PESTO: Proactively Secure Distributed Single Sign-On, or How to Trust a Hacked Server

Carsten Baum[1], Tore K. Frederiksen[2], Julia Hesse[3], Anja Lehmann[4], and Avishay Yanai[5]

[1]Aarhus University, Denmark, cbaum@cs.au.dk
[2]Alexandra Institute, Denmark, tore.frederiksen@alexandra.dk
[3]IBM Research – Zurich, Switzerland, jhs@zurich.ibm.com
[4]Hasso-Plattner-Institute, University of Potsdam, Germany, anja.lehmann@hpi.de
[5]VMware Research, Israel, yanaia@vmware.com

***Abstract*—Single Sign-On (SSO) is becoming an increasingly popular authentication method for users that leverages a trusted Identity Provider (IdP) to bootstrap secure authentication tokens from a single user password. It alleviates some of the worst security issues of passwords, as users no longer need to memorize individual passwords for all service providers, and it removes the burden of these service to properly protect huge password databases. However, SSO also introduces a single point of failure. If compromised, the IdP can impersonate all users and learn their master passwords. To remedy this risk while preserving the advantages of SSO, Agrawal et al. (CCS'18) recently proposed a *distributed* realization termed PASTA (password-authenticated threshold authentication) which splits the role of the IdP across n servers. While PASTA is a great step forward and guarantees security as long as not all servers are corrupted, it uses a rather inflexible corruption model: servers cannot be corrupted adaptively and — even worse — cannot recover from corruption. The latter is known as *proactive security* and allows servers to re-share their keys, thereby rendering all previously compromised information useless.**

In this work, we improve upon the work of PASTA and propose a distributed SSO protocol with proactive and adaptive security (PESTO), guaranteeing security as long as not all servers are compromised at the *same* time. We prove our scheme secure in the UC framework which is known to provide the best security guarantees for password-based primitives. The core of our protocol are two new primitives we introduce: partially-oblivious distributed PRFs and a class of distributed signature schemes. Both allow for non-interactive refreshs of the secret key material and tolerate adaptive corruptions. We give secure instantiations based on the gap one-more BDH and RSA assumption respectively, leading to a highly efficient 2-round PESTO protocol. We also present an implementation and benchmark of our scheme in Java, realizing OAuth-compatible bearer tokens for SSO, demonstrating the viability of our approach.

1. Introduction

Until today, passwords are still the primary means of user-authentication towards online services. While stronger approaches, such as two-factor or key-based authentication have been considered and are partially deployed, passwords are still ubiquitous due to their usability advantages: users do not have to securely manage cryptographic keys or hardware tokens, but only need to type-in a human-memorizable phrase.

Unfortunately, password databases are a prime target of potential attackers and even large companies seem to struggle to protect this information properly [4], [5], [7]. In fact, server compromise is currently the biggest threat to password security and has lead to a compromise of over 1 billion passwords to date [40]. In combination with user tendencies to re-use passwords at different services, such data breaches can have a devastating impact for those affected by the breach.

Using single sign-on services (SSO) alleviates some of the worst security issues with passwords, and also enhances the usability aspects for the end user. The SSO approach centralizes the authentication task via a trusted Identity Provider (IdP). The user only has to login towards the IdP which then generates a short-term cryptographic access token such as a JWT [38] or SAML assertion [3], allowing the user to authenticate herself towards other services. The main advantage of SSO is that it avoids tedious password handling with every single service provider, and does not require users to trust each of these providers to keep their passwords safe. Hence such approaches have been standardized though frameworks like OAuth [29], and more concretely through protocols like OpenID Connect (OIDC) [44] based on a JWT [38].

On the negative side, SSO introduces a *single point of failure*: an attacker that compromises an IdP can take over *all* of the users' registered accounts and learns their master passwords. For the latter, even protective measures such as storing salted password hashes only have rather limited impact, as any such information is still vulnerable to offline guessing attacks due to the low entropy of human-memorizable passwords [6].

1.1. Distributed Single Sign-On

The problem of offline dictionary attacks is inherent in all solutions where a *single* server can test the correctness of passwords: as soon as the server gets compromised, the attacker can exploit it's capability to mount massive guessing attacks against the password hashes. The natural solution to remedy such attacks is to distribute the task of password verification over n servers. By carefully splitting the verification process, security can be guaranteed as long as not all (or a threshold) of these servers got hacked. This approach has been used to salvage security for a number of password-based primitives, such as threshold

password-authenticated key exchange (TPAKE) [30], [39], [43], [47], threshold password-authenticated secret sharing (TPASS) [10], [15], [13], [31], as well as plain distributed password verification [22], [14], [21].

The newest addition to this line of research is password-based threshold authentication (PASTA) [8] which allows to split the role of the IdP for SSO. The PASTA protocol combines distributed password verification with a distributed signature scheme, and achieves many of the properties needed for distributed SSO: If t out of n servers remain honest, the users' master passwords are safe against offline attacks and impersonation attacks via forged signatures on access tokens are prevented too.

Proactive Security. While PASTA avoids the single point of failure of standard SSO, it does not provide significantly stronger security though due to the lack of a *recovery* strategy — also known as proactive security. A crucial feature in distributed protocols is to allow servers to refresh their keys so that they can securely re-initialize after a compromise. Without such a recovery mechanism, it is only a matter of time until all servers have been hacked and their data can be combined to an offline-testable password table. For a distributed SSO, one would probably not want to deploy much more than 2 or 3 servers, i.e., in a static setting the costs and time of an attack would only double or triple.

UC Security. Another more subtle challenge is that of an appropriate security model: Agrawal et al. [8] define the desired security properties in the form of game-based notions. It is well-known that for password-based primitives this cannot properly capture the way users (mis)handle passwords [19]. When formulated through games, users are assumed to choose their passwords at random from known and independent distributions. The adversary also only gains access to "perfect" users which always use the correct password. In reality, however, users share, re-use, and leak information related to their passwords, and often make typos when using them which leads to running the protocol on incorrect yet related passwords. In the Universal Composability (UC) model, this is modeled naturally as the environment provides the passwords. That is, a UC security notion guarantees the desired properties without making any assumptions regarding the passwords' distributions, dependencies, or leakages.

Key Management. When aiming for a distributed system, the key and secret state handling by the individual servers should be as simple and concise as possible, as any complexity increases the risks of (implementation) mistakes and the attack surface for adversaries. Translated to the task of SSO, each server should ideally have a single secret key (per time epoch) only, whereas the dynamic user-specific information should be non-sensitive. This way, the secret key can enjoy stronger protection, e.g., through hardware, which is much harder to realize for the large and dynamic user database. Unfortunately, PASTA does not satisfy this design criteria, as the servers obtain dedicated secret key material from each user which needs to be protected accordingly. In fact, this choice is also one of the reasons the PASTA protocol is not amendable to proactive security measures.

Online Attacks. Finally, one reason to move to a distributed setting for password-based protocols such as SSO, is to rely on honest servers for detecting and preventing *online* guessing attacks. That is, an adversary targets a dedicated account and tries to guess the users password to gain access to her account. Honest servers can block or throttle login attempts for a specific account when they notice suspicious behaviour and thereby significantly limit the adversary's amount of guesses — roughly mimicking the hardware protection our short 4-digit PIN numbers for ATM cards enjoy. In the PASTA protocol the servers do not learn whether a password attempt was successful though, which restricts their capabilities to detect such online guessing attacks. For instance, an adversary can camouflage its attack by spreading many password guesses across a day and gain several hundreds or thousands of undetected guesses, whereas a system in which servers learn the outcome of the password verification would have stopped the attack after a few failed attempts already.

1.2. Our Contributions

In this work we propose a distributed SSO protocol — PESTO — which achieves all the aforementioned properties, while still being compatible with existing authentication standards such as OAuth [29] and OIDC [44]. Similar to PASTA, we model SSO through a password-authentication distributed signature functionality. Therein, the distributed servers sign messages (uid, m), i.e., they bind a user-provided message m to the verified user name uid. They only do so after they have verified that the user knows the password which has been setup for uid. For SSO applications, the message m will contain the unique ID of the targeted service provider and a nonce that is usually specified by the provider. We compare the overall features of PESTO with PASTA, lined up against a standard, single-server, JWT-based scheme [38] in Fig. 1.

Proactive and UC Security for Distributed SSO. We define security of PESTO in form of an ideal functionality in the UC framework, avoiding any unrealistic assumptions inherent in game-based definitions. We admit a fine grained corruption model: first, servers can be corrupted in an *adaptive* manner, i.e., the adversary can take control of any initially honest party at any time. This improves upon [8] which allows only static corruptions. Second, we allow both *transient* and *permanent* corruptions. Transiently corrupted parties can recover from an attack by going through a dedicated *refresh* procedure, whereas permanently corrupted ones cannot recover. Security is guaranteed as long as not all servers are corrupt at the *same* time.

Our PESTO Protocol. We propose a protocol that securely satisfies this strong notion. While our PESTO protocol uses similar building blocks as PASTA, it significantly differs in *how* they are used.

Roughly, we start with a *distributed partially-oblivious PRF* (dpOPRF) to let users deterministically derive a signing key pair from their username and password as $(upk, usk) \leftarrow \mathsf{dpOPRF}(K, (uid, pw))$. Thereby, the key K is split among n servers and the evaluation is done partially-blind: the servers learn uid but not pw. Disclosing the uid allows us to use a single dpOPRF key per server, yet allow user-specific rate limiting.

	Security							Efficiency		
	Threshold	Serv. Corruption	Security Model	Proactive Security	Offline Attack Resist.	Online Attack Resist.	Secret Keys per Server	Rounds	Exponentiation / Pairings per Server	User
Std. JWT	$(1,1)$	-	-	✘	✘	✓	$\mathbf{O(1)}$	**1**	**1**ex	**0**
PASTA	$(\mathbf{t},\mathbf{n})$	static	Game	✘	✓	(✓)	$O(\#\text{Users})$	**1**	2ex	2ex
PESTO	(n,n)	**adaptive**	**UC**	✓	✓	✓	$O(n)$	2	4ex + 1p	5ex

Figure 1: Comparison between PESTO (Our Work), PASTA [8], and a standard Json Web Token (JWT) used for SSO. For efficiency we count the most expensive operations, i.e., exponentiations and pairings per party.

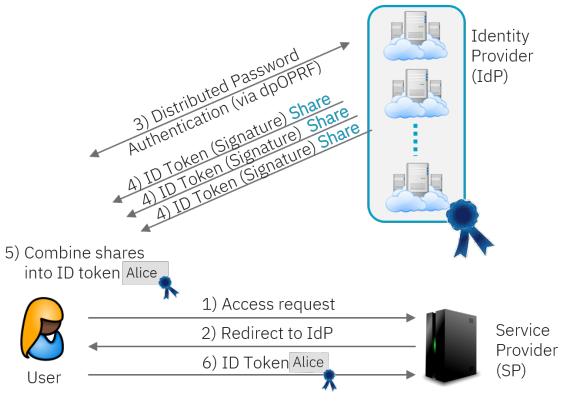

Figure 2: High-level overview of SSO with PESTO

When creating her account, the user sends the derived upk to all servers which store (uid, upk) along with a signature on these values that is generated in a distributed way. This jointly computed signature is crucial to ensure that an adversary cannot gradually plant malicious public keys for honest user accounts, when he transiently corrupts individual servers. In contrast to PASTA, all user-specific information stored by the servers is not security sensitive.

When asking for a SSO token on a message m for username uid, the user repeats the dpOPRF part for a password attempt pw' to re-construct her key pair (upk, usk), and uses usk to sign a fresh nonce towards all servers. The servers verify the signature against the stored upk to determine whether the user provided the correct password. If so, the servers jointly sign (uid, m) using a proactively-secure distributed signature scheme DSIG.

Proactively Secure dpOPRF *and* DSIG. Our protocol is built from generic building blocks dpOPRF and DSIG we introduce and which constitute contributions of independent interest. For both we first present security models that capture the strong type of adaptive and transient corruptions, and then propose secure instantiations. The dpOPRF is based on the classical 2Hash-Diffie-Hellman OPRF construction [31] and DSIG is realized for a distributed version of RSA signatures. The challenge in making these schemes proactively *and* adaptively secure is to avoid a loss in security by having to guess corrupted servers upfront. While this loss can be tolerated once, it would blow exponentially when moving into a proactive setting where such a guess needs to be done at every epoch.

We avoid this by ensuring that none of the values sent by servers commits them to their secret key shares. To do this efficiently, we rely on a trusted setup that generates pairwise seeds for all servers from which they derive consistent blinding values to mask all outgoing messages. In the security proof, this allows us to choose a server's keys only at the moment that it is corrupted, without having to guess anything upfront. Finally, to allow servers to recover from transient corruptions we assume that each server has access to a special backup tape. Such a backup tape is necessary to allow servers to re-boot with a clean and uncorrupted state. We also leverage it to run the refresh in a *non-interactive* (yet synchronized) manner.

The backup tape is only needed during the refresh procedure but does not have to (and *should* not) be online during normal operations. Transiently corrupted parties leak their full state to the adversary, but not the content of their backup tape, and recover with the next refresh. Our model also supports more damaging attacks which additionally leak the backup tape to the adversary. Such parties are considered permanently corrupted and cannot become uncorrupted again.

Implementation & Efficiency. Using the aforementioned DDH-based dpOPRF and RSA-based DSIG instantiations, we obtain an efficient 2-round PESTO protocol which we test in a proof-of-concept implementation in Java. We use our protocol to construct *standard* RSA-signed JWTs, i.e., they can be verified by any service provider that accepts such tokens in an OAuth or OIDC authentication flow. Our implementation utilizes REST and TLS in order to give a realistic benchmark on the efficiency to be expected in a real-world deployment. Overall, our protocol requires 4 exp. and 1 pairing operation per server and authentication request, which adds only minimal overhead compared with PASTA yet provides significantly stronger security. Concretely, a complete sign-in operation is only 124 ms even when all entities are located in distinct countries.

Distributed vs. Threshold Setting. While our scheme provides stronger security than PASTA (proactive and adaptive vs. static security), it does pay a price in terms of robustness: PESTO is fully distributed, i.e., requires n-out-of-n servers to participate, whereas PASTA offers a threshold (t, n) solution. Thus, our solution is more vulnerable to Denial-of-Service attacks as a single offline server will result in a shutdown of the entire protocol.

The (n, n) setting is the result of our solution coping with *adaptive* corruptions: to avoid that any protocol message commits to a server's secret share, we let each server add blinding values to their messages which cancel out when shares of *all* servers are combined. This approach seems to inherently conflict with a threshold solution, and thus it is an interesting open problem how an adaptively secure *and* threshold version can be achieved.

A straightforward "solution" to just get a threshold version would be to downgrade the security model from adaptive to static corruptions (as in PASTA). This makes the blinding values, as the main obstacle, obsolete. The

rest of the protocol then needs to use threshold building blocks that are secure against this weaker attacker. Currently, PESTO exploits the trusted setup and (n, n) approach to also bootstrap an efficient *non-interactive* key refresh procedure, whereas a (t, n) solution most likely needs to revert to an adaptive refresh protocol.

1.3. Related Work

Apart from PASTA [8] which we discussed already, our work takes inspiration from a number of previous works we detail here. We also refer to [8] for a comprehensive overview of concepts related to distributed SSO.

Distributed Password Verification. The use of blinding values and offline backup tapes to recover from corruptions is inspired by the distributed password verification protocol by Camenisch et al. [14], that also aimed at proactive security in the UC framework. The targeted problems are different though, as their protocol considers a single login server that protects it's password database by relying on n back-end servers to re-compute the password "hashes". The back-end servers in [14] are also fully oblivious: they neither learn the uid's for which they verify a password nor the result of the verification. Both significantly limit the servers' capabilities to prevent online attacks.

Furthermore, we make the proactive security properties accessible in a more modular and re-usable way, as we already integrate them in our generic building blocks: distributed partially-oblivious OPRFs and distributed signatures. Both can be used in a plug-and-play fashion to enhance security of protocols where their distributed yet non-proactive versions have been used.

Partially-Oblivious PRF with Key Rotation. The Pythia protocol [21] introduced the concept of *partially-oblivious PRFs* which they used to build a password hardening service with fine-grained rate limiting. However, Pythia only uses a single backend PRF server. As a single key cannot be re-shared, proactive security is achieved through key rotation instead, i.e., the PRF server switches to a fresh key and provides a short update token that allows to update all previous PRF outputs accordingly.

Our work extends these partially-oblivious PRF to a fully distributed setting, which naturally allows for a more elegant refresh protocol without the need to update any previously computed PRF outputs. Further, Pythia's approach requires direct access to the algebraic structure of the OPRF which prevented them of using the outer hash of the 2Hash-DDH construction and lead to weaker overall security of the OPRF. Recently, Jarecki et al. [34] presented a distributed partially-oblivious PRF that achieves partial blindness in a different and more efficient way. However, they did not propose a security notion and their construction inherently does not allow to re-share the key.

Password-Authenticated Secret Sharing. Finally, the first part of our protocol essentially can be seen as a proactively-secure version of password-authenticated secret sharing (TPASS) [10], [31], [32], [33] and might provoke the question why we need the second protocol step and distributed SSO at all: the user could simply use the reconstructed signing key pair (usk, upk) to directly authenticate to the service providers in a secure, password-less manner. However, using TPASS for such user authentication would lead to a full loss of security when the user's device gets compromised during key reconstruction. As the adversary then learns the user's master key it can impersonate the user towards all service providers. In our PESTO approach, a compromised device or password alone is not sufficient to gain access to the user's accounts. All servers need to sign every single access token, which provides a second line of defense: They can refuse to complete these requests when they notice suspicious access patterns, and the user can block her account with the online servers when she suspects a breach of her device — thereby rendering the compromised information useless.

2. Preliminaries

We denote with $[k]$ the set $\{1, \ldots, k\}$ and say that a non-negative function $f(n)$ is negligible in n, or $f(n) \leq \epsilon(n)$, if for every polynomial $p(X)$ there exists a bound t such that $\forall t' \geq t : \quad f(t') \leq \frac{1}{|p(t')|}$. We denote the computational security parameter throughout this work as τ and the statistical security parameter as s.

2.1. Bilinear Groups

Definition 1 (Asymmetric Pairing)**.** *Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be cyclic groups of order p with generators g_1, g_2, g_T, respectively. Furthermore, let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable non-degenerate function such that $\forall a, b \in \mathbb{Z}_p : \quad e(g_1^a, g_2^b) = g_T^{ab}$. Then e is called an* asymmetric pairing. *$\mathbb{G} = (p, g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ is called an* asymmetric bilinear group setting*, or* bilinear group *for short.*

We formulate two assumptions on $\mathbb{G}$. The first assumption is the standard DDH-generalization groups with asymmetric pairings and has been used e.g. in [20].

Definition 2 (Bilinear DDH Assumption)**.** *Let $\tau \in \mathbb{N}$ be a security parameter and $\mathbb{G} = (p, g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ be a bilinear group where $\log(p) = poly(\tau)$. We say that the* Bilinear Decisional Diffie-Hellman (BDDH) *assumption holds for $\mathbb{G}$ if for any PPT (in τ) algorithm $\mathcal{A}$*

$$\left| \begin{array}{c} \Pr[\mathcal{A}(\mathbb{G}, g_r^a, g_s^b, g_t^c, R) = 1] - \\ \Pr[\mathcal{A}(\mathbb{G}, g_r^a, g_s^b, g_t^c, e(g_1, g_2)^{abc}) = 1] \end{array} \right| \leq \epsilon(\tau)$$

where a, b, c are uniformly random in $\mathbb{Z}_p$, $r, s, t \in \{1, 2\}$ and R is uniformly random in $\mathbb{G}_T$.

For the second assumption we first define the following experiment with an algorithm $\mathcal{A}$:

Experiment $\mathsf{Exp}^{\mathbb{G}}_{\mathcal{A},\mathsf{Gapom\text{-}BDH}}(\tau)$:
$k \leftarrow_{\mathrm{R}} \mathbb{Z}_p$, $q_C \leftarrow 0$, $X_1 \leftarrow \emptyset$, $X_2 \leftarrow \emptyset$.
$\{(x_i, y_i, z_i)\}_{i \in [\ell]} \leftarrow \mathcal{A}^{\mathcal{O}_{\mathbb{G}\text{-}1}, \mathcal{O}_{\mathbb{G}\text{-}2}, \mathcal{O}_{\mathsf{D\text{-}help}}, \mathcal{O}_{\mathsf{C\text{-}help}}}(\mathbb{G}, g_2^k)$
return 0 if
$0 \leq \ell - 1 < q_C$ or
$\exists i \in [\ell] : (x_i \notin X_1 \vee y_i \notin X_2)$ or
$\exists i, j \in [\ell], i < j : (x_i = x_j \wedge y_i = y_j)$
return 1 if $\forall i \in [\ell] : e(x_i, y_i)^k = z_i$ and 0 otherwise.

where the experiment uses the following oracles:

$\underline{\mathcal{O}_{\mathbb{G}\text{-}\mathsf{r}}()}$
return $\perp$ if $r \notin \{1, 2\}$
$x \leftarrow_{\mathrm{R}} \mathbb{G}_r$
$X_r \leftarrow X_r \cup \{x\}$
return x

$\underline{\mathcal{O}_{\mathsf{C\text{-}help}}(m)}$
return $\perp$ if $m \notin \mathbb{G}_T$
$q_C \leftarrow q_C + 1$
return m^k

$\mathcal{O}_{\mathsf{D\text{-}help}}(m, w, m', w')$
return $\perp$ if either m, w, m', w' not in $\mathbb{G}_T$
return 1 if $\log_m(w) = \log_{m'}(w')$ and else 0

To win, $\mathcal{A}$ needs to find pairs $(x, y, e(x, y)^k)$ without querying $e(x, y)$ to $\mathcal{O}_{\mathsf{C\text{-}help}}$ and where $\mathcal{A}$ could not re-randomize previous such pairs as it does not know the discrete logarithm of any x, y (enforced by sampling them at random using $\mathcal{O}_{\mathbb{G}\text{-}\mathsf{r}}$). $\mathcal{A}$ is equipped with a DDH oracle $\mathcal{O}_{\mathsf{D\text{-}help}}$ in the group $\mathbb{G}_T$. The game Gapom-BDH follows the definition in [21].

Definition 3 (Gap One-More BDH Assumption)**.** *Let $\tau \in \mathbb{N}$ be a security parameter and $\mathbb{G} = (p, g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ be a bilinear group with $\log(p) = poly(\tau)$, then we then say that the Gap One-More Bilinear Diffie-Hellman (Gapom-BDH) assumption holds for $\mathbb{G}$ if for all PPT adversaries $\mathcal{A}$ there is a negligible function $\epsilon(\cdot)$ such that $\Pr[\mathsf{Exp}^{\mathbb{G}}_{\mathcal{A},\mathsf{Gapom\text{-}BDH}}(\tau) = 1] \leq \epsilon(\tau)$.*

2.2. Digital Signatures

We use a signature scheme SIG, consisting of algorithms $\mathsf{SIG.Setup}(1^\tau) \to_{\mathrm{R}} pp, \mathsf{SIG.KGen}(pp; r) \to_{\mathrm{R}} (sk, pk), \mathsf{SIG.Sign}(sk, m) \to_{\mathrm{R}} \sigma$ and $\mathsf{SIG.Vf}(pk, m, \sigma) \to \{0, 1\}$. Note that we sometimes make the randomness r used in key generation explicit. When used on the same input and randomness, SIG.KGen behaves deterministically.

It is required that except with negligible probability over τ, for a key pair (pk, sk), where sk output by $\mathsf{SIG.SKGgen}(pp)$ and pk output by $\mathsf{SIG.PKGgen}(pp, sk)$, it holds that $\mathsf{SIG.Vf}(pk, m, \mathsf{SIG.Sign}(sk, m)) = 1$ for every message $m \in \{0, 1\}^*$ (for implicit pp generated by SIG.Setup).

Security of a signature scheme is defined through the standard unforgeability experiment $\mathsf{Exp}^{\mathsf{SIG}}_{\mathcal{A},\mathsf{forge}}(\tau)$.

Experiment $\mathsf{Exp}^{\mathsf{SIG}}_{\mathcal{A},\mathsf{forge}}(\tau)$:
$pp \leftarrow_{\mathrm{R}} \mathsf{SIG.Setup}(1^\tau)$, $(sk, pk) \leftarrow_{\mathrm{R}} \mathsf{SIG.KGen}(pp)$
$(m^*, \sigma^*) \leftarrow_{\mathrm{R}} \mathcal{A}^{\mathcal{O}_{\mathsf{Sign}}}(pk)$
where $\mathcal{O}_{\mathsf{Sign}}(m)$:
add m to $\mathcal{Q}$, return $\mathsf{SIG.Sign}(sk, m)$
return 1 if $\mathsf{SIG.Vf}(pk, m^*, \sigma^*) = 1$ and $m^* \notin \mathcal{Q}$

Definition 4. *A signature scheme* SIG *is existentially unforgeable under adaptive chosen-message attacks, if for all PPT adversaries $\mathcal{A}$, there is a negligible function $\epsilon(\cdot)$ such that $\Pr[\mathsf{Exp}^{\mathsf{SIG}}_{\mathcal{A},\mathsf{forge}}(\tau) = 1] \leq \epsilon(\tau)$.*

3. Proactively Secure Distributed Signatures

In this section we formalize the notion of a proactively secure distributed signature and present an instantiation of it based on RSA signatures. A distributed signature scheme DSIG is a tuple of polynomial-time algorithms $(\mathsf{Setup}, \mathsf{KGen}, \mathsf{Sign}, \mathsf{Comb}, \mathsf{Vf}, \mathsf{Refresh})$ such that:

$\mathsf{DSIG.Setup}(1^\tau, 1^s) \to_{\mathrm{R}} pp$**:** Given a computational security parameter 1^τ and a statistical security parameter 1^s, outputs public parameters pp.

$\mathsf{DSIG.KGen}(pp, n) \to_{\mathrm{R}} (pk, (sk_i)_{i\in[n]}, (bk_i)_{i\in[n]})$**:** Given public parameters pp and the number of parties n, output a public key pk, secret signing key shares $sk_1, \ldots, sk_n$ and backup key shares $bk_1, \ldots, bk_n$.

$\mathsf{DSIG.Sign}(sk_i, m, \ell) \to \sigma_i$**:** Given a key share sk_i, message m, and query label ℓ, output a signature share σ_i.

$\mathsf{DSIG.Comb}(\sigma_1, \ldots, \sigma_n) \to \sigma$**:** Given signature shares $\sigma_1, \ldots, \sigma_n$, output signature σ or $\perp$.

$\mathsf{DSIG.Vf}(pk, m, \sigma) \to b$**:** Given a public key pk, a message m and a signature σ, output a bit b such that $b = 1$ means valid and $b = 0$ means invalid.

$\mathsf{DSIG.Refresh}(bk_i) \to_{\mathrm{R}} (sk'_i, bk'_i)$**:** Given a backup key bk_i, output new keys sk'_i and bk'_i.

We assume that the server's index i can be efficiently recovered from sk_i and bk_i, and that the public parameters pp are available to all algorithms.

The Sign algorithm generates a signature share on a message m and an additional parameter ℓ, while verification only requires m. The parameter ℓ is used to bind signature shares to a query label ℓ. The label ensures that signature share $\sigma_1, \ldots, \sigma_n$ can only be combined if *all* where received for the same ℓ, i.e., even shares for the same message cannot be combined if they were derived for $\ell \neq \ell'$. This will prevent so-called "mix-and-match" attacks.

Correctness. We define *correctness* of DSIG in terms of providing the required functionality of a signature when used as intended. Let $(sk_i^{\mathsf{ep}}, bk_i^{\mathsf{ep}}) \leftarrow \mathsf{DSIG.Refresh}^{\mathsf{ep}}(bk_i)$ result from applying $\mathsf{DSIG.Refresh}(\ldots(\mathsf{DSIG.Refresh}(bk_i))$ for ep times where $bk_i^0 = bk_i$. DSIG is *correct* if for every $\ell \in \{0, 1\}^*$, $\mathsf{ep} \in \mathbb{N}^+$ and $n > 1$, it follows that $\mathsf{Vf}(pk, m, \sigma) = 1$ with overwhelming probability over the randomness of Setup, KGen, Refresh; where $\sigma = \mathsf{Comb}(\sigma_1, \ldots, \sigma_n)$, $pp \leftarrow \mathsf{Setup}(1^\tau)$, $\sigma_i \leftarrow \mathsf{Sign}(sk_i^{\mathsf{ep}}, m, \ell)$ and $(pk, (sk_i)_{i\in[n]}, (bk_i)_{i\in[n]}) \leftarrow \mathsf{KGen}(pp, n)$.

3.1. Proactive Security Model for DSIG

We now present the desired security definitions for DSIG capturing proactive security against adaptive corruptions, and allowing the adversary to corrupt servers in both a transient and permanent way.

We formalize three security properties: *proactive unforgeability*, which can be seen as the strengthening of standard unforgeability to the proactive setting, *share simulatability* and *signature indistinguishability*. The latter two are needed to give us the strong simulation-based security needed prove UC security for PESTO. The security games in the remainder of this section require us to explicitly keep track of the different epochs using an extra variable ep. This is done to express the adversaries knowledge: for each epoch, the adversary may obtain a different subset of keys sk_i^{ep} and different sets of signatures shares. Our security properties must hold as long as the adversary does not corrupt all servers at the same epoch. The security experiment thus has to keep track of which information the adversary gets at which time.

Proactive Unforgeability. This notion generalizes standard (strong) unforgeability to the proactive, distributed setting. It is defined with oracles given in Fig 3 which allow the adversary $\mathcal{A}$ to obtain signature and key shares (either sk_i or (sk_i, bk_i) depending on the corruption mode) at will. The adversary can also trigger honest and transiently corrupted servers to refresh their keys, thereby moving the epoch forward and turning all transiently corrupted parties

into honest ones. He wins if he successfully forges a valid signature without either corrupting all parties or requesting signature shares for the same message m and query label ℓ for all uncorrupted parties in one epoch.

Definition 5 (Proactive Unforgeability)**.** *A distributed signature scheme* DSIG *is* proactively unforgeable *if for all PPT adversaries $\mathcal{A}$, there is a negligible function $\epsilon(\cdot)$ such that* $\Pr[\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A},\mathsf{forge}}(\tau,s)=1] \leq \epsilon(\tau,s)$.

Experiment $\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A},\mathsf{forge}}(\tau,s)$:
$pp \leftarrow_{\mathrm{R}} \mathsf{DSIG.Setup}(1^\tau,1^s)$, set $\mathsf{ep}\leftarrow 0$, $\mathcal{C}_{t,\mathsf{ep}},\mathcal{C}_{p,\mathsf{ep}}\leftarrow\emptyset$
$(pk,(sk_i)_{i\in[n]},(bk_i)_{i\in[n]}) \leftarrow_{\mathrm{R}} \mathsf{DSIG.KGen}(pp,n)$
$(m^*,\sigma^*) \leftarrow_{\mathrm{R}} \mathcal{A}^{\mathcal{O}_{\mathsf{sign}},\mathcal{O}_{\mathsf{corrupt}},\mathcal{O}_{\mathsf{refresh}}}(pk)$
return 0 if $\mathsf{DSIG.Vf}(pk,m^*,\sigma^*)=0$ or
$\exists \mathsf{ep}$ where $|\mathcal{C}_{t,\mathsf{ep}}\cup\mathcal{C}_{p,\mathsf{ep}}|=n$ or
$\exists \mathsf{ep},\ell$ s.t. $\mathsf{DSIG.Comb}(\sigma_1,,\ldots,\sigma_n)=\sigma^*$
where $\forall i\notin\mathcal{C}_{t,\mathsf{ep}}\cup\mathcal{C}_{p,\mathsf{ep}}\ \ (\mathsf{ep},i,m^*,\ell,\sigma_i)\in\mathcal{Q}$
and $\forall i\in\mathcal{C}_{t,\mathsf{ep}}\cup\mathcal{C}_{p,\mathsf{ep}}\ \ \sigma_i\leftarrow\mathsf{DSIG.Sign}(i,sk_i^{\mathsf{ep}},m^*,\ell)$
else **return** 1

Share Simulatability. The second property is needed to use DSIG as a building block in a UC-secure protocol (aiming at proactive security and allowing adaptive corruptions), as therein security is proven by constructing a simulator that mimics the ideal functionality in a way that is not noticeable to the environment. The notion of share simulatability expresses exactly that and ensures that we can simulate signature and key shares of honest parties in a security game without knowledge of the actual signing key. The definition follows the simulation paradigm and either gives the adversary access to real signature and key oracles (the same ones we used in our unforgeability definition above) or simulated ones. Share simulatability is satisfied when the adversary cannot distinguish between both worlds.

Definition 6 (Share Simulatability)**.** *A distributed signature scheme* DSIG *is* share simulatable *if for all PPT adversaries $\mathcal{A}$, there exists a stateful PPT algorithm* SIM *with input pk and a negligible function $\epsilon(\cdot)$ such that* $\left|\Pr[\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A},\mathsf{sim},1}(\tau,s)=1]-\Pr[\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A},\mathsf{sim},0}(\tau,s)=1]\right|\leq\epsilon(\tau,s)$.

Experiment $\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A},\mathsf{sim},b}(\tau,s)$:
$pp \leftarrow_{\mathrm{R}} \mathsf{DSIG.Setup}(1^\tau,1^s)$
$(pk,(sk_i)_{i\in[n]},(bk_i)_{i\in[n]}) \leftarrow_{\mathrm{R}} \mathsf{DSIG.KGen}(pp,n)$
set $\mathsf{ep}\leftarrow 0$, $\mathcal{C}_{t,\mathsf{ep}}\leftarrow\emptyset$, $\mathcal{C}_{p,\mathsf{ep}}\leftarrow\emptyset$
$b^* \leftarrow_{\mathrm{R}} \mathcal{A}^{\mathcal{O}_{\mathsf{sign}},\mathcal{O}_{\mathsf{corrupt}},\mathcal{O}_{\mathsf{refresh}}}(pk)$
if $b=0$: left oracles as in Fig 3
if $b=1$: right oracles as in Fig 3 (for $\mathsf{SIM}(\mathsf{init},pk)$)
return 1 if $b^*=b$ and $\forall\mathsf{ep}$: $|\mathcal{C}_{t,\mathsf{ep}}\cup\mathcal{C}_{p,\mathsf{ep}}|<n$

Roughly, $\mathcal{A}$ interacts with DSIG when $b=0$, while for $b=1$ the signature and key shares are simulated by SIM. The simulator thereby has no information about the underlying secret key shares. We have to be careful to be prevent trivial wins though: Eventually, signature shares can be completed into a full signature by $\mathcal{A}$ and verified against the public key. Thus, we have to keep track when SIM is about to reveal enough information (either via signatures or key shares) that allow $\mathcal{A}$ to complete a signature. If so, we generate the full signature $\mathcal{A}$ is about to learn via $\sigma\leftarrow\mathsf{Comb}(\mathsf{Sign}(sk_1,m,\ell),\ldots,\mathsf{Sign}(sk_n,m,\ell))$ and give σ as advice to SIM.

There are two scenarios where this can occur: in a signature or in a corruption query. For the former, we use a function $\mathsf{compSig}_{\mathcal{Q},\mathcal{C}_t,\mathcal{C}_p}(\mathsf{ep},i,m,\ell)\rightarrow\{\sigma,\perp\}$ that tests whether answering a signing query for (i,m,ℓ) would, together with the information already revealed to $\mathcal{A}$ (contained in $\mathcal{Q},\mathcal{C}_t,\mathcal{C}_p,$), allow him to complete the signature. If so, it returns the *full* signature σ for m.

For corruption queries, we use a similar function $\mathsf{compList}_{\mathcal{Q},\mathcal{C}_t,\mathcal{C}_p}(e,i)\rightarrow\mathcal{L}$ which checks which previously answered sign queries (ep,j,m,ℓ) the adversary can complete after corrupting party i in epoch ep. For each such query, the function computes the corresponding full signature and returns a set $\mathcal{L}$ of tuples $(\mathsf{ep},j,m,\ell,\sigma)$.

For sake of brevity, we also sometimes omit the subindex $\mathcal{Q},\mathcal{C}_t,\mathcal{C}_p$ from the name of these functions. The functions $\mathsf{compSig}_{\mathcal{Q},\mathcal{C}_t,\mathcal{C}_p}(\mathsf{ep},i,m,\ell)$ and $\mathsf{compList}_{\mathcal{Q},\mathcal{C}_t,\mathcal{C}_p}(\mathsf{ep},i)$ are defined as follows:

$\mathsf{compSig}_{\mathcal{Q},\mathcal{C}_t,\mathcal{C}_p}(\mathsf{ep},i,m,\ell)$:
if $\forall j\notin\mathcal{C}_{t,\mathsf{ep}}\cup\mathcal{C}_{p,\mathsf{ep}}\cup\{i\}\ \ \exists(\mathsf{ep},j,m,\ell)\in\mathcal{Q}$
get $\sigma_i\leftarrow(\mathsf{DSIG.Sign}(sk_i^{\mathsf{ep}},m,\ell)$ for $i\in[n]$
return $\sigma\leftarrow\mathsf{DSIG.Comb}(\sigma_1,\ldots,\sigma_n)$
else **return** $\perp$

$\mathsf{compList}_{\mathcal{Q},\mathcal{C}_t,\mathcal{C}_p}(\mathsf{ep},i)$:
set $\mathcal{L}\leftarrow\emptyset$
for all $(\mathsf{ep},j,m,\ell)\in\mathcal{Q}$:
if $\forall j^*\notin\mathcal{C}_{t,\mathsf{ep}}\cup\mathcal{C}_{p,\mathsf{ep}}\cup\{i\}\ \ \exists(\mathsf{ep},j^*,m,\ell)\in\mathcal{Q}$
get $\sigma_i\leftarrow(\mathsf{DSIG.Sign}(sk_i^{\mathsf{ep}},m,\ell)$ for $i\in[n]$
set $\sigma\leftarrow\mathsf{DSIG.Comb}(\sigma_1,\ldots,\sigma_n)$
and set $\mathcal{L}\leftarrow\mathcal{L}\cup(\mathsf{ep},j,m,\ell,\sigma)$
return $\mathcal{L}$

Signature Indistinguishability. Our last definition also covers some simulation-based aspects and guarantees that signatures generated in a distributed way cannot be distinguished from directly computed ones. The notion guarantees that algorithms $\mathsf{Sign}^*,\mathsf{Comb}^*$ exist, such that $\mathsf{Sign}^*(\mathsf{Comb}^*(sk_1,\ldots,sk_n),m)\ \approx\ \mathsf{Comb}(\mathsf{Sign}(sk_1,m,\ell),\ldots\mathsf{Sign}(sk_n,m,\ell))$.

Signature indistinguishability can again be seen as an artifact of using DSIG in a UC-protocol: therein the functionality produces signatures generated by Sign^*, while the simulator has to mimic our distributed signature scheme where each party runs Sign locally.

Definition 7 (Signature Indistinguishability)**.** *A distributed signature scheme* DSIG *is signature indistinguishable if for all PPT adversaries $\mathcal{A}$, there is a negligible function $\epsilon(\cdot)$ such that* $\left|\Pr[\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A},\mathsf{ind},1}(\tau,s)=1]-\Pr[\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A},\mathsf{ind},0}(\tau,s)=1]\right|\leq\epsilon(\tau,s)$.

Experiment $\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A},\mathsf{ind},b}(\tau,s)$:
$pp \leftarrow_{\mathrm{R}} \mathsf{DSIG.Setup}(1^\tau,1^s)$
$(pk,(sk_i)_{i\in[n]},(bk_i)_{i\in[n]}) \leftarrow_{\mathrm{R}} \mathsf{DSIG.KGen}(pp,n)$
$sk\leftarrow\mathsf{DSIG.Comb}^*(pk,sk_1,\ldots,sk_n)$
$b^* \leftarrow_{\mathrm{R}} \mathcal{A}^{\mathcal{O}_{\mathsf{chall}_b}(pk,(sk_1,\ldots,sk_n),(bk_1,\ldots,bk_n))}$
where $\mathcal{O}_{\mathsf{chall}_b}(\sigma_1,\ldots,\sigma_n,m)$:
$\sigma_0^*\leftarrow\mathsf{DSIG.Comb}(\sigma_1,\ldots,\sigma_n)$
$\sigma_1^*\leftarrow\mathsf{DSIG.Sign}^*(sk,m)$
abort if $\mathsf{DSIG.Vf}(pk,m,\sigma_0^*)=0$
else return σ_b^*
return 1 if $b^*=b$

3.2. Our RSA-DSIG Instantiation

We now describe an instantiation of a proactively secure distributed signature scheme. Our construction is a distributed variant of RSA signatures where the secret key is distributed among the n parties; it also uses two hash functions $H,\tilde{H}$, modeled as random oracles in the security proof, necessary to achieve proactive security. The

$\underline{\mathcal{O}_{\mathsf{sign}}(i, m, \ell)}$
abort if $i \in \mathcal{C}_{t,\mathsf{ep}} \cup \mathcal{C}_{p,\mathsf{ep}}$
$\sigma_i \leftarrow \mathsf{DSIG.Sign}(i, sk_i^{\mathsf{ep}}, m, \ell)$
add $(\mathsf{ep}, i, m, \ell, \sigma_i)$ to $\mathcal{Q}$
return σ_i

$\underline{\mathcal{O}_{\mathsf{refresh}}()}$
for all $i \notin \mathcal{C}_{p,\mathsf{ep}}$:
$\mathsf{DSIG.Refresh}(bk_i^{\mathsf{ep}}) \to (sk_i^{\mathsf{ep}+1}, bk_i^{\mathsf{ep}+1})$
set $\mathsf{ep} \leftarrow \mathsf{ep}+1$, $\mathcal{C}_{t,\mathsf{ep}} \leftarrow \emptyset, \mathcal{C}_{p,\mathsf{ep}} \leftarrow \mathcal{C}_{p,\mathsf{ep}-1}$

$\underline{\mathcal{O}_{\mathsf{corrupt}}(i, mode)}$
if $mode = \mathtt{trans}$:
  abort if $i \in \mathcal{C}_{p,\mathsf{ep}}$
  set $\mathcal{C}_{t,\mathsf{ep}} \leftarrow \mathcal{C}_{t,\mathsf{ep}} \cup \{i\}$
  return sk_i^{ep}
if $mode = \mathtt{perm}$:
  set $\mathcal{C}_{p,\mathsf{ep}} \leftarrow \mathcal{C}_{p,\mathsf{ep}} \cup \{i\}$
  return $(sk_i^{\mathsf{ep}}, bk_i^{\mathsf{ep}})$

$\underline{\mathcal{O}_{\mathsf{sign}}(i, m, \ell)}$
abort if $i \in \mathcal{C}_{t,\mathsf{ep}} \cup \mathcal{C}_{p,\mathsf{ep}}$
$\sigma \leftarrow \mathsf{compSig}(\mathsf{ep}, i, m, \ell)$
$\sigma_i^* \leftarrow_{\mathrm{R}} \mathsf{SIM}(\mathsf{sign}, i, m, \ell, \sigma)$
add $(\mathsf{ep}, i, m, \ell, \sigma_i)$ to $\mathcal{Q}$
return σ_i^*

$\underline{\mathcal{O}_{\mathsf{refresh}}()}$
invoke $\mathsf{SIM}(\mathsf{refresh})$
set $\mathsf{ep} \leftarrow \mathsf{ep}+1$, and
$\mathcal{C}_{t,\mathsf{ep}} \leftarrow \emptyset, \mathcal{C}_{p,\mathsf{ep}} \leftarrow \mathcal{C}_{p,\mathsf{ep}-1}$

$\underline{\mathcal{O}_{\mathsf{corrupt}}(i, mode)}$
get $\mathcal{L} \leftarrow \mathsf{compList}(\mathsf{ep}, i)$
if $mode = \mathtt{trans}$:
  abort if $i \in \mathcal{C}_{p,\mathsf{ep}}$
  set $\mathcal{C}_{t,\mathsf{ep}} \leftarrow \mathcal{C}_{t,\mathsf{ep}} \cup \{i\}$
  $sk_i^* \leftarrow_{\mathrm{R}} \mathsf{SIM}(\mathsf{corr}, \mathtt{trans}, i, \mathcal{L})$
  return sk_i^*
if $mode = \mathtt{perm}$:
  set $\mathcal{C}_{p,\mathsf{ep}} \leftarrow \mathcal{C}_{p,\mathsf{ep}} \cup \{i\}$
  $(sk_i^*, bk_i^*) \leftarrow_{\mathrm{R}} \mathsf{SIM}(\mathsf{corr}, \mathtt{perm}, i, \mathcal{L})$
  return (sk_i^*, bk_i^*)

Figure 3: **Left**: Oracles for our unforgeability and share-simulatability experiment (for $b = 0$). **Right**: simulated oracles for share-simulatability and $b = 1$. Note that compSig used in the simulated sign-oracle will only return $\sigma \neq \perp$ when the adversary is about to complete the signature. Also, we omit the epoch or any previously computed output of SIM as explicit input to the simulator, as SIM can simply keep track of that information itself.

reason we chose RSA and not ECDSA is that even without proactive security, signing requires interaction between the servers [**?**]. Thus the latency of executing this over WAN would overshadow the local computation time of generating an RSA signature.

Generation and Refresh of Key Shares. The signature scheme during KGen generates both the RSA modulus N and the keys d, e as in regular RSA signatures. Next, it generates n shares of d over the integers. This is necessary as the multiplicative order $\varphi(N)$ of the underlying group must be unknown. The algorithm generates two different levels of shares of key for each party, namely "offline" key shares d_i and "online" shares f_i.

First, it chooses d_i from an exponentially larger interval than what the value d can have, which hides d even when the adversary has shares d_i of $n-1$ parties by a standard argument. This "master" share d_i is stored per party inside its backup key bk_i, which is only used during refresh and the adversary gets it only when he permanently corrupts a certain party i. The actual signing share per party and epoch is f_i which each party derives by adding an epoch-specific share of 0 to their d_i share. We again choose f_i to be from an exponentially larger interval than d_i, thus hiding the "master" share from bk_i for any $\mathcal{A}$ only accessing sk_i.

The adding of additive (pseudorandom) shares of 0 is done in a *non-interactive* way by relying on pairwise exchanged seeds $mk_{i,j}$ which the servers can apply locally. These seeds, generated during key generation in a consistent manner, are stored as part of the backup key and used to obtain a fresh f_i at every epoch.

Blinding of Signature Shares. To achieve proactive and adaptive security servers should not commit to their key shares. We achieve this by applying fresh shares β_i of 1 to each signing share. We use a similar trick for the f_i values to do this in non-interactive manner: All servers share pairwise seeds $s_{i,j}$ that are generated during key generation and every refresh. The blinding β_i is derived through a random oracle call on these seeds and a fresh label ℓ. The random oracle gives the flexibility we need in the security proof, and the uniqueness of ℓ ensures that we use every blinding value at most once. These blinding values also prevent mix-and-match attacks, i.e., combining shares for different labels ℓ, as required by the proactive unforgeability notion, as the β_i values only cancel out when the user combines signature shares for the same ℓ.

Detailed RSA-DSIG *Description.* The algorithms for our distributed and proactively secure signature scheme RSA-DSIG are defined as follows. To ease presentation, we define the function $a : \mathbb{Z}^n \to \mathbb{Z}$ as $a(i, \{x_j\}_{j\in[n]\setminus\{i\}}) = \sum_{j\in[i-1]} x_j - \sum_{j\in[i+1,n]} x_j$.

We also stress that we assume that all servers use each label ℓ only once, this can be ensured, e.g., by letting all servers contribute to ℓ before using it. In fact, when using DSIG in our PESTO protocol, the label will simply be the *ssid* which is assumed to be unique for every call.

RSA-DSIG.Setup$(1^\tau, 1^s)$**:** Output $pp \leftarrow (H, \tilde{H})$ where $H : \{0,1\}^* \to \{0,1\}^\tau$ and $\tilde{H}_N : \{0,1\}^* \to [N]$.

RSA-DSIG.KGen(pp, n)**:**

1) Pick random primes p, q of length $\tau/2$, set $N \leftarrow p \cdot q$.
2) Let e be a prime ≥ 3 and set $d \leftarrow e^{-1} \mod \varphi(N)$.
3) Choose random $\gamma_{i,j}, \delta_{i,j}$ for $i, j \in [n], i \neq j$:
   - $\gamma_{i,j} \leftarrow_{\mathrm{R}} [-nN2^s, nN2^s]$ for $i < j$
   - $\delta_{i,j} \leftarrow [-n^2N2^{2s}, +n^2N2^{2s}]$ for $i < j$
   - $\gamma_{j,i} \leftarrow \gamma_{i,j}$, $\delta_{j,i} \leftarrow \delta_{i,j}$ for $i > j$
4) For $i \in [n-1]$ set $d_i \leftarrow a(i, \gamma_{i,1}, \ldots, \gamma_{i,n})$ while $d_n \leftarrow a(n, \gamma_{n,1}, \ldots, \gamma_{n,n-1}) + d$. Furthermore, set $f_i \leftarrow a(i, \delta_{i,1}, \ldots, \delta_{i,n}) + d_i$ for all $i \in [n]$.
5) Pick random blinding seeds $s_{i,j}$ and master keys $mk_{i,j}$ for all $i, j \in [n], i \neq j$:
   - $s_{i,j} \leftarrow_{\mathrm{R}} \{0,1\}^\tau$, $mk_{i,j} \leftarrow_{\mathrm{R}} \{0,1\}^\tau$ for $i < j$
   - $s_{j,i} \leftarrow s_{i,j}$, $mk_{j,i} \leftarrow mk_{i,j}$ $i > j$
6) Output $pk = (N, e)$ and $sk_i = (f_i, \{s_{i,j}\}_{j\in[n]\setminus\{i\}})$, $bk_i = (d_i, \{mk_{i,j}\}_{j\in[n]\setminus\{i\}})$ for every $i \in [n]$.

RSA-DSIG.Sign(sk_i, m, ℓ)**:**

1) Parse $(f_i, \{s_{i,j}\}_{j\in[n]\setminus\{i\}}) \leftarrow sk_i$.
2) Let $\Delta_{i,j} = 1$ if $i > j$ and else -1.
3) Compute $\beta_i = \prod_{j\in[n]\setminus\{i\}} \tilde{H}_N(\ell, s_{i,j})^{\Delta_{i,j}} \mod N$.
4) Output $\sigma_i = H(m)^{f_i} \cdot \beta_i \mod N$.

RSA-DSIG.Comb$(\sigma_1, \ldots, \sigma_n)$**:**
Output $\sigma = \prod_{i=1}^n \sigma_i \mod N$.

RSA-DSIG.Vf(pk, m, σ)**:** Parse $(N, e) \leftarrow pk$. Output $b = 1$ if $\sigma^e = H(m) \bmod N$ and $b = 0$ otherwise.

RSA-DSIG.Refresh(bk_i)**:**

1) Parse $(d_i, \{mk_{i,j}\}_{j\in[n]\setminus\{i\}}) \leftarrow bk_i$.
2) Compute $(mk'_{i,j}, s'_{i,j}, \delta_{i,j}) \leftarrow \mathsf{PRG}(mk_{i,j})$ for every $j \in [n]\setminus\{i\}$ such that $mk'_{i,j} \in \{0,1\}^\tau, s'_{i,j} \in \{0,1\}^\tau$ and $\delta_{i,j} \in [-n^2N2^{2s}, +n^2N2^{2s}]$.
3) Set $f_i = d_i + a(i, \delta_{i,1}, \ldots, \delta_{i,n})$. Then output $sk'_i =$

$(f_i, \{s'_{i,j}\}_{j\in[n]\setminus\{i\}})$ and $bk'_i = (d_i, \{m'_{i,j}\}_{j\in[n]\setminus\{i\}})$.

Theorem 1. *Assuming hardness of RSA and modeling $\tilde{H}$ as a random oracle, the distributed signature* RSA-DSIG *satisfies proactive unforgeability, share simulatability and signature indistinguishability.*

Towards correctness, note that $\sum_{i=1}^{n} a(i, \gamma_{i,1}, \ldots, \gamma_{i,n}) = \sum_{i=1}^{n} a(i, \delta_{i,1}, \ldots, \delta_{i,n}) = 0$, and thus $\sum_{i=1}^{n} d_i = \sum_{i=1}^{n} f_i = d$. Signature indistinguishability follows as the f_i form an additive sharing of d and from the uniqueness of the (combined) signatures for each m.

For share simulatability we can generate all bk_i without knowledge of d as a secret-sharing of 0, which also sets the f_i to be a 0-sharing. This is ok because in order to win, an adversary can obtain at most $n-1$ of these shares for which the secret-sharing is statistically hiding. All but one of the signatures σ_i for a certain epoch ep and input(m, ℓ) can then be generated from the dummy f_i shares, whereas the last signature has to be made fit such that $\sigma = \prod_i \sigma_i$. To make this consistent with transient or permanent corruptions we exploit the programmability of $\tilde{H}$ — and the fact that all labels ℓ will only be used once per server — in the construction of β_i.

Proactive unforgeability follows from the unforgeability of the basic RSA signature scheme for one party together with the share simulatability and signature indistinguishability argument.

The full proof can be found in Appendix A.

Comparison to other Threshold RSA Signatures. Early works on threshold RSA were only proven secure against static corruptions [23], [45], [26], [28]. Later, after the introduction of the proactive security model [41] this was extended to 'static-proactive' security, where the adversary may corrupt parties only in the onset of each operational phase [25], [24], [42], [37]. These works were strengthened towards security for adaptive-proactive corruptions in [18], [27], [36], [9]. However, note that all those works use a different definition for proactive security than ours: they do not assume an inaccessible storage for temporarily corrupted parties whereas our approach, following [14], does (in the form of backup tapes). This allows us to technically distinguish between corruptions inside and outside the refresh phase, leading to different treatment of permanent and transient corruptions. Our use of a backup tape also means that the refresh is completely local, which is not just highly efficient but also means that the overall corruption bound is always optimal. Previous works had to allow for a "cooling-down" after refresh of corrupted parties, leading to non-optimal corruption threshold right after parties got "uncorrupted".

4. Proactively Secure Distributed Partially-Oblivious PRF

In this section we provide both the ideal functionality as well as a construction of a *distributed partially-blind oblivious PRF* (dpOPRF). Both functionality and protocol run in the presence of an arbitrary number of users as well as n servers $S_1, \ldots, S_n$. For each dpOPRF evaluation there will only be one user present and we denote this user as U. The dpOPRF is partially blind in the sense that in addition to the user input x_{priv} (which remains hidden towards the dpOPRF servers) the evaluation is also parameterized by a public value x_{pub}.

4.1. Functionality $\mathcal{F}_{\mathsf{dpOPRF}}$

We follow the previous line of work [32], [33], [35] and model security for our dpOPRF via an ideal functionality $\mathcal{F}_{\mathsf{dpOPRF}}$ in the UC framework [16]. The functionality allows evaluation of a random function on chosen inputs $(x_{\mathsf{priv}}, x_{\mathsf{pub}})$, given all servers $S_1, \ldots, S_n$ agree to participate. Implementing a partially-oblivious function, $\mathcal{F}_{\mathsf{dpOPRF}}$ tells the servers x_{pub} before they have to make their decision. The adversary may also evaluate the function on arbitrary inputs, but crucially requires participation of all servers. Note that our $\mathcal{F}_{\mathsf{dpOPRF}}$ does not contain an adversarial interface for offline evaluations, i.e., without all servers participating, as is the case for other OPRF functionalities [32], [33], [35]. We do not need such an interface since we are only interested in security when less than n servers are corrupted concurrently.

To model servers deviating from the protocol, $\mathcal{F}_{\mathsf{dpOPRF}}$ allows $\mathcal{A}$ to modify the PRF key if not all servers are honest. This is implemented by letting $\mathcal{A}$ input a label lbl and $\mathcal{F}_{\mathsf{dpOPRF}}$ maintaining a different random function for each lbl. Lastly, our $\mathcal{F}_{\mathsf{dpOPRF}}$ provides strong guarantees by preventing "mix-and-match" attacks, where participation agreement of servers can be collected among different evaluation requests (as possible in, e.g., [32], [33]). $\mathcal{F}_{\mathsf{dpOPRF}}$ prevents such attacks by maintaining evaluation ticket counters $\mathsf{ctr}[U, x_{\mathsf{pub}}]$ that are increased only when all servers agree to participate in an evaluation initiated by U using x_{pub} as public input. Neither can other parties "steal" those evaluation tokens, nor can U later decide to instead evaluate on a different public input value.

Corruption and Proactive Security. $\mathcal{F}_{\mathsf{dpOPRF}}$ is designed to work with adversaries performing adaptive permanent (often called *standard* or *Byzantine*) and *transient* corruptions of servers S_i. Transient corruptions are a special type of corruption that we use to model proactive security. Upon a transient corruption, a party's internal state is given to the adversary and he gains control over the party's actions. However, the adversary can decide to "uncorrupt" all transiently corrupted parties. $\mathcal{F}_{\mathsf{dpOPRF}}$ is informed about such corruption recovery via an input `Refresh` from one of the servers. The detailed functionality is given in Figure 4. For brevity, we use the following conventions:

Writing Conventions for $\mathcal{F}_{\mathsf{dpOPRF}}$ (and $\mathcal{F}_{\mathsf{PESTO}}$).

1) The functionality considers a specific session $sid = (S_1, \ldots, S_n, sid')$ and only accepts inputs from servers S_i that are contained in the sid.
2) We assume sub-session identifiers $ssid$ to be unique. All interfaces that take as input an $ssid$ will only accept one input per party and such $ssid$.
3) When the functionality is supposed to retrieve an internal record, but no such record can be found, then the query is ignored.
4) We assume private delayed outputs and inputs, i.e., the adversary can schedule their delivery but not learn the private content. E.g., for calls to "$(\texttt{Eval}, sid, ssid, \mathsf{payload})$ from a party P", the adversary receives $(\texttt{Eval}, sid, ssid, P)$ i.e., he learns all meta-data but *not* the payload.

The functionality is parametrized by a security parameter τ. It interacts with servers $\mathcal{S} := \{S_1, ..., S_n\}$ (specified in the sid), arbitrary other parties and an adversary $\mathcal{A}$. $\mathcal{F}_{\mathsf{dpOPRF}}$ maintains a table $T(\mathsf{lbl}, x_{\mathsf{pub}}, x_{\mathsf{priv}})$ initially undefined everywhere, counters $\mathsf{ctr}[U, x_{\mathsf{pub}}]$ initially set to 0 and sets $\mathcal{C}_t, \mathcal{C}_p$ initially set to $\emptyset$. The label lbl is an arbitrary string $\{0,1\}^*$, where hon denotes the "honest" label.

Key Generation

- **On receiving** $(\texttt{KeyGen}, sid)$ **from** S_i**:**
  - Ignore if the sid is marked `ready`.
  - If $(\texttt{KeyGen}, sid)$ was received from all S_i, mark sid as `ready`, and give output $(\texttt{KeyConf}, sid)$ to all S_i.

Evaluation

- **On receiving** $(\texttt{Eval}, sid, ssid, x_{\mathsf{pub}}, x_{\mathsf{priv}})$ **from any party** U **(including** $\mathcal{A}$**):**
  - Record $(\texttt{eval}, sid, ssid, U, x_{\mathsf{pub}}, x_{\mathsf{priv}})$, and send output $(\texttt{Eval}, sid, ssid, x_{\mathsf{pub}})$ to every S_i.
- **On receiving** $(\texttt{EvalProceed}, sid, ssid)$ **from** S_i**:**
  - Retrieve record $(\texttt{eval}, sid, ssid, U, x_{\mathsf{pub}}, x_{\mathsf{priv}})$.
  - If $(\texttt{EvalProceed}, sid, ssid)$ has been received from all S_i, set $\mathsf{ctr}[U, x_{\mathsf{pub}}] \leftarrow \mathsf{ctr}[U, x_{\mathsf{pub}}] + 1$.
- **On receiving** $(\texttt{EvalComplete}, sid, ssid, \mathsf{lbl}^*)$ **from** $\mathcal{A}$**:**
  - Retrieve record $(\texttt{eval}, sid, ssid, U, x_{\mathsf{pub}}, x_{\mathsf{priv}})$, only proceed if $\mathsf{ctr}[U, x_{\mathsf{pub}}] > 0$, set $\mathsf{ctr}[U, x_{\mathsf{pub}}] \leftarrow \mathsf{ctr}[U, x_{\mathsf{pub}}] - 1$.
  - Set $\mathsf{lbl} \leftarrow$ hon if all servers are honest, and $\mathsf{lbl} \leftarrow \mathsf{lbl}^*$ else.
  - If $T(\mathsf{lbl}, x_{\mathsf{pub}}, x_{\mathsf{priv}})$ is undefined, then pick $\rho \leftarrow_{\mathsf{R}} \{0,1\}^\tau$ and set $T(\mathsf{lbl}, x_{\mathsf{pub}}, x_{\mathsf{priv}}) \leftarrow \rho$.
  - Output $(\texttt{EvalComplete}, sid, ssid, T(\mathsf{lbl}, x_{\mathsf{pub}}, x_{\mathsf{priv}}))$ to U.

Corruption and Refresh

- **On receiving** $(\texttt{Corrupt}, sid, S_i, mode)$ **from** $\mathcal{A}$ **with** $S_i \in \mathcal{S}$ **and** $mode \in \{\texttt{trans}, \texttt{perm}\}$**:**
  - If $mode = \texttt{trans}$, set $\mathcal{C}_t \leftarrow S_i \cup \mathcal{C}_t$; if $mode = \texttt{perm}$, then set $\mathcal{C}_p \leftarrow S_i \cup \mathcal{C}_p$.
- **On receiving** $(\texttt{Refresh}, sid)$ **from a server** S_i**:**
  - Set $\mathcal{C}_t \leftarrow \emptyset$, abort all ongoing `Eval` processes, reset all counters $\mathsf{ctr}[*] \leftarrow 0$ and output $(\texttt{Refresh}, sid)$ to $\mathcal{A}$.

Figure 4: Ideal functionality $\mathcal{F}_{\mathsf{dpOPRF}}$

5) Calls to all interfaces other than $(\texttt{KeyGen}, sid)$ will only be processed after `KeyGen` is completed.

4.2. Our DH-dpOPRF Construction

We now describe our protocol DH-dpOPRF, which has at its core the distributed 2HashDH protocol for computing $H'(x_{\mathsf{priv}}, H(x_{\mathsf{priv}})^K)$ [31], but combines additional features inspired by other works: partial obliviousness using pairings [21], and proactive security [14].

In the 2HashDH protocol, the user blinds his input x_{priv} with randomness r and sends $\bar{x} \leftarrow H_1(x_{\mathsf{priv}})^r$ to the server, which she can remove again after receiving $\bar{x}^K$ from the server. The protocol can be made distributed by simply setting $K \leftarrow \sum_{i \in [n]} k_i$ where k_i are the individual server keys and letting the user multiply the individual PRF shares she receives. To achieve *partial* blindness, servers contribute $e(\bar{x}, H_2(x_{\mathsf{pub}}))^{k_i}$ with an asymmetric pairing e and a public input x_{pub} obtained from the user in the clear. Overall, our protocol computes $y \leftarrow \mathsf{PRF}_K(x_{\mathsf{priv}}, x_{\mathsf{pub}})$ with:

$$y = H_4\left(x_{\mathsf{priv}}, x_{\mathsf{pub}}, e\left(H_1(x_{\mathsf{priv}}), H_2(x_{\mathsf{pub}})\right)^{\sum_{i \in [n]} k_i}\right)$$

Setup. For our construction, we assume a bilinear group $(p, g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and hash functions $H_1 : \{0,1\}^* \to \mathbb{G}_1$, $H_2 : \{0,1\}^* \to \mathbb{G}_2$, $H_3 : \{0,1\}^* \to \mathbb{G}_T$, and $H_4 : \{0,1\}^* \times \mathbb{G}_T \to \{0,1\}^\tau$.

Key Generation. For simplicity we assume the existence of a trusted dealer that generates master keys for the servers. Each server disposes of an offline backup tape that can be plugged in for performing read and write operations on it. A server S_i, upon input $(\texttt{KeyGen}, sid)$, sends $(\texttt{KeyGen}, sid)$ to the trusted dealer. The trusted dealer, upon the first message $(\texttt{KeyGen}, sid)$ for a particular sid from a server S_i, generates the signing keys for all n servers in $sid = (S_1, \ldots, S_n, sid')$. More precisely, the dealer generates the secret key shares k_i of an implicit joint PRF key $K \leftarrow_{\mathsf{R}} \mathbb{Z}_q$, seeds $s_{i,j}$ for blinding factors as well as master keys $mk_{i,j}$ as follows:

- For all $i \in [n]$: choose $k_i \leftarrow_{\mathsf{R}} \mathbb{Z}_q$
- For all $i, j \in [n], i \neq j$:
  - $s_{i,j} \leftarrow_{\mathsf{R}} \{0,1\}^\tau$, $mk_{i,j} \leftarrow_{\mathsf{R}} \{0,1\}^\tau$ for $i < j$
  - $\mathrm{s}_{j,i} \leftarrow s_{i,j}$, $mk_{j,i} \leftarrow mk_{i,j}$ for $j > i$

It sends $(k_i, \{s_{i,j}\}_{j \in [n] \setminus \{i\}}, \{mk_{i,j}\}_{j \in [n] \setminus \{i\}})$ to S_i over a secure channel. Note that the seeds $s_{i,j} = \mathrm{s}_{j,i}$ and master keys $mk_{i,j} = mk_{j,i}$ will be known only to the servers S_i and S_j, i.e., each pair of servers will share a common master key and blinding seed that is unknown to all other servers. For all further messages $(\texttt{KeyGen}, sid)$ from the other servers S_j contained in sid, the dealer simply responds with the already generated key shares for that server. Upon receiving $(k_i, \{s_{i,j}\}_{j \in [n] \setminus \{i\}}, \{mk_{i,j}\}_{j \in [n] \setminus \{i\}})$, S_i stores $(k_i, \{s_{i,j}\}_{j \in [n] \setminus \{i\}})$ as the current epoch key, stores $(k_i, \{mk_{i,j}\}_{j \in [n] \setminus \{i\}})$ on the backup tape and erases any $mk_{i,j}$ from the memory.

Evaluation. The detailed protocol is given in Figure 5 and lets the user U and all n servers jointly and (partially) blind compute the function $\mathsf{PRF}_K(x_{\mathsf{priv}}, x_{\mathsf{pub}})$ stated above for private input x_{priv} and public input x_{pub}. For achieving proactive and adaptive security it is again important to avoid any commitment to secret key shares. We use the same approach as in our RSA-DSIG construction and let servers mask all messages with a blinding value β_i which is a pseudorandom multiplicative share of 1 that is constructed from the pairwise shared $s_{i,j}$-values and the (unique) $ssid$. Similar as in RSA-DSIG the purpose of the servers' blinding values β_i is two-fold as they also prevent mix-and-match attacks, i.e., combining shares from different sub-sessions $ssid$, as required by our functionality.

Refresh. The refresh is triggered when a server S_i receives the input $(\texttt{Refresh}, sid)$, upon which it sends $(\texttt{Refresh}, sid)$ over a secure broadcast channel to all other servers $S_j \in \mathcal{S} \setminus S_i$ which now perform their updates in a synchronized manner. They first retrieve their values $(k_i, \{mk_{i,j}\}_{j \in [n] \setminus \{i\}})$ from the backup tape and then use

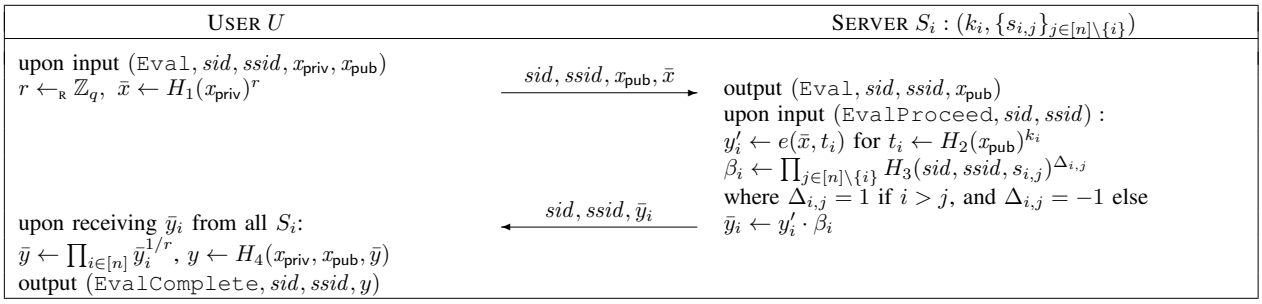

Figure 5: Evaluation protocol of our DH-dpOPRF construction.

a PRG to derive a new master key, fresh seeds $s'_{i,j}$ for the blinding values, and seeds $\delta_{i,j}$ for their update share. The latter is used to refresh the PRF key share, which is simply done by updating the old share k_i with a share of 0. Relying on $\delta_{i,j}$ which are deterministically derived from their pairwise exchanged master keys $mk_{i,j}$, the servers can compute these shares of 0 in a non-interactive fashion. More precisely, each server S_i (either triggered via $(\texttt{Refresh}, sid)$ or by receiving $(\texttt{Refresh}, sid)$ from a server $S_j \in \mathcal{S}$) runs the following update procedure:

- Retrieve $(k_i, \{mk_{i,j}\}_{j\in[n]\setminus\{i\}})$ from the backup tape.
- For all $j \in [n] \setminus \{i\}$: $(mk'_{i,j}, s'_{i,j}, \delta_{i,j}) \leftarrow \mathsf{PRG}(mk_{i,j})$.
- Set $k'_i \leftarrow k_i + \sum_{j=1,j\neq i}^{n} \Delta_{i,j}\delta_{i,j} \bmod q$, where $\Delta_{i,j}$ is $\Delta_{i,j} = 1$ if $i > j$, and $\Delta_{i,j} = -1$ else.
- Store $(k'_i, \{s'_{i,j}\}_{j\in[n]\setminus\{i\}})$ as the current epoch key and $(k'_i, \{mk'_{i,j}\}_{j\in[n]\setminus\{i\}})$ on the backup tape.
- Securely delete $k_i, mk_{i,j}, mk'_{i,j}, s_{i,j}, \delta_{i,j}$ for all $j \in [n] \setminus \{i\}$ from the memory, and $(k_i, \{mk_{i,j}\}_{j=1,j\neq i}^{n})$ from the backup tape.

Note that retrieving an *uncorrupted* k_i from the backup tape is necessary to recover from transient corruptions, as the adversary may have altered k_i.

4.2.1. Security of DH-dpOPRF. For our analysis, we require that transiently corrupted parties are always "uncorrupted" before plugging in their backup tape, such that transient corruptions never reveal the backup tape to the adversary. Formally, this is handled by defining a corruption model for UC where, upon sending `Refresh`, the adversary neither controls nor observes the state of any formerly transient corrupted party anymore, and all these parties are considered honest again. As a result, they give inputs and receive outputs directly to/from $\mathcal{F}_{\mathsf{dpOPRF}}$ instead of sending them via the adversary. For this corruption modeling to be meaningful, we assume that the adversary can corrupt servers at any time, but if he does so during `KeyGen` or `Refresh`, then he must perform a permanent corruption.

We assume private and server-sided authenticated channels between each user and each server. Such a channels can be implemented, e.g., using TLS.

Theorem 2. *The protocol* DH-dpOPRF *securely realizes* $\mathcal{F}_{\mathsf{dpOPRF}}$ *in the random oracle model if* PRG *is secure and the Gapom-BDH assumption holds in* $\mathbb{G}$*, w.r.t environments corrupting at most* $n-1$ *servers concurrentlyusing adaptive permanent and transient corruptions.*

The Gapom-BDH assumption (Def 3), stated in Section 2, is a generalization of the assumption introduced by [21] which in turn is the bilinear variant of the one-more DDH assumption that underlies the standard 2HashDH construction. The overall proof idea is similar to the one by [33], extended to our partially-blind and proactive setting. An informal proof is given in Appendix B.

5. Proactively-Secure Distributed SSO

We now present our approach for proactively-secure distributed SSO (PESTO). We start by describing the high-level functionality and desired security properties. In Section 5.1 we then present our formal security for PESTO and give our protocol in Section 5.2.

High-Level Idea. From a user perspective, PESTO should work just as standard SSO, but simply interacting with n servers instead of only one. The servers hereby possess a joint public verification key vk.

First, the user can create a user account for a certain user name uid and a chosen password pw. Note that he only uses *one* password to register with all n servers. Upon successful registration, the user can request joint signatures of all servers on messages m of his choice. To do so, the user must authenticate under uid and a password attempt pw'. If the provided password matches the one used at account creation for uid, the user receives a signature σ, that was created by all servers for (uid, m), i.e., they bind the message to the verified user name. For SSO applications, the message will contain the unique ID of the targeted service provider and a nonce that is usually specified by the provider. Finally, the signature σ can be verified by everyone against the public key vk. We want PESTO to satisfy the following security properties:

Offline-Attack Resistance: An adversary corrupting at most $n-1$ servers must not be able to run offline attacks against the users' account passwords.

Unforgeability of Tokens: An adversary corrupting at most $n-1$ servers cannot forge signatures that verify under the joint public key vk. The only way to receive a valid signature for an honestly created uid and adversarially chosen message m^* is by guessing the associated password *and* getting the remaining honest server(s) to explicitly approve the signature request.

Server-Controlled Rate Limiting: The servers have strong control over the user accounts, meaning that passwords can only be verified if *all* servers explicitly agree to do so for the given uid. That is, servers can prevent further verification when they detect online guessing attempts against a certain user, or the user asked to block or pause her account. Furthermore, the servers explicitly learn at every signing request whether the provided user password was correct or not (w/o violating the offline-attack resistance) such that they

can base their further actions on this information, e.g., blocking accounts after 10 failed attempts.

Adaptive and Proactive Security: For modelling realistic attacks, we allow the adversary to corrupt servers in an *adaptive* fashion, i.e., he can take control of any initially honest party at any time. We further allow *transient corruptions* as detailed in Section 4.2.1, ensuring that our protocol features proactive security and the above properties hold even when all servers get corrupted, as long as not all of them are corrupted at the same time.

5.1. Security Model

In this section we present our formal security for PESTO in form of an ideal functionality $\mathcal{F}_{\mathsf{PESTO}}$ in the UC framework. The main entities in our system are a set of servers $\mathcal{S} := \{S_1, ..., S_n\}$ which are specified in the $sid = (S_1, \ldots, S_n, sid')$, where sid' is a unique string. We now briefly discuss the interfaces our functionality provides and how they enforce the desired security properties sketched above. The detailed definition is given in Figure 6. For the sake of brevity, our $\mathcal{F}_{\mathsf{PESTO}}$ definition assumes the writing conventions from Section 4.

Key Generation. The functionality is parametrized by algorithms $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify})$, and internally generates a key pair $(vk, sk) \leftarrow \mathsf{KGen}(1^\tau)$ when all servers have explicitly triggered key generation by sending $(\texttt{KeyGen}, sid)$. From then on, $\mathcal{F}_{\mathsf{PESTO}}$ accepts calls to the other interfaces and will guarantee unforgeability of signatures w.r.t. vk. We stress that we will not rely on any security properties of these algorithms but only use them to let $\mathcal{F}_{\mathsf{PESTO}}$ output well-formed cryptographic values.

Account Creation. The creation of a new account for username uid and password pw is initiated by a user on input $(\texttt{Register}, sid, ssid, uid, pw)$. To distinguish between several account creation (and signing) sessions, we use unique sub-session identifiers $ssid$. If uid has not been registered yet, all servers $S_1, \ldots, S_n$ are notified about the request (but without learning the password) and must approve it by sending $(\texttt{ProceedReg}, sid, ssid, uid)$. The functionality internally stores (uid, pw) and from then on allows signing requests for uid. We flag accounts created by malicious users, as the adversary will have more control over these when signing.

Password-Authenticated Distributed Signing. After an account for uid was created, a user can request password-authenticated signatures for a message m by sending $(\texttt{Sign}, sid, ssid', uid, m, pw')$ where pw' denotes the password attempt the user is logging in with. The servers are notified — again without learning the password pw' — and the functionality only proceeds with the password verification when all servers have responded with $(\texttt{ProceedSign}, sid, ssid', uid)$.

Awaiting explicit approval of all servers gives each the opportunity to block a session if they detect some suspicious behaviour, or they have been asked by the user to suspend her account. This is crucial to prevent offline attacks against the password, as well as to detect and stop online guessing attacks. Note that this implicitly implements our rate limiting requirement: servers will not send the `ProceedSign` command when they suspect malicious activities, e.g., a certain number of failed login attempts. By not specifying the exact method how servers come to the decision on whether to proceed or not, our model remains flexible, i.e., it can support any rate limiting policy.

When given approval, the functionality now checks whether pw' matches the original password pw the user has created her account with. If all servers and the user are honest, the decision bit is simply set to $b \leftarrow (pw = pw')$. If malicious parties are involved, the adversary has a bit of wiggle room: If at least one server is corrupt, the adversary can always make a correct login fail and enforce $b \leftarrow 0$ but not vice versa, i.e., he cannot make a mismatch of the passwords look like a match. If the user account for uid has been corrupted, either because it was created by the adversary or the adversary correctly guessed the password of an initially honest account, $\mathcal{A}$ now can freely set the decision bit, modeling the fact that he has full control over the account anyway. These capabilities of the attacker are modeled by first telling $\mathcal{A}$ whether login was successful via output `match-ok`, and then asking $\mathcal{A}$ for a bit b^*. The functionality enforces its influence according to the corruption setting. All servers receive the decision bit via an output $(\texttt{Match}, sid, ssid', b)$.

Finally, when the password was determined to be correct, the signature is created. $\mathcal{F}_{\mathsf{PESTO}}$ generates $\sigma \leftarrow \mathsf{Sign}(sk, (uid, m))$ and stores a record $(\texttt{sigrec}, uid, m, \sigma, vk, \texttt{true})$ that will allow successful verification of σ. Note that the record also contains uid which enforces that the signature is only valid for that particular user. The user receives the computed signature via the output $(\texttt{Signature}, sid, ssid', \sigma)$.

Verification. Everyone can check the validity of signatures by sending $(\texttt{Verify}, sid, uid, m, \sigma, vk')$. If $vk' = vk$, i.e,. verification is requested for the verification key associated with sid, then $\mathcal{F}_{\mathsf{PESTO}}$ uses its internal records to determine whether σ is valid for (uid, m). The output is set to $f \leftarrow \texttt{true}$ only if a record $(\texttt{sigrec}, uid, m, \sigma, vk, \texttt{true})$ exist. As such records only get created through successful signing requests, the desired unforgeability is enforced. For other verification keys, the functionality uses the Verify algorithm to determine f. Allowing such verification for "incorrect" public keys is necessary to avoid that $\mathcal{F}_{\mathsf{PESTO}}$ must realize a trusted certification authority too.

Corruptions and Refresh. We use the same transient corruption model in UC as in Section 4.2.1, and thus $\mathcal{F}_{\mathsf{PESTO}}$ provides the same corruption interfaces `Corrupt` and `Refresh` for corrupting servers as $\mathcal{F}_{\mathsf{dpOPRF}}$.

We note that the functionality does not prevent the adversary from calling `Refresh` when he has corrupted a server. Since a refresh models "cleaning" of all transiently corrupted servers, it does not seem desireable for a corrupted server to call `Refresh`: if the calling server is transiently corrupted, such a request will only result in the adversary being booted from that server.

$\mathcal{F}_{\mathsf{PESTO}}$ does not provide a specific interface for adaptive user corruption, but nonetheless captures such attacks: It does not perform any checks based on the user's "identity" U, i.e., an account created by an honest user can later be accessed through the adversary if it knows the correct password, which models adaptive corruption of a user's device. Note that there is no long-term secret stored by the user and that password guessing and corruption is

The functionality is parametrized by algorithms $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify})$ and security parameter τ. It interacts with servers $\mathcal{S} := \{S_1, ..., S_n\}$ (specified in the sid), as well as arbitrary users, verifiers and an adversary $\mathcal{A}$. Let $\mathcal{C}_t, \mathcal{C}_p$ denote initially empty sets.

Key Generation

- **On receiving** $(\texttt{KeyGen}, sid)$ **from server** S_i**:**
  - Ignore if a record $(\texttt{key}, sk, vk)$ already exists.
  - If $(\texttt{KeyGen}, sid)$ was received from all S_i:
    - Create a record $(\texttt{key}, sk, vk)$ with $(vk, sk) \leftarrow \mathsf{KGen}(1^\tau)$ and output $(\texttt{KeyConf}, sid, vk)$ to all S_i.

Account Creation

- **On receiving** $(\texttt{Register}, sid, ssid, uid, pw)$ **from user** U**:**
  - Proceed only if *no* record $(\texttt{account}, uid, *)$ exists. Create record $(\texttt{register}, ssid, U, uid, pw)$.
  - Send a delayed output $(\texttt{Register}, sid, ssid, uid)$ to all $S_i \in \mathcal{S}$.
- **On receiving** $(\texttt{ProceedReg}, sid, ssid, uid)$ **from server** S_i**:**
  - Retrieve record $(\texttt{register}, ssid, U, uid, pw)$. If $(\texttt{ProceedReg}, sid, ssid, uid)$ has been received from all S_i:
    - Record $(\texttt{account}, uid, pw)$; if U is corrupt, mark uid $\texttt{corrupted}$.
    - Send a delayed output $(\texttt{Registered}, sid, ssid, uid)$ to U.

Signing Request and Verification

- **On receiving** $(\texttt{Sign}, sid, ssid', uid, m, pw')$ **from party** U**:**
  - Proceed only if a record $(\texttt{account}, uid, pw)$ exists. Create record $(\texttt{sign}, ssid', U, uid, m, pw', b)$ with $b \leftarrow \bot$.
  - Send a delayed output $(\texttt{Sign}, sid, ssid', uid, m)$ to all $S_i \in \mathcal{S}$.
- **On receiving** $(\texttt{ProceedSign}, sid, ssid', uid)$ **from server** S_i**:**
  - Retrieve records $(\texttt{sign}, ssid', U, uid, m, pw', b)$ and $(\texttt{account}, uid, pw)$.
  - If $(\texttt{ProceedSign}, sid, ssid', uid)$ has been received from all S_i:
    - If U is corrupt and $pw = pw'$, mark uid $\texttt{corrupted}$ and set $b \leftarrow 1$.
    - Send $(\texttt{match-ok}, sid, ssid', b)$ to $\mathcal{A}$ and receive back $(\texttt{match-ok}, sid, ssid', b^*)$.
    - Update the $\texttt{sign}$ record by (re-)setting the password verification bit b:
      - If U and uid are corrupted, set $b \leftarrow b^*$.
      - Else, if all servers are honest set $b \leftarrow (pw = pw')$. If at least one $\mathcal{S}_i$ is corrupt, set $b \leftarrow b^* \wedge (pw = pw')$.
    - Send a delayed output $(\texttt{Match}, sid, ssid', b)$ to all $S_i \in \mathcal{S}$.
    - Only proceed if $b = 1$, retrieve $(\texttt{key}, sk, vk)$ and create the signature:
      - Send $(\texttt{sign-ok}, sid, ssid')$ to $\mathcal{A}$ and receive back $(\texttt{sign-ok}, sid, ssid')$
      - Create $\sigma \leftarrow \mathsf{Sign}(sk, (uid, m))$, abort if $(\texttt{sigrec}, uid, m, \sigma, vk, \texttt{false})$ exists.
      - Record $(\texttt{sigrec}, uid, m, \sigma, vk, \texttt{true})$ and output $(\texttt{Signature}, sid, ssid', \sigma)$ to U. *(Correctness)*
- **On receiving** $(\texttt{Verify}, sid, uid, m, \sigma, vk')$ **from party** V**:**
  - If $(\texttt{sigrec}, uid, m, \sigma, vk', f')$ exists, set $f \leftarrow f'$ *(Consistency)*
  - Else, create a record $(\texttt{sigrec}, uid, m, \sigma, vk', f)$ where f is determined as follows:
    - If $vk = vk'$, set $f \leftarrow \texttt{false}$ *(Strong Unforgeability)*, else set $f \leftarrow \mathsf{Verify}(vk', (uid, m), \sigma)$.
  - Output $(\texttt{Verified}, sid, uid, m, \sigma, vk', f)$ to V.

Corruption and Refresh

- **On receiving** $(\texttt{Corrupt}, sid, S_i, mode)$ **from** $\mathcal{A}$ **with** $S_i \in \mathcal{S}$ **and** $mode \in \{\texttt{trans}, \texttt{perm}\}$**:**
  - If $mode = \texttt{trans}$, set $\mathcal{C}_t \leftarrow S_i \cup \mathcal{C}_t$; if $mode = \texttt{perm}$, then set $\mathcal{C}_p \leftarrow S_i \cup \mathcal{C}_p$.
- **On receiving** $(\texttt{Refresh}, sid)$ **from server** S_i**:**
  - Set $\mathcal{C}_t \leftarrow \emptyset$, abort all ongoing $\texttt{Register}$ and $\texttt{Sign}$ processes and output $(\texttt{Refresh}, sid)$ to $\mathcal{A}$.

Figure 6: Ideal functionality $\mathcal{F}_{\mathsf{PESTO}}$ for proactively-secure distributed SSO

captured by the UC model, as the environment can leak passwords of honest users to the adversary.

5.1.1. SSO-specific Modeling Choices. Apart from binding the username uid to signed messages, there is another subtle SSO-related aspect to our model. Essentially, $\mathcal{F}_{\mathsf{PESTO}}$ can be seen as a complex *signature functionality*. The common approach for UC signature definitions [17] is to let the adversary provide the signature values σ (either directly or by imputing algorithms with hardcoded signing keys). $\mathcal{F}_{\mathsf{PESTO}}$ instead generates signatures within the functionality for a key that is unknown to the adversary. While this might seem like a benign modeling choice — unforgeability does not depend on σ but the records $\mathcal{F}_{\mathsf{PESTO}}$ maintains — it makes a big difference for our SSO application: Allowing the adversary to determine and learn signatures of honest users would render the SSO aspect useless where signatures serve as authentication tokens! The internal approach taken by $\mathcal{F}_{\mathsf{PESTO}}$ ensures that only the affected user learns her signature, but also makes proving security much more challenging.

5.2. Our PESTO Construction

On a high level, our protocol Π_{PESTO} combines a distributed partially-oblivious PRF dpOPRF, a distributed signature scheme DSIG and standard signature SIG. The dpOPRF is evaluated on the username uid (public) and password (private), and the PRF value is interpreted as long-term secret signing key usk of SIG. Servers store the corresponding verification key upk in the user's account. When the user later requests a signature for message m and uid, she re-derives usk and signs a session-specific nonce to convince the servers of her possessing the right password. Servers then jointly sign the message (uid, m) using the distributed signature scheme DSIG.

The distributed building blocks are as defined in Sections 3 and 4, and SIG is a conventional signature scheme consisting of algorithms $(\mathsf{SIG.KGen}, \mathsf{SIG.Sign}, \mathsf{SIG.Vf})$. In our construction we will make the randomness r used in SIG.KGen explicit, and assume that key generation behaves deterministically when using the same r.

Key Generation. We assume a trusted dealer that pro-

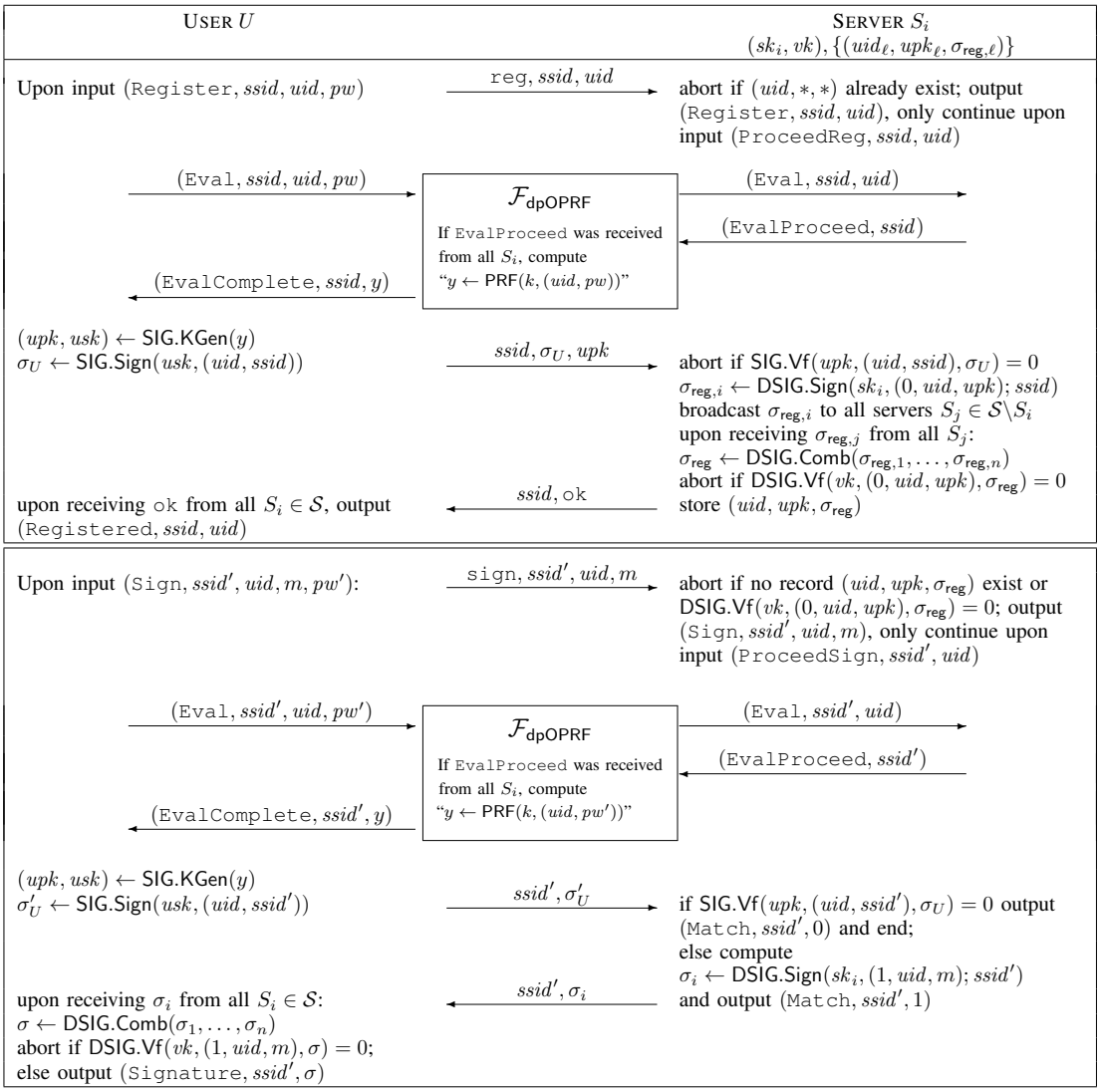

Figure 7: Registration and sign procedures of Π_{PESTO}. For concise notation, we omit the sid from all in- and outputs.

vides all servers with their initial key shares and backup keys. That is, when a server S_i receives input $(\texttt{KeyGen}, sid)$ it checks that $sid = (S_1, \ldots, S_n, sid')$. It sends $(\texttt{KeyGen}, sid)$ to $\mathcal{F}_{\mathsf{dpOPRF}}$ and $(\texttt{keygen}, sid)$ to the trusted dealer of the DSIG scheme.

The trusted dealer, upon first message $(\texttt{keygen}, sid)$ for a particular sid from a server S_i generates the signing keys for all n servers as $(pk, \{sk_i\}_{i\in[n]}, \{bk_i\}_{i\in[n]}) \leftarrow_{\mathrm{R}} \mathsf{DSIG.KGen}(pp, n)$. It sets $vk \leftarrow pk$ and returns (vk, sk_i, bk_i) over a secure channel to S_i. For all further messages $(\texttt{keygen}, sid)$ from the other servers in sid, the dealer responds with the already generated key shares. Each server stores (sid, sk_i, vk) as the online signing key, and keeps bk_i on a secure and offline backup tape.

Account Creation. A user can create an account for username uid and password pw with all n servers, by running the protocol depicted in Figure 7. Roughly, the servers perform a partially blind $\mathcal{F}_{\mathsf{dpOPRF}}$ evaluation where the uid is the public and pw the blinded input. From the received $\mathcal{F}_{\mathsf{dpOPRF}}$ output y, the user deterministically derives a key pair usk, upk of a standard signature scheme, and sends upk and a signature computed from usk back to the servers. The servers now check the validity of the upk and user signature, and then use their distributed signature scheme to sign $(0, uid, upk)$. The jointly computed signature σ_{reg} on these values ensures that an adversary cannot gradually plant malicious public keys for honest user accounts, when he transiently corrupts individual servers. Prepending the 0-bit to the signed message is done to enforce domain separation from the messages the servers are signing for the users later on (using the same distributed signature scheme). The account information stored by each server is $(uid, upk, \sigma_{\mathsf{reg}})$.

Signature Request & Verification. When an account has been created successfully, the user can request signatures for messages m of his choice, and authenticating as uid using a password attempt pw'. If the servers have valid account information stored for uid and agree to proceed, they first blindly compute the $\mathcal{F}_{\mathsf{dpOPRF}}$ output for uid, pw'. The user then (re-)derives its key pair usk, upk, where usk will be the matching secret key to the public key stored by the servers. This is tested by letting the user sign a fresh value, which is simply the query identifier $ssid'$, with its usk. Each server then verifies the received signature to check if the user provided the correct password. If that check passes, each server provides the user with a signature share σ_i for $(1, uid, m)$. That is, the

servers sign the user's message and bind it to its verified username uid. Again, the prefix bit 1 serves for domain separation. The user combines all shares into the final signature. The detailed protocol is given in Figure 7.

Refresh. If a server S_i receives input $(\texttt{Refresh}, sid)$ it broadcasts $(\texttt{refresh}, sid)$ to all other servers $S_j \in \mathcal{S} \backslash S_i$, and sends $(\texttt{Refresh}, sid)$ to $\mathcal{F}_{\mathsf{dpOPRF}}$. It then updates its signature key share with the help of the secure backup key, and securely deletes the old keys.

- retrieve backup key bk_i
- get $(sk_i', bk_i') \leftarrow_{\mathrm{R}} \mathsf{DSIG.Refresh}(bk_i)$
- set $sk_i \leftarrow sk_i'$ and $bk_i \leftarrow bk_i'$

Server S_j receiving $(\texttt{refresh}, sid)$ from S_i, runs the same update of the signature key shares described above. Note that only the key shares change, but vk and the signatures computed under the old keys remain valid.

The servers could now also engage in a protocol to recover from potential loss or compromise of their states (as the user accounts are not contained in the trusted backup). That is, each server could first check if they have records $\{(uid_\ell, upk_\ell, \sigma_{\mathsf{reg},\ell})\}$ for the same set of users, and whether these tuples contain valid signatures $\sigma_{\mathsf{reg},\ell}$ under vk. If not, they can ask the other servers to send their stored information. As long as one server is honest, the others will receive the correct information and can recover from state loss or compromise. Also note that the signature $\sigma_{\mathsf{reg},\ell}$ is computed with the distributed and proactively secure DSIG scheme, thus malicious servers cannot distribute incorrect yet valid account information.

We do not make such a recovery process explicit in our protocol, and instead check the validity of the user account at each login session.

5.2.1. Security of our PESTO Construction. We assume private and server-sided authenticated channels between each user and each server, e.g., using TLS. Further, we assume a private and authenticated broadcast channel with guaranteed delivery among the n servers and which is used only when a new user is registered. We further assume a trusted dealer during setup and key generation. All servers possess secure backup tapes that are assumed to be not corruptible via transient corruptions (cf. Section 4.2.1).

In the following theorem we slightly abuse notation and let $\mathsf{Comb}^* \circ \mathsf{DSIG.KGen}$ denote the algorithm that first runs $(vk, (sk_i)_{i\in[n]}, (bk_i)_{i\in[n]}) \leftarrow \mathsf{DSIG.KGen}(1^\tau)$, discards all bk_i from the output and obtains $sk \leftarrow \mathsf{Comb}^*(vk, sk_1, \ldots, sk_n)$ (with Comb^* being the algorithm from the signature indistinguishability of DSIG).

Theorem 3. *Consider protocol Π_{PESTO} from Figure 7 and Section 5.2. If SIG is an existentially unforgeable signature scheme, DSIG a distributed signature scheme that is proactively unforgeable, signature indistinguishable w.r.t algorithms $\mathsf{Comb}^*, \mathsf{Sign}^*$ and share simulatable, then Π_{PESTO} securely realizes $\mathcal{F}_{\mathsf{PESTO}}$ with algorithms $(\mathsf{Comb}^* \circ \mathsf{DSIG.KGen}, \mathsf{Sign}^*, \mathsf{DSIG.Vf})$ in the $\mathcal{F}_{\mathsf{dpOPRF}}$-hybrid model, w.r.t adaptively corrupting environments controlling at most $n-1$ servers concurrently.*

To prove the above theorem we need to show that for any environment $\mathcal{Z}$ and adversary $\mathcal{A}$ there exists an efficient ideal-world adversary algorithm, called simulator $\mathcal{S}$, such that the view of $\mathcal{Z}$ interacting with Π_{PESTO} and $\mathcal{A}$ is indistinguishable from the view that $\mathcal{F}_{\mathsf{PESTO}}$ and $\mathcal{S}$ provide. The detailed proof is in Appendix C.

Proof. (Sketch) Before describing the simulation strategy, we first point to some challenges we face in the proof.

- While $\mathcal{S}$ has control over the trusted dealer, signature tokens are verified w.r.t vk chosen by $\mathcal{F}_{\mathsf{PESTO}}$. $\mathcal{S}$ has to, e.g., simulate signing key shares upon server corruption that "look like" belonging to vk. We stress that $\mathcal{S}$ even needs to simulate "full sets" of n signature shares that $\mathcal{Z}$ observes through a corrupted user, and that combine to a signature verifying under vk.
- As discussed in Sec. 5.1.1 $\mathcal{F}_{\mathsf{PESTO}}$ differs from standard UC signature definitions in that it generates signatures itself for a key that is unknown to $\mathcal{S}$. $\mathcal{S}$ needs to provide a view that is consistent with these internally generated signatures and the public verification key.
- Modeling a password-authenticated primitive, $\mathcal{F}_{\mathsf{PESTO}}$ protects against offline dictionary attacks and admits to $\mathcal{S}$ only one online password guess (via the `match-ok` message). $\mathcal{S}$ has to simulate the protocol transcript from only this one guess.

Honest server & corrupted user: $\mathcal{S}$ needs to simulate signature shares of honest servers towards the corrupted user, as well as key shares when a server gets corrupted. The simulator has to manage this without knowing any key shares, yet remain a consistent view w.r.t. the vk of $\mathcal{F}_{\mathsf{PESTO}}$ which allows to verify (simulated) signatures. For this, we introduced and proved a special property of the DSIG scheme called *share simulatability* (cf. Def. 6), which guarantees the availability of a simulator SIM that produces signature and key shares indistinguishable from real ones. In case $\mathcal{Z}$ sees a "full set" of signature shares, it can test whether they combine to a signature verifying under vk. Right before revealing a final share for such a set, the SIM algorithm requires a valid completed signature as auxiliary input (which allows to set the final share correctly). Fortunately, in case of a corrupted user, $\mathcal{S}$ receives such a signature from $\mathcal{F}_{\mathsf{PESTO}}$ and thus can full advantage of the simulation that SIM provides.

Honest user & corrupted server: The challenge is to simulate messages from an honest user, e.g., attempting to get a signature token for uid, m, towards a corrupted server, without $\mathcal{S}$ knowing her password pw'. However, the simulator (playing the role of corrupt servers towards $\mathcal{F}_{\mathsf{PESTO}}$) learns the necessary information just in time via the `match-ok` output which reveals whether pw' matches the password stored with uid's account. Thus $\mathcal{S}$ knows whether to simulate a successful login or not.

Input Extraction: In UC, the simulator needs to provide inputs of corrupted parties to the ideal functionality. In protocols where inputs are supposed to be hidden — like the passwords in $\mathcal{F}_{\mathsf{PESTO}}$ — this is often challenging to realize, as it requires to extract secrets from adversarially generated messages. In our proof, extraction of a password from a corrupted user's messages is easy due to the usage of the hybrid $\mathcal{F}_{\mathsf{dpOPRF}}$ simulated by $\mathcal{S}$. □

5.2.2. Adding Privacy to Π_{PESTO}. In our protocol, all n servers learn the message m which in the SSO context will contain the identity of the service provider the user wishes to access. Learning this information might be useful as it allows the servers to deploy additional security checks. For example, they could refuse to sign access requests towards blacklisted service providers, or identify

unreasonable access patterns. However, this also means that all servers can track the online behavior of the user.

To increase user's privacy, one could simply let servers sign a commitment $h = H(m, r)$ instead of m, where r is randomly chosen by the user and only revealed in the final signature. Note that signing a commitment to m hides the message in the signing process but is not a full-blown blind signature, as it does not guarantee the unlinkability of the produced signature. However, in our SSO application there is no unlinkability anyway as the servers *must* still learn and sign the username uid along with h to provide the expected user authentication.

6. Implementation & Deployment

We now report on our proof-of-concept implementation of PESTO and further deployment considerations.

Implementation & Benchmarks. Our implementation of PESTO is open-source and available online [1]. The implementation is in Java and uses the MIRACL-AMCL library [46] for pairing computations and Java's BigInteger class for implementing DSIG. We use BLS-461 [12] for the pairings, DSIG is RSA with 2048 bits and for SIG we chose ECDSA with secp256r1. Our hash function of choice is SHA-256, and we used Java's `KeyPairGenerator` class for non-distributed key generation and signing. We chose to use ECDSA instead of RSA for non-distributed signing since its key generation algorithm is significantly faster than RSA. In fact, preliminary numbers showed that the RSA key generation contributed an average of 95 ms to the client execution time, whereas for ECDSA it was at most 1 ms. Furthermore, ECDSA signatures are smaller than RSA signatures so the choice also limits bandwidth usage.

We ran our implementation on AWS Elastic Compute Cloud (EC2) on m5.xlarge instances. That is, machines running on Intel Xeon Platinum 8000 series CPU with 4 virtual CPUs, 16 GiB of RAM[1], solid state storage and with up to a 10 Gbps network. We ran our servers (and user machine) on different data centers throughout Europe to simulate a realistic deployment setting. Alternatively we could have deployed all servers in the same data center on the same physical machine as done in [14]. However we do believe that deploying machines at different physical locations (that could be run by different providers) increases the security significantly to merit the small overhead in runtime from the communication to different data centers.

We show the benchmarks of our Π_{PESTO} protocol in Tab. 1, using a two server setup with one in England, one in Ireland and a user in Germany. We note that the latency of a ping between the user and the slowest server is 22ms. All of our tests were repeated 30 times, after 30 dummy iterations to make sure the JVM is properly "warmed up". Our implementation utilizes a full REST stack with a TLS connection in place through Jetty - which closer emulates the setting we imagine our scheme to be deployed in. This of course gives a penalty on efficiency, which can be seen by comparing the latency with the micro benchmarks.

1. We note that the memory requirements of our implementation is far, far below that offered by the server.

	Throughput (ops/s)	Latency (ms)	
	mean	mean	std
Registration	51	138	8.0
Sign in	60	124	5.8

	dpOPRF		Other local computation			
			Registration		Sign in	
	mean	std	mean	std	mean	std
Server	31.1	6.4	8.3	2.1	1.8	0.6
Client	28.5	1.9	2.2	0.4	1.9	0.7

Table 1: Benchmark of full execution of Π_{PESTO} (upper table), and sub-parts (lower table) for $n = 2$.

We note that our server implementation is single threaded. The reason for not implementing a multi-threaded version is that the bottleneck of the implementation is the underlying computation of pairings and standard signatures, which we did not implement as this has not been the focus of our work. We computed our throughput benchmarks for the *servers only* and excluded network communication for these numbers. Concretely we forced an instance of our program to run on a single core through `taskset` and interpolated the throughput for the amount of available cores. The numbers for latency express the entire time it takes for the user from beginning to end of the protocol, including network latency, establishing a TLS connection and waiting for server replies.

The micro benchmark numbers only include local computation, in particular for the dpOPRF, which is the same for both registration and sign in. It also expresses the other local computation required for the registration, respectively the sign in, in particular key generation, signature verification and distributed signing. The micro benchmarks do not include any of the overhead of the REST/TLS communication stack.

Comparison. For comparison we note that PASTA [8] takes 94.6 ms for sign in when comparing to the most equivalent setting (WAN with 2 servers using an RSA signature). Although for fairness we note that the latency in their WAN setting is 80 ms whereas for us it is 22 ms. However, our implementation also includes constructing a full TLS connection which theirs (to the best of our knowledge) doesn't. For further comparison we also ran benchmarks for the construction of a JWT token (based on RSA) using the our code base. This execution took an average of 30.4 ms. We can thus conclude that PESTO does add some overhead compared to previous schemes, in particular due to the pairing operations. However we believe that the added overhead is still small enough, in absolute terms, for the scheme to be practical, in particular when considering its security benefits over the competition.

SSO Integration. PESTO is designed for "user-centric" SSO authentication flows, i.e., where the IdP sends the time-constrained bearer token to the user who then relays the token to the SP. This specific flow is used by OpenID Connect (OIDC) [44], and is also *one* of the supported flows in the OAuth standard [29], which specifies a general SSO framework. Whereas OAuth supports many different formats for the token, e.g., JSON Web Token

(JWT) or signed XML, OIDC works exclusively with JWT and also specifies the content of the token.

For compatibility with these standards, our implementation focuses on OAuth, with JWTs signed using RSA. Our protocol generates standard JWTs, which we verify in our tests using the Auth0 library. By simply selecting the appropriate claim/value pairs to be included in the JSON message, our implementation becomes fully compatible with OIDC.

Because of these choices, and the fact that our PESTO protocol lets the user combine the signature shares into a standard RSA signature, only the IdPs and the clients need to run PESTO-specific code, whereas the SPs adhere to the normal JWT standard. The user part of the protocol is relatively lightweight: 4 exponentiations in total with only one being in the target group of the bilinear map. As our benchmark shows, this can be done efficiently in Java, and we envision client deployment via, e.g., a simple Android app or browser plugin. The user part could also be implemented entirely in Javascript, but this setup has inherent disadvantages as it requires to trust the code that is fetched on demand.

We also note that other SSO authentication flows in OAuth exist, e.g., where the SP directly contacts the IdP on behalf of the user, or where the IdP sends the signed token directly to the SP, such as in Shibboleth [2]. Such protocols will not be directly compatible with PESTO as there the (distributed) IdPs must send the signed token *directly* to the service provider, thus requiring the SP to run custom code to combine the partial tokens first.

6.1. Deployment Considerations

Finally, we address further considerations for the real-world deployment of our scheme.

Implementing the backup tape. First, for key refreshing our scheme uses "offline" backup tapes that are assumed to be harder to corrupt than the online server itself. There are multiple realizations for this assumption in practice. A simple one is to store a USB-stick containing the respective data in a safe. Each server administrator will then use her stick to run the refresh procedure. Furthermore, modern cloud computing platforms such as Openstack have an architecture that is compatible with the backup tape approach: they strongly separate between the virtual machines that are "online", and the cloud management interfaces that run in a protected environment, which can also function as backup tape.

Password administration. Second, our protocol focuses solely on the account generation and login. In practice, users also want to change or reset their password. Resetting could be done in the classic way by simply sending a password reset link to the stored user's email address. With PESTO this will trigger a new run of the Registration procedure, overwriting the previously stored password information. For changing a password, one could use a PESTO-generated SSO token to let the user first authenticate to all PESTO servers (with the current password) and then establish a new password by re-running the Registration procedure.

Acknowledgements. This work received funding from the EU Horizon 2020 research and innovation programme under grant agreement No 786725 OLYMPUS and the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. We thank Michael Bladt Stausholm for his prototype implementation and performance testing. Most of Anja's work was done while she was at IBM Research – Zurich. Part of the work of Carsten, Avishay and Tore was done while the authors were at Bar-Ilan University.

References

[1] PESTO open source implementation. https://bitbucket.alexandra.dk/projects/OL/repos/pesto.

[2] Shibboleth. https://www.shibboleth.net. Accessed: 2020-03-01.

[3] Security assertion markup language (SAML) v2.0 technical overview. http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html, 2008. Accessed: 2020-03-01.

[4] 117 million linkedin emails and passwords from a 2012 hack just got posted online. https://techcrunch.com/2016/05/18/117-million-linkedin-emails-and-passwords-from-a-2012-hack-just-got-posted-online/, 2016.

[5] Hack brief: 4-year-old dropbox hack exposed 68 million people's data. https://www.wired.com/2016/08/hack-brief-four-year-old-dropbox-hack-exposed-68-million-peoples-data/, 2016.

[6] Forum cracks the vintage passwords of ken thompson and other unix pioneers. https://arstechnica.com/information-technology/2019/10/forum-cracks-the-vintage-passwords-of-ken-thompson-and-other-unix-pioneers/, 2019.

[7] Password data for 2.2 million users of currency and gaming sites dumped online. https://arstechnica.com/information-technology/2019/11/password-data-dumped-online-for-2-2-million-users-of-currency-and-gaming-sites/, 2019.

[8] S. Agrawal, P. Miao, P. Mohassel, and P. Mukherjee. Pasta: Password-based threshold authentication. In *CCS*, pages 2042–2059. ACM, 2018.

[9] J. F. Almansa, I. Damgård, and J. B. Nielsen. Simplified threshold RSA with adaptive and proactive security. In *EUROCRYPT*, pages 593–611, 2006.

[10] A. Bagherzandi, S. Jarecki, N. Saxena, and Y. Lu. Password-protected secret sharing. In *CCS*, pages 433–444, 2011.

[11] C. Baum, T. K. Frederiksen, J. Hesse, A. Lehmann, and A. Yanai. PESTO: Proactively secure distributed single sign-on, or how to trust a hacked server. Cryptology ePrint Archive, Report 2019/1470, 2019. https://eprint.iacr.org/2019/1470.

[12] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[13] J. Camenisch, A. Lehmann, A. Lysyanskaya, and G. Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In *CRYPTO*, pages 256–275, 2014.

[14] J. Camenisch, A. Lehmann, and G. Neven. Optimal distributed password verification. In *CCS*, pages 182–194. ACM, 2015.

[15] J. Camenisch, A. Lysyanskaya, and G. Neven. Practical yet universally composable two-server password-authenticated secret sharing. In *CCS*, pages 525–536, 2012.

[16] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. *ePrint*, 2000:67, 2000.

[17] R. Canetti. Universally composable signature, certification, and authentication. In *(CSFW-17*, page 219, 2004.

[18] R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Adaptive security for threshold cryptosystems. In *CRYPTO*, pages 98–115, 1999.

[19] R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT*, pages 404–421, 2005.

[20] L. Chen, Z. Cheng, and N. P. Smart. Identity-based key agreement protocols from pairings. *International Journal of Information Security*, 6(4):213–241, 2007.

[21] A. Everspaugh, R. Chaterjee, S. Scott, A. Juels, and T. Ristenpart. The pythia prf service. In *USENIX*, pages 547–562, 2015.

[22] W. Ford and B. S. K. Jr. Server-assisted generation of a strong secret from a password. In *(WETICE*, pages 176–180, 2000.

[23] Y. Frankel. A practical protocol for large group oriented networks. In *EUROCRYPT*, pages 56–61, 1989.

[24] Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung. Optimal resilience proactive public-key cryptosystems. In *FOCS*, pages 384–393, 1997.

[25] Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung. Proactive RSA. In *CRYPTO*, pages 440–454, 1997.

[26] Y. Frankel, P. Gemmell, and M. Yung. Witness-based cryptographic program checking and robust function sharing. In *STOC*, pages 499–508, 1996.

[27] Y. Frankel, P. D. MacKenzie, and M. Yung. Adaptive security for the additive-sharing based proactive RSA. In *PKC*, pages 240–263, 2001.

[28] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust and efficient sharing of RSA functions. In *CRYPTO*, pages 157–172, 1996.

[29] D. Hardt. The OAuth 2.0 authorization framework. Technical report, 2012.

[30] D. P. Jablon. Password authentication using multiple servers. In *CT-RSA*, pages 344–360, 2001.

[31] S. Jarecki, A. Kiayias, and H. Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT*, pages 233–253, 2014.

[32] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *EuroS&P*, pages 276–291, 2016.

[33] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. Toppss: Cost-minimal password-protected secret sharing based on threshold oprf. In *ACNS*, pages 39–58. Springer, 2017.

[34] S. Jarecki, H. Krawczyk, and J. K. Resch. Threshold partially-oblivious prfs with applications to key management. ePrint Archive 2018/733, 2018.

[35] S. Jarecki, H. Krawczyk, and J. Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, pages 456–486, 2018.

[36] S. Jarecki and J. Olsen. Proactive RSA with non-interactive signing. In *FC*, pages 215–230, 2008.

[37] S. Jarecki and N. Saxena. Further simplifications in proactive RSA signatures. In *TCC*, pages 510–528, 2005.

[38] M. Jones, J. Bradley, and N. Sakimura. JSON web token (JWT). Technical report, 2015.

[39] P. D. MacKenzie, T. Shrimpton, and M. Jakobsson. Threshold password-authenticated key exchange. In *CRYPTO*, pages 385–400, 2002.

[40] K. O'Flaherty. Collection 1 breach – how to find out if your password has been stolen. *Forbes*, 2019.

[41] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *PODC*, pages 51–59, 1991.

[42] T. Rabin. A simplified approach to threshold and proactive RSA. In *CRYPTO*, pages 89–104, 1998.

[43] M. D. Raimondo and R. Gennaro. Provably secure threshold password-authenticated key exchange. In *EUROCRYPT*, pages 507–523, 2003.

[44] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID connect core 1.0 incorporating errata set 1. https://openid.net/specs/openid-connect-core-1_0.html, 2014. Accessed: 2020-03-01.

[45] A. D. Santis, Y. Desmedt, Y. Frankel, and M. Yung. How to share a function securely. In *STOC*, pages 522–533, 1994.

[46] M. Scott, K. McCusker, A. Budroni, and S. Andreoli. The miracl core cryptographic library. https://github.com/miracl/core, 2019.

[47] M. Szydlo and B. S. K. Jr. Proofs for two-server password authentication. In *CT-RSA*, pages 227–244, 2005.

Appendix A.
Proof of Theorem 1 (Security of RSA-DSIG)

The proof mainly relies on using "noise drowning" techniques that hide a value s using an added x that comes from an exponentially larger interval. This can be formalized as follows:

Lemma 1. *Let $\ell, B, \beta \in \mathbb{N}$ such that $B > \beta$. Furthermore, let χ be a random variable on $\mathbb{Z}$ whose outputs are bounded by β. Then the following holds:*

1) *Let $x \leftarrow [-B, B]$ and $s \leftarrow \chi$. Then any (potentially computationally unbounded) algorithm $\mathcal{A}$ can distinguish x from $x+s$ with probability at most $\frac{4s+4}{2B+2s+1}$.*
2) *Let $x_1, \ldots, x_\ell \leftarrow [-B, B]$ and $s \leftarrow \chi$. Then any (potentially computationally unbounded) algorithm $\mathcal{B}$ can distinguish $\sum_{i\in[\ell]} x_i$ from $s + \sum_{i\in[\ell]} x_i$ with probability at most $\frac{4s+4}{2B+2s+1}$.*

Proof. Consider the random variable X being input to $\mathcal{A}$ which is either sampled as x or $x+s$. We can see that X must be bounded by $B+s$. By the bound on s it holds that $\Pr[X \in [-B+\beta, B-\beta] \mid X = x+s] = \Pr[X \in [-B+\beta, B-\beta] \mid X = x]$, so in that case the output of $\mathcal{A}$ on X must be identically distributed. Thus $\mathcal{A}$ can only distinguish the cases when X falls outside this interval, which happens with probability at most $\frac{4s+4}{2B+2s+1}$.

Assume that there exists an algorithm $\mathcal{B}$ which can distinguish the second experiment with probability at least ϵ. Then, knowing $\mathcal{B}$, we can take the random variable X as defined above for the first experiment for $\mathcal{A}$, sample $x_2, \ldots, x_\ell \leftarrow [-B, B]$ and set $Y \leftarrow X + x_2 + \cdots + x_\ell$, which implies that the distribution of Y exactly matches the input distribution from the second experiment - here, $X = x$ translates into the setting where $Y = x + 1 + \cdots + x_\ell$ while $X = x + s$ translates into $Y = x_1 + \cdots + x_\ell + s$. Thus, we have constructed a distinguisher for X which succeeds with the same probability ϵ. By the aforementioned argument, we must have $\epsilon \leq \frac{4s+4}{2B+2s+1}$. □

Proof. The aforementioned Lemma can now be used in the proof of Theorem 1, which we present below for the individual security properties.

Signature Indistinguishability. Algorithm $\mathsf{DSIG.Comb}^*(pk, sk_1, \ldots, sk_n)$ parses $(f_i, \{s_{i,j}\}_{j\in[n]\setminus\{i\}}) \leftarrow sk_i$ and computes $sk = \sum_{i=1}^{n} f_i \bmod N$. Algorithm $\mathsf{DSIG.Sign}^*(sk, m)$ outputs $H(m)^{sk} \bmod N$. Now, since the $\{f_i\}_i$ form an additive sharing of $sk = d$ over the integers, the result of running $\mathsf{DSIG.Comb}^*$ on the secret keys to produce a single secret key and then $\mathsf{DSIG.Sign}$ using that resulted secret key is equivalent to running $\mathsf{DSIG.Sign}$ individually with every sk_i to produce partial signatures and then running $\mathsf{DSIG.Comb}$ on the partial signatures. The outputs of both processes are identical by the linearity in the exponent.

Share Simulatability. We now describe the simulator SIM and will afterwards argue indistinguishability.

Upon first call SIM will sample "shadow" $\overline{sk}_i^0, \overline{bk}_i^0$. It therefore generates both keys according to Steps 3-6 of $\mathsf{DSIG.KGen}$, except that SIM will not add d to d_n and not add d_i to f_i.

On query $(\mathsf{Sign}, m, \ell, \sigma)$, if $\sigma = \perp$ then use $\overline{sk}_i^{\mathsf{ep}}$ to generate σ_i honestly. If $\sigma \neq \perp$ then output $\sigma/(\prod_{j\in[n]\setminus\{i\}} \sigma_j)$ where $\sigma_j = \mathsf{DSIG.Sign}(j, \overline{sk}_j^{\mathsf{ep}}, m, \ell)$.

On query refresh, for all $i \in [n]$ run $\mathsf{DSIG.Refresh}(i, \overline{bk}_i^{\mathsf{ep}}) \rightarrow (\overline{sk}_i^{\mathsf{ep}+1}, \overline{bk}_i^{\mathsf{ep}+1})$ on all keys that SIM generated.

On query $(\mathsf{corr}, mode, i, \mathcal{L})$, with $mode \in \{\texttt{trans}, \texttt{perm}\}$, the list $\mathcal{L}$ contains all those signatures of the current epoch ep for which SIM has to 'adjust' the output of the random oracle $\tilde{H}_N$ for key $\overline{sk}_i^{\mathsf{ep}}$. This is done as follows: For each row $(\mathsf{ep}, i, m, \ell, \sigma) \in \mathcal{L}$, we already have that the equation $\sigma = \prod_{j\in[n]} \sigma_j$ holds, but we need to adjust the computation of σ_i appropriately as we need to reveal to the adversary either $\overline{sk}_i^{\mathsf{ep}}$ such that $\sigma_i = H(m)^{f_i} \cdot \prod_{j\neq i} \tilde{H}_N(\ell, s_{i,j}) = \sigma/\big(\prod_{j\neq i} \sigma_j\big)$ or $\overline{bk}_i^{\mathsf{ep}}$ with the same properties. Therefore, we can pick the smallest $k \in [n]\setminus(\mathcal{C}_{p,\mathsf{ep}} \cup \mathcal{C}_{t,\mathsf{ep}})$ and program $\tilde{H}_N$ as $\tilde{H}_N(\ell, s_{i,k}) = \sigma/\left(\left(\prod_{j\neq i} \sigma_j\right) \cdot H(m)^{f_i} \cdot \prod_{j\neq i,k} \tilde{H}_N(\ell, s_{i,j})\right) \bmod N$. Observe that such a k must always exist unless i is the last uncorrupted party, in which case SIM will only output $sk_i, bk_i = \perp$. If the random oracle has been programmed on $\tilde{H}_N(\ell, s_{i,k})$ already then by linearity of the above expression it must have been programmed to the same value.

We now argue why SIM's output is indistinguishable. First, observe that we only program the random oracle on inputs where the key $s_{i,j}$ has not yet been revealed before. Therefore, we in the following assume that the adversary has not queried $\tilde{H}_N$ on this input before since $s_{i,j}$ has length τ bits and is either chosen uniformly at random or as the output of a PRG on a (by assumption) uniformly random input of the same length.

In the generation of the $\overline{bk}_i$ we generate $mk_{i,j}$ as before, but now sample $\overline{d}_n$ differently, namely without adding d. By Lemma 1 we have that the distributions of d_n are statistically indistinguishable due to the gap on the bounds of d, d_n.

Furthermore, by Lemma 1 it also follows that each $\overline{sk}_i$ is statistically indistinguishable from its real counterpart generated by KGen, due to the gap of the bound of d_i and f_i.

For the generation of the σ_i we have that due to the factor h_i they are distributed uniformly at random in the protocol, subject to the constraint that their product is σ which is preserved by SIM.

When corrupting the last party, SIM will output $\perp$ whereas the real experiment would output the correct keys. Observe that in such a setting $\mathcal{A}$ cannot win the experiment anymore, so the output of SIM can be arbitrary.

Unforgeability. First, consider the standard RSA signature scheme SIG_1 which generates $\sigma(m) \leftarrow H(m)^d$ and verifies by checking that $H(m) = \sigma^e$. Here $sk = d$ and $pk = (e, N)$ and d is also chosen such that $d \cdot e = 1 \bmod \varphi(N)$.

We show that the existence of an adversary $\mathcal{A}_n$ such that $\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A}_n,\mathsf{forge}}(\tau) = 1$ implies an adversary $\mathcal{A}_1$ such that $\mathsf{Exp}^{\mathsf{DSIG},1}_{\mathcal{A}_1,\mathsf{forge}}(\tau) = 1$ with essentially the same success probability. That is, the existence of an adversary who forges a signature in a setting where there are n signers implies the existence of an adversary in a setting with only one signer. Now, since the case with one signer is identical to a standard signature scheme SIG, this means that by existential unforgeability of SIG implies existential unforgeability of DSIG.

$\mathcal{A}_1$ participates in experiment $\mathsf{Exp}^{\mathsf{DSIG},1}_{\mathcal{A}_1,\mathsf{forge}}(\tau)$ from which it obtains the public key $pk_{\mathsf{own}} = (N, e)$ and runs $\mathcal{A}_n$ internally such that $\mathcal{A}_n$ participates in a simulation of $\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A}_n,\mathsf{forge}}(\tau)$. To do so, $\mathcal{A}_1$ simulates $\mathsf{DSIG.KGen}(pp, n)$ as in the proof of share simulatability to obtain (sk_i, bk_i) for all $i \in [n]$ where $sk_i = (f_i, \{s_{i,j}\}_{j\in[n]\setminus\{i\}})$ and $bk_i = (d_i, \{mk_{i,j}\}_{j\in[n]\setminus\{i\}})$ and public key $pk = pk_{\mathsf{own}}$.

$\mathcal{A}_1$ initializes $\mathsf{ep} \leftarrow 0$, $\mathcal{C}_{t,\mathsf{ep}} \leftarrow \emptyset$, $\mathcal{C}_{p,\mathsf{ep}} \leftarrow \emptyset$ and forwards pk to $\mathcal{A}_n$. In addition, $\mathcal{A}_1$ simulates the oracles responses to $\mathcal{A}_n$ by executing their code exactly as it appears in Figure 3 except in the case in which the requested signature is from the only party remaining for which the adversary does not have a signature already (or a potential to generate one). In that case $\mathcal{A}_1$ would generate the output as in SIM.

Thus, the view of $\mathcal{A}_n$ in the internal simulation is distributed identically to its view in a real experiment assuming share simulatability, meaning that $\mathcal{A}_n$ wins in $\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A}_n,\mathsf{forge}}(\tau)$ with the same probability that it wins in the simulation, meaning that $\mathcal{A}_1$ wins $\mathsf{Exp}^{\mathsf{DSIG},1}_{\mathcal{A}_1,\mathsf{forge}}(\tau)$ with the same probability as $\mathcal{A}_n$. □

Appendix B. Proof of Theorem 2 (Security of DH-dpOPRF)

Informal proof. We show security through a hybrid argument with indistinguishability between a real execution and the ideal execution through a simulator which is allowed to "fail" in certain cases. We then modify the simulator such that it uses an instance of Gapom-BDH internally, arguing indistinguishability between the first and second simulator and then showing that if "fail" occurs then the adversary has actually been able to win the experiment. We note that we also allow the simulator to abort, but argue that it will only be allowed to do so in events that have negligible probability. Concretely, only if the adversary is able to either guess or find a collision on random oracle queries on inputs of at least τ bits of entropy.

We start by constructing our simulator to run simulated versions of users and each S_i for $i \in [n]$ until the real world adversary $\mathcal{A}$ corrupts one. Let the set of corrupt parties be denoted by $\mathcal{C} := \mathcal{C}_t \cup \mathcal{C}_p$ and $\bar{\mathcal{C}} := [n] \setminus \mathcal{C}$. Because of the proactive security the size and parties in $\mathcal{C}$ and $\bar{\mathcal{C}}$ can change.

Key Generation: $\mathcal{S}$ simulates the trusted third party and constructs the long term keys $(k_i, \{mk_{i,j}\}_{j\in[n]\setminus\{i\}})$ along with epoch keys $\{s_{i,j}\}_{j\in[n]\setminus\{i\}}$ for all $i \in [n]$ s.t. $mk_{i,j} = mk_{j,i}$ and $s_{i,j} = s_{j,i}$. Define $k = \sum_{i\in[n]} k_i \bmod p$.

Random Oracles: Every oracle stores a table of each query which is used to answer repeat queries.

A new query $H_1(x_{\mathsf{priv}})$ is answered by the value $\tilde{x} \leftarrow g_1^v$ for $v \leftarrow_{\mathrm{R}} \mathbb{Z}_p$. The simulator then stores $\langle H_1, x_{\mathsf{priv}}, v, \tilde{x}\rangle$.

A new query $H_2(x_{\mathsf{pub}})$ is answered by the value $c \leftarrow g_2^w$ for $w \leftarrow_{\mathrm{R}} \mathbb{Z}_p$. The simulator then stores $\langle H_2, x_{\mathsf{pub}}, w, c\rangle$.

A new query $H_3(sid, ssid, s_{i,j})$ is answered by a random $d_{i,j} \leftarrow_R \mathbb{Z}_p$. The simulator then stores $\langle H_3, d_{i,j}, sid, ssid, s_{i,j}\rangle$.

A new query to H_4 on input $(x_{\mathsf{priv}}, x_{\mathsf{pub}}, \bar{y})$ is handled as follows: Check if tuples $\langle H_1, x_{\mathsf{priv}}, v, \tilde{x}\rangle$ and $\langle H_2, x_{\mathsf{pub}}, w, c\rangle$ exist. If not, pick and return a random value $y \leftarrow_R \{0,1\}^\tau$, otherwise look up all tuples $\langle \mathsf{lbl}_l, *, *, \tilde{y}_l\rangle$, then check if $\tilde{y}_l = \bar{y}^{1/(vw)}$ for any l and abort if it holds for multiple l's. If there is such an index then let $\mathsf{lbl} \leftarrow \mathsf{lbl}_l$, otherwise pick a new random label lbl. Sample a new $ssid'$ and call $(\texttt{Eval}, sid, ssid', x_{\mathsf{pub}}, x_{\mathsf{priv}})$ and delay the message $(\texttt{Eval}, sid, ssid', x_{\mathsf{pub}})$ otherwise returned to all honest parties by the environment. Then call $(\texttt{EvalComplete}, sid, ssid, \mathsf{lbl})$ to get the message $(\texttt{EvalComplete}, sid, ssid, y)$ and learn $y \leftarrow T(\mathsf{lbl}, x_{\mathsf{pub}}, x_{\mathsf{priv}})$. $\mathcal{S}$ then stores $\langle \mathsf{lbl}, \tilde{x}, c, \bar{y}\rangle$. Return y as output to the query. In case the ideal functionality does not return $(\texttt{EvalComplete}, sid, ssid, y)$ from the call, then $\mathcal{S}$ outputs fail.

The simulator aborts if a collision is detected for any of the oracles.

The idea is that H_4 will verify consistent malicious behavior by checking if consistent keys (exponent) have been used by an adversary. If so, the ideal functionality will return a previously stored result, if not the ideal functionality will return a new result on a different label. It will later become apparent that fail *will only occur when the adversary has not received shares from all "honest" parties.*

Evaluation: We consider two scenarios: the querying party U is honest and the querying party U is dishonest.

U **is honest:** In this setting the simulation starts when $\mathcal{S}$ receives $(\texttt{Eval}, sid, ssid, U, x_{\mathsf{pub}})$ from the ideal functionality and it must then simulate the query of an honest U. It does so by emulating a query to H_1, thus constructing the value $\tilde{x} \leftarrow g_1^v$ for $v \leftarrow_R \mathbb{Z}_p$ and storing $\langle H_1, \cdot, v, \tilde{x}\rangle$. It then sends $(\texttt{Eval}, sid, ssid, x_{\mathsf{pub}}, \tilde{x})$ to each $S_i \in \mathcal{C}$. When $\mathcal{S}$ has received $(\texttt{EvalComplete}, sid, ssid, S_i, \bar{y}_i)$ from $\mathcal{A}$ for all $S_i \in \mathcal{C}$ and $(\texttt{EvalProceed}, sid, ssid, S_i)$ from the ideal functionality on behalf of $S_i \in \bar{\mathcal{C}}$ it must ask the ideal functionality to give output to the honest user which must be consistent with previous queries in case the adversary has used different keys for the malicious servers. It does so by simulating the honest servers' part of the computation by computing $\bar{y} = \left(\prod_{S_i \in \mathcal{C}} \bar{y}_i\right) \cdot \left(\prod_{S_i \in \bar{\mathcal{C}}} \cdot \prod_{S_j \in \mathcal{C}} d_{i,j}^{\Delta_{i,j}}\right) \cdot \left(\prod_{S_i \in \bar{\mathcal{C}}} e(\tilde{x}, c)^{k_i}\right)$ where $\Delta_{i,j} = 1$ if $i > j$ and $\Delta_{i,j} = -1$ otherwise. The values $d_{i,j}$ are found by looking up $\langle H_3, d_{i,j}, sid, ssid, s_{i,j}\rangle$. (If they don't exist or $\langle H_2, x_{\mathsf{pub}}, w, c\rangle$ does not exist then emulate the oracle calls.) If the adversary has acted honestly (or is lucky) then $\bar{y}^{1/(vw)} = e(\tilde{x}, c)^{v^{-1}w^{-1}\cdot\sum_{i\in[n]} k_i} = e(g_1, g_2)^{\sum_{i\in[n]} k_i}$. If that is the case we let $\mathsf{lbl} = \mathsf{hon}$, otherwise we look up the tuples $\langle \mathsf{lbl}_l, *, *, \tilde{y}_l\rangle$ and check if $\tilde{y}_l = \bar{y}^{1/(vw)}$. If we find a label lbl_l where this holds let $\mathsf{lbl} \leftarrow \mathsf{lbl}_l$, otherwise let lbl be a new random label, then call the ideal functionality with $(\texttt{EvalComplete}, sid, ssid, \mathsf{lbl}_l)$. Receive back $(\texttt{EvalComplete}, sid, ssid, y)$ and store $\langle \mathsf{lbl}, \tilde{x}, c, \bar{y}^{1/(vw)}\rangle$.[2]

U **is dishonest:** The message $(\texttt{Eval}, sid, ssid, x_{\mathsf{pub}}, \bar{x})$ is received by $\mathcal{A}$ on behalf of U towards some $S_i \in \bar{\mathcal{C}}$. Look up tuple $\langle H_2, x_{\mathsf{pub}}, w, c\rangle$. If it doesn't exist, construct it. Pick a random dummy value $x_{\mathsf{priv}} \leftarrow_R \{0,1\}^\tau$ then $\mathcal{S}$ sends the query $(\texttt{Eval}, sid, ssid, x_{\mathsf{pub}}, x_{\mathsf{priv}})$ to the ideal functionality and asks it to give delayed output $(\texttt{Eval}, sid, ssid, x_{\mathsf{pub}})$ to S_i.
Each time $\mathcal{S}$ receives $(\texttt{EvalProceed}, sid, ssid, S_i)$ from a $S_i \in \bar{\mathcal{C}}$ from the ideal functionality it ignores it if it cannot find $(\texttt{eval}, sid, ssid, x_{\mathsf{pub}}, \bar{x})$ for this $ssid$ and S_i. Otherwise it looks up the values $\langle H_2, x_{\mathsf{pub}}, w, c\rangle$ and $\langle H_3, d_{i,j}, sid, ssid, s_{i,j}\rangle$ (and emulating the construction of these if they don't exist), computes the value $\bar{y}_i$ like in the real protocol. Send $(\texttt{EvalComplete}, sid, ssid, S_i, \bar{y}_i)$ to $\mathcal{A}$ as values returned to U from honest S_i. Note that the rest of the simulation for dishonest U is handled by calls to H_4.

Refresh: During refresh the simulator computes $(mk'_{i,j}, s'_{i,j}, \delta_{i,j}) \leftarrow \mathsf{PRG}(mk_{i,j})$ for all $j \in [n] \setminus \{i\}$ for all permanently corrupted servers where $mk_{i,j}$ was the master key given to the adversary when permanently corrupting the server S_i. For the honest, and previously transiently corrupted servers it picks $s'_{i,j}, \delta_{i,j}$ uniformly at random and computes the new state for the honest parties like in the real protocol.

Corruption: When $\mathcal{A}$ transiently corrupts a server S_i then $\mathcal{S}$ can hand over current epoch keys and transcript as it knows the entire epoch state.

If an adaptive corruption happens of a user U then the simulator must explain the user's true input, x_{priv}, and choice of nonce r' where the simulator previously sent a value $\tilde{x}$. See if there is an entry $\langle H_1, x_{\mathsf{priv}}, v, g_1^v\rangle$, otherwise construct such an entry for x_{priv}. Then construct a simulated nonce as $r' \leftarrow r/v$ and use this to return to $\mathcal{A}$, thus $\bar{x} = H_1(x_{\mathsf{priv}})^{r'} = g_1^{v\cdot r/v} = g_1^r$.

When $\mathcal{A}$ permanently corrupts a server S_i then $\mathcal{S}$ picks $mk_{i,j} \leftarrow_R \{0,1\}^\tau$ (consistent with previously picked keys) and use this to simulate the construction of the current epoch keys, $s_{i,j}$. See that the backup keys $mk_{i,j}$ don't define the keys of the current and previous epoch keys, $s_{i,j}$ because of the generation through the PRG. Thus the epoch keys used in the simulation for the previous epochs can easily be given to the adversary as transcript.

Indistinguishability: It is easy to see that the random oracles all return random values and that the key generation is done like in the real execution. When U is dishonest then everything sent to $\mathcal{A}$ is computed like in the real execution. There are a few places where the simulator emulates the oracles. Emulated calls to H_2 do not pose a problem since they are queried on the public value x_{pub} and thus whether we define the output when an adversary

2. Note that the ideal functionality will always return, even if $\mathsf{lbl} = \mathsf{hon}$.

performs a query or the first time the simulator needs it does not matter. In regards to H_3 we notice that this is again only emulated for a query the adversary knows how to issue, based on $s_{i,j}$.

Finally notice that in relation to oracle H_4 the simulator will only query the ideal functionality in case of a corrupt user and thus we don't have the problem of the simulator's query being leaked to the environment through the output to an honest user. In particular this is so since if $\mathcal{A}$ queries H_4 without corrupting the user then even if $\mathcal{A}$ uses the correct x_{priv} (given by the environment) it cannot return a correctly computed value $\bar{y}$ since it does not know the honest user's randomization exponent, r, which it cannot guess with non-negligible probability. Furthermore we see that if U has been corrupt then the only time where the ideal functionality will not return $(\texttt{EvalComplete}, sid, ssid, y)$ to the simulator is if $(\texttt{EvalProceed}, sid, ssid)$ has not been received from all the honest S_i for the same amount of concurrent queries of U on x_{pub} as the adversary has started. This is is the only time fail will be output.

Reduction to the Gap om-BDH Problem: Suppose we are given a PPT adversary $\mathcal{A}$ and a PPT environment $\mathcal{Z}$ that can cause fail to happen. We now show how to use these to construct an algorithm $\mathcal{B}$ that solves the gap om-BDH problem with essentially the same probability. $\mathcal{B}$ is given $\mathbb{G}$ and g_2^k from the Gapom-BDH experiment and also access to the oracles $\mathcal{O}_{\mathbb{G}\text{-}1}, \mathcal{O}_{\mathbb{G}\text{-}2}$, $\mathcal{O}_{\mathsf{D\text{-}help}}$ and $\mathcal{O}_{\mathsf{C\text{-}help}}$. We use this to implicitly define keys k_i s.t. $k = \sum_{i\in[n]} k_i \mod p$. However, since we don't know k we can at most explicitly define $n-1$ of these. Each of these along with the blinding seeds, $s_{i,j}$ will only be fixed once a party gets corrupted. Thus we avoid having $\mathcal{B}$ needing to guess a server that will remain uncorrupted in each epoch. The key observation in the following is then to notice that $\mathcal{B}$ never needs to simulate values for all k_i, because otherwise the fail event will not occur since $\mathcal{B}$ will need to be missing at least one share $\bar{y}_i$ from an honest server to make it occur.

Key Generation is handled the same as before. However at the beginning we also store the tuple $\langle \mathsf{hon}, g_1, g_2, e(g_1, g_2^k)\rangle$ For the random oracles we now use the $\mathcal{O}_{\mathbb{G}\text{-}1}, \mathcal{O}_{\mathbb{G}\text{-}2}$ oracles to sample random elements for H_1 and H_2, $\tilde{x}$, respectively c. $\mathcal{B}$ then stores tuples $\langle H_1, x_{\mathsf{priv}}, \cdot, \tilde{x}\rangle$ and $\langle H_2, x_{\mathsf{pub}}, \cdot, c\rangle$ now, since it does not know the discrete logarithm of $\tilde{x}$, respectively c.

A new query to H_4 on input $(x_{\mathsf{priv}}, x_{\mathsf{pub}}, \bar{y})$ is now handled as follows: $\mathcal{B}$ checks if tuples $\langle H_1, x_{\mathsf{priv}}, \cdot, \tilde{x}\rangle$ and $\langle H_2, x_{\mathsf{pub}}, \cdot, c\rangle$ exist. If not, pick and return a random value $y \leftarrow_{\mathrm{R}} \{0,1\}^\tau$, otherwise look up all tuples $\langle \mathsf{lbl}_l, \tilde{x}_l, c_l, \tilde{y}_l\rangle$, and query $1 \leftarrow^? \mathcal{O}_{\mathsf{D\text{-}help}}(e(\tilde{x}, c), \bar{y}, e(\tilde{x}_l, c_l), \tilde{y}_l)$ to try to find an index l where this holds. If that happens then let $\mathsf{lbl} \leftarrow \mathsf{lbl}_l$, otherwise pick a new random label lbl. Sample a new $ssid'$ and call $(\texttt{Eval}, sid, ssid', x_{\mathsf{pub}}, x_{\mathsf{priv}})$ and delay the message $(\texttt{Eval}, sid, ssid', x_{\mathsf{pub}})$ otherwise returned to all honest parties by the environment. Then call $(\texttt{EvalComplete}, sid, ssid, \mathsf{lbl})$ to get the message $(\texttt{EvalComplete}, sid, ssid, y)$ and learn $y \leftarrow T(\mathsf{lbl}, x_{\mathsf{pub}}, x_{\mathsf{priv}})$. $\mathcal{B}$ then stores $\langle \mathsf{lbl}, \tilde{x}, c, \bar{y}\rangle$. Return y as output to the query. In case the ideal functionality does not return $(\texttt{EvalComplete}, sid, ssid, y)$ from the call, then $\mathcal{S}$ outputs fail and adds $(\tilde{x}, c, \bar{y})$ to the list of solutions to the experiment.

Note that we are here able to complete the check on the stored tuples without knowing the discrete log of $\tilde{x}$ and c because we have access to the DDH oracle instead. We see that the check will verify if there is any tuple stored with the same exponent to the query $e(\tilde{x}, c)$, which should be the base if $\bar{y}$ is constructed correctly. Furthermore we see that we can also detect when this occurs for the true key k because we have stored a tuple for the base $e(g_1, g_2)$ since we got g_2^k from the experiment.

Now consider the simulation for evaluation. Again we have two cases; U is honest and U is dishonest.

U **is honest:** Simulate like before except for when receiving the message $(\texttt{EvalComplete}, sid, ssid, S_i, \bar{y}_i)$ from the adversary. In that case do as before when constructing the variable $\bar{y}$. However since we don't have the honest parties' key shares, instead use the oracle $e(\tilde{x}, c)^k \leftarrow \mathcal{O}_{\mathsf{C\text{-}help}}(e(\tilde{x}, c))$. Add $(\tilde{x}, c, e(\tilde{x}, c)^k)$ to the list of solutions to give to the Gapom-BDH experiment. Next compute $\bar{y} = \left(\prod_{i\in\mathcal{C}} \bar{y}_i\right) \cdot \left(\prod_{i\in\bar{\mathcal{C}}} \cdot \prod_{j\in\mathcal{C}} d_{i,j}^{\Delta_{i,j}}\right) \cdot \left(e(\tilde{x}, c)^k\right)^{-\sum_{i\in\mathcal{C}} k_i}$. Then look up the tuples $\langle \mathsf{lbl}_l, \tilde{x}_l, c_l, \tilde{y}_l\rangle$ and use the $\mathcal{O}_{\mathsf{D\text{-}help}}$ to compute $1 \leftarrow^? \mathcal{O}_{\mathsf{D\text{-}help}}(e(\tilde{x}, c), \tilde{y}, e(\tilde{x}_l, c_l), \tilde{y}_l)$. If we find a label lbl_l where this holds let $\mathsf{lbl} \leftarrow \mathsf{lbl}_l$, otherwise let lbl be a new random label, then call the ideal functionality with $(\texttt{EvalComplete}, sid, ssid, \mathsf{lbl}_l)$ and then stores $\langle \mathsf{lbl}, \tilde{x}, c, \bar{y}\rangle$. The user U then receives back $(\texttt{EvalComplete}, sid, ssid, y)$ from the ideal functionality.

U **dishonest:** On receiving $(\texttt{EvalProceed}, sid, ssid, S_i)$ from all, except the last honest party $S_i \in \bar{\mathcal{C}}$, simply return $(\texttt{EvalComplete}, sid, ssid, S_i, \bar{y}_i)$ for $\bar{y}_i \leftarrow_{\mathrm{R}} \mathbb{G}_T$. We see that this is fine, because there will be at least one value $d_{i,j}$ unknown in this case. For the last honest party we need to make things consistent. To do so use the oracle $e(\bar{x}, c)^k \leftarrow \mathcal{O}_{\mathsf{C\text{-}help}}(e(\bar{x}, c))$ using $\bar{x}$ already received and c from the H_2 table. Based on this compute $\bar{y}_i \leftarrow \left(e(\bar{x}, c)^k\right)^{-\sum_{j\in\mathcal{C}} k_j} \cdot \left(\prod_{j\in\mathcal{C}} d_{i,j}^{\Delta_{i,j}}\right)^{-1} \cdot \prod_{i\in\bar{\mathcal{C}}} \bar{y}_i$ and return $(\texttt{EvalComplete}, sid, ssid, S_i, \bar{y}_i)$. Furthermore add $(\bar{x}, c, e(\bar{x}, c)^k)$ to the list of solutions to give to the experiment Gapom-BDH. Note that multiple values for $\bar{x}$ may have been received. In that case use the one sent to the last honest party.

Corruption and refresh: When $\mathcal{A}$ corrupts S_i then $\mathcal{B}$ chooses a random key k_i and blinding seeds $\{s_{i,j}\}_{j\in[n]\setminus\{i\}}$ (unless already defined) at random. It then defines the outputs of oracle H_3 to match the simulated random values sent, that is to ensure that $\bar{y}_i = e(\bar{x}, c)^{k_i} \cdot \prod_{j\in[n]\setminus\{i\}} H_3(sid, ssid, s_{i,j})^{\Delta_{i,j}}$ (if U was dishonest of that specific query). This is trivial as long as there is at least one $s_{i,j}$ unknown to the adversary, i.e. if there are at least two honest parties. If there is only one honest party remaining then notice that its value $\bar{y}_i$ (when U is corrupt) was defined such that it is consistent with the random values picked for all the other honest parties. Next see that if we need to simulate the response for honest servers when U was honest (that is when the simulator didn't construct the values $\bar{y}_i$) the simulator can again simply

hand out random values as long as one $s_{i,j}$ is unknown and then for the last values simulate like in the case when U is dishonest, using the value $\bar{y}$ already defined internally in the simulator.

If $\mathcal{A}$ permanently corrupts S_i then $\mathcal{B}$ additionally chooses random master keys $mk_{i,j}$ for $j \in [n] \setminus \{i\}$ to simulate the backup tape. When a transiently corrupted server is refreshed then $\mathcal{B}$ takes back control and forget all previously chosen keys for that server.

CDH Solutions: When fail occurs then $\mathcal{B}$ adds one more CDH solution to the list of solutions than the number of times it invoked the oracle. Specifically note that the oracle is only invoked when $\mathcal{B}$ must simulate values from all honest servers (when the message $(\texttt{EvalProceed}, sid, ssid)$ is given to $\mathcal{B}$ by the ideal functionality on request from $\mathcal{Z}$). There $\mathcal{B}$ wins the experiment by returning a list of values that wins the Gapom-BDH experiment. □

Appendix C. Proof of Theorem 3 (Security of PESTO)

We restrict the analysis to static user corruptions, although our protocol is adaptively secure. This works because users delete all secret values after usage (e.g., y and usk), and the adversary does not gain any advantage from corrupting a user ITI after the protocol started. For this, note that party identifiers are meaningless besides providing routing information, and thus we can assume that $\mathcal{Z}$ chooses a fresh user party for every fresh subsession. Therefore, statically corrupting a user who attempts to log in with a previously honestly uid naturally models adaptive user corruptions.

Game 1: Real execution. This is the real protocol execution running with adversary $\mathcal{A}$ and hybrid functionality $\mathcal{F}_{\mathsf{dpOPRF}}$.

Game 2: Adding $\mathcal{F}$ with `KeyGen` and `Register`. We regroup the real execution into one ITI called simulator or $\mathcal{S}$ ($\mathcal{S}$ now includes $\mathcal{A}, \mathcal{F}_{\mathsf{dpOPRF}}$, trusted dealer and private backup tapes of servers). We add an ITI $\mathcal{F}$ with all interfaces of $\mathcal{F}_{\mathsf{PESTO}}$ except `ProceedSign` and `Verify`, modified s.t. $\mathcal{F}$ discloses passwords to $\mathcal{A}$. For `KeyGen`, we let $\mathcal{F}$ read n, the number of servers, from sid. Then, $\mathcal{F}$ runs $pp \leftarrow \mathsf{DSIG.Setup}(1^\tau)$, $(vk, sk_1, \ldots, sk_n) \leftarrow \mathsf{DSIG.KGen}(pp, n)$ and sends $sk_1, \ldots, sk_n$ to $\mathcal{S}$ (we stress that we take away these keys from $\mathcal{S}$ in a later game).

$\mathcal{S}$ uses key shares $sk_1, \ldots, sk_n$ to simulate honest servers and answer corruption queries of $\mathcal{Z}$. $\mathcal{S}$ sends the `KeyConf` output of an honest server to $\mathcal{F}$. $\mathcal{S}$ delivers outputs as soon as the corresponding message is delivered, or instantaneously if it was already delivered. Correspondances are as follows:

- message $(\texttt{Register}, sid, ssid, uid)$ sent $\longleftrightarrow$ deliver $(\texttt{Register}, sid, ssid, uid)$ to S_i
- message $(\texttt{Sign}, sid, ssid, uid, m)$ sent $\longleftrightarrow$ deliver $(\texttt{Sign}, sid, ssid, uid, m)$ to S_i
- message $(sid, ssid, \texttt{ok})$ sent $\longleftrightarrow$ deliver $(\texttt{Registered}, sid, ssid, uid)$ to U

$\mathcal{S}$ directly acknowledges all inputs and starts simulating the corresponding messages:

- input $(\texttt{Register}, sid, ssid, U) \longleftrightarrow$ ack. and simulate $(\texttt{reg}, sid, ssid, uid)$
- input $(\texttt{ProceedReg}, sid, ssid, S_i) \longleftrightarrow$ acknowledge and simulate $\mathcal{F}_{\mathsf{dpOPRF}}$ usage
- input $(\texttt{Sign}, sid, ssid, U) \longleftrightarrow$ acknowledge and simulate $(\texttt{sign}, sid, ssid, uid, m)$

Simulate corrupted parties' inputs to $\mathcal{F}$: whenever $\mathcal{A}$ sends an input $(\texttt{Eval}, sid, ssid, uid, pw)$ through a corrupted user U to $\mathcal{F}_{\mathsf{dpOPRF}}$, if $\mathcal{A}$ sent $(\texttt{Register}, sid, ssid, uid)$ to any S_i before, $\mathcal{S}$ sends an input $(\texttt{Register}, sid, ssid, uid, pw)$ to $\mathcal{F}$ on behalf of U. Else, if $\mathcal{A}$ sent $(\texttt{Sign}, sid, ssid, uid, m)$ to any S_i before, $\mathcal{S}$ sends $(\texttt{Sign}, sid, ssid, uid, pw)$ to $\mathcal{F}$ on behalf of U. In case of a corrupted server S_i, $\mathcal{S}$ sends inputs `KeyGen` and `ProceedReg` to $\mathcal{F}$ on behalf of S_i as soon as $\mathcal{A}$ sends a key request to the trusted dealer or $(\texttt{EvalProceed}, sid, ssid)$ to $\mathcal{F}_{\mathsf{dpOPRF}}$, respectively, where the latter query's identifier $ssid$ was formerly received within an input $(\texttt{Sign}, sid, ssid, ...)$ from $\mathcal{F}$. As a last change in this game, we add dummy parties between $\mathcal{F}$ and $\mathcal{Z}$, namely one per real party.

Considering indistinguishability, the output pattern of $\mathcal{F}$ matches the protocol and $\mathcal{F}$ starts working in all corruption scenarios thanks to $\mathcal{S}$'s inputs to $\mathcal{F}$. Regarding completeness of inputs note that servers do not input any secret values into $\mathcal{F}_{\mathsf{PESTO}}$ - the only secret input value is pw provided by U. If U is corrupted, $\mathcal{S}$ learns pw when U queries $\mathcal{F}_{\mathsf{dpOPRF}}$.

If furthermore key generation and user registration is equally successful in both games and all real-world protocol aborts can be enforced by $\mathcal{S}$ via $\mathcal{F}$, then outputs are equally distributed in both games (since they come unmodified from the real execution). For key generation, in the real execution all honest servers react only on the first `KeyGen` input and output vk obtained from $\mathcal{F}_{DKG}$, which is exactly what happens in Game 2. For user registration, we have to argue that U outputs `Registered` in Game 2 if and only if U outputs `Registered` in Game 1. This holds by correctness of SIG and DSIG and lists given above showing how $\mathcal{S}$ can abort the protocol run in Game 2 appropriately for all protocol messages.

Game 3: Abort upon SIG forgery. Whenever $\mathcal{Z}$ sends a message $(sid, ssid, upk, \sigma_U)$ to a server on behalf of a corrupted user with $\mathsf{SIG.Vf}(upk, (uid, ssid), \sigma_U) = 1$, where there was already a query $(sid, ssid', uid', upk, \sigma')$ with $ssid \neq ssid'$ and a message $(\texttt{Registered}, sid, ssid', uid')$ from $\mathcal{F}$ to some *honest* user U delayed by $\mathcal{S}$, then $\mathcal{S}$ aborts.

Similarly, $\mathcal{S}$ aborts whenever $\mathcal{Z}$ sends a message $(sid, ssid, uid, \sigma'_U)$ to a server on behalf of a corrupted user with $\mathsf{SIG.Vf}(upk, (uid, ssid), \sigma'_U) = 1$ and upk being the public key registered with an *uncorrupted* uid (i.e., a uid for which $\mathcal{S}$ never received $(\texttt{match-ok}, sid, ssid, 1)$ from $\mathcal{F}$ for a session $ssid$ run with a corrupted user).

The first case corresponds to $\mathcal{Z}$ maliciously registering a user uid with public key upk, where upk of another honest user uid'. $\mathcal{Z}$ does not know the corresponding usk and thus a verifying σ would constitute a forgery.

The second case corresponds to $\mathcal{Z}$ impersonating an honestly registered uid during signing. Since uid was honestly registered, $\mathcal{Z}$ does not know the corresponding

usk. As before, unforgeability of SIG signatures is enough to argue computational indistinguishability in this case.

Game 4: Abort when forged DSIG signature is verified. We let $\mathcal{S}$ abort if $\mathcal{Z}$ sends $(\texttt{Verify}, sid, uid, m, \sigma, vk)$ to any party V where $\texttt{Verify}(vk, (uid, m), \sigma) = 1$ and σ was never contained in any former Signature output, nor could it be assembled using keys of corrupted parties and signature shares.

We construct an adversary $\mathcal{B}$ winning the proactive unforgeability experiment $\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{B},\mathsf{forge}}(\tau)$ (cf. Def. 5) with non-negligible probability given a distinguisher $\mathcal{D}$ between Game 4 and Game 3. $\mathcal{B}$ uses outputs from his oracles to emulate either one of the consecutive games. $\mathcal{B}$ gets as input pk and works as follows:

- Whenever $\mathcal{F}$ outputs $(\texttt{KeyConf}, sid, vk)$, $\mathcal{B}$ overwrites this with $(\texttt{KeyConf}, sid, pk)$.
- Whenever $\mathcal{F}$ generates an output containing $\sigma \leftarrow \texttt{Sign}(sk, (uid, m))$ within a ProceedSign query with $ssid$, $\mathcal{B}$ obtains $\sigma_i \leftarrow \mathcal{O}_{\mathsf{sign}}(i, (uid, m), ssid)$ for all $i \in [n]$ with S_i being honest, and generates $\sigma_i \leftarrow \mathsf{DSIG.Sign}(sk_i, (uid, m); ssid)$ using sk_i for all corrupted S_i. Then $\mathcal{B}$ overwrites the output with $\sigma \leftarrow \mathsf{DSIG.Comb}(\sigma_1, \ldots, \sigma_n)$.
- Similarly, $\mathcal{B}$ overwrites signatures σ_{reg} in uid records stored at a server receiving a $(\texttt{Corrupt}, sid, S_i, *)$ query, using his $\mathcal{O}_{\mathsf{sign}}$ oracle and $\mathsf{DSIG}.comb()$ in the same way as above.
- Whenever a server sends an input $(\texttt{Refresh}, sid)$ to $\mathcal{F}$, $\mathcal{B}$ calls his $\mathcal{O}_{\mathsf{refresh}}$ oracle once.
- Whenever $\mathcal{Z}$ issues $(\texttt{Corrupt}, sid, \mathcal{S}_i, \texttt{mode})$ for a currently honest S_i, $\mathcal{B}$ queries $k \leftarrow \mathcal{O}_{\mathsf{corrupt}}(i, \texttt{mode})$ and sets k to be the key(s) of S_i.
- Whenever an honest server S_i generates a DSIG signature share on message m to send it to either a corrupted server or a corrupted user, i.e., within a ProceedSign or ProceedReg query with $ssid$, $\mathcal{B}$ queries $\mathcal{O}_{\mathsf{sign}}(i, m, ssid)$ and sets the answer σ_i to be the signature share send by the honest server.

This emulation is exactly a UC execution of Game 3 and Game 4, the only difference being that the DSIG key pair is generated by the oracles of $\mathcal{B}$ instead of being created within $\mathcal{F}$. For this, it is crucial to see that the above is a complete list of all key-related values that $\mathcal{Z}$ can observe from the execution of Game 3 or Game 4.

A distinguisher $\mathcal{D}$ between both consecutive games can only be successful if he causes the execution in Game 4 to abort, so if he submits a signature σ^* on message m^* for verification with the properties described above. Since σ^* cannot be assembled from secret key shares, we know that there is no e such that $|\mathcal{C}_{t,e} \cup \mathcal{C}_{p,e}| \neq n$. Since σ^* was also never output by $\mathcal{F}_{\mathsf{PESTO}}$ nor could be assembled by combining signature shares (obtained through a corrupted party or using known key shares), there is also no epoch in which $\mathcal{B}$ requests $\mathcal{O}_{\mathsf{sign}}(i, m^*, ssid)$ for all honest servers S_i and some $ssid$. By this, we see that (m^*, σ^*) constitutes a forgery in $\mathcal{B}$'s proactive unforgeability game.

Game 5: Abort when DSIG forgery is stored with user account. $\mathcal{S}$ aborts if S_i overwrites a record with $(uid, upk, \sigma_{\mathsf{reg}})$ in some epoch e, where $\mathsf{DSIG.Vf}(vk, (0, uid, upk), \sigma_r eg) = 1$ and where there was a $(\texttt{Corrupt}, sid, S_i, \texttt{trans})$ query in epoch e.

We use the same reduction to proactive unforgeability as in Game 4, i.e., the same adversary $\mathcal{B}$ to emulate an execution of Game 4 and Game 5. Due to the changes made in this game and usage of a secure broadcast channel, in every execution all honest servers that do not abort during a signing procedure are guaranteed to have the same user signing key upk stored for each account uid.

Game 6: Simulate DSIG key shares. We let $\mathcal{S}$ generate and refresh secret key shares (to give them to $\mathcal{Z}$ upon corruption) and signature shares (simulating honest servers) using algorithms $\mathsf{SIM}(\mathsf{refresh})$, $\mathsf{SIM}(\mathsf{corr}, ...)$ and $\mathsf{SIM}(\mathsf{sign}, ...)$ as in Figure 3. Whenever the two latter algorithms are used to compute the last key or signature share missing to let $\mathcal{Z}$ compute signatures using only $\mathsf{DSIG.Sign}()$ and $\mathsf{DSIG.Comb}()$, $\mathcal{S}$ uses key shares $sk_1, \ldots, sk_n$ forwarded by $\mathcal{F}_{\mathsf{PESTO}}$ to compute all those signatures as $\mathsf{DSIG.Comb}(\mathsf{DSIG.Sign}(sk_1, \cdot), \ldots, \mathsf{DSIG.Sign}(sk_n, \cdot))$ and feed them to the $\mathsf{SIM}()$ algorithms. Else, $\mathcal{S}$ simply runs $\mathsf{SIM}(\mathsf{sign}, \cdot, \cdot, \cdot, \perp)$ or $\mathsf{SIM}(\mathsf{corr}, \cdot, \cdot, \emptyset)$.

We show that if DSIG is share simulatable (cf. Def. 6), then Game 6 and Game 5 are indistinguishable. The adversary $\mathcal{B}$ running with experiment $\mathsf{Exp}^{\mathsf{DSIG},n}_{\mathcal{A},\mathsf{sim},b}(\tau)$ obtains pk and works as follows:

- Whenever $\mathcal{F}$ outputs $(\texttt{KeyConf}, sid, vk)$, $\mathcal{B}$ overwrites this with $(\texttt{KeyConf}, sid, pk)$.
- Whenever a server sends an input $(\texttt{Refresh}, sid)$ to $\mathcal{F}$, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{refresh}}()$.
- Whenever $\mathcal{Z}$ issues $(\texttt{Corrupt}, sid, S_i, \texttt{mode})$ for a currently honest S_i, $\mathcal{B}$ queries $k \leftarrow \mathcal{O}_{\mathsf{corrupt}}(i, \texttt{mode})$ and sets k to be the key(s) of S_i.
- When an honest S_i generates a DSIG signature share on message m, i.e., within a ProceedSign or ProceedReg query with $ssid$, $\mathcal{B}$ overwrite this share with $\sigma_i \leftarrow \mathcal{O}_{\mathsf{sign}}(i, m, ssid)$.

We need to argue that $\mathcal{B}$ using oracles from Figure 3 emulates the execution in Game 5, while $\mathcal{B}$ using oracles from Figure 3 emulates the execution in Game 6.

For the former, we observe that the oracles from Figure 3 follow the protocol instructions, e.g., signature shares are computed using $\mathsf{DSIG.Sign}(sk_i, \cdot, \cdot)$, which matches Game 5. For the latter, $\mathcal{S}$ uses the simulation algorithms in the same way as the oracles from Figure 3, and using knowledge of the true $sk_1, \ldots, sk_n$ he can compute the auxiliary inputs to these algorithms as done by the $\mathsf{compList}, \mathsf{compSig}$ algorithms. Thus, indistinguishability of this and the previous game follows from the share simulatability of DSIG.

Game 7: $\mathcal{F}$ generates, records and verifies signatures. We add interfaces ProceedSign and Verify to $\mathcal{F}$. For this, we first let $\mathcal{F}$ assemble a single signing key $sk \leftarrow \mathsf{Comb}^*(sk_1, \ldots, sk_n)$ from the key shares generated as detailed in Game 2. Then, $\mathcal{F}$ sets $\mathsf{Sign}() := \mathsf{Sign}^*()$ and no longer forwards $sk_1, \ldots, sk_n$ to $\mathcal{S}$. (Note that $\mathcal{F} \neq \mathcal{F}_{\mathsf{PESTO}}$ since $\mathcal{S}$ still obtains passwords).

The simulation is changed as follows: $\mathcal{S}$ acknowledges ProceedSign for S_i directly and then starts simulating usage of $\mathcal{F}_{\mathsf{dpOPRF}}$. $\mathcal{S}$ sends ProceedSign to $\mathcal{F}$ on behalf of corrupted users whenever $\mathcal{A}$ sends EvalProceed for an $ssid$ from an earlier Sign input. $\mathcal{S}$ sets $b^* = 1$ if all of the $(sid, ssid', \sigma'_U)$ messages delivered to the n servers have $\mathsf{SIG.Vf}(upk, (uid, ssid'), \sigma'_U) = 1$; else $\mathcal{S}$ sets $b^* =$

0. $\mathcal{S}$ replies to a $(\texttt{match-ok}, sid, ssid, b)$ message from $\mathcal{F}$ with $(\texttt{match-ok}, sid, ssid, b^*)$. Lastly, whenever $\mathcal{S}$ receives output $(\texttt{Signature}, sid, ssid, \sigma)$ (where σ is a verifying signature on message m) towards a corrupted U, $\mathcal{S}$ sets σ to be the result of $\mathsf{compSig}(\mathsf{ep}, i, m, ssid)$, where ep is the current epoch computable from the number of times $\mathcal{S}$ called $\mathsf{SIM}(\mathsf{refresh})$, and i is the last index missing an entry $(\mathsf{ep}, \cdot, m, ssid)$ in $\mathcal{Q}$ for an honest S_i. If S_i is corrupted, then $\mathcal{S}$ adds σ to the output of $\mathsf{compList}(\mathsf{ep}, i)$. Our indistinguishability argument will be split in four cases. The first three argue indistinguishability of outputs generated upon $(\texttt{Verify}, ...)$ and $(\texttt{Sign}, ...)$ queries, and the latter argues that values computed by $\mathcal{S}$ are distributed as in the preceding game.

First, in `Verify` queries, $\mathcal{F}$ overwrites protocol outputs according to `sigrec` records, or when no `sigrec` record exists and $vk = vk'$. The former is indistinguishable from Game 6 since (1) when `sigrec` contains `true`, then $b = 1$ (honest signing & verifying signature). (2), when `sigrec` contains `false`, then the record was created by a verification request (i.e., no regular signing procedure was completed for the signature to be verified). Such a query resulted in V outputting `false` already in Game 5 due to the changes made in that particular game.

Secondly, consider U *not* outputting a signature. In Game 6 this happens when password authentication failed (due to wrong password or $\mathcal{A}$ tampering with message delivery). In Game 7 `Signature` output depends on `sigrec` recording, which in turn only happens when $b = 1$ (assuming that all servers participate and $\mathcal{A}$ always continues with `ok` messages). Thus, we need to argue that $b = 0$ in Game 7 whenever U was not outputting `Signature` in Game 6. By the protocol code (cf. Figure 7), the latter happens if $\mathsf{SIG.Vf}(upk, (uid, ssid'), \sigma'_U) = 0$ for at least one of the messages sent by U to a server. But in that case, the above simulation ensures $\mathcal{S}$ sending $b^* = 0$ and $\mathcal{F}$ adopting this bit by setting $b \leftarrow b^*$ (due to either a corrupted server or a corrupted user).

Thirdly, consider the case where U outputs a signature. We have to argue that signatures $\sigma \leftarrow \mathsf{Sign}(sk, (uid, m))$ generated by $\mathcal{F}$ in Game 7 are indistinguishable from signatures generated as $\mathsf{DSIG.Comb}(\sigma_1, \ldots, \sigma_n)$, as done in Game 6. This follows from the signature indistinguishability of DSIG (cf. Definition 7).

Lastly, regarding outputs of $\mathcal{S}$, it is enough to verify that $\mathcal{S}$ in Game 7 obtains all signatures from $\mathcal{F}$ that he would compute in Game 6 within $\mathsf{compSig}$ and $\mathsf{compList}$. But this follows from the fact that $\mathcal{Z}$ cannot corrupt all servers. $\mathcal{Z}$ can thus compute signatures not only from secret key shares, but needs to observe signature shares sent via secure channels. Thus, $\mathcal{Z}$ can only compute signatures on behalf of corrupted users. And since $\mathcal{F}$ sends $(\texttt{Signature}, ..., \sigma)$ outputs for corrupted users directly to $\mathcal{S}$, $\mathcal{S}$ obtains a signature from $\mathcal{F}$ in case $\mathcal{Z}$ can reconstruct it using $\mathsf{DSIG.Comb}()$. Notably, $\mathcal{S}$ never has to simulate the "last" signature or signing key share *before* seeing the full signature.

Game 8: Simulate without pw in honest case. We let $\mathcal{S}$ use a dummy password $pw_{\mathcal{S}}$ in every `Register` or `Sign` input to an honest user.

Usage of secure channels makes simulation of the transcript trivial in the honest case. Regarding outputs $\mathcal{F}$ will create and output a verifying signature in Game 8 (i.e., a `sigrec` records with $f = \texttt{true}$. However, in case of everybody being honest, in Game 7 a `Sign` query only leads to a signature output to $\mathcal{Z}$ in case of matching passwords (otherwise $\mathcal{F}$ will ignore all `ProceedSign` queries), and in this case it will be a verifying signature since secure channels prevent $\mathcal{A}$ from tampering with signature shares.

Game 9: Simulate user's transkript without pw in case of corrupted servers. We consider the case where some servers are corrupted. Opposed to the all-honest case, we now have to provide a transkript of messages of honest users, but generated without knowing passwords. We change the simulation in case of an honest $\texttt{Sign}, uid$ input, where $(\texttt{account}, uid, *)$ is already registered in $\mathcal{F}$. Let upk denote the public key corresponding to uid that was generated in the protocol simulation during registration, usk the corresponding secret key and $|\mathsf{bl}|$ the label that was sent by $\mathcal{A}$ in the corresponding $\mathcal{F}_{\mathsf{dpOPRF}}$ session (i.e., $|\mathsf{bl}|$ represents the pseudo-random function that was evaluated during the registration of uid). Upon receiving an output $(\texttt{Match}, sid, ssid, b)$ for a corrupted S_i and a session $ssid$ invoked by an honest user U, if $b = 1$ then $\mathcal{S}$ sends $(sid, ssid, \sigma'_U)$ on behalf of U to all S_i in the protocol simulation, where $\sigma'_U \leftarrow \mathsf{SIG.Sign}(usk, (uid, ssid))$. If $b = 0$, $\mathcal{S}$ sends a random non-verifying σ'_U. In case of receiving $(\texttt{Eval}, sid, ssid, uid, pw)$ from a corrupted U intended to $\mathcal{F}_{\mathsf{dpOPRF}}$, in case $\mathcal{S}$ received $(\texttt{match-ok}, sid, ssid, 1)$ from $\mathcal{F}$, he sends $(\texttt{EvalComplete}, sid, ssid, y)$ to the corrupted U where y is the former output of $\mathcal{F}_{\mathsf{dpOPRF}}$ from registration of uid if $\mathcal{A}$ sent $(\texttt{EvalComplete}, sid, ssid, |\mathsf{bl}|)$ (i.e., using the same label $|\mathsf{bl}|$ as during registration) and a freshly drawn y otherwise. In case of receiving $(\texttt{match-ok}, sid, ssid, 0)$ from $\mathcal{F}$, $\mathcal{S}$ answers to U with a freshly drawn y.

Regarding indistinguishability, let us first emphasize again why we do not have to adjust the simulation to simulate honest servers. Namely, the protocol simulation is already a perfect simulation of honest servers since $\mathcal{F}$ keeps no secret server inputs from $\mathcal{S}$. If an honest U attempts to sign, only the message $(sid, ssid, \sigma'_U)$ has to be simulated. Since $\mathcal{S}$ obtains the information of whether σ'_U should be a verifying signature of $(uid, ssid)$ from $\mathcal{F}$ via a `Match` output towards a corrupted server, this message is equally distributed as in the previous game. If U is corrupted, we only need to argue that simulation of $\mathcal{F}_{\mathsf{dpOPRF}}$ is indistinguishable in both games. For this, we have to argue that the `match-ok` message by $\mathcal{F}$ actually provides $\mathcal{S}$ with the crucial information of whether to adjust $\mathcal{F}_{\mathsf{dpOPRF}}$ outputs to the corresponding output in the registration session of uid. To see this, note that `match-ok` leaks whether input passwords matched or not in case of a corrupted user.

Since $\mathcal{S}$ does not use the password provided by $\mathcal{F}$ anymore, we remove its forwarding and now have $\mathcal{F} = \mathcal{F}_{\mathsf{PESTO}}$. This concludes the proof of Thm. 3.