIM. At the beginning of the game, let SIM_{SIG} generates pk_U and sk_U using the key generation algorithm Sig.KeyGen, then let SIM_{SIG} reveals sks_U when $\mathcal{A}^*$ computes it, i.e.

1. At the beginning of the game, set $(\mathsf{pk}_\mathsf{U}, \mathsf{sks}_\mathsf{U}) \leftarrow \mathsf{Sig.KeyGen}(1^\lambda)$
2. When the signature is needed in the simulation, SIM just sign using sks_U.

On $(\mathsf{ptsig.uinit}, sid, \mathsf{U})$ from $\mathcal{F}_{\text{aptSIG}}$ for honest U:

1. Send $(\mathsf{ppss.uinit}, sid, \mathsf{U})$ to $\mathcal{A}^*$
2. Wait to receive:
   (a) $(\mathsf{ptsig.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})$ from $\mathcal{F}_{\text{aptSIG}}$ for all $\mathsf{P}_i \in \mathbf{P}_{sid} \setminus \mathbf{Corr}$
   (b) $(\mathsf{ppss.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})$ from $\mathcal{A}^*$ for all $\mathsf{P}_i \in \mathbf{P}_{sid} \cap \mathbf{Corr}$
   (c) $(\mathsf{ppss.fininit}, sid)$ from $\mathcal{A}^*$
3. Set $sid^+ := \mathsf{add}(sid, \mathsf{U})$ and send $(\mathsf{tsig.keygen}, sid^+, \mathsf{U})$ to SIM_{tSIG}
4. Wait to receive
   (d) $(\mathsf{tsig.keygen}, sid^+, \mathsf{P}_i)$ from SIM_{tSIG} for all $\mathsf{P}_i \in \mathbf{P}_{sid} \cap \mathbf{Corr}$
   (e) $(\mathsf{tsig.publickey}, sid^+, \mathsf{V})$ from SIM_{tSIG}
   (f) $(\mathsf{tsig.keygencomplete}, sid^+, \mathsf{U})$ from SIM_{tSIG}
5. Set $\mathsf{aec_U} := \mathsf{SIM}_{AE}(\mathsf{len_m})$ and $(\mathsf{pk_U}, \mathsf{sks_U}) \leftarrow \mathsf{Sig.KeyGen}(1^\lambda)$, send $\{(\mathsf{send}, sid, \mathsf{U}, \mathsf{P}_i, \mathsf{aec_U}, \mathsf{pk_U})\}_{\mathsf{P}_i \in \mathbf{P}_{sid}}$ to $\mathcal{A}^*$,
   send $\{(\mathsf{ptsig.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})\}_{\mathsf{P}_i \in \mathbf{P}_{sid} \cap \mathbf{Corr}}$ and $(\mathsf{ptsig.uinit}, sid, \mathsf{V})$ to $\mathcal{F}_{\text{aptSIG}}$.

On $(\mathsf{ptsig.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ from $\mathcal{F}_{\text{aptSIG}}$ for honest $\mathsf{S} \in \mathbf{P}_{sid}$ (assuming honest U):

1. Send $(\mathsf{ppss.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ to $\mathcal{A}^*$
2. Set $sid^+ := \mathsf{add}(sid, \mathsf{U})$ and send $(\mathsf{tsig.keygen}, sid^+, \mathsf{S})$ to SIM_{tSIG}
3. Wait to receive
   (a) $(\mathsf{ptsig.uinit}, sid, \mathsf{U})$ from $\mathcal{F}_{\text{aptSIG}}$
   (b) $(\mathsf{ptsig.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})$ from $\mathcal{F}_{\text{aptSIG}}$ for all $\mathsf{P}_i \in \mathbf{P}_{sid} \setminus (\mathbf{Corr} \cup \{\mathsf{S}\})$
   (c) $(\mathsf{ppss.sinit}, sid, i, \mathsf{P}_i, \mathsf{U})$ from $\mathcal{A}^*$ for all $\mathsf{P}_i \in \mathbf{P} \cap \mathbf{Corr}$
   (d) $(\mathsf{tsig.keygen}, sid^+, \mathsf{P}_i)$ from SIM_{tSIG} for all $\mathsf{P}_i \in \mathbf{P} \cap \mathbf{Corr}$
   (e) $(\mathsf{tsig.publickey}, sid^+, \mathsf{V})$ from SIM_{tSIG}
   (f) $(\mathsf{tsig.keygencomplete}, sid^+, \mathsf{U})$ from SIM_{tSIG}
   (g) $(\mathsf{sent}, sid, \mathsf{U}, \mathsf{S}, \mathsf{aec_U}, \mathsf{pk_U})$ from $\mathcal{A}^*$
4. If all messages received, save record $(sid, sid^+, \mathsf{aec_U}, \mathsf{pk_U})$, mark S as *initialized*.

Fig. 22: Simulator for Π_{aptSIG} with Perfect Forward Secrecy (1): Initialization for honest parties

3. SIM verifies the signature σ_U^* sent by $\mathcal{A}^*$ when simulating the honest servers in signing and abort if $0 := \mathsf{Sig.Verify}(\mathsf{pk_U}, \sigma_U^*, \mathsf{m})$;

Observe that G_1 and G_{1b} are indistiguiable to $\mathcal{A}^*$ unless SIM does not abort in the case that σ_U^* sent by $\mathcal{A}^*$ meaning $1 := \mathsf{Sig.Verify}(\mathsf{pk_U}, \sigma_U^*, \mathsf{m})$. We have that

$$Dist_{\mathcal{Z}}^{G_1, G_{1b}} \leq \mathsf{Adv}_{\mathcal{Z}}^{\text{euf-cma}, \mathsf{sig}}(\lambda)$$

which is a negligible function of λ as Sig is EUF-CMA secure.

Game G_2: creating $\mathsf{aec_U}$ via encryption equivocability simulator SIM_{AE}. At the beginning of the game, let SIM_{AE} simulate $\mathsf{aec_U}$ and leave sk undefined, then let SIM_{AE} simulate sk when $\mathcal{A}^*$ computes it, i.e.

1. At the beginning of the game, set $\mathsf{aec_U} \leftarrow \mathsf{SIM}_{AE}(\mathsf{len_m})$
2. When $\mathcal{A}^*$ obtains the correct password pw through either the online or offline password test avenue ($\mathsf{ppss.testpw}$ with enough tickets $\mathsf{tx(S)}$). In such an

On $(\mathsf{ptsig.usign}, sid, ssid, \mathsf{U}', \mathbf{S}, \mathsf{m})$ from $\mathcal{F}_{\text{aptSIG}}$ for honest U':

1. Send $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \mathbf{S})$ to $\mathcal{A}^*$
2. When $\mathcal{A}^*$ sends message $(\mathsf{ppss.finrec}, sid, ssid, \mathbf{C}, \mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*)$ and $\mathcal{A}^*$ sends $(sid, \mathsf{aec_U}^*)$ as a message from all $\mathsf{S} \in \mathbf{S}$ to U' (else go to step 2d) then:
   (a) If ($\mathsf{flag} = 1$ and $\mathsf{aec_U}^* = \mathsf{aec_U}$),
   set $\sigma_{\mathsf{U}} \leftarrow \mathsf{Sig.Sign}(\mathsf{sks_U}, m)$ and sends it to $\mathcal{A}^*$
   then send $(\mathsf{ptsig.pretest}, sid, ssid, \mathbf{C}, \mathsf{flag}, \cdot)$ to $\mathcal{F}_{\text{aptSIG}}$, if $\mathcal{F}_{\text{aptSIG}}$'s replies $b = 0$ go to step 2d, else do:
      i. set $sid^+ := \mathsf{add}(sid, \mathsf{U})$ and send $(\mathsf{tsig.sign}, sid^+, \mathsf{U}', \mathsf{m})$ to SIM_{tSIG};
      ii. wait for $(\mathsf{tsig.signature}, sid^+, \mathsf{m}, \mathbf{P}, \sigma, \mathsf{P}^*)$ from SIM_{tSIG};
      iii. if $\sigma = \perp$ go to step 2d, else wait for the following messages:
         – $(\mathsf{ptsig.ssign}, sid, \cdot, \mathsf{S}, \cdot, \mathsf{m})$ from $\mathcal{F}_{\text{aptSIG}}$ for all $\mathsf{S} \in \mathbf{P} \setminus \mathbf{Corr}$
         – $(\mathsf{tsig.sign}, sid^+, \mathsf{m}, \mathsf{S})$ from SIM_{tSIG} for all $\mathsf{S} \in \mathbf{P} \cap \mathbf{Corr}$;
      iv. wait for $(\mathsf{tsig.signcomplete}, sid^+, \mathsf{U}')$ from SIM_{tSIG};
      v. send $(\mathsf{ptsig.finsign}, sid, ssid, \mathbf{P}, \mathsf{flag}, \sigma, \perp)$ to $\mathcal{F}_{\text{aptSIG}}$;
   (b) If ($\mathsf{flag} = 1$ and $\mathsf{aec_U}^* \neq \mathsf{aec_U}$ and U is marked $\mathsf{attacked}$), then reset $\mathsf{flag} := 2$, $\mathsf{sk}^* := \mathsf{sk_U}$, and $\mathsf{pw}^* := \mathsf{pw_U}$, and go to step 2c.
   (c) If $\mathsf{flag} = 2$, let b be $\mathcal{F}_{\text{aptSIG}}$'s reply to $(\mathsf{ptsig.pretest}, sid, ssid, \perp, \mathsf{flag}, \mathsf{pw}^*)$:
      i. If $b = 0$ or $\mathsf{AE.AuthDec}_{\mathsf{sk}^*}(\mathsf{aec_U}^*) = \perp$ then go to step 2d;
      ii. Else set $(\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U}, \mathsf{sks_U}) = \mathsf{AE.AuthDec}_{\mathsf{sk}^*}(\mathsf{aec_U}^*)$, $sid^+ = \mathsf{add}(sid, \mathsf{U})$, run $\sigma_{\mathsf{U}} \leftarrow \mathsf{Sig.Sign}(\mathsf{sk_U}, m, ssid)$ and send it to $\mathcal{A}^*$, and then run $\mathsf{tSIG}.\Pi_{\mathsf{TSign}^+}$ on input $(sid^+, \mathsf{ts_U}, \mathsf{tcs_U}, \mathsf{m})$ on behalf of a virtual party U^*, interacting with SIM_{tSIG} as an adversary. If U^* completes the protocol with local output σ^* then send $(\mathsf{ptsig.finsign}, sid, ssid, \perp, \mathsf{flag}{=}2, \sigma^*, \perp)$ to $\mathcal{F}_{\text{aptSIG}}$;
   (d) Else send $(\mathsf{ptsig.finsign}, sid, ssid, \perp, \mathsf{flag}{=}0, \perp, \perp)$ to $\mathcal{F}_{\text{aptSIG}}$ and abort;

On $(\mathsf{ptsig.ssign}, sid, ssid, i, \mathsf{S}, \mathsf{U}', \mathsf{m})$ from $\mathcal{F}_{\text{aptSIG}}$ for honest S marked *initialized*:

1. Send $(\mathsf{ppss.srec}, sid, ssid, \mathsf{S}, \mathsf{U}')$ and $(sid, \mathsf{aec_U},)$ to $\mathcal{A}^*$;
2. When $\mathcal{A}^*$ sends σ^*_{U} abort if $0 := \mathsf{Sig.Verify}(\mathsf{pk_U}, \sigma^*_U, \mathsf{m})$;
3. Recover record $(sid, sid^+, \mathsf{aec_U}, \mathsf{pk_U})$ and send $(\mathsf{tsig.sign}, sid^+, \mathsf{m})$ to $\mathsf{SIM}_{tSIG}+$.

Fig. 23: Simulator for Π_{aptSIG} with Perfect Forward Secrecy (2): Signing for honest parties

event SIM would go ahead and compromise U in the tSIG scheme and obtain the simulated local state of the user U $(\mathsf{ts_U}, \mathsf{tcs_U})$ from SIM_{tSIG}. Then SIM asks SIM_{AE} to equivocate $\mathsf{sk} \leftarrow \mathsf{SIM}_{AE}(\mathsf{aec_U}, (\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U}), \mathsf{sks_U})$.

Observe that in G_1, $\mathcal{Z}$ sees $\mathsf{aec_U} \leftarrow \mathsf{AuthEnc}_{\mathsf{sk}}(\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U}, \mathsf{sks_U})$, and unless and until $\mathcal{A}^*$ obtains the correct password pw and learns sk, secret key sk is not used by any party except in generating $\mathsf{aec_U}$, hence is a random string in $\{0,1\}^\lambda$ and independent of everything else (except for $\mathsf{aec_U}$) in $\mathcal{Z}$'s view. In G_1, all processing is based on whether $\mathsf{flag} = 1$, $\mathsf{pw} = \mathsf{pw}'$, and $\mathsf{aec_U}^* \neq \mathsf{aec_U}$. Therefore, in G_5 $\mathsf{aec_U}$ followed by sk are formed as in the real game in the equivocability game of AE, where $\mathcal{A}^*$ sees the encryption $\mathsf{aec_U}$ of $(\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U}, \mathsf{sks_U})$ under key sk. On the

On $(\mathsf{ppss.urec}, sid, ssid, \mathbf{S}, \mathsf{pw}')$ from $\mathcal{A}^*$ for corrupt U':

1. Send $(\mathsf{ptsig.usign}, sid, ssid, \mathbf{S}, \mathsf{pw}', \perp)$ to $\mathcal{F}_{\text{aptSIG}}$
2. If $\mathcal{A}^*$ sends $(\mathsf{ppss.finrec}, sid, ssid, \mathbf{C}, \mathsf{flag}, \mathsf{pw}^*, \mathsf{sk}^*)$ then:
   (a) If $\mathsf{flag} = 0$ send $(\mathsf{ppss.finrec}, sid, ssid, \perp)$ to $\mathcal{A}^*$.
   (b) If $\mathsf{flag} = 1$ then add $ssid$ to Set_{ssid}:
      i. Send $(\mathsf{ptsig.pretest}, sid, ssid, \mathbf{C}, \mathsf{flag}{=}1, \perp)$ to $\mathcal{F}_{\text{aptSIG}}$ and if $\mathcal{F}_{\text{aptSIG}}$ sends $b = 0$ then send $(\mathsf{ppss.finrec}, sid, ssid, \perp)$ to $\mathcal{A}^*$;
      ii. Otherwise if $\mathcal{F}_{\text{aptSIG}}$ sends $b = 1$ then and:
         A. If U is not yet marked $\mathsf{attacked}$ then mark U $\mathsf{attacked}$ and:
            – send $(\mathsf{tsig.compromise}, sid, \mathsf{U})$ to SIM_{tSIG} to obtain $(\mathsf{ts_U}, \mathsf{tcs_U})$
            – set $\mathsf{sk_U} \leftarrow \mathsf{SIM}_{AE}(\mathsf{aec_U}, (\mathsf{U}, \mathsf{ts_U}, \mathsf{tcs_U}, \mathsf{sks_U}))$
         B. send $(\mathsf{ppss.finrec}, sid, ssid, \mathsf{sks_U})$ to $\mathcal{A}^*$
   (c) If $\mathsf{flag} = 2$ then send $(\mathsf{ppss.finrec}, sid, ssid, \mathsf{sk}^*)$ to $\mathcal{A}^*$ if $\mathsf{pw}^* = \mathsf{pw}'$, otherwise send $(\mathsf{ppss.finrec}, sid, ssid, \perp)$.

On $(\mathsf{tsig.sign}, sid^+, \mathsf{U}, \mathsf{m})$ from SIM_{tSIG} for U marked $\mathsf{attacked}$:

1. Wait to receive $(\mathsf{tsig.signature}, sid^+, \mathsf{m}, \mathbf{P}, \sigma^*, \mathsf{P}^*)$ from SIM_{tSIG}. If there were at least $t{+}1$ messages received of either of the following two types:
   (a) $(\mathsf{ptsig.ssign}, sid, \cdot, \mathsf{S}, \cdot, \mathsf{m})$ from $\mathcal{F}_{\text{aptSIG}}$ for $\mathsf{S} \in \mathbf{P}_{sid} \setminus \mathbf{Corr}$
   (b) $(\mathsf{tsig.sign}, sid^+, \mathsf{S}, \mathsf{m})$ from SIM_{tSIG} for $\mathsf{S} \in \mathbf{P}_{sid} \cap \mathbf{Corr}$
   Then remove one $ssid$ from Set_{ssid} and send $(\mathsf{ptsig.finsign}, sid, ssid, \mathbf{P}\backslash\{\mathsf{U}\}, \mathsf{flag}{=}1, \sigma^*, \mathsf{m})$ to $\mathcal{F}_{\text{aptSIG}}$.

On $(\mathsf{ppss.srec}, sid, \cdot, \mathsf{S}, \cdot)$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{P}_{sid} \cap \mathbf{Corr}$:

1. Send $(\mathsf{ptsig.ssign}, sid, ssid{=}\perp, \mathsf{S}, \mathsf{U}'{=}\perp, \mathsf{m}{=}\perp, 1)$ to $\mathcal{F}_{\text{aptSIG}}$.

On $(\mathsf{tsig.sign}, sid^+, \mathsf{S}, \mathsf{m})$ from SIM_{tSIG} for $\mathsf{S} \in \mathbf{P}_{sid} \cap \mathbf{Corr}$:

1. Send $(\mathsf{ptsig.ssign}, sid, ssid{=}\perp, \mathsf{S}, \mathsf{U}'{=}\perp, \mathsf{m}, 0)$ to $\mathcal{F}_{\text{aptSIG}}$.

Fig. 24: Simulator for Π_{aptSIG} with Perfect Forward Secrecy (3): Signing for corrupt parties

other hand, in G_2, the ciphertext $\mathsf{aec_U}$ followed by key sk are formed as in the ideal game in the equivocability game of AE. We can thus conduct a reduction $\mathcal{R}_{EQV}$ to the equivocability of $\mathsf{AuthEnc}$: $\mathcal{R}_{EQV}$ runs the code of G_1 except that it uses input as $\mathsf{aec_U}$ and sk, and copies $\mathcal{Z}$'s output. We have that

$$Dist_{\mathcal{Z}}^{G_1,G_2} \leq \mathsf{Adv}^{\text{eqv},\mathsf{ae}}_{\mathcal{R}_{EQV}}(\lambda)$$

which is a negligible function of λ.

H Concrete implementation of aptSIG-PFS

In Section 5 we describe protocol aptSIG-BLS, which is a full instantiation of our generic aptSIG protocol using threshold BLS and the aPPSS scheme from Section 3 with OPRF implemented as 2HashDH. In this section we show a full instantiation, using the same components, of the aptSIG-PFS protocol shown in

Appendix G, i.e. a perfect forward secure extension of our main aptSIG scheme. We call this fully instantiated protocol *aptSIG-PFS-BLS* and we show it in Figure 25.

Protocol aptSIG-PFS-BLS introduces the following changes to its non-PFS version, i.e. to protocol aptSIG-BLS. In the initialization phase after step 6, the user generates a key pair $(\mathsf{sks}_\mathsf{U}, \mathsf{pk}_\mathsf{U}) \leftarrow_\$ \mathsf{KeyGen}$. Then in step 7 secret key sks_U is appended to the vector $(\mathsf{U}, \mathsf{V}, \vec{\mathsf{V}}, v_0)$ which is encrypted as aec_U, and the public key pk_U is attached to aec_U and sent through an authenticated channel to each server. In the signing phase party U in step 6 decrypts aec_U and obtains sk_U which she uses to sign $(\mathsf{m}, \mathit{ssid})$ and send it back to each server. The server executes step 3 in the signing phase only after receiving a a signature on $(\mathsf{m}, \mathit{ssid})$ valid under the verification key pk_U stored in the initialization record. In Figure 25 we show the resulting protocol, and we mark therein all the above differences compared to protocol aptSIG-BLS.

Parameters: Security parameter l, threshold parameters t, n, $t \leq n$, field $\mathbb{F} = GF(2^l)$, cyclic group G of prime order p, bilinear map group $\hat{G}$ of prime order $\hat{p}$ and generator $\hat{g}$; hash functions H_1, H_2, H_3, H_4 with ranges G, $\{0,1\}^l$, $\{0,1\}^{2l}$, $\hat{G}$. Let $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ be an Equivocable Authenticated Encryption and $\mathsf{Sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a standard Signature scheme. $\mathsf{auth}_{A\to B}\{m\}$ and $\mathsf{sec}_{A\to B}\{m\}$ stand for A sending message m to B via resp. authenticated and secure $A \to B$ channel.

Initialization for user U on input $(sid, \mathsf{S}_1, ..., \mathsf{S}_n, \mathsf{pw})$:

1. Pick $\alpha \leftarrow_\$ \mathbb{Z}_p$, set $a = (\mathsf{H}_1(\mathsf{pw}))^\alpha$, and send $((sid||i||0), a)$ to S_i for $i \in [n]$.
2. Receive $\mathsf{auth}_{\mathsf{S}\to\mathsf{U}}\{a_i, b_i(= a^{k_i})\}$ for each S_i, abort if $(a_i \neq a)$ for any i.
3. Pick $s \leftarrow_\$ \mathbb{F}$, generate shares $(s_1, ..., s_n)$ as a (t, n)-secret-sharing of s over $\mathbb{F}$. Set $\rho_i = \mathsf{H}_2(\mathsf{pw}, b_i^{1/\alpha})$ and $e_i = s_i \oplus \rho_i$ for all $i \in [n]$.
4. Set $\mathbf{e} := (e_1, ..., e_n)$, $(C||\mathsf{sk}) := \mathsf{H}_3(\mathsf{pw}, \mathbf{e}, s)$, and $\omega := (\mathbf{e}, C)$.
5. Send $\mathsf{auth}_{\mathsf{U}\to\mathsf{S}_i}\{(sid||i||1), \omega\}$ for all $i \in [n]$.
6. Pick $v', v_0 \leftarrow_\$ \mathbb{Z}_{\hat{p}}$, set $v = v_0 + v' \bmod \hat{p}$, generate shares $(v_1, ..., v_n)$ as (t, n)-sharing of v' over $\mathbb{Z}_{\hat{p}}$. Send $\mathsf{sec}_{\mathsf{U}\to\mathsf{S}_i}\{(sid||i), v_i\}$ for all $i \in [n]$.
7. Generate $(\mathsf{sks}_\mathsf{U}, \mathsf{pk}_\mathsf{U}) \leftarrow_\$ \mathsf{KeyGen}$.
8. Set $\mathsf{V} = \hat{g}^v$ and $\vec{\mathsf{V}} = (\mathsf{V}_1, ..., \mathsf{V}_n)$ where $\mathsf{V}_i = \hat{g}^{v_i}$ for every $i \in [n]$. Set $\mathsf{aec}_\mathsf{U} := \mathsf{AE.AuthEnc}_{\mathsf{sk}}(\mathsf{U}, \mathsf{V}, \vec{\mathsf{V}}, v_0, \mathsf{sks}_\mathsf{U})$, send $\mathsf{auth}_{\mathsf{U}\to\mathsf{S}_i}\{(sid, \mathsf{aec}_\mathsf{U}, \mathsf{pk}_\mathsf{U})\}$ for all $i \in [n]$. Output $(\mathsf{ptsig.verificationkey}, sid, \mathsf{V})$.

Initialization for server S on input (sid, i, U):

1. Set $k \leftarrow_\$ \mathbb{Z}_p$, on $((sid||i||0), a)$ from U, abort if $a \notin G$, else send $\mathsf{auth}_{\mathsf{S}\to\mathsf{U}}\{a, a^k\}$.
2. On message $\mathsf{auth}_{\mathsf{U}\to\mathsf{S}}\{((sid||i||1), \omega\}$ from U, store (sid, i, ω, k).
3. Receive $\mathsf{sec}_{\mathsf{U}\to\mathsf{S}}\{(sid||i), v_i\}$, abort if $v_i \notin \mathbb{Z}_{\hat{p}}$. Save (sid, v_i).
4. On $\mathsf{auth}_{\mathsf{U}\to\mathsf{S}}\{sid, \mathsf{aec}_\mathsf{U}, \mathsf{pk}_\mathsf{U}\}$ save $(sid, \mathsf{aec}_\mathsf{U}, \mathsf{pk}_\mathsf{U})$.

Signing for user U on input $(sid, ssid, \mathbf{S} = \{\mathsf{S}_1, ..., \mathsf{S}_{t+1}\}, \mathsf{pw}, \mathsf{m})$:

1. Pick $\alpha \leftarrow_\$ \mathbb{Z}_p$, set $a = (H_1(\mathsf{pw}))^\alpha$, send $((sid, ssid, j), a)$ to $\mathsf{S}_j \in \mathbf{S}$.
2. Given $((sid, ssid, j), (b_j, i_j, \omega_j))$ and $(sid, \mathsf{aec}_{\mathsf{U}j})$ from each S_j, set $\phi_j = \mathsf{H}_2(\mathsf{pw}, b_j^{1/\alpha})$ for $j \in [t+1]$. Abort if $i_{j_1} = i_{j_2}$ or $\omega_{j_1} \neq \omega_{j_2}$ for any $j_1 \neq j_2$. Otherwise set $\rho_{i_j} := \phi_j$ for all $j \in [t+1]$ and $I := \{i_j | j \in [t+1]\}$.
3. Parse any ω_j as $(\mathbf{e}, C)$ and $\mathbf{e}$ as $(e_1, ..., e_n)$. Set $s_i := e_i \oplus \rho_i$ for each $i \in I$.
4. Recover s and the shares s_i for $i \notin I$ by interpolating points (i, s_i) for $i \in I$.
5. Parse $\mathsf{H}_3(\mathsf{pw}, \mathbf{e}, s)$ as $(C'||\mathsf{sk})$. Abort if $C' \neq C$.
6. Abort if $\mathsf{aec}_{\mathsf{U}j_1} \neq \mathsf{aec}_{\mathsf{U}j_2}$ for any $j_1, j_2 \in [t+1]$, else set aec_U to any $\mathsf{aec}_{\mathsf{U}j}$. Abort if $\mathsf{AE.AuthDec}_{\mathsf{sk}}(\mathsf{aec}_\mathsf{U}) = \perp$, else parse $\mathsf{AE.AuthDec}_{\mathsf{sk}}(\mathsf{aec}_\mathsf{U})$ as $(\mathsf{U}, \mathsf{V}, \vec{\mathsf{V}}, v_0, \mathsf{sks}_\mathsf{U})$. Send $\sigma_\mathsf{U} \leftarrow \mathsf{Sign}(\mathsf{sks}_\mathsf{U}, (\mathsf{m}, ssid))$ to each $\mathsf{S} \in \mathbf{S}$.
7. On messages (j, σ_j) from each $\mathsf{S}_j \in \mathbf{S}$ if $e(g, \sigma_j) \neq e(\mathsf{V}_j, \mathsf{H}_4(\mathsf{m}))$ for any $j \in [t+1]$, output $(\mathsf{ptsig.finsign}, sid, ssid, m, \perp)$. Else compute $\sigma := \sigma_0 \cdot (\prod_{j\in S}(\sigma_i)^{\lambda_i})$, where $\sigma_0 = \mathsf{H}_4(\mathsf{m})^{v_0}$ and λ_i's are Lagrange interpolation coefficients corresponding to the set of indexes in S corresponding to $\mathbf{S}$.
8. Output $(\mathsf{ptsig.finsign}, sid, ssid, \mathsf{m}, \sigma)$.

Signing for server S on input $(sid, ssid, \mathsf{U}, \mathsf{m})$:

1. Given $((sid, ssid, j), a)$ from U, abort if $a \notin G$ or S does not hold records (sid, i, ω, k), (sid, v_i) and $(sid, \mathsf{aec}_\mathsf{U}, \mathsf{pk}_\mathsf{U})$ with the matching sid.
2. Otherwise set $b := a^k$ and send $((sid, ssid, j), (b, i, \omega))$, $(sid, \mathsf{aec}_\mathsf{U})$ to P.
3. On σ_U from U, if $\mathsf{Verify}(\mathsf{pk}_\mathsf{U}, (\mathsf{m}, ssid), \sigma_\mathsf{U}) = 1$, then send (i, σ) to U, where $\sigma := \mathsf{H}_4(m)^{v_i}$.

Fig. 25: *aptSIG-PFS-BLS*: an aptSIG-PFS protocol instantiated similarly to aptSIG-BLS of Figure 8, with all PFS-related additions marked in gray.