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 refreshes 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 [1], [2], [4]. 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 [35]. 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, 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.

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 [3].

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) [26], [34], [38], [41], threshold password-authenticated secret sharing (TPASS) [7], [11], [9], [27], as well as plain distributed password verification [19], [10], [18].
The newest addition to this line of research is password-based threshold authentication (PASTA) [5] 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. [5] define the desired security properties in form of game-based notions. It is well-known that for password-based primitives these notions cannot properly capture the way users (mis)handle passwords [15]. When formulated through games, users are assumed to choose their passwords at random from known and independent distributions. The security notion also gives the adversary access to “perfect” users only which are always using 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. 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.

**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 [5] 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 cor-

<table>
<thead>
<tr>
<th>Security</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Std. JWT</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>PASTA</td>
<td>((t, n))</td>
</tr>
<tr>
<td>PESTO</td>
<td>((n, n))</td>
</tr>
</tbody>
</table>
rupted 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 ($\text{dpOPRF}$) to let users deterministically derive a signing key pair from their username and password as $(\text{upk}, \text{usk}) \leftarrow \text{dpOPRF}(K, (\text{uid}, \text{pw}))$. Thereby, the key $K$ is split among $n$ servers and the evaluation is done partially-blind: the servers learn $\text{uid}$ but not $\text{pw}$. Disclosing the $\text{uid}$ allows us to use a single dpOPRF key per server, yet allow user-specific rate limiting.

When creating her account, the user sends the derived $\text{upk}$ to all servers which store $(\text{uid}, \text{upk})$ 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 $\text{uid}$, the user repeats the dpOPRF part for a password attempt $\text{pw}'$ to re-construct her key pair $(\text{apk}, \text{ask})$, and uses $\text{usk}$ to sign a fresh nonce towards all servers. The servers verify the signature against the stored $\text{upk}$ to determine whether the user provided the correct password. If so, the servers jointly sign $(\text{uid}, m)$ 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 $\text{dpOPRF}$ is based on the classical 2Hash-Diffie-Hellman OPRF construction [27] 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 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. Permanently corrupted parties additionally leak the backup tape 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. 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 2 exp. and 1 pairing operation per server and authentication request, which adds only minimal overhead compared with PASTA yet provides significantly stronger security. Concretely the time it takes for a complete sign-in operation, using a single threaded implementation, is only 124 ms even when both the user and servers are all located in distinct countries.

1.3. Related Work

Apart from PASTA [5] which we discussed already, our work takes inspiration from a number of previous works we detail here. We also refer to [5] 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. [10], that also aimed at proactive security in the UC framework. The targeted problems are different though, as their protocol considers a single login service with fine-grained rate limiting. However, Pythia’s approach requires direct access to the algebraic structure any previously computed PRF outputs. Further, Pythia’s method for 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.

Partial-Proactively PRF with Key Rotation. The Pythia protocol [18] 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. [30] presented a distributed partially-oblivious PRF that uses a different and more
efficient way to achieve partial blindness. However, they did not propose a security notion and, to the best of our knowledge, 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) [7], [27], [28], [29] 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. Whereas in our PESTO approach, a compromised device or password alone is not sufficient to gain access to the user’s accounts. All servers still 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 that her device has been breached — 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 \(f(n) \leq \text{negl}(n)\), if for every polynomial \(p(X)\) there exists a bound \(t\) such that \(\forall t' > t : f(t') \leq \frac{1}{|p(t)|}\). We denote the computational security parameter throughout this work as \(\tau\) and the statistical security parameter as \(s\).

2.1. **Bilinear Groups**

**Definition 1 (Asymmetric Pairing).** Let \(G_1, G_2, G_T\) be cyclic groups of order \(p\) with generators \(g_1, g_2, g_T\), respectively. Furthermore, let \(e : G_1 \times G_2 \rightarrow G_T\) be an efficiently computable non-degenerate function such that \(\forall a, b \in \mathbb{Z}_p : e(g_1^a, g_2^b) = g_T^{ab}\). Then \(e\) is called an asymmetric pairing. \(G = \langle p, g_1, g_2, g_T, G_1, G_2, G_T, e \rangle\) is called an asymmetric bilinear group setting, or bilinear group for short.

We formulate two assumptions on \(G\). The first assumption is the standard DDH-generalization groups with asymmetric pairings and has been used e.g. in [16].

**Definition 2 (Bilinear DDH Assumption).** Let \(\tau \in \mathbb{N}\) be a security parameter and \(G = \langle p, g_1, g_2, g_T, G_1, G_2, G_T, e \rangle\) be a bilinear group where \(\log(p) = \text{poly}(\tau)\). We say that the Bilinear Decisional Diffie-Hellman (BDDH) assumption holds for \(G\) if for any \(\tau\)-PPT (in \(\tau\)) algorithm \(A\):

\[
\Pr[A(G, g_1^r, g_2^s, g_T^t, R) = 1] - \Pr[A(G, g_1^r, g_2^s, g_T^t, e(g_1, g_2)^{abc}) = 1] \leq \text{negl}(\tau)
\]

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

For the second assumption we first define the following experiment with an algorithm \(A\):

**Experiment** \(\text{Exp}_{A, \text{Gapom-BDH}}(\tau)\):

\[
\begin{align*}
&k \leftarrow \mathbb{Z}_p, \ qc \leftarrow 0, \quad X_1 \leftarrow 0, \quad X_2 \leftarrow 0, \\
&(x_i, y_i, z_i) \in [\ell] \to A_{\text{GKS} \circ \text{O}_{\text{Gapom}}, \text{O}_{\text{Gapom}}}(G, g_2^k) \\
&\text{return } 0 \text{ if } \\
&\quad 0 \leq \ell < 1 - qc \text{ or } \\
&\quad \exists i \in [\ell] : (x_i \notin X_1 \lor y_i \notin X_2) \text{ or } \exists i, j \in [\ell], i < j : (x_i = x_j \land y_i = y_j) \\
&\text{return } 1 \text{ if } \forall i \in [\ell] : e(x_i, y_i)^k = z_i \text{ and } 0 \text{ otherwise.}
\end{align*}
\]

where the experiment uses the following oracles

\[
\text{O}_{\text{G KS}}() \quad \text{return } 1 \text{ if } r \notin \{1, 2\} \\
\text{O}_{\text{C help}}(m) \quad \text{return } 1 \text{ if } m \notin G_T.
\]

To win, \(A\) needs to find pairs \((x, y, e(x, y)^k)\) without querying \(e(x, y)\) to \(O_{\text{C help}}\) and where \(A\) could not rerandomize previous such pairs as it does not know the discrete logarithm of any \(x, y\) (enforced by sampling them at random using \(O_{\text{G KS}}\)). \(A\) is equipped with a DDH oracle \(O_{\text{C help}}\) in the group \(G_T\). The game Gapom-BDH follows the definition in [18].

**Definition 3 (Gap One-More BDH Assumption).** Let \(\tau \in \mathbb{N}\) be a security parameter and \(G = \langle p, g_1, g_2, g_T, G_1, G_2, G_T, e \rangle\) be a bilinear group with \(\log(p) = \text{poly}(\tau)\), then we say that the Gap One-More Bilinear Diffie-Hellman (Gapom-BDH) assumption holds for \(G\) if for all \(\tau\)-PPT adversaries \(A\) there is a negligible function \(\text{negl}(\tau)\) such that \(\Pr[\text{Exp}_{A, \text{Gapom-BDH}}(\tau) = 1] \leq \text{negl}(\tau)\).

2.2. **Digital Signatures**

We use a signature scheme SIG, consisting of algorithms \(\text{SIG.Setup}(1^\tau) \to (\text{pp}, \text{SIG.KGen}(\text{pp}, \tau)) \to (\text{sk}, \text{pk})\), \(\text{SIG.Sign}(\text{sk}, m) \to (\sigma)\) and \(\text{SIG.Vf}(\text{pk}, m, \sigma) \to \{0, 1\}\). 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 \(\tau\), for a key pair \((\text{pk}, \text{sk})\), where \(\text{sk}\) output by SIG.SKGen(\(\text{pp}\)) and \(\text{pk}\) output by SIG.PKGen(\(\text{pp}, \text{sk}\)), it holds that \(\text{SIG.Vf}(\text{pk}, m, \text{SIG.Sign}(\text{sk}, m)) = 1\) for every message \(m \in \{0, 1\}^\ell\) (for implicit \(\text{pp}\) generated by SIG.Setup).

Security of a signature scheme is defined through the standard unforgeability experiment \(\text{Exp}_{A, \text{forge}}(\tau)\):

**Experiment** \(\text{Exp}_{A, \text{forge}}(\tau)\):

\[
\begin{align*}
&pp \leftarrow \text{SIG.Setup}(1^\tau), (\text{sk}, \text{pk}) \leftarrow \text{SIG.KGen}(\text{pp}) \\
&(m^*, \sigma^*) \leftarrow A_{\text{O}_{\text{Sig}}, \text{O}_{\text{Sig}}}((\text{pk}, \text{sk})) \\
&\text{where } O_{\text{Sig}}(m): \quad \text{add } m \text{ to } Q, \text{return } \text{SIG.Sign}(\text{sk}, m) \\
&\text{return } 1 \text{ if } \text{SIG.Vf}(\text{pk}, m^*, \sigma^*) = 1 \text{ and } m^* \notin Q
\end{align*}
\]

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

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 $\text{DSIG}$ is a tuple of polynomial-time algorithms ($\text{Setup}, \text{KGen}, \text{Sign}, \text{Comb}, \text{Vf}, \text{Refresh}$) such that:

- $\text{DSIG.\text{Setup}}(1^n, 1^\ell) \rightarrow_{\text{a.p.}} (pk, (sk_i)_{i \in [n]}, (bk_i)_{i \in [n]})$: Given a computational security parameter $1^n$ and a statistical security parameter $1^\ell$, outputs public parameters $pp$.

- $\text{DSIG.\text{KGen}}(pp, n) \rightarrow_{\text{a.p.}} (pk, secret signing key shares $sk_1, \ldots, sk_n$ and backup key shares $bk_1, \ldots, bk_n$).

- $\text{DSIG.\text{Sign}}(sk_i, m, \ell) \rightarrow \sigma_i$: Given a key share $sk_i$, message $m$, and query label $\ell$, output a signature share $\sigma_i$.

- $\text{DSIG.\text{Comb}}(\sigma_1, \ldots, \sigma_n) \rightarrow \sigma$: Given signature shares $\sigma_1, \ldots, \sigma_n$, output signature $\sigma$ or $\perp$.

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

- $\text{DSIG.\text{Refresh}}(bk_i) \rightarrow_{\text{a.p.}} (sk_i', bk_i')$: Given a backup key $bk_i$, output new keys $sk_i'$ and $bk_i'$.

The latter two are needed to give us the strong simulation-based security needed to 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 in order 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. Therefore, the security experiment has to keep track of which information the adversary gets which point of time.

**Proactive Unforgeability.** This notion generalizes standard (strong) unforgeability to the proactive, distributed setting. It is defined with oracles given in Fig 2 which allow the adversary $A$ to obtain signature shares, and key shares (either only $sk_i$ 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.

The adversary then 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 $\ell$ for all uncorrupted parties in one epoch.

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

**Experiment** $\text{Exp}_{A,\text{forge}}^{\text{SIG}, n}(\tau, s)$:

- $pp \leftarrow_{\text{a.p.}} \text{DSIG.\text{Setup}}(1^n, 1^\ell)$, set $ep \leftarrow 0$, $C_\text{tap}, C_\text{p,ap} \leftarrow \emptyset$, $(pk, (sk_i)_{i \in [n]}, (bk_i)_{i \in [n]}) \leftarrow_{\text{a.p.}} \text{DSIG.\text{KGen}}(pp, n)$ ($m^*, \sigma^*) \leftarrow_{\text{a.p.}} \text{O_{ap,refresh}}$, $\text{Setup}(pk)$

- return $0$ if $\text{DSIG.\text{Vf}}(pk, m^*, \sigma^*) = 0$ or $\exists ep$ where $|C_\text{tap} \cup C_\text{p,ap}| = n$ or $\exists ep, \ell$ s.t. $\text{DSIG.\text{Comb}}(\sigma_1, \ldots, \sigma_n) = \sigma^*$

- where $\forall i \notin C_\text{tap} \cup C_\text{p,ap}$ $(ep, i, m^*, \ell, \sigma_i) \in \mathcal{Q}$ and $\forall i \in C_\text{tap} \cup C_\text{p,ap} \sigma_i = \text{DSIG.\text{Sign}}(i, sk_i^{ep}, m^*, \ell)$

else return $1$

**Share Simulatability.** The second property is needed to use $\text{DSIG}$ 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 $\text{DSIG}$ is share simulatable if for all PPT adversaries $A$, there exists a stateful PPT algorithm $\text{SIM}$ with input $pk$ and a negligible function $\text{negl}(\cdot)$ such that $\Pr[\text{Exp}_{A,\text{sim,1}}^{\text{SIG}, n}(\tau, s) = 1] - \Pr[\text{Exp}_{A,\text{sim,0}}^{\text{SIG}, n}(\tau, s) = 1] \leq \text{negl}(\tau, s)$.

3.1. Proactive Security Model for $\text{DSIG}$

We now present the desired security definitions for $\text{DSIG}$ 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, as well as share simulatability and signature indistinguishability.
\[ O_{\text{sig}}(i, m, \ell) \]
\[
\text{abort if } \exists c \in C_{\text{sig, app}} : \ell \notin C_{\text{app}} \\
\text{set } C_{\text{app}} \leftarrow C_{\text{app}} \cup \{i\}
\]
\[
\text{return } \sigma_i
\]
\[ O_{\text{corrupt}}(i, \text{mode}) \]
\[
\text{if mode} = \text{trans:} \\
\text{abort if } i \in C_{\text{app}} \\
\text{set } C_{\text{app}} \leftarrow C_{\text{app}} \cup \{i\} \\
\text{return } \sigma_i
\]
\[
\text{if mode} = \text{perm:} \\
\text{set } C_{\text{app}} \leftarrow C_{\text{app}} \cup \{i\} \\
\text{return } (sk_i^p, b_i^p)
\]
\[ O_{\text{return}}() \]
\[
\text{for all } i \notin C_{\text{app}}: \\
\text{DSIG.Refresh}(b_i^p) \rightarrow (sk_i^{p+1}, b_i^{p+1}) \\
\text{set } ep \leftarrow ep + 1, C_{\text{app}} \leftarrow \emptyset, C_{\text{app}} \leftarrow C_{\text{app}} - 1
\]

Figure 2: 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 \( \sigma \neq \bot \) 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.

\[ \text{Experiment } \text{Exp}_{\text{DSIG, n}}^{\text{SIM}, k}(\tau, s): \]
\[
pp \leftarrow \text{DSIG.Setup}(1^\tau, 1^s) \\
(pk, (sk_i)_i \in [n], (bk_i)_i \in [n]) \leftarrow \text{DSIG.KGen}(pp, n) \\
set ep \leftarrow 0, C_{\text{app}} \leftarrow \emptyset, C_{\text{app}} \leftarrow \emptyset \\
b^* \leftarrow A_{\text{Comp}} (pk, C_{\text{app}}, C_{\text{app}} (sk_j)) \\
\text{if } b = 0: \text{left oracles as in Fig 2} \\
\text{if } b = 1: \text{right oracles as in Fig 2 (for SIM(init, pk))} \\
return 1 \text{ if } b^* = b \text{ and } \text{Vep} : (C_{\text{app}} \cup C_{\text{app}}) < n
\]

Roughly, \( A \) 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 \( A \) 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 \( A \) to create a signature. If so, we generate the full signature \( A \) is about to learn via \( \sigma \leftarrow \text{Com}(\text{Sign}(sk_1, m, \ell), \ldots, \text{Sign}(sk_n, m, \ell)) \) and give \( \sigma \) as advise 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 \( \text{CompSig}_{C_{\text{app}}, \ell} (ep, i, m, \ell) \rightarrow \{ \sigma, \bot \} \) that tests whether answering a signing query for \( (i, m, \ell) \) would, together with the information already revealed to \( A \) (contained in \( Q, C_{\text{app}}, pp \)), allow him to complete the signature. If so, it returns the full signature \( \sigma \) for \( m \).

For corruption queries, we use a similar function \( \text{compList}_{C_{\text{app}}, \ell} (ep, i) \rightarrow L \) which checks which previously answered sign queries \( (ep, j, m, \ell) \) 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 \( L \) of tuples \( (ep, j, m, \ell, \sigma) \).

For sake of brevity, we also sometimes omit the subindex \( Q, C_{\text{app}}, pp \) from the name of these functions. The functions \( \text{CompSig}_{Q, C_{\text{app}}, \ell} (ep, i, m, \ell) \) and \( \text{compList}_{Q, C_{\text{app}}, \ell} (ep, i) \) are defined as follows:

\[ \text{compSig}_{Q, C_{\text{app}}, \ell}(ep, i, m, \ell); \]
\[
\text{if } \forall j \notin C_{\text{app}} : \ell \notin C_{\text{app}} \cup \{j\} \\
\exists (ep, j, m, \ell) \in Q \\
\text{get } \sigma \leftarrow \text{DSIG.Sign}(sk_j, m, \ell) \text{ for } i \in [n] \\
\text{return } \sigma \leftarrow \text{DSIG.Sign}(\sigma_1, \ldots, \sigma_u) \\
\text{else return } \bot
\]

\[ \text{compList}_{Q, C_{\text{app}}, \ell}(ep, i); \]
\[
\text{if } \forall j \notin C_{\text{app}} : \ell \notin C_{\text{app}} \cup \{j\} \exists (ep, j, m, \ell) \in Q \\
\text{get } \sigma \leftarrow \text{DSIG.Sign}(sk_j, m, \ell) \text{ for } i \in [n] \\
\text{return } \sigma \leftarrow \text{DSIG.Sign}(\sigma_1, \ldots, \sigma_u) \\
\text{else return } \bot
\]

3.2. Our RSA-DSIG Instantiation

We now describe an instantiation of such 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 \( H, H \), modeled as random oracles in the security proof, necessary to achieve proactive security. The reason we chose RSA and not ECDSA is that even...
without proactive security, signing requires interaction between the servers [17]. Thus the latency of executing this over a 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 $\phi(N)$ 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 $A$ 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 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.** For achieving proactive and adaptive security it is crucial that servers do not commit to their secret key share. We achieve this by applying fresh shares $\beta_i$ of 1 to each signing share. We use a similar trick for the $f_i$ values to do this in non-interactive yet consistent manner: All servers share pairwise seeds $s_{i,j}$ that are generated during initial key generation and every refresh. The blinding $\beta_i$ is derived through a random oracle call on these seeds and a fresh label $\ell$. The random oracle gives thus the flexibility we need in the security proof, and the uniqueness of $\ell$ guarantees 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 $\ell$, as required by the proactive unforgeability notion, as the $\beta_i$ values only cancel out when the user combines signature shares for the same $\ell$.

**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 $a : \mathbb{Z}^n \rightarrow \mathbb{Z}$ as $a(i, \{x_j\}_{j\in[n]\setminus\{i\}}) = \sum_{j \in [n]\setminus\{i\}} x_j - \sum_{j \in [n]\setminus\{i\}} x_j$.

We also stress that we assume that all servers use each label $\ell$ only once, this can be ensured, e.g., by letting all servers contribute to $\ell$ 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**($\Gamma^*$, $\ell^*$): Output $pp \leftarrow (H, \tilde{H})$ where $H : \{0, 1\}^* \rightarrow \{0, 1\}^*$ and $\tilde{H}_N : \{0, 1\}^* \rightarrow [N]$.

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

1. Pick random primes $p, q$ of length $\tau/2$, set $N = p \cdot q$.
2. Let $e$ be a prime $\geq 3$ and set $d \leftarrow e^{-1} \mod \varphi(N)$.
3. Choose random $\gamma_{i,j}, \delta_i$ for $i, j \in [n], i \neq j$:
   - $\gamma_{i,j} \leftarrow \left[-n^2 N^2, n^2 N^2\right]$ for $i < j$
   - $\delta_i \leftarrow \left[-n^2 N^2 2^{2^i}, n^2 N^2 2^{2^i}\right]$ for $i < j$
   - $\gamma_{j,i} \leftarrow \gamma_{i,j}, \delta_i \leftarrow \delta_j$ for $i > j$
4. For $i \in [n - 1]$ set $d_i \leftarrow a(i, \gamma_{i,1}, \ldots, \gamma_{i,n})$ while $d_n \leftarrow a(n, \gamma_{1,1}, \ldots, \gamma_{n,n}) + d$. 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 $i, j \in [n], i \neq j$:
   - $s_{i,j} \leftarrow \left\{0, 1\right\}^\tau$, $mk_{i,j} \leftarrow \left\{0, 1\right\}^\tau$ for $i < j$
   - $s_{j,i} \leftarrow s_{i,j}$, $mk_{j,i} \leftarrow mk_{i,j}$ for $i > j$
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**($sk_i, m, \ell$):

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 $\beta_i = \prod_{j \in [n]\setminus\{i\}} H_N(\ell, s_{i,j})^{\Delta_{i,j}} \mod N$.
4. Output $\sigma_i = H(m)^{\tau} \cdot \beta_i \mod N$.

**RSA-DSIG.Comb**($\sigma_1, \ldots, \sigma_n$):

Output $\sigma = \prod_{i=1}^n \sigma_i \mod N$.

**RSA-DSIG.Vf**($pk, m, \sigma$): Parse $(N, e) \leftarrow pk$. Output $b = 1$ if $\sigma' = H(m) \mod N$ 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 $(mk_{i,j}', \delta_{i,j}') \leftarrow \text{PRG}(mk_{i,j})$ for every $j \in [n]\setminus\{i\}$ such that $mk_{i,j}' \in \{0, 1\}^{\tau}$, $\delta_{i,j}' \in \{0, 1\}^{\tau}$ and $\delta_{i,j}' \in \left[-n^2 N^2 2^{2^i}, n^2 N^2 2^{2^i}\right]$.
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 $bk_i' = (d_i, \{mk_{i,j}'\}_{j\in[n]\setminus\{i\}})$.

**Theorem 1.** Assuming hardness of RSA and modeling $H$ as a random oracle, the distributed signature scheme 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 $\sigma_i$ for a certain epoch $ep$ and input $(m, \ell)$ can then be generated from the dummy $f_i$ shares, whereas the last signature has to be made fit such that $\sigma = \prod_{i=1}^n \sigma_i$. To make this consistent with transient or permanent corruptions we exploit the programmability of $H$ — and the fact that all labels $\ell$ will only be used once per server — in the construction of $\beta_i$.

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 [20], [39], [23], [25]. Later, after the introduction of the proactive security model [36] this was extended to ‘static-proactive’ security, where the adversary may corrupt parties only in the onset of each operational phase [22], [21], [37], [33]. These works were strengthened towards security for adaptive-proactive corruptions in [14], [24], [32], [6]. 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 [10], 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 \( x_{\text{priv}} \) (which remains hidden towards the dpOPRF servers) the evaluation is also parameterized by a public value \( x_{\text{pub}} \).

4.1. Functionality \( \mathcal{F}_{\text{dpOPRF}} \)

We follow the previous line of work [28], [29], [31] and model security for our dpOPRF via an ideal functionality \( \mathcal{F}_{\text{dpOPRF}} \) in the UC framework [12]. The functionality allows evaluation of a random function on chosen inputs \( (x_{\text{priv}}, x_{\text{pub}}) \), given all servers \( S_1, \ldots, S_n \) agree to participate. Implementing a partially-oblivious function, \( \mathcal{F}_{\text{dpOPRF}} \) tells the servers \( x_{\text{pub}} \) 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 \( \mathcal{F}_{\text{dpOPRF}} \) does not contain an adversarial interface for offline evaluations, i.e., without all servers participating, as is the case for other OPRF functionalities [28], [29], [31]. 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, \( \mathcal{F}_{\text{dpOPRF}} \) allows \( A \) to modify the PRF key if not all servers are honest. This is implemented by letting \( A \) input a label \( \text{lbl} \) and \( \mathcal{F}_{\text{dpOPRF}} \) maintaining a different random function for each lbl. Lastly, our \( \mathcal{F}_{\text{dpOPRF}} \) 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., [28], [29]). \( \mathcal{F}_{\text{dpOPRF}} \) prevents such attacks by maintaining evaluation ticket counters \( \text{ctr}[U, x_{\text{pub}}] \) that are increased only when all servers agree to participate in an evaluation initiated by \( U \) using \( x_{\text{pub}} \) 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. \( \mathcal{F}_{\text{dpOPRF}} \) 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. \( \mathcal{F}_{\text{dpOPRF}} \) is informed about such corruption recovery via an input Refresh from one of the servers.

The detailed functionality is given in Figure 3. For brevity, we use the following writing conventions:

Writing Conventions for \( \mathcal{F}_{\text{dpOPRF}} \) (and \( \mathcal{F}_{\text{PESTO}} \)).
1) The functionality considers a specific session \( \text{sid} = (S_1, \ldots, S_n, \text{sid}') \) and only accepts inputs from servers \( S_i \) that are contained in the \( \text{sid} \).
2) We assume sub-session identifiers \( \text{ssid} \) to be unique. All interfaces that take as input an \( \text{ssid} \) will only accept one input per party and such \( \text{ssid} \).
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 "(Eval, sid, ssid, payload)" from a party \( P \), the adversary receives (Eval, sid, ssid, sid, \( P \)) i.e., he learns all meta-data but nor the payload.
5) Calls to all interfaces other than \( \text{KeyGen, sid} \) 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 \( H'(x_{\text{priv}}, H(x_{\text{priv}})K) \) [27], but combines additional features inspired by other works: partial obliviousness using pairings [18], and proactive security [10].

In the 2HashDH protocol, the user binds his input \( x_{\text{priv}} \) with randomness \( r \) and sends \( x \leftarrow H_1(x_{\text{priv}})^r \) to the server, which she can remove again after receiving \( x^K \) from the server. The protocol can be made distributed by simply setting \( K \leftarrow \sum_{i=1}^{n} k_i \) where \( k \) are the individual server keys and letting the user multiply the individual PRF shares she receives.

To achieve partial blindness, servers contribute \( e(x\bar{x}, H_2(x_{\text{pub}}))^{k_i} \) with an asymmetric pairing \( e \) and a public input \( x_{\text{pub}} \) obtained from the user in the clear. Overall, our protocol computes \( y \leftarrow \text{PRF}_K(x_{\text{priv}}, x_{\text{pub}}) \) with:

\[
y = H_4 (x_{\text{priv}}, x_{\text{pub}}, e(H_1(x_{\text{priv}}), H_2(x_{\text{pub}}))^{\sum_{i=1}^{n} k_i})
\]

Setup. For our construction, we assume a bilinear group \( (p, g_1, g_2, g_T, G_1, G_2, G_T, e) \) and hash functions \( H_1 : \)
The functionality is parametrized by a security parameter $\tau$. It interacts with servers $S := \{S_1, \ldots, S_n\}$ (specified in the $sid$), arbitrary other parties and an adversary $A$. $F_{dpORF}$ maintains a table $T(\text{bl}, x_{pub}, x_{pub}, x_{priv})$ initially undefined everywhere, counters $ctr[U, x_{pub}]$ initially set to 0 and sets $C_r, C_p$ initially set to 0. The label $\text{bl}$ is an arbitrary string $\{0,1\}^\tau$, where $\text{bl}$ denotes the “honest” label.

**Key Generation.** For simplicity we assume the existence of a trusted dealer that generates master keys for the servers. Each server devises of an offline backup tape that can be plugged in for performing read and write operations. Each server disposes of an offline backup tape that can be plugged in for performing read and write operations.

$F_{dpORF}$ makes its initial setup in a non-interactive fashion. Relying on the RSA-DSIG construction and letting servers mask all messages with a blinding value $\beta_i$ which is a pseudorandom multiplicative share of 1 that is constructed from the pairwise shared $s_{i,j}$-values and the (unique) $sid$.

**Refresh.** The refresh is triggered when a server $S_i$ receives the input $(\text{Refresh}, sid)$, upon which it sends $(\text{Refresh}, sid)$ over a secure broadcast channel to all other servers $S_j \in S \backslash S_i$, 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 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 $k_{i,j}$ 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 $(\text{Refresh}, sid)$ or by receiving $(\text{Refresh}, sid)$ from a server $S_j \in S \backslash S_i$) runs the following update procedure:

- Retrieve $(k_i, \{mk_{i,j}\}_{j \in [n] \setminus \{i\}})$ from the backup tape.
- For all $j \in [n] \setminus \{i\}$: $(mk_{i,j}, s'_{i,j}, \delta_{i,j}) \leftarrow \text{PRG}(mk_{i,j})$.
- Set $k'_i \leftarrow k_i + \sum_{j=1, j \neq i}^{n} \Delta_{i,j} \delta_{i,j} \mod q$, where $\Delta_{i,j} = \delta_{i,j} = 1$ 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 $(k_i, \{mk_{i,j}\}_{j=1, j \neq i})$ from the backup tape.
Figure 4: Evaluation protocol of our DH-dpOPRF construction.

Note that retrieving an uncorrupted \( k_i \) from the backup tape is necessary to properly recover from transient corruptions, as the adversary may have altered \( k_i \) during its corruption.

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 \( F_{dpOPRF} \) 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.

Regarding communication channels, 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 \( F_{dpOPRF} \) in the random oracle model if PRG is secure and the Gapom-BDH assumption holds in \( \mathbb{G} \), w.r.t environments restricted to corrupt at most \( n-1 \) servers concurrently using adaptive permanent and transient corruptions.

Our proof leverages proof techniques used by other works in a similar setting. In particular, the overall idea is very similar to the proof of Theorem 3 by Jarecki et al. [29], which shows security of a fully oblivious PRF in a static and non-proactive threshold setting, based on the gap one-more threshold Diffie-Hellman assumption.

We follow their overall structure, but generalize to the bilinear setting (as required by the partial obliviousness of our scheme) and make a reduction directly to a non-threshold assumption.\(^1\)

However, as we need both adaptive and proactive security, their techniques are not enough. Hence we have taken inspiration of blinding values and security of backup keys from Camenisch et al. [10] for adaptive and proactive password verification, also based on the Gap one-more Diffie-Hellman assumption.

To be able to provide a transcript in case of adaptive corruption, our simulator must ensure that every message sent is not “committing” to the servers’ private keys. This is handled by using blinding values \( H_3(\text{sid}, \text{ssid}, s_{i,j}) \) that will pad the only message exchanged that is not (randomized) output of a random oracle, namely \( H_3(x_{\text{priv}}) \). Besides allowing the simulator to be able to simulate values \( y_i \) without committing to \( k_i \) for each \( S_i \), it also ensures randomness of messages from each server sent in different epochs, even if a corrupt user makes the same query; thus allowing for simulation. It also prevents the mixing of unfinished executions of evaluation under different ssids.

We must ensure that the simulator can extract the input \( x_{\text{priv}} \) of a malicious party and use this for the query in the ideal functionality. However we notice that this cannot be done from the message \( x = H_1(x_{\text{priv}}) \) sent by a corrupt user. The reason being that this \( r \) is random and unknown to the simulator. The simulator must instead wait until it learns it from a query \( H_4(x_{\text{priv}}, x_{\text{pub}}, y) \). The problem is however that the ideal functionality must output an acknowledgment to the honest servers on \( x_{\text{pub}} \) and \( \text{ssid} \) after receiving a query from a user. Thus the simulator already here needs to input a value for \( x_{\text{priv}} \) to the ideal functionality. What happens is that it inputs a random dummy value for \( x_{\text{priv}} \) and then later when it receives the query \( H_4(x_{\text{priv}}, x_{\text{pub}}, y) \) it re-queries the ideal functionality on a different \( \text{ssid} \), but as the same corrupt user and with the same \( x_{\text{pub}} \). It then learns the true output of the ideal functionality and can simply substitute this into the response of \( H_4 \). This trick is achieved through the counters ctr which basically allows several (possibly unfinished) queries from the same user to take place. This trick is also used in a similar setting in a proof by Camenisch et al. [10, Sec. 4.2].

We must also ensure a consistent simulation even
when the adversary is using different keys than it was assigned by the trusted third party. To achieve this, we have to store a transcript of every query, and a unique label for all different choices of keys the adversary has used. Letting \( k^i = k - \sum_{i \in C} k_i \) with \( k = \sum_{i \in [n]} k_i \) and \( C \) denoting the set of currently corrupt parties, then the problem basically reduces to the simulator having to detect if \( y = e(H_1(x_{\text{priv}}), H_2(x_{\text{pub}}))^k \) for some \( k^i \) seen before. However since it cannot define all the honest parties’ keys in the simulation and since the amount of corrupted parties and the value of their keys change between epochs, it will have to use the \( \mathcal{O}_{C,\text{help}} \) oracle to get the value \( e(H_1(x_{\text{priv}}), H_2(x_{\text{pub}}))^k \) and then use this to get \( e(H_1(x_{\text{priv}}), H_2(x_{\text{pub}}))^{k^i} \) where \( \epsilon \) is any additive error the adversary might have added to the sum of its key shares. The simulator can then use the \( \mathcal{O}_{D,\text{help}} \) to figure out if this is an \( \epsilon \) it has seen before and then use the ideal functionality to give correct output accordingly. See Appendix B for more details.

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 will 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 \( \sigma \), 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 \( \sigma \) can be verified by everyone against the public key \( vk \).

In terms of security we want PESTO to satisfy the following 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. Also, 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 modeling 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 \( F_{\text{PESTO}} \) in the UC framework. The main entities in our system are a set of servers \( S := \{S_1,\ldots,S_n\} \) 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 5. For the sake of brevity, our \( F_{\text{PESTO}} \) definition assumes the writing conventions from Section 4.

Key Generation. The functionality is parametrized by algorithms (KGen, Sign, Verify), and internally generates a key pair \((vk, sk) \leftarrow \text{KGen}(1^n)\) when all servers have explicitly triggered key generation by sending (KeyGen, sid). From then on, \( F_{\text{PESTO}} \) 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 \( F_{\text{PESTO}} \) 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 (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 (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 (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 (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.
The functionality is parameterized by algorithms (KGen, Sign, Verify) and security parameter $\tau$. It interacts with servers $S := \{S_1, \ldots, S_n\}$ (specified in the $sid$), as well as arbitrary users, verifiers and an adversary $A$. Let $C_i, C_p$ denote initially empty sets.

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

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

**Signing Request and Verification**
- **On receiving** (Sign, $sid$, ssid, uid, m, pw) from party $V$:
  - Proceed only if a record (account, uid, pw) exists. Create record (sign, ssid, $U$, uid, m, pw, $b$) with $b \leftarrow \bot$.
  - Send a delayed output (Sign, ssid, uid, m, pw, $b$) to all $S_i \in S$.
- **On receiving** (ProceedSign, $sid$, ssid, uid) from server $S_j$:
  - Retrieve records (sign, ssid, $U$, uid, m, pw, $b$) and (account, uid, pw).
  - If (ProceedSign, ssid, uid) has been received from all $S_i$:
  - If $U$ is corrupt and $pw = pw'$, mark $uid$ corrupted and set $b \leftarrow 1$.
  - Send (match-ok, ssid, $b$) to $A$ and receive back (match-ok, ssid, $b'$).
  - Update the 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 $S_i$ is corrupt, set $b \leftarrow b' \land (pw = pw')$.
  - Send a delayed output (match, ssid, $b$) to all $S_i \in S$.
  - Only proceed if $b = 1$, retrieve ($key$, $sk$, $vk$) and create the signature:
    - Send (sign-ok, ssid, $sid$) to $A$ and receive back (sign-ok, ssid, $sid'$).
    - Create $\sigma \leftarrow$ Sign($key$, (uid, $m$)), abort if ($sigrec$, uid, $m$, $\sigma$, $vk$, false) exists.
    - Record (sigrec, uid, $m$, $\sigma$, $vk$, true) and output (Signature, $sid$, ssid, $\sigma$) to $U$. (Correctness)
- **On receiving** (Verify, sid, uid, m, $\sigma$, $vk'$) from party $V$:
  - If ($sigrec$, uid, $m$, $\sigma$, $vk'$) exists, set $f \leftarrow f'$ (Consistency).
  - Else, create a record ($sigrec$, uid, $m$, $\sigma$, $vk'$, $f$) where $f$ is determined as follows:
    - If $vk = vk'$, set $f \leftarrow false$ (Strong Unforgeability), else set $f \leftarrow Verify$(vk', (uid, $m$), $\sigma$).
  - Output (Verified, sid, uid, $m$, $\sigma$, $vk'$, $f$) to $V$.

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

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 it was because it was created by the adversary or the adversary correctly guessed the password of an initially honest account, $A$ can now 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 $A$ whether login was successful via output match-ok, and then asking $A$ for a bit $b'$. The functionality enforces its influence according to the corruption setting. All servers receive the decision bit via an output (match, sid, ssid, $b$).

Finally, when the password was determined to be correct, the signature is created. $\mathcal{F}_{\text{PESTO}}$ generates $\sigma \leftarrow$ Sign($key$, (uid, $m$)) and stores a record ($sigrec$, uid, $m$, $\sigma$, $vk$, true) that will allow successful verification of $\sigma$. 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 (Signature, sid, ssid, $\sigma$).

**Verification.** Everyone can check the validity of signatures by sending (Verify, sid, uid, $m$, $\sigma$, $vk'$). If $vk = vk'$, i.e., verification is requested for the verification key associated with $sid$, then $\mathcal{F}_{\text{PESTO}}$ uses its internal records to determine whether $\sigma$ is valid for (uid, $m$). The output is set to $f \leftarrow$ true only if a record ($sigrec$, uid, $m$, $\sigma$, $vk$, 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 $\mathcal{F}_{\text{PESTO}}$ must realize a

![Figure 5: Ideal functionality $\mathcal{F}_{\text{PESTO}}$ for proactively-secure distributed SSO](image-url)
trusted certification authority too.

**Corruptions and Refresh.** We use the same transient corruption model in UC as in Section 4.2.1, and thus \( F_{\text{PESTO}} \) provides the same corruption interfaces \( \text{Corrupt} \) and \( \text{Refresh} \) for corrupting servers as \( F_{\text{dpOPRF}} \).

\( F_{\text{PESTO}} \) does not provide a specific interface for user corruption, but nonetheless is informed about such and marks accounts belonging to the adversary accordingly. We note that \( F_{\text{PESTO}} \) naturally models adaptive user corruption though even if the adversary only corrupts machines before they start the protocol. This is because \( F_{\text{PESTO}} \) ignores which machine a user chooses, i.e., a user might choose an honest machine for registering and a corrupted one for logging in afterwards, leaking all formerly hidden secrets to the adversary in an adaptive fashion. Also, note that password guessing or leakage is absorbed by the UC model already, 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 but important SSO-related aspect to our model. Essentially, \( F_{\text{PESTO}} \) can be seen as a complex *signature functionality*. The common approach for UC signature definitions [13] is to let the adversary provide the signature values \( \sigma \) (either directly or by imputing algorithms with hardcoded signing keys). Our \( F_{\text{PESTO}} \) 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 \( \sigma \) but the records the functionality maintains — it makes a big difference for our targeted 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 \( F_{\text{PESTO}} \) 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 \( \Pi_{\text{PESTO}} \) combines a distributed partially-obliviou PRF \( \text{dpOPRF} \), a distributed signature scheme \( \text{SIG} \) and standard signature \( \text{SIG} \). The \( \text{dpOPRF} \) is evaluated on the username \( uid \) (public) and password \( pw \) (private), and the \( \text{SIG} \) value is interpreted as long-term secret signing key \( \text{usk} \) of \( \text{SIG} \). Servers store the corresponding verification key \( \text{upk} \) in the user's account. When the user later requests a signature for message \( m \) and \( uid \), she re-derives \( \text{usk} \) 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 \( \text{SIG} \).

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

**Key Generation.** We assume a trusted dealer that provides all servers with their initial key shares and backup keys. That is, when a server \( S_i \) receives input \((\text{KeyGen}, sid)\) it checks that \( sid = (S_1, \ldots, S_n, sid') \). It sends \((\text{KeyGen}, sid)\) to \( F_{\text{dpOPRF}} \) and \((\text{Keygen}, sid)\) to the trusted dealer of the \( \text{SIG} \) scheme.

The trusted dealer, for a particular \( sid \) from the first message \((\text{keygen}, sid)\) to all other servers \( S_i \) generates the signing keys for all \( n \) servers as \((pk, (sk_1, \ldots, sk_n), (bk_1, \ldots, bk_n)) \leftarrow_{\mu} \text{SIG.KGen}(pp, n)\). It sets \( vk \leftarrow pk \) and returns \((vk, sk_i, bk_i)\) over a secure channel to \( S_i \). For all further messages \((\text{keygen}, sid)\) from the other servers contained in \( sid \), the dealer simply 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 6. Roughly, the servers perform a partially blind \( F_{\text{dpOPRF}} \) evaluation where the \( uid \) is the public and \( pw \) the blinded input. From the received \( F_{\text{dpOPRF}} \) output \( y \), the user deterministically derives a key pair \( \text{usk}, \text{upk} \) of a standard signature scheme, and sends \( \text{upk} \) and a signature computed from \( \text{usk} \) back to the servers. The servers now check the validity of the \( \text{upk} \) and user signature, and then use their distributed signature scheme to sign \((0, uid, \text{upk})\). The jointly computed signature \( \sigma_{\text{sig}} \) on these values ensures that an adversary cannot gradually plant malicious public keys for honest user accounts, when he transiently corrupts individual servers. Preparing the \( 1 \)-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 \((uid, \text{upk}, \sigma_{\text{sig}})\).

**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 \( F_{\text{dpOPRF}} \) output for \( uid, pw' \). The user then (re-)derives its key pair \( \text{usk}, \text{upk} \), where \( \text{uske} \) 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 \( \text{usk} \). 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 \( \sigma_{\text{sig}} \) for \((1, \text{uid}, m) \). 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 6.

**Refresh.** If a server \( S_i \) receives input \((\text{Refresh}, sid)\) it broadcasts \((\text{refresh}, sid)\) to all other servers \( S_j \in S\setminus S_i \), and sends \((\text{Refresh}, sid)\) to \( F_{\text{dpOPRF}} \). 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 \((sk_i', bk_i') \leftarrow_{\mu} \text{SIG.Refresh}(bk_i)\)
- set \( sk_i \leftarrow sk_i' \) and \( bk_i \leftarrow bk_i' \)

Server \( S_j \), receiving \((\text{refresh}, sid)\) from \( S_i \), simply runs the same update of the signature key shares described
Users, and whether these tuples contain valid signatures (as the user accounts are not contained in the trusted recover from potential loss or compromise of their states and require no updates.

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.

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 \( \{(uid, upk, \sigma_{reg, \ell})\} \) for the same set of users, and whether these tuples contain valid signatures \( \sigma_{reg, \ell} \) under \( upk \). 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 \( \sigma_{reg, \ell} \) 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. This channel is used only when a new user is registered. We further assume a trusted dealer during the setup and key generation phase. 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 Comb\(^*\) o DSIG.KGen denote the algorithm that first runs \( (vk, (sk_i)_{i \in [n]}, (bk_i)_{i \in [n]}) \leftarrow DSIG.KGen(1^n) \), discards all \( bk_i \) from the output and then obtains \( sk \leftarrow \text{Comb}^*(vk, sk_1, \ldots, sk_n) \) (with Comb\(^*\) being the algorithm inherited from the signature indistinguishability notion of DSIG).

**Theorem 3.** Consider the protocol described in Figure 6 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 Comb*, Sign* and share simulatable, then \( \Pi_{PESTO} \) securely realizes \( F_{PESTO} \) with algorithms (Comb* \& DSIG.KGen, Sign*, DSIG.Vf) in the \( F_{dpOPRF} \)-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 \( Z \) and adversary \( A \) there exists an efficient ideal-world adversary algorithm, called simulator \( S \), such that the view of \( Z \) interacting with the real scheme and \( A \), is indistinguishable from the view that \( F_{PESTO} \) and \( S \) provide. Below, we sketch the behavior of our simulator \( S \), and give the detailed proof in Appendix C.

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

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

Honest server & corrupted user: \( S \) 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 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 \( S \) knowing her password \( pw' \). However, the simulator (playing the role of corrupt servers towards \( F_{PESTO} \)) learns the necessary information just in time via the match-ok output which reveals whether \( pw' \) matches the password stored with \( uid \)'s account. This way, \( S \) knows whether to simulate a successful login of the user 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 \( F_{PESTO} \) — this is often challenging or expensive 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 \( F_{dpOPRF} \) (which is run by \( S \) in the simulation).

5.2.2. Adding Privacy to \( \Pi_{PESTO} \). 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. Benchmarking

Here we report on our proof-of-concept implementation of our full PESTO protocol for signing a JWT token. The implementation is in Java and used the MIRACL - AMCL library [40] for pairing computation and Java’s BigInteger class for implementing DSIG. We used a Boneh-Lynn-Shacham (BLS) pairing [8] with 461 bits curves. For DSIG we used RSA of 2048 bits and for SIG we used 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, 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 setting of deployment and use of our scheme. Alternatively we could have deployed all servers in the same data center on the same physical machine.

2. We note that the memory requirements of our implementation is far, far below that offered by the server.
as done in [10]. However we do believe that deploying machines at different physical locations (that could be run by different providers) increases the security significantly enough to merit the small overhead in runtime from the communication to different data centers.

We show the benchmarks of our $\Pi_{\text{PESTO}}$ protocol in Tab. 1 and Tab. 2, 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.

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. In particular the micro benchmarks do not include any of the overhead of the REST/TLS communication stack.

Comparison. For comparison we note that PASTA [5] 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) don’t. For further comparison we also ran benchmarks for 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.

Acknowledgement

This paper has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 786725 OLYMPUS, as well as 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.

The authors would like to thank Michael Bladt Stausbolm for implementing our proof-of-concept.

Part of the work of Carsten and Avishay was done while the authors were working at Bar-Ilan University, Israel. Tore would like to thank Bar-Ilan University for hosting a research visit that led collaboration with Carsten and Avishay in the beginning of this project.

References


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 $\chi$ be a random variable on $\mathbb{Z}$ whose outputs are bounded by $\beta$. Then the following holds:

1) Let $x \leftarrow [-B, B]$ and $s \leftarrow \chi$. Then any (potentially computationally unbounded) algorithm $A$ can distinguish $x + s$ with probability at most $\frac{2\beta + 2\ell + 1}{B}$.

2) Let $x_1, \ldots, x_t \leftarrow [-B, B]$ and $s \leftarrow \chi$. Then any (potentially computationally unbounded) algorithm $B$ can distinguish $\sum_{i \in [t]} x_i$ from $s + \sum_{i \in [t]} x_i$ with probability at most $\frac{2^{t+2\beta+1}}{B}$.

Proof. Consider the random variable $X$ being input to $A$ 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 $Pr[X \in [-B - \beta, B - \beta) | X = x + s] = Pr[X \in [-B - \beta, B - \beta) | X = x]$, so in that case the output of $A$ on $X$ must be identically distributed. Thus $A$ can only distinguish the cases when $X$ falls outside this interval, which happens with probability at most $\frac{2\beta + 2\ell + 1}{B}$.

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

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 DSIG.Sign$(pk, sk, m)$ parses $(f_1, \ldots, f_n) \rightarrow sk_i$ and computes $sk = \sum_{i=1}^{n} f_i \mod N$. Algorithm DSIG.Sign$(sk, m)$ outputs $H(m)^s \mod N$. Now, since the $\{f_i\}$ form an additive sharing of $sk = d$ over the integers, the result of running DSIG.Sign$(pk, sk, m)$ on the secret keys to produce a single secret key and then DSIG.Sign using that resulted secret key is equivalent to running DSIG.Sign individually with
every $sk_i$ to produce partial signatures and then running DSIG. Comb on the partial signatures. The outputs of both processes are identical by the linearity in the exponent.

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

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

On query $(\text{Sign}, m, \ell, \sigma)$, if $\sigma = \bot$ then use $\overrightarrow{sk^0_i}$ to generate $\sigma_i$ honestly. If $\sigma \neq \bot$ then output $\sigma/\prod_{j \in [n] \setminus \{i\}} \sigma_j$ where $\sigma_j = \text{DSIG.Sign}(j, \overrightarrow{sk^0_j}, m, \ell)$. On query refresh, for all $i \in [n]$ run DSIG.Refresh$(i, \overrightarrow{bk^0_i}) \rightarrow (\overrightarrow{sk^{op+1}_i}, \overrightarrow{bk^{op+1}_i})$ on all keys that SIM generated.

On query $(\text{corr}, \text{mode}, i, L)$, with mode $\in \{\text{trans, perm}\}$, the list $L$ contains all those signatures of the current epoch $ep$ for which SIM has to “adjust” the output of the random oracle $H_N$ for key $\overrightarrow{sk^0_i}$. This is done as follows: For each row $(ep, i, m, \ell, \sigma) \in L$, we already have that the equation $\sigma = \prod_{j \in [n]} \sigma_j$ holds, but we need to adjust the computation of $\sigma_i$ appropriately as we need to reveal to the adversary either $\overrightarrow{sk^0_i}$ such that $\sigma_i = H(m)_i/\prod_{j \neq i} H_N(\ell, s_{i,j}) = \sigma/\prod_{j \neq i} \sigma_j$ or $\overrightarrow{bk^0_i}$ with the same properties. We can pick the smallest $k \in [n] \setminus (C_{ep} \cup \ell C_{ep})$ and program $H_N$ as $H_N(\ell, s_{i,k}) = \sigma/\prod_{j \neq i} \sigma_j \cdot H(m)_i/\prod_{j \neq i} H_N(\ell, s_{i,j}) \mod N$. Observe that such a $k$ must always exist unless $i$ is the last uncorrupted party, in which case SIM will only output $\overrightarrow{sk^0_i}, \overrightarrow{bk^0_i} = \bot$. If the random oracle has been programmed on $H_N(\ell, s_{i,k})$ 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 $H_N$ on this input before since $s_{i,j}$ has length $\tau$ 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 $\overrightarrow{bk_i}$ we generate $mk_{i,j}$ as before, but now sample $\overrightarrow{d_n}$ differently, namely without adding $d_i$. By Lemma 1 we have that the distributions of $\overrightarrow{d_n}$ are statistically indistinguishable due to the gap on the bounds of $d_i, d_n$.

Furthermore, by Lemma 1 it also follows that each $\overrightarrow{sk_i}$ 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 $\sigma_i$ we have that due to the factor $b_i$ they are distributed uniformly at random in the protocol, subject to the constraint that their product is $\sigma$ which is preserved by SIM.

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

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

We show that the existence of an adversary $A_n$ such that $\text{Exp}_A^{\text{DSIG, forge}}(\tau) = 1$ implies an adversary $A_1$ such that $\text{Exp}_A^{\text{DSIG, forge}}(\tau) = 1$ 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.

$A_1$ participates in the experiment $\text{Exp}_A^{\text{DSIG, forge}}(\tau)$ from which it obtains the public key $pk_{\\text{own}} = (N, e)$ and runs $A_n$ internally such that $A_n$ participates in a simulation of $\text{Exp}_A^{\text{DSIG, forge}}(\tau)$. To do so, $A_1$ simulates DSIG.KGen$(pp, n)$ as in the proof of share simulatibility to obtain $(\overrightarrow{sk_i}, \overrightarrow{bk_i})$ 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 $pk = \text{pk}_{\\text{own}}$.

$A_1$ initializes $ep \leftarrow \emptyset, C_{ep} \leftarrow \emptyset, \ell C_{ep} \leftarrow \emptyset$ and forwards $pk$ to $A_n$. In addition, $A_1$ simulates the oracles responses to $A_n$ by executing their code exactly as it appears in Figure 2 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 $A_1$ would generate the output as in SIM.

Thus, the view of $A_n$ in the internal simulation is distributed identically to its view in a real experiment assuming share simulatibility, meaning that $A_n$ wins in $\text{Exp}_A^{\text{DSIG, forge}}(\tau)$ with the same probability that it wins in the simulation, meaning that $A_1$ wins $\text{Exp}_A^{\text{DSIG, forge}}(\tau)$ with the same probability as $A_n$.

### 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 Gaporn-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 $\tau$ 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 $A$ corrupts one. Let the set of corrupt parties be denoted by $C := C_i \cup \ell C_i$ and $\overline{C} := [n] \setminus C$. Because of the proactive security the size and parties in $C$ and $\overline{C}$ can change.

**Key Generation:** $S$ 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 $k = \sum_{i \in [n]} k_i \mod p$.
Random Oracles: Every oracle stores a table of each query which is used to answer repeat queries.

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

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

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

A new query to \( H_4 \) on input \( (x_{\text{priv}}, x_{\text{pub}}, \hat{y}) \) is handled as follows: Check if tuples \( \langle H_1, x_{\text{priv}}, v, \hat{x} \rangle \) and \( \langle H_2, x_{\text{pub}}, w, c \rangle \) exist. If not, pick and return a random value \( y \leftarrow \{0,1\}^r \). Otherwise, look up all tuples \( \langle lb_l, \ast, \ast, y_l \rangle \), then check if \( y_l = y_l^{1/(ew)} \) for any \( l \) and abort if it holds for multiple \( l's \). If there is such an index then let \( lb_l = lb_l', \) otherwise pick a new random label \( lb_l \). Sample a new \( ssid' \) and call \( \langle Eval, sid, ssid', x_{\text{pub}}, x_{\text{priv}} \rangle \) and delay the message \( \langle Eval, sid, ssid', x_{\text{pub}} \rangle \) otherwise returned to all honest parties by the environment. Then call \( \langle EvalComplete, sid, ssid, lb_l \rangle \) to get the message \( \langle EvalComplete, sid, ssid, y \rangle \) and learn \( y \leftarrow T(lb_l, x_{\text{pub}}, \hat{x}_{\text{priv}}) \). Then stores \( \langle lb_l, c, y \rangle \). Return \( y \) as output to the query. In case the ideal functionality does not return \( \langle EvalComplete, sid, ssid, y \rangle \) from the call, then \( S \) 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 simulator starts when \( S \) receives \( \langle Eval, sid, ssid, x_{\text{pub}} \rangle \) 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 \( \hat{x} \leftarrow g_1^v \) for \( v \leftarrow \mathbb{Z}_p \) and storing \( \langle H_1, v, \hat{x} \rangle \). It then sends \( \langle Eval, sid, ssid, x_{\text{pub}}, \hat{x} \rangle \) to each \( S_i \in \mathcal{C} \). When \( S \) has received \( \langle EvalComplete, sid, ssid, S_i, \hat{y} \rangle \) from \( A \) for all \( S_i \in \mathcal{C} \) and \( \langle EvalProceed, sid, ssid, S_i \rangle \) from the ideal functionality on behalf of \( S_i \in \mathcal{C} \) 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 \( \hat{y} = \left( \prod_{S_i \in \mathcal{E}} \hat{y}_i \right) \cdot \left( \prod_{S_i \in \mathcal{C}} \prod_{S_{i'} \in \mathcal{E}} d_{i,j} \right) \cdot \left( \prod_{S_{i'} \in \mathcal{E}} e(\hat{x}, c)^{e_{i,j}} \right) \) where \( d_{i,j} = 1 \) if \( i < j \) and \( d_{i,j} = -1 \) otherwise. The values \( d_{i,j} \) are found by looking up \( \langle H_3, d_{i,j}, sid, ssid, si_j \rangle \). (If they don’t exist or \( \langle H_2, x_{\text{pub}}, w, c \rangle \) does not exist then emulate the oracle calls.) If the adversary has acted honestly (or is lucky) then \( y = y^{1/(ew)} = e(\hat{x}, c)^{e_{i,j}} \cdot \sum_{i \in [n]} e_{i} = e(g_1, g_2) \sum_{i \in [n]} e_{i} \). If that is the case we let \( lb_l = hon \), otherwise we look up the tuples \( \langle lb_l, \ast, \ast, \hat{y} \rangle \) and check if \( \hat{y} = y^{1/(ew)} \). If we find a label \( lb_l \), where this holds let \( lb_l = lb_l' \), otherwise let \( lb_l \) be a new random label, then call the ideal functionality with \( \langle EvalComplete, sid, ssid, lb_l \rangle \). Receive back \( \langle EvalComplete, ssid, ssid, y \rangle \) and store \( \langle lb_l, \hat{x}, c, y^{1/(ew)} \rangle \).

\( U \) is dishonest: The message \( \langle Eval, sid, ssid, x_{\text{pub}}, \hat{x} \rangle \) is received by \( A \) on behalf of \( U \) towards some \( S_i \in \mathcal{C} \). Look up tuple \( \langle H_2, x_{\text{pub}}, w, c \rangle \). If it doesn’t exist, construct it. Pick a random dummy value \( x_{\text{priv}} \leftarrow \{0,1\}^r \) then \( S \) sends the query \( \langle Eval, sid, ssid, x_{\text{pub}}, priv \rangle \) to the ideal functionality and asks it to give delayed output \( \langle Eval, sid, ssid, x_{\text{pub}} \rangle \) to \( S_i \). Each time \( S \) receives \( \langle EvalProceed, sid, ssid, S_i \rangle \) from a \( S_i \in \mathcal{C} \) from the ideal functionality it ignores it if it cannot find \( \langle Eval, sid, ssid, x_{\text{pub}}, \hat{x} \rangle \) for this \( ssid \) and \( S_i \). Otherwise it looks up the values \( \langle H_2, x_{\text{pub}}, w, c \rangle \) and \( \langle H_3, d_{i,j}, ssid, ssid, si_j \rangle \) (and emulating the construction of these if they don’t exist), computes the value \( \hat{y}_i \) like in the real protocol. Send \( \langle EvalComplete, sid, ssid, S_i, \hat{y} \rangle \) to \( A \) 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 \( \langle mk_{i,j}, si_j, \delta_{i,j} \rangle \leftarrow PRG(mk_{i,j}) \) 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 \( si_j, \delta_{i,j} \) uniformly at random and computes the new state for the honest parties like in the real protocol.

Corruption: When \( A \) transiently corrupts a server \( S_i \) then \( S \) 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, \( x_{\text{priv}} \), and choice of nonce \( r' \) where the simulator previously sent a value \( \hat{x} \). See if there is an entry \( \langle H_1, x_{\text{priv}}, v, \hat{x} \rangle \), otherwise construct such an entry for \( x_{\text{priv}} \). Then construct a simulated nonce as \( r' \leftarrow r/v \) and use this to return to \( A \), thus \( \hat{x} = H_1(x_{\text{priv}}) = g_1^{r/v} = g_1^{r'v} \).

When \( A \) permanently corrupts a server \( S_i \) then \( S \) picks \( mk_{i,j} \leftarrow \{0,1\}^r \) (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 \( A \) 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 \( x_{\text{pub}} \) and thus whether we define the output when an adversary
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 $A$ queries $H_4$ without corrupting the user then even if $A$ uses the correct $x_{\text{priv}}$ (given by the environment) it cannot return a correctly computed value $\hat{y}$ 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 $\text{EvalComplete}(sid, ssid, S_i, \hat{y}_i)$ to the simulator is if $\text{EvalProceed}(sid, ssid)$ has not been received from all the honest $S_i$ for the same amount of concurrent queries of $U$ on $x_{\text{pub}}$ as the adversary has started. This is the only time fail will be output.

**Reduction to the Gap om-BDH Problem:** Suppose we are given a PPT adversary $A$ and a PPT environment $Z$ that can cause fault to happen. We now show how to use these to construct an algorithm $B$ that solves the gap om-BDH problem with essentially the same probability. $B$ is given $G$ and $g^2_k$ from the Gapom-BDH experiment and also access to the oracles $\text{O}_{1,2}, \text{O}_{\text{help}}$ and $\text{O}_{\text{help}}$. 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 $B$ needing to guess a server that will remain uncorrupted in each epoch. The key generation in the following is then to notice that $B$ never needs to simulate values for all $k_i$, because otherwise the fail event will not occur since $B$ will need to be missing at least one share $\hat{y}_i$ 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 $(\text{hon}, g_1, g_2, e(g_1, g_2^k))$ For the random oracles we now use $\text{O}_{1,2}$ oracles to sample random elements for $H_1$ and $H_2$, $\bar{x}$ respectively c. $B$ then stores tuples $\langle H_1, x_{\text{priv}}, \cdot, x \rangle$ and $\langle H_2, x_{\text{pub}}, \cdot, c \rangle$ now, since it does not know the discrete logarithm of $\bar{x}$ respectively c.

A new query to $H_4$ on input $(x_{\text{priv}}, x_{\text{pub}}, y)$ is now handled as follows: $B$ checks if tuples $\langle H_1, x_{\text{priv}}, \cdot, x \rangle$ and $\langle H_2, x_{\text{pub}}, \cdot, c \rangle$ exist. If not, pick and return a random value $y \leftarrow \{0, 1\}^\ell$, otherwise look up all tuples $\langle \text{lbl}, \bar{x}, c_i, \hat{y}_i \rangle$, and $\bar{y}$ and find an index $l$ where this holds. If that happens then let $\text{lbl} \leftarrow \text{lbl}_i$ otherwise pick a new random label $\text{lbl}$. Sample a new $\text{ssid}$ and call $\text{Eval}(\text{sid}, \text{ssid}, x_{\text{pub}}, x_{\text{priv}})$ and the receive the message $\langle \text{EvalComplete}(\text{sid}, \text{ssid}, \text{lbl}) \rangle$ otherwise return to all honest parties by the environment. Then call $\text{EvalComplete}(\text{sid}, \text{ssid}, \text{lbl})$ to get the message $\langle \text{EvalComplete}(\text{sid}, \text{ssid}, y) \rangle$ and learn $y \leftarrow T'(\text{lbl}, x_{\text{pub}}, x_{\text{priv}})$. $B$ then stores $\langle \text{lbl}, \bar{x}, c, y \rangle$. Return $y$ as output to the query. In case the ideal functionality does not return $\text{EvalComplete}(\text{sid}, \text{ssid}, y)$ from the call, then $S$ outputs fail and adds $(\bar{x}, c, y)$ 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 $\bar{x}$ 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 $e(\bar{x}, c)$, which should be the base if $\hat{y}$ 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 $\langle \text{EvalComplete}, \text{sid}, \text{ssid}, S_i, \hat{y}_i \rangle$ from the adversary. In that case do as before when constructing the variable $\hat{y}_i$. However since we don’t have the honest parties’ key shares, instead use the oracle $e(\bar{x}, c)^k \leftarrow \text{O}_{\text{help}}(e(\bar{x}, c))$. Add $(\bar{x}, c, e(\bar{x}, c)^k)$ to the list of solutions to give to the Gapom-BDH experiment. Next compute $y_i = (\prod_{y \in C} y_i) \cdot (\prod_{y \in C^*} \prod_{i,j \in C} d_{i,j}^{e(\bar{x}, c)})$. Then look up the tuples $(\text{lbl}_i, \bar{x}_i, c_i, \hat{y}_i)$ and use the $\text{O}_{\text{help}}$ to compute $1 \leftarrow \text{O}_{\text{help}}(e(\bar{x}, c), y_i \cdot e(\bar{x}_i, c_i), \hat{y}_i)$. If we find a label $\text{lbl}_i$ where this holds let $\text{lbl} \leftarrow \text{lbl}_i$, otherwise let $\text{lbl}$ be a new random label, then call the ideal functionality with $\langle \text{EvalComplete}, \text{sid}, \text{ssid}, \text{lbl} \rangle$. Receive back $\langle \text{EvalComplete}, \text{sid}, \text{ssid}, y \rangle$ and store $\langle \text{lbl}, \bar{x}, c, y \rangle$.

**$U$ dishonest:** On receiving $\langle \text{EvalProceed}, \text{sid}, \text{ssid}, S_i \rangle$ from all, except the last honest party $S_i \in C$, simply return $\langle \text{EvalComplete}, \text{sid}, \text{ssid}, S_i, \hat{y}_i \rangle$ for $\hat{y}_i \leftarrow \text{G}_T$. 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 $e(\bar{x}, c)^k \leftarrow \text{O}_{\text{help}}(e(\bar{x}, c))$ using $\bar{x}$ already received and $c$ from the $H_2$ table. Based on this compute $y_i \leftarrow e(\bar{x}, c)^k \leftarrow \text{O}_{\text{help}}(e(\bar{x}, c)) \leftarrow \text{G}_T$. 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 one the sent to the last honest party.

**Corruption and refresh:** When $A$ corrupts $S_i$ then $B$ chooses a random key $k_i$ and blinding seeds $s_{i,j} \in \{0, 1\}^\ell$ (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 $\hat{y}_i = e(\bar{x}, c)^k \cdot \prod_{y \in C} y_i$, $H_3(\text{sid}, \text{ ssid}, s_{i,j})$ (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 $\hat{y}_i$ (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 $\hat{y}_i$) 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 $y$ already defined internally in the simulator.

If $A$ permanently corrupts $S_i$ then $B$ 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 $B$ takes back control and forget all previously chosen keys for that server.

**CDH Solutions:** When fail occurs then $B$ 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 $B$ must simulate values from all honest servers (when the message $(\text{Eval}, \text{sid}, \text{ssid}, \text{uid})$ is given to $B$ by the ideal functionality on request from $\mathcal{Z}$). There $B$ wins the experiment by returning a list of values that wins the Gapm-BDH experiment.

**Appendix C.**

**Proof of Theorem 3 (Security of PESTO)**

We first note that we can restrict the analysis to static user corruptions, although our protocol provides security against adaptively corrupted users. The reasoning for the restriction is as follows: since users are supposed to delete any secret values after usage (e.g., $y$ and usk), an adversary does not gain any advantage from corrupting a user ITI after the protocol started. For this, note that party identifiers do not play any role in our protocol besides providing routing information, and thus we can assume that $\mathcal{Z}$ chooses a fresh user party for every fresh subsession. Thus, statically corrupting a user who attempts to log in with a previously honestly $\text{uid}$ naturally models adaptive user corruptions.

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

**Game 2:** Adding $\mathcal{F}$ with KeyGen and Register.

We regroup the real protocol execution into one ITI called simulator or $\mathcal{S}$ (we often refer to this as protocol simulation; this now includes $\mathcal{A}, \mathcal{F}_{\text{dpOPRF}}$, the trusted dealer and the private backup tapes of servers). Additionally, we add an ITI $\mathcal{F}$ with all interfaces of $\mathcal{F}_{\text{PESTO}}$ except for $\text{ProcedReg}$ and Verify but modified such that $\mathcal{F}$ discloses passwords to the adversary. For KeyGen, we let $\mathcal{F}$ read $\mathcal{n}$, the number of participating servers, from its $\text{sid}$. Then, $\mathcal{F}$ runs $pp \leftarrow \text{DSIG.Setup}(1^n)$, $(sk_1, \ldots, sk_n) \leftarrow \text{DSIG.KGen}(pp, n)$ 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).

The simulator $\mathcal{S}$ is changed in the following way: $\mathcal{S}$ uses key shares $sk_1, \ldots, sk_n$ to simulate honest servers and answer corruption queries issued by $\mathcal{Z}$. $\mathcal{S}$ 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. The correspondences are as follows:

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

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

- input $(\text{Register}, \text{sid}, \text{ssid}, \text{uid}) \leftrightarrow$
  ack. and simulate $(\text{reg}, \text{sid}, \text{ssid}, \text{uid})$
- input $(\text{ProcedReg}, \text{sid}, \text{ssid}, S_i) \leftrightarrow$
  acknowledge and simulate $\mathcal{F}_{\text{dpOPRF}}$ usage
- input $(\text{Sign}, \text{sid}, \text{ssid}, \text{uid}) \leftrightarrow$
  acknowledge and simulate $(\text{sign}, \text{sid}, \text{ssid}, \text{uid}, m)$

We now detail how $\mathcal{S}$ provides inputs to $\mathcal{F}$ in case of corrupted protocol participants. In case of a corrupted user, whenever $\mathcal{A}$ sends an input $(\text{Eval}, \text{sid}, \text{ssid}, \text{uid}, pw)$ to $\mathcal{F}_{\text{dpOPRF}}$, if $\mathcal{A}$ sent $(\text{Register}, \text{sid}, \text{ssid}, \text{uid})$ to any $S_i$ before, then $\mathcal{S}$ sends an input $(\text{Register}, \text{sid}, \text{ssid}, \text{uid}, pw)$ to $\mathcal{F}$ on behalf of the corrupted $U$. Else, if $\mathcal{A}$ sent $(\text{Sign}, \text{sid}, \text{ssid}, \text{uid}, m)$ to any $S_i$ before, then $\mathcal{S}$ sends an input $(\text{Sign}, \text{sid}, \text{ssid}, \text{uid}, pw)$ to $\mathcal{F}$ on behalf of the corrupted user. In case of a corrupted server $S_i$, $\mathcal{S}$ sends 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 $(\text{Eval}, \text{proceed}, \text{sid}, \text{ssid})$ to $\mathcal{F}_{\text{dpOPRF}}$, respectively, where the latter query’s sub-session identifier $\text{ssid}$ was formerly received within an input $(\text{Sign}, \text{sid}, \text{ssid}, ...)$. from $\mathcal{F}$.

As a last change in this game, we also add dummy parties between $\mathcal{F}$ and $\mathcal{Z}$, namely one per real party.

For indistinguishability, we first observe that the output pattern of $\mathcal{F}$ matches the protocol and $\mathcal{F}$ starts working in all corruption scenarios thanks to $\mathcal{S}$ providing timely and complete inputs to $\mathcal{F}$. Regarding completeness of inputs it is worth noting that servers do not input any secret values into $\mathcal{F}_{\text{PESTO}}$ - the secret input value is $pw$ provided by the user. However, in case the user is corrupted, $\mathcal{S}$ learns $pw$ when the corrupted user queries $\mathcal{F}_{\text{dpOPRF}}$.

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 the 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 $ok$ obtained from $\mathcal{F}_{\text{DKG}}$, which is exactly what happens in Game 2. For user registration, we have to argue that $\mathcal{U}$ outputs Registered in Game 2 if and only if $\mathcal{U}$ 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 between user and servers. This concludes the indistinguishability argument.

**Game 3:** Abort whenever SIG signature is forged. Whenever $\mathcal{Z}$ sends a message $(\text{sid}, \text{ssid}, \text{upk}, \sigma_U)$ to a server on behalf of a corrupted user with $\text{SIG.V}(\text{upk}, (\text{uid}, \text{ssid}), \sigma_U) = 1$, where there was already a query $(\text{sid}, \text{ssid}, \text{uid}, \text{upk}, \sigma_U')$ with $\text{ssid} \neq \text{ssid}'$ and a message $(\text{Registered}, \text{sid}, \text{ssid}, \text{uid})$ from $\mathcal{F}$ to some honest user $U$ delayed by $\mathcal{S}$, then $\mathcal{S}$ aborts the simulation.

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

The first case corresponds to \( Z \) maliciously registering a user \( \text{uid} \) with public key \( \text{upk} \), where \( \text{upk} \) is already the public key of a successfully registered honest user \( \text{uid}' \). Since the user is honest, \( Z \) does not know the corresponding \( \text{usk} \) and thus a verifying \( \sigma \) would constitute a forgery. It follows from existential unforgeability of SIG that this happens only with negligible probability.

The second case corresponds to \( Z \) trying to impersonate an honestly registered \( \text{uid} \) within a signing request. Similar to the case before, since \( \text{uid} \) got registered in a request issued by an honest user, \( Z \) does not know the corresponding signing key \( \text{usk} \). As before, unforgeability of SIG signatures is enough to argue computational indistinguishability in this case. Altogether, it follows that Game 2 is computationally indistinguishable from Game 3.

**Game 4: Abort when forged DSIG signature is verified.** We let \( S \) abort the protocol simulation in case of \( Z \) providing an input \((\text{Verify}, \text{sid}, \text{uid}, m, \sigma, e, vk)\) to any party \( V \) where \( \text{Verify}(vk, (\text{uid}, m), \sigma) = 1 \) and \( \sigma \) was never contained in any former Signature output of the execution, nor could it be assembled using keys of corrupted parties and signature shares.

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

- Whenever \( F \) outputs \((\text{KeyConf}, \text{sid}, vk)\), \( B \) overwrites this with \((\text{KeyConf}, \text{sid}, pk)\)
- Whenever \( F \) generates an output containing a signature \( \sigma \leftarrow \text{Sign}(sk, (\text{uid}, m)) \) within a query \((\text{ProceedSign}, \text{sid}, \text{ssid}, ..., \text{...})\), \( B \) obtains \( \sigma_i \leftarrow O_{\text{sign}}(i, (\text{uid}, m), \text{ssid}) \) for all \( i \in [n] \) with \( S_i \) being honest and generates \( \sigma \leftarrow \text{DSIG.Sign}(sk_i, (\text{uid}, m); \text{ssid}) \) using \( sk_i \) for all corrupted \( S_i \). Then \( B \) overwrites the output with \( \sigma \leftarrow \text{DSIG.IComb}(\sigma_1, ..., \sigma_n) \)
- Similarly, \( B \) overwrites signatures \( \sigma_{\text{reg}} \) in \( uid \) records stored at a server receiving a \((\text{Corrupt}, \text{sid}, S_i, *)\) query, using his \( O_{\text{sign}} \) oracle and DSIG.IComb() in the same way as above.
- Whenever a server sends an input \((\text{refresh}, \text{sid})\) to \( F \), \( B \) calls his \( O_{\text{refresh}} \) oracle once
- Whenever \( Z \) issues a request \((\text{Corrupt}, \text{sid}, S_i, \text{mode})\) for a currently honest \( S_i \), \( B \) queries \( k \leftarrow O_{\text{corrupt}}(i, \text{mode}) \) 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 \((\text{ProceedSign}, \text{sid}, \text{ssid}, ...)\) or a \((\text{ProceedReg}, \text{sid}, \text{ssid}, ...)\) query, \( B \) queries \( O_{\text{sign}}(i, m, \text{ssid}) \) and sets the answer \( \sigma_i \) 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 \( B \) instead of being created within \( F \). For this, it is crucial to see that the above is a complete list of all key-related values that \( Z \) can observe from the execution of Game 3 or Game 4.

A distinguisher \( D \) between both consecutive games can only be successful if he causes the execution in Game 4 to abort, so if he submits a signature \( \sigma^* \) on message \( m^* \) for verification with the properties described above. Since \( \sigma^* \) cannot be assembled from secret key shares, we know that there is no \( e \) such that \( |C_{t,e} \cup C_{p,e}| \neq n \). Since \( \sigma^* \) was also never output by \( F_{\text{PESTO}} \) 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 \( B \) requests \( O_{\text{sign}}(i, m^*, \text{ssid}) \) for all honest servers \( S_i \) and some \( \text{ssid} \). By this, we see that \((m^*, \sigma^*) \) constitutes a forgery and lets \( B \) win his proactive unforgeability game.

**Game 5: Abort when forged DSIG signature is stored with user account.** \( S \) aborts in case of \( S_i \) storing record \((\text{uid}, \text{upk}, \sigma_{\text{reg}})\) in some epoch \( e \) that differs from the record stored in epoch \( e = 1 \), where \( \text{DSIG.Vi}(vk, (\text{uid}, \text{upk}), \sigma_{\text{reg}}) = 1 \) and where there was a \((\text{Corrupt}, \text{sid}, S_i, \text{trans})\) query in epoch \( e \).

To argue indistinguishability, we can use the same reduction to proactive unforgeability as in Game 4, i.e., the same adversary \( B \) to emulate an execution of Game 4 and Game 5.

Let us note that, due to the changes made in this game and usage of a secure broadcast channel, in every execution all honest servers who do not abort during a signing procedure are guaranteed to have the same user signing key \( \text{upk} \) stored for each account \( \text{uid} \).

**Game 6: Simulate DSIG key shares.** We let \( S \) generate and refresh secret key shares (to give them to \( Z \) upon corruption) and signature shares (simulating honest servers) using algorithms \( \text{Sim.refresh} \), \( \text{Sim.corrupt,...} \) and \( \text{Sim.sign,...} \) as mentioned in Figure 2. Whenever the two latter algorithms are used to compute the last key or signature share missing to let \( Z \) complete signatures (using only DSIG.Sign() and DSIG.IComb()), \( S \) uses the key shares \( sk_1, ..., sk_n \) forwarded by \( F_{\text{PESTO}} \) to compute all those signatures as DSIG.Comb(DSIG.Sign(sk_1, ...), ..., DSIG.Sign(sk_n, ...)) and feed them to the \( \text{Sim}() \) algorithms. Else, \( S \) simply runs \( \text{Sim}(\text{sign}, \text{...,}, \text{shed}) \) or \( \text{Sim.corrupt...,0} \).

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

- Whenever \( F \) outputs \((\text{KeyConf}, \text{sid}, vk)\), \( B \) overwrites this with \((\text{KeyConf}, \text{sid}, pk)\)
- Whenever a server sends an input \((\text{refresh}, \text{sid})\) to \( F \), \( B \) calls \( O_{\text{refresh}}() \)
- Whenever \( Z \) issues \((\text{Corrupt}, \text{sid}, S_i, \text{mode})\) for a currently honest \( S_i \), \( B \) queries \( k \leftarrow O_{\text{corrupt}}(i, \text{mode}) \) and sets \( k \) to be the key(s) of \( S_i \)
- Whenever an honest server \( S_i \) generates a DSIG signature share on message \( m \), i.e., within a \((\text{ProceedSign}, \text{sid}, \text{ssid}, ...)\) or a \((\text{ProceedReg}, \text{sid}, \text{ssid}, ...)\) query, \( B \) queries \( O_{\text{sign}}(i, m, \text{ssid}) \) and sets \( \sigma_i \) to be the signature share send by the honest server
For indistinguishability, we need to argue that $B$ using oracles from Figure 2 emulates the execution from Game 5, while $B$ using oracles from Figure 2 emulates the execution from Game 6.

For the former, we observe that the oracles from Figure 2 follow the protocol instructions, e.g., signature shares are computed using $\text{DSIG.Sign}(sk_1, \ldots, sk_n)$, which matches the protocol simulation from Game 5. For the latter, $S$ uses the simulation algorithms in the same way as the oracles from Figure 2, and using knowledge of the true $sk_1, \ldots, sk_n$, he can compute the auxiliary inputs to these algorithms in the same way as done by the compList, compSig algorithms. Thus, indistinguishability of this and the previous game follows from the share simulatability of DSIG.

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

The simulation is changed as follows: $S$ acknowledges $\text{ProceedSign}$ for $S_i$ directly and then starts simulating usage of $\text{FpeOPRF}$. $S$ sends $\text{ProceedSign}$ to $F$ on behalf of a corrupted user whenever $A$ sends $(\text{EvalProced}, sid, ssid)$ for an ssid that was contained in a former input $(\text{Sign}, sid, ssid, \ldots)$. $S$ sets $b^* = 1$ if all of the $(sid, ssid, \sigma, \text{sid}, \text{ssid})$ messages delivered to the $n$ servers have $\text{DSIG.Vf}((upk, (uid, ssid'), \sigma')$; $\sigma') = 1$; else $S$ sets $b^* = 0$. $S$ replies to a $(\text{match-ok}, sid, ssid, b)$ message from $F$ with $(\text{match-ok}, sid, ssid, b^*)$. As a last change, whenever $S$ receives output $(\text{Signature}, sid, ssid, \sigma)$ (where $\sigma$ is a verifying signature on message $m$) towards a corrupted $U$, $S$ sets $\sigma$ to be the result of $\text{compSig}((ep, i, m, ssid))$, where $ep$ is the current epoch computable from the number of times $S$ called $\text{SIM(refresh)}$, and $i$ is the last index missing an entry $(ep, \cdot, ssid)$ in $Q$ for an honest $S_i$. If $S_i$ is corrupted, then $S$ adds $\sigma$ to the output of $\text{compList}(ep, i)$. Our indistinguishability argument will be split in four cases. The first three argue indistinguishability of outputs generated upon $(\text{Verify}, \ldots)$ and $(\text{Sign}, \ldots)$ queries, and the latter argues that values computed by $S$ are distributed as in the last game.

First, regarding outputs for Verify queries, $F$ 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 $b = 1$, in which case the signing procedure was completed honestly and resulted in a 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 the case where the signing procedure does not lead to $U$ outputting Signature with a signature. In Game 6 this happens when password authentication failed (due to wrong password or $A$ 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 the adversary 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 6), the latter happens if $\text{SIG.Vf}(upk,(uid, ssid'), \sigma') = 0$ for at least one of the messages sent by $U$ to a server. But in that case, the above simulation ensures $S$ sending $b^* = 0$ and $F$ adopting this bit by setting $b \leftarrow b^*$ (due to either a corrupted server or a corrupted user).

Thirdly, consider the case where the signing procedure leads to $U$ outputting a signature. In this case, we have to argue that signatures $\sigma \leftarrow \text{Sign}(sk,(uid,m))$ generated by $F$ in Game 7 are indistinguishable from signatures generated as in the real protocol execution using $\text{DSIG.Comb}(\sigma_1, \ldots, \sigma_n)$, as done in Game 6. This follows from the signature indistinguishability property of DSIG (cf. Definition 7).

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

Altogether, it follows that both games are indistinguishable.

**Game 8: Simulate without pw in honest case.** We let $F$ no longer tell $S$ about passwords (from Register and Sign queries) in case everybody is honest. $S$ is changed as follows: it uses a dummy password $pw_S$ in the protocol simulation in every Register or Sign input of an honest user.

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

**Game 9: Simulate user’s transcript without pw in case of corrupted servers.** We consider the case where some subset of servers is corrupted. Opposed to the all-honest case before, we now have to provide a transcript of messages from an honest user, but generated without knowing the password of the user. We change the simulation in case of a Sign, uid request of an honest user, where (account, uid, *) is already registered in $F$. Let upk denote the public key corresponding to uid that was generated in the protocol simulation during the registration
phase, usk the corresponding secret key and lbl the label that was sent by A in the corresponding \( F_{dpOPRF} \) session (i.e., lbl represents the pseudo-random function that was evaluated during the registration of uid). Upon receiving an output \((\text{Match}, sid, ssid, b)\) for a corrupted \( S_i \) and a session ssid invoked by an honest user \( U \), if \( b = 1 \) then \( S \) sends \((sid, ssid, \sigma'_U)\) on behalf of \( U \) to all \( S_i \) in the protocol simulation, where \( \sigma'_U \leftarrow \text{SIG}.\text{Sign}(usk, (uid, ssid)) \). If \( b = 0 \), \( S \) sends a random non-verifying \( \sigma'_U \).

In case of receiving \((\text{Eval}, sid, ssid, uid, pw)\) from a corrupted \( U \) intended to \( F_{dpOPRF} \), in case \( S \) received \((\text{match-ok}, sid, ssid, 1)\) from \( F \), he sends \((\text{EvalComplete}, sid, ssid, y)\) to the corrupted \( U \) where \( y \) is the former output of \( F_{dpOPRF} \) from registration of uid if \( A \) sent \((\text{EvalComplete}, sid, ssid, lbl)\) (i.e., using the same label lbl as during registration) and a freshly drawn \( y \) otherwise. In case of receiving \((\text{match-ok}, sid, ssid, 0)\) from \( F \), \( S \) 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 \( F \) keeps no secret server inputs from \( S \). For the case of the honest user attempting to sign, only the message \((sid, ssid, \sigma'_U)\) has to be simulated. Since \( S \) obtains the information of whether \( \sigma'_U \) should be a verifying signature of \((uid, ssid)\) or not from \( F \) via a \text{Match} output towards a corrupted server, this message is equally distributed as in the previous game. For the case of a corrupted user, we only need to argue that simulation of \( F_{dpOPRF} \) is indistinguishable in both games. For this, we only have to argue that the \text{match-ok} message by \( F \) actually provides \( S \) with the crucial information of whether to adjust \( F_{dpOPRF} \) outputs to the corresponding output in the registration session of uid or not. To see this, note that \text{match-ok} leaks whether input passwords matched or not in case of a corrupted user.

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