# One TPM to Bind Them All: Fixing TPM 2.0 for Provably Secure Anonymous Attestation

Jan Camenisch[1], Liqun Chen[2], Manu Drijvers[1,3], Anja Lehmann[1], David Novick[4], and Rainer Urian[5]

[1] IBM Research – Zurich
[2] University of Surrey
[3] ETH Zurich
[4] Intel
[5] Infineon

**Abstract.** The Trusted Platform Module (TPM) is an international standard for a security chip that can be used for the management of cryptographic keys and for remote attestation. The specification of the most recent TPM 2.0 interfaces for direct anonymous attestation unfortunately has a number of severe shortcomings. First of all, they do not allow for security proofs (indeed, the published proofs are incorrect). Second, they provide a Diffie-Hellman oracle w.r.t. the secret key of the TPM, weakening the security and preventing forward anonymity of attestations. Fixes to these problems have been proposed, but they create new issues: they enable a fraudulent TPM to encode information into an attestation signature, which could be used to break anonymity or to leak the secret key. Furthermore, all proposed ways to remove the Diffie-Hellman oracle either strongly limit the functionality of the TPM or would require significant changes to the TPM 2.0 interfaces. In this paper we provide a better specification of the TPM 2.0 interfaces that addresses these problems and requires only minimal changes to the current TPM 2.0 commands. We then show how to use the revised interfaces to build $q$-SDH- and LRSW-based anonymous attestation schemes, and prove their security. We finally discuss how to obtain other schemes addressing different use cases such as key-binding for U-Prove and e-cash.

## 1 Introduction

The amount of devices connected to the Internet grows rapidly and securing these devices and our electronic infrastructure becomes increasingly difficult, in particular because a large fraction of devices cannot be managed by security professional nor can they be protected by firewalls. One approach to achieve better security is to equip these devices with a root of trust, such as a Trusted Platform Module (TPM), a Trusted Execution Environment (TEE), and Software Guard Extensions (SGX), and then have that root of trust attest to the state of the device or to computations made. When doing such attestations, it is crucial that they be privacy-protecting. On the one hand, to protect the privacy of users of such devices, and on the other hand, to minimize the information available to attackers. Realizing this, the Trusted Computing Group (TCG) has developed a protocol called direct anonymous attestation (DAA) [BCC04] and included it in their TPM 1.2 specification [Tru04]. The protocol allows a device to authenticate as a genuine device (i.e., that it is certified by the manufacturer) and attest to messages without the different attestations being linkable to each other and has since been implemented in millions of chips.

Later, Brickell and Li [BL11] proposed a scheme called Enhanced-privacy ID (EPID) that is based on elliptic curves and adds *signature-based* revocation which is a revocation capability based on a previous signature of a platform. This scheme has become Intel's recommendation for attestation of a trusted system, has been incorporated in Intel chipsets and processors, and is recommended by Intel to serve as the industry standard for authentication in the Internet of Things. Being based on elliptic curves, EPID is much more efficient than the original RSA-based DAA scheme. Therefore, the TCG has revised the specification of the TPM and switched to elliptic curve-based attestation schemes [Tru14, CL13]. The design idea of this new specification is rather beautiful: the TPM only executes a simple core protocol that can be extended to build different attestation schemes. Essentially, the core protocol is a Schnorr proof of knowledge of a discrete logarithm [Sch91], the discrete logarithm being the secret key stored and protected inside the TPM. Chen and Li [CL13] describe how to extend this proof of knowledge to DAA schemes, one based on the $q$-SDH assumption [BB08] and one based on the LRSW assumption [LRSW99]. The idea here is that the host in which the TPM is embedded extends the protocol messages output by the TPM into messages of the DAA protocol. They further show how to extend it to realize device-bound U-Prove [PZ13], so that the U-Prove user secret key is the one stored inside the TPM.

Unfortunately, the core protocol as specified has severe shortcomings. First, the random oracle based security proof for attestation unforgeability by Chen and Li is flawed [XYZF14] and indeed it seems impossible to prove that a host cannot attest to a message without involving the TPM. Second, the core protocol can be abused as a Diffie-Hellman oracle w.r.t. the secret key $tsk$ inside the TPM. It was shown that such an oracle weakens the security, as it leaks a lot of information about $tsk$ [BG04]. Further, the presence of the oracle prevents forward anonymity, as an attacker compromising a host can identify the attestations stemming from this host.

These issues were all pointed out in the literature before and fixes have been proposed [XYZF14, CDL16c, CDL16a]. However, the proposed fixes either introduce new problems or are hard to realize. Xi et al. [XYZF14] propose a change to the TPM specification that allows one to prove the unforgeability of TPM-based attestations. This change introduces a subliminal channel though, i.e., a subverted TPM could now embed information into the values it produces and thereby into the final attestation. This covert channel could be used to break anonymity of the platform and its user, or to leak the secret key held in the TPM. The proposed fixes to remove the static Diffie-Hellman oracle [XYZF14, CDL16c, CDL16a] either require substantial changes to the TPM to the extend that they are not implementable, or restrict the functionality of the TPM too much, excluding some major DAA schemes from being supported. For instance, it was priorly proposed to have the host prove in zero knowledge that a new base is safe to use for the TPM, who then needs to verify that proof. This does not only take a heavy toll on the resources of the TPM but also excludes signature-based revocation, thus not meeting the requirements of the TCG. We refer to Section 3 for a detailed discussion of the existing proposals and their shortcomings.

*Our Contributions.* In this paper we provide a new specification of the DAA-related interfaces of the TPM that requires only minimal changes to the current TPM 2.0 commands. It is the first one that addresses all the issues discussed and that can easily be implemented on a TPM. We then show what kind of proof of knowledge statements can be proven with the help of our new TPM interfaces and how to build secure DAA schemes with them. Our specification supports both LRSW-based and $q$-SDH-based direct anonymous attestation, signature-based revocation, and extensions to attributes. Our LRSW-based scheme has a new way to issue credentials that is much more efficient than prior ones that aimed to avoid a DH-oracle in the TPM interfaces. To achieve this, we use a slight modification of the LRSW assumption (which we prove to hold in the generic group model). Avoiding this modification would be possible, but would require a second round of communication with the issuer.

We further show how to extend our DAA schemes to support attributes and signature-based revocation and give security proofs for all of that. The TPM interfaces that we give can also be used to realize other schemes, such as device-bound U-Prove [PZ13] and e-cash [CHL05], for which it is beneficial that a secret key be kept securely inside a TPM.

To make the construction of such schemes easier, we give for the first time a thorough characterization of statements that can be proven with a TPM w.r.t. a secret key inside the TPM. We provide a generic protocol that orchestrates our new TPM interfaces and allows one to generate TPM-based proofs for a wide class of statements. We further prove the security of such generated TPM-based proofs. This facilitates the use of the TPM interfaces for protocol designers who can simply use our generic proof protocol to devise more complex protocols.

Some of the changes to the TPM 2.0 interfaces we propose have already been adopted by the TCG and will appear in the forthcoming revision of the TPM 2.0 specifications. The remaining changes are currently under review by the TPM working group. Furthermore, the authors are in discussion with the other bodies standardizing DAA protocols to adopt our changes and schemes, in particular ISO w.r.t. to ISO/IEC 20008-2, Intel for EPID, and with the FIDO alliance for their specification of anonymous attestation [CDE$^+$], so that all of these standards will define provably secure protocols that are compatible with each other.

*Outline.* We start by presenting the necessary preliminaries in Section 2. In Section 3, we describe the current TPM 2.0 commands and their inherent security issues and also discuss how previous work aims to overcome these problems. Section 4 then presents our proposed changes to the TPM 2.0 specification and our generic proof protocol to create TPM-based attestations. How to build direct anonymous attestation with signature-based revocation and attributes is described in Section 5. We discuss forward anonymity separately in Section 6, show other applications of the revised TPM interfaces in Section 7, and conclude in Section 8.

## 2 Building Blocks and Assumptions

This section introduces the notation for signature proofs of knowledge and the complexity assumptions required for our schemes. Here we also present the new generalized version of the LRSW assumption.

### 2.1 Bilinear Maps

Let $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ be groups of prime order $p$. A bilinear map $\mathbf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ must satisfy bilinearity, i.e., $\mathbf{e}(g_1^x, g_2^y) = \mathbf{e}(g_1, g_2)^{xy}$ for all $x, y \in \mathbb{Z}_q$; non-degeneracy, i.e., for all generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, $\mathbf{e}(g_1, g_2)$ generates $\mathbb{G}_T$; and efficiency, i.e., there exists an efficient algorithm $\mathcal{G}(1^\tau)$ that outputs the bilinear group $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ and an efficient algorithm to compute $\mathbf{e}(a, b)$ for any $a \in \mathbb{G}_1$, $b \in \mathbb{G}_2$.

Galbraith et al. [GPS08] distinguish three types of pairings: Type-1, in which $\mathbb{G}_1 = \mathbb{G}_2$; Type-2, in which $\mathbb{G}_1 \neq \mathbb{G}_2$ and there exists an efficient isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$; and Type-3, in which $\mathbb{G}_1 \neq \mathbb{G}_2$ and no such isomorphism exists. Type-3 pairings currently allow for the most efficient operations in $\mathbb{G}_1$ given a security level using Barreto-Naehrig curves with a high embedding degree [BN06]. Therefore it is desirable to describe a cryptographic scheme in a Type-3 setting, i.e., without assuming $\mathbb{G}_1 = \mathbb{G}_2$ or the existence of an efficient isomorphism from $\mathbb{G}_2$ to $\mathbb{G}_1$.

### 2.2 Complexity Assumptions

We recall some existing complexity assumptions and introduce a variation of one of them (which we prove to hold in the generic group model). Let $\mathcal{G}(1^\tau)$ generate random groups $\mathbb{G}_1 = \langle g_1 \rangle$, $\mathbb{G}_2 = \langle g_2 \rangle$, $\mathbb{G}_T = \langle \mathbf{e}(g_1, g_2) \rangle$, all of prime order $p$ where $p$ has bith length $\tau$, with bilinear map $\mathbf{e}$.

Recall the $q$-SDH assumption [BB08] and the LRSW assumption [LRSW99] in a bilinear group.

**Assumption 1 ($q$-SDH)** *Define the advantage of $\mathcal{A}$ as:*

$$\mathsf{Adv}(\mathcal{A}) = \mathsf{Pr}\Big[(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \mathbf{e}, q) \leftarrow \mathcal{G}(1^\tau), x \xleftarrow{\$} \mathbb{Z}_p^*, \\ (c, h) \leftarrow \mathcal{A}(g_1, g_1^x, g_1^{(x^2)}, \ldots, g_1^{(x^q)}, g_2, g_2^x) : h = g_1^{\frac{1}{x+c}}\Big] \ .$$

*No PPT adversary has $\mathsf{Adv}(\mathcal{A})$ non-negligible in $\tau$.*

**Assumption 2 (LRSW)** *Let $X = g_2^x$ and $Y = g_2^y$, and let $\mathcal{O}_{X,Y}(\cdot)$ be an oracle that, on input a value $m \in \mathbb{Z}_p$, outputs a triple $(a, a^y, a^{x+xym})$ for a randomly chosen $a$. Define the advantage of $\mathcal{A}$ as follows:*

$$\mathsf{Adv}(\mathcal{A}) = \mathsf{Pr}\Big[(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \mathbf{e}, q) \leftarrow \mathcal{G}(1^\tau), (x, y) \xleftarrow{\$} \mathbb{Z}_p^2, \\ X \leftarrow g_2^x, Y \leftarrow g_2^y, (a, b, c, m) \leftarrow \mathcal{A}^{\mathcal{O}_{X,Y}(\cdot)}(X, Y) : \\ m \notin Q \wedge a \in \mathbb{G}_1 \wedge a \neq 1_{\mathbb{G}_1} \wedge b = a^y \wedge c = a^{x+xym}\Big].$$

*No PPT adversary has $\mathsf{Adv}(\mathcal{A})$ non-negligible in $\tau$.*

We introduce a generalized version of the LRSW assumption where we split the oracle $\mathcal{O}_{X,Y}$ into one that first gives the values $a$ and $b$, the two elements that do not depend on the message, and one that later provides $c$ upon input of $m$. That is, after receiving $a, b$, the adversary may specify a message $m$ to receive $c = a^{x+xym}$.

**Assumption 3 (Generalized LRSW)** *Let $X = g_2^x$ and $Y = g_2^y$, and let $\mathcal{O}_X^{\mathsf{a,b}}(\cdot)$ return $(a, b)$ with $a \xleftarrow{\$} \mathbb{G}_1$ and $b \leftarrow a^y$. Let $\mathcal{O}_{X,Y}^{\mathsf{c}}(\cdot)$ on input $(a, b, m)$, with $(a, b)$ generated by $\mathcal{O}_{X,Y}^{\mathsf{a,b}}$, output $c = a^{x+xym}$. It ignores queries with input $(a, b)$ not generated by $\mathcal{O}_{X,Y}^{\mathsf{a,b}}$ or inputs $(a, b)$ that were queried before. Define the advantage of $\mathcal{A}$ as follows.*

$$\begin{aligned}\mathsf{Adv}(\mathcal{A}) = \mathsf{Pr}\Big[&(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, q) \leftarrow \mathcal{G}(1^\tau), (x, y) \xleftarrow{\$} \mathbb{Z}_p^2,\\ &X \leftarrow g_2^x, Y \leftarrow g_2^y, (a, b, c, m) \leftarrow \mathcal{A}^{\mathcal{O}_X^{\mathsf{a,b}}(\cdot), \mathcal{O}_{X,Y}^{\mathsf{c}}(\cdot)}(X, Y) :\\ &m \notin Q \wedge a \in \mathbb{G}_1 \wedge a \neq 1_{\mathbb{G}_1} \wedge b = a^y \wedge c = a^{x+xym}\Big].\end{aligned}$$

*No PPT adversary has $\mathsf{Adv}(\mathcal{A})$ non-negligible in $\tau$.*

Note that our assumption implies the LRSW assumption, but the contrary is not true. In our assumption, the adversary may let $m$ depend on $(a, b)$. Intuitively, it is clear that this does not give any meaningful advantage, as $a$ is random in $\mathbb{G}_1$. We formalize this intuition and prove that Assumption 3 holds in Shoup's generic group model [Sho97] in Appendix A.

### 2.3 Proof Protocols

For zero-knowledge proofs of knowledge of discrete logarithms and statements about them, we will follow the notation introduced by Camenisch and Stadler [CS97] and formally defined by Camenisch, Kiayias, and Yung [CKY09]. For instance, $\mathsf{PK}\{(a) : y = g^a\}$ denotes a "*zero-knowledge Proof of Knowledge of integer $a$ such that $y = g^a$ holds.*" $\mathsf{SPK}\{\ldots\}(m)$ denotes a signature proof of knowledge on $m$, that is a non-interactive transformation of a zero-knowledge proof *PK* with the Fiat-Shamir heuristic [FS87] in the random oracle model [BR93].

(S)PK protocols have three moves: In the first move the prover sends to the verifier what is often referred to as a commitment message or $t$-values. In the second move, the verifier sends a random challenge $c$ to which the prover responds with the so-called $s$-values.

When describing our protocols at a high-level, we use the following, more abstract notation. By $\mathsf{NIZK}\{(w) : \mathit{statement}(w)\}(\mathit{ctxt})$ we denote any non-interactive zero-knowledge proof that is bound to a certain context *ctxt* and proves knowledge of a witness $w$ such that the statement *statement*$(w)$ is true.

## 3 Related Work & Current TPM 2.0 Specification

We now summarize the specification of current TPM 2.0 DAA interfaces and discuss its inherent security and privacy issues and how existing work aims to overcome them.

*TPM 2.0 Interface and SPKs.* For realizing DAA, and signature proofs of knowledge of a TPM secret key in general, the TPM 2.0 specification offers four main commands `TPM.Create`, `TPM.Hash`, `TPM.Commit`, and `TPM.Sign`. Calling `TPM.Create` triggers the creation of a secret key $tsk \in \mathbb{Z}_p$ and a public key $tpk \leftarrow \bar{g}^{tsk}$, where $\bar{g}$ and $\mathbb{Z}_p$ are fixed parameters. Roughly, for signing a message $m$ via a signature proof of knowledge (SPK) of $tsk$ w.r.t. a basename $bsn_L$, the host first invokes `TPM.Commit` on input a group element $g$ and basename $bsn_L$ upon which the TPM outputs $(commitId, E, K, L)$ with $K \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn_L)^{tsk}$, and the $t$-values of the SPK, denoted $E \leftarrow g^r$ and $L \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn_L)^r$. The TPM also internally stores $(commitId, r)$. The host then calls `TPM.Hash` to obtain a hash $c$ on the message $(m, (E, L))$. If the TPM wants to sign this message, it marks $c$ as safe to sign. The proof gets completed by invoking the `TPM.Sign` command on input a safe-to-sign hash $c$ and a reference *commitId* to the randomness $r$ upon which the TPM outputs $s \leftarrow r + c \cdot tsk$.

Due to this generic interface, the TPM 2.0 can be used to construct multiple DAA schemes. Chen and Li [CL13] show that the TPM 2.0 supports both LRSW-based DAA [CPS10] and $q$-SDH-based DAA [BL10], whereas the TPM 1.2 only supported the original RSA-based DAA scheme [BCC04]. Unfortunately, the current TPM 2.0 interfaces have some drawbacks: the signature proofs of knowledge the TPM makes cannot be proven to be unforgeable and there exists a static Diffie-Hellman oracle leaking information about the TPM key.

### 3.1 Unforgeability Flaw for TPM 2.0-based SPKs

The SPKs that are created via the TPM commands should be unforgeable, i.e., a host must not be able to compute an SPK on message $m$ without calling `TPM.Sign` on a hash $c$ that was previously cleared via a `TPM.Hash` call on $m$. Chen and Li [CL13] aim to prove this property, but the proof is incorrect, as pointed out by Xi et al. [XYZF14]. In the proof, the authors simulate the TPM without knowing its secret key $tsk$. To simulate an SPK on message $m$, the authors use the standard approach of randomly choosing the $c$ and $s$ values, and then derive the $t$-values $E$ and $L$ in `TPM.Commit` based on $c$, $s$, and $tpk$. For the reduction to go through, one must ensure that the randomly chosen $c$ becomes the hash value of $(m, t)$ (via `TPM.Hash` and modeling the hash as random oracle), and then let `TPM.Sign` respond with $s$ whenever that $c$ is given as input. However, given that an adversary has arbitrary access to the TPM interfaces, it can query `TPM.Hash` on many different messages $(m_1, t), \ldots, (m_n, t)$ containing the same $t$ value. The reduction does not know which of these queries the adversary will later use to complete the signature, and thus only has a $1/n$ chance to correctly simulate the proof.

*Unforgeability Fix Breaks Privacy.* This problem is inherent in the current TPM interface, but could be solved by a simple modification to the `TPM.Sign` method as proposed by Xi et al. [XYZF14]: when signing, the TPM first chooses a nonce $n_t$ and computes $c' \leftarrow \mathsf{H}(n_t, c)$ and $s \leftarrow r + c' \cdot tsk$. This allows to prove the unforgeability of TPM generated SPKs, as the reduction can now program the random oracle on $c'$ only when the `TPM.Sign` query is made.

However, this would also introduce a subliminal channel for the TPM, as $n_t$ would be part of the final signature and a subverted TPM can embed arbitrary information in that nonce, breaking the anonymity without a host noticing. Recent revelations of subverted cryptographic standards and tampered hardware indicate that such attacks are very realistic. We propose changes to the TPM that provably prevent such subliminal challenges and at the same time allow to prove the unforgeability of the SPKs, as we will show in Section 4.

### 3.2 Static Diffie-Hellman Oracle

Another problem in the TPM 2.0 interface is the static Diffie-Hellman (DH) oracle, as pointed out by Acar et al. [ANZ13]. For any chosen point $g \in \mathbb{G}_1$, the host can learn $g^{tsk}$ by calling $(commitId, E, K, L) \leftarrow$ $\texttt{TPM.Commit}(g, bsn)$, $s \leftarrow \texttt{TPM.Sign}(commitId, c)$ and computing $g^{tsk} \leftarrow (g^s \cdot E^{-1})^{1/c}$. This leaks a lot of information about $tsk$, Brown and Gallant [BG04] and Cheon [Che06] show that the existence of such an oracle makes finding the discrete log much easier. The reason is that the oracle can be used to compute a $q$-SDH sequence $g^{tsk}, g^{tsk^2}, \ldots, g^{tsk^q}$ for very large $q$, which in turn allows to recover $tsk$ faster than had one been given only $\bar{g}^{tsk}$. On Barreto-Naehrig (BN) curves [BN06], one third of the security strength can be lost due to a static DH oracle. For example, a 256 bit BN curve, which should offer 128 bits of security, only offers 85 bits of security with a static DH oracle.

The static DH oracle also prevents *forward anonymity*. Forward anonymity guarantees that signatures made by an honest platform remain anonymous, even when the host later becomes corrupted. In existing schemes, even anonymous signatures contain a pair $(g_i, U_{i,k})$ where $g_i$ is a random generator and $U_{i,k} = g_i^{tsk_k}$. With a static DH oracle, a host upon becoming corrupt can use the TPM to compute $U_i' = g_i^{tsk}$ for all previous signatures, test whether $U_i' = U_{i,k}$, breaking the anonymity of these signatures.

*Cleared Generators for LRSW-based Schemes.* Xi et al. [XYZF14] propose an approach to remove the static DH oracle while preserving the support for the both LRSW- and $q$-SDH-based DAA schemes. They introduce a new `TPM.Bind` command that takes as input two group elements $P$ and $K$ and a proof $\pi_P \leftarrow \mathsf{SPK}\{\alpha : P = \bar{g}^{\alpha} \ \wedge\ K = tpk^{\alpha}\}$. The TPM verifies the proof and, if correct, stores $P$ as cleared generator. The `TPM.Commit` interface will then only accept such cleared generators as input for $g$. This removes the static DH oracle because the proof $\pi_P$ shows that $P^{tsk} = K$ is already known. A similar approach was also used in the recent LRSW-DAA scheme by Camenisch et al. [CDL16c].

However, this approach has two crucial problems. First, it is very hard to implement this functionality on a TPM. The TPM stores only a small number of root keys due to the very limited amount of storage available. For all other keys, the TPM creates a "key blob" that contains the public part of the key in the clear and the private part of the key encrypted with one of the root keys. The TPM would have to similarly store an authenticated list of generators which have been cleared via the `TPM.Bind` interface. However, this would be a new type of key structure, which is a significant change to the current TPM 2.0 specification.

Second, this interface does not support *signature-based revocation*, which is an important extension to anonymous signatures. This type of revocation was introduced in EPID [BL11] and allows one to revoke a platform given a signature from that platform. Roughly, for signature-based revocation, every signature includes a pair $(B, nym)$ where $B \xleftarrow{\$} \mathbb{G}_1$ and $nym \leftarrow B^{tsk}$. The signature revocation list `SRL` contains tuples $\{(B_i, nym_i)\}$ from signatures of the platforms that are revoked. When signing, the TPM must also prove that it is not the producer of any of these revoked signatures. To do so, it proves $\pi_{\mathtt{SRL},i} \leftarrow \mathsf{SPK}^*\{(tsk) : nym = B^{tsk} \ \wedge\ nym_i \neq B_i^{tsk}\}$ for each tuple in `SRL`. Using the changes proposed by Xi et al. [XYZF14], a host cannot input the generators $B_i$ to the TPM anymore as it is not able to produce proofs $\pi_{B_i}$ that are required in the `TPM.Bind` interface.

*Random Generators via Hashing.* Another approach to remove the static DH oracle is to determine the base $g$ by hashing. That is, instead of inputing $g$ in `TPM.Commit`, the host provides a basename $bsn_E$ upon which the TPM derives $g \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn_E)$. By assuming that the hash function is a random oracle, $g$ is now enforced to be a random instead of a chosen generator which avoids the static DH oracle, as the host can no longer create the large $q$-SDH sequences that are the basis of the static DH attacks.

Interestingly, this approach was included in the revision from TPM 1.2 to TPM 2.0 to avoid another static DH oracle that was present in the earlier standard. In TPM 1.2, the `TPM.Commit` interface received a generator $j$ instead of $bsn_L$ and directly computed $K \leftarrow j^{tsk}$ and $L \leftarrow j^r$, whereas TPM 2.0 now receives $bsn_L$ and first derives $j \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn_L)$.

While applying the same idea on $g$ would solve the problem, it would also significantly limit the functionality of the TPM interface. Recall that TPM 2.0 was designed to support both, LRSW- and $q$-SDH-based DAA schemes. While $q$-SDH schemes could be easily ported to these new interfaces, no current LRSW-based scheme would be supported. All existing LRSW-based schemes require the TPM to prove knowledge of $d = b^{tsk}$ for a generator $b \leftarrow a^y$ chosen by the DAA issuer. As the issuer must be privy of the discrete logarithm $y$, it cannot produce a basename $bsn_E$ such that $b = \mathsf{H}_{\mathbb{G}_1}(bsn_E)$ holds at the same time.

Another protocol that would, in its current forms, not be compatible with this change is the aforementioned signature-based revocation [BL11], which needs the TPM to use basenames $B_i$ defined in the revocation list `SRL`. Camenisch et al. [CDL16a] recently proposed to use $B \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn)$ instead of $B \xleftarrow{\$} \mathbb{G}_1$ to avoid the DH oracle, i.e., the TPM gets $bsn$ as input and the `SRL` has the form $\{(bsn_i, nym_i)\}$. However, the authors did not detail how the TPM interfaces have to be changed to support this approach. In fact, their protocol is not easily instantiable, as their proposed computations by the TPM for generating the proofs $\pi_{\mathtt{SRL},i}$ would require the TPM to keep state, which in turn would require new TPM commands.

*Our Approach.* In this work we follow the idea of using hash-based generators but thoroughly describe the necessary changes to the TPM 2.0 specification and, in addition, are very conscious to optimize our solutions. Most importantly, our proposed modifications do not require any new TPM commands, but modify the existing ones only slightly. To demonstrate the flexibility of our TPM interface we present a generic protocol that allows to create a wide class of signature proofs of knowledge using these TPM commands. The existing LRSW-based DAA and signature-based revocation protocols cannot be used with our interface due to the

Session system parameters: $\mathbb{G}_1 = \langle \bar{g} \rangle$ of prime order $q$, nonce bit length $l_n$, random oracles $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_p$ and $\mathsf{H}_{\mathbb{G}_1} : \{0,1\}^* \to \mathbb{G}_1$. Initialize `Committed` $\leftarrow \emptyset$ and $\mathit{commitId} \leftarrow 0$.

---

**Init.** On input `TPM.Create`():

- If this is the first invocation of `TPM.Create`, choose a fresh secret key $\mathit{tsk} \xleftarrow{\$} \mathbb{Z}_p$ and compute public key $\mathit{tpk} \leftarrow \bar{g}^{\mathit{tsk}}$.
- Store $\mathit{tsk}$ and output $\mathit{tpk}$.

**Hash.** On input `TPM.Hash`$(m_t, m_h)$:

- If $m_t \neq \perp$, the TPM checks whether it wants to attest to $m_t$.
- Compute $c \leftarrow \mathsf{H}(\text{“}\mathit{TPM}\text{”}, m_t, m_h)$.
- Mark $c$ as “safe to sign” and output $c$.

**Commit.** On input `TPM.Commit`$(\mathit{bsn}_E, \mathit{bsn}_L)$:

- If $\mathit{bsn}_E \neq \perp$, set $\tilde{g} \leftarrow \mathsf{H}_{\mathbb{G}_1}(\mathit{bsn}_E)$, otherwise set $\tilde{g} \leftarrow \bar{g}$.
- Choose $r \xleftarrow{\$} \mathbb{Z}_p$, $n_t \xleftarrow{\$} \{0,1\}^{l_n}$ and store $(\mathit{commitId}, r, n_t)$ in `Committed`.
- Set $\bar{n}_t \leftarrow \mathsf{H}(\text{“}\mathit{nonce}\text{”}, n_t)$, $E \leftarrow \tilde{g}^r$, and $K, L \leftarrow \perp$.
- If $\mathit{bsn}_L \neq \perp$, set $j \leftarrow \mathsf{H}_{\mathbb{G}_1}(\mathit{bsn}_L)$, $K \leftarrow j^{\mathit{tsk}}$ and $L \leftarrow j^r$.
- Output $(\mathit{commitId}, \bar{n}_t, E, K, L)$ and increment $\mathit{commitId}$.

**Sign.** On input `TPM.Sign`$(\mathit{commitId}, c, n_h)$:

- Retrieve record $(\mathit{commitId}, r, n_t)$ and remove it from `Committed`, output an error if no record was found.
- If $c$ is safe to sign, set $c' \leftarrow \mathsf{H}(\text{“}\mathit{FS}\text{”}, n_t \oplus n_h, c)$ and $s \leftarrow r + c' \cdot \mathit{tsk}$.
- Output $(n_t, s)$.

**Fig. 1.** Our proposed modified TPM 2.0 interface (changes w.r.t. the current specification are highlighted in blue).

aforementioned issues. We therefore also propose new protocols for signature-based revocation and LRSW-based DAA that are compatible with the proposed TPM interfaces and provably secure.

## 4 The Revised TPM 2.0 Interface

This section introduces new TPM 2.0 interfaces for creating signature proofs of knowledge. The TPM creates keys with the `TPM.Create` command. Messages can be signed by first calling `TPM.Commit`, followed by a `TPM.Hash` and a `TPM.Sign` command. We first discuss our proposed modifications to these commands and how they address the problems mentioned in Section 3. Indeed, we are able to do that by making only minor modifications to the commands. The description of our revised TPM interfaces is presented in Figure 1. We again use a simplified notation and refer for the full specification of our TPM 2.0 interfaces to Appendix E for details.

*Avoiding a Subliminal Channel.* To solve the unforgeability problem discussed in Section 3, a nonce to which the TPM contributed needs to be included in the computation of the Fiat-Shamir challenge $c'$. Thereby, a malicious TPM must not be able to alter the distribution of the signature proofs of knowledge, as this would break the privacy, which is the key goal of anonymous attestation. For this reason, the nonce needs to be computed jointly at random by the TPM and the host. In `TPM.Commit`, the TPM chooses a nonce $n_t$ and commits to that nonce by computing $\bar{n}_t \leftarrow \mathsf{H}(\text{“}\mathit{nonce}\text{”}, n_t)$. The host picks another nonce $n_h$, and gives that as input to `TPM.Sign`. The TPM must use $n_t \oplus n_h$ in the Fiat-Shamir hash. An honest host takes $n_h$ uniformly at random, which guarantees that $n_t \oplus n_h$ is uniform random, preventing a malicious TPM from hiding messages in the nonce.

*Avoiding the DH Oracle.* The `TPM.Commit` command is changed to prevent a static Diffie-Hellman oracle. The oracle exists in the current TPM 2.0 interface because therein a host can pass any value $g$ to the TPM and obtain $g^{\mathit{tsk}}$. Our revised TPM prevents this as it will only use a generator $\tilde{g}$ that is either $\tilde{g} \leftarrow \mathsf{H}_{\mathbb{G}_1}(\mathit{bsn}_E)$

for some $bsn_E$ it receives, or set to $\tilde{g} \leftarrow \bar{g}$ if $bsn_E = \perp$ where $\bar{g}$ denotes the fixed generator used within the TPMs.

Clearly, the host can no longer abuse this interface to learn information about the TPM secret key $tsk$. If $\tilde{g} = \bar{g}$, the host receives $tpk$ which it already knows. If $\tilde{g} = \mathsf{H}_{\mathbb{G}_1}(bsn_E)$ and we model the hash function as a random oracle, the host receives a random element raised to power $tsk$, which does not give the host useful information. More precisely, the proof of Lemma 2 shows that we can simulate a TPM without knowing $tsk$, which proves that the TPM does not leak information on $tsk$. Although our changes limit the generators that the host can choose, Section 5.2 shows that we can still build DAA schemes based on $q$-SDH and LRSW on top of this interface, including support for signature-based revocation.

### 4.1 Zero-knowledge Proofs with the TPM

We now describe how our proposed TPM interfaces can be used to create a wide class of signature proofs of knowledge. To demonstrate the flexibility of our interface we propose a generic proof protocol $\mathsf{Prove}$ that orchestrates the underlying TPM commands. We then show that proofs generated by $\mathsf{Prove}$ are unforgeable, device-bound and remain zero-knowledge even if the TPM is subverted. Thus, protocol designers can use our $\mathsf{Prove}$ protocol as generic building block for more complex protocols instead of having to use the TPM command and proving these security properties from scratch. Our DAA protocols presented in Section 5 use exactly that approach.

*A Generic* $\mathsf{Prove}$ *Protocol.* Using the proposed TPM interfaces, a host can create signature proofs of knowledge of the following structure:

$$\mathsf{SPK}^*\{(\gamma \cdot (tsk + hsk), \alpha_1, \ldots, \alpha_l) : y_1 = (\hat{g}^{\delta})^{\gamma \cdot (tsk+hsk)} \cdot \prod_i {b_i}^{\alpha_i} \quad \wedge$$
$$y_2 = \mathsf{H}_{\mathbb{G}_1}(bsn_L)^{\gamma \cdot (tsk+hsk)} \cdot \prod_i b_i'^{\alpha_i} \quad \wedge \quad y_3 = \prod_i b_i''^{\alpha_i}\}(m_h, m_t) \ , \quad (1)$$

for values $\delta$, $hsk, tsk$, and $\gamma$ in $\mathbb{Z}_p$, strings $bsn_L, m_h, m_t \in \{0,1\}^*$, group elements $y_1, y_2, y_3, \hat{g}$, and set $\{(\alpha_i, b_i, b_i', b_i'')\}_i$, with $\alpha_i \in \mathbb{Z}_p$. Either $y_1$, $\hat{g}$, and all $b_i$'s are in $\mathbb{G}_1$ or they are all in $\mathbb{G}_T$. All $b_i'$ values and $y_2$ must be in $\mathbb{G}_1$. If $bsn_L = \perp$, the second equation proving a representation of $y_2$ is omitted from the proof. We could also lift this part of the proof to $\mathbb{G}_T$ but as we do not require such proofs, we omit this to simplify the presentation. The values $y_3$ and $b_i''$ must either all be in $\mathbb{G}_1$, in $\mathbb{G}_2$, or in $\mathbb{G}_T$.

In addition we require that the TPM works with a cleared generator, meaning either $\hat{g} = \tilde{g}$ or $\hat{g} = \mathbf{e}(\tilde{g}, \hat{g}_2)$ with $\tilde{g}$ denoting the cleared generator being either $\bar{g}$, i.e., the fixed generator or it is $\mathsf{H}_{\mathbb{G}_1}(bsn_E)$ for some $bsn_E$.

The protocol allows the host to add a key $hsk$ to the witness for $tsk$ because, as we will see in the later sections, this can improve the privacy of DAA schemes. Note that we could trivially generalize the proof statement (4.1) to include additional terms that do not contain $\gamma \cdot (tsk + hsk)$ as witness, but for ease of presentation we omit these additional terms.

The host can add any message $m_h$ to the proof. It also chooses $m_t$, but this is a value the TPM attests to and will be checked by the TPM.

The host can create such a proof using the $\mathsf{Prove}$ protocol described in Figure 3. We assume a perfectly secure channel between the host and TPM, i.e., the adversary does not notice the host calling TPM commands. Note that before starting the proof, the host may not know $y_2$, as it does not know $tsk$, but learns this value during the proof because it is given as output of the $\mathsf{Prove}$ protocol. How to verify such proofs using the $\mathsf{VerSPK}$ algorithm is shown in Figure 3 as well. Note that verification does not require any participation of the TPM. Figure 2 gives a brief overview of the required parameters and their respective types and conditions.

The completeness of these proofs can easily be verified. The proof is sound as we can extract a valid witness using the standard rewinding technique.

| Variable | Type | Explanation |
|---|---|---|
| TPM Variables | | |
| $tsk$ | $\mathbb{Z}_p$ | secret key held inside the TPM (in DAA part of the platform secret key) |
| $tpk$ | $\mathbb{G}_1$ | public key corresponding to $tsk$, i.e., $tpk = \bar{g}^{tsk}$ |
| $\bar{g}$ | $\mathbb{G}_1$ | fixed generator in all TPMs |
| $\tilde{g}$ | $\mathbb{G}_1$ | cleared generator created in `TPM.Commit`, with $\tilde{g} \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn_E)$ if $bsn_E \neq \perp$ and $\tilde{g} \leftarrow \bar{g}$ else |
| Prove Variables | | |
| $hsk$ | $\mathbb{Z}_p$ | secret key held by the host (in DAA part of the platform secret key), set $hsk = 0$ if not needed |
| $y_1$ | $\mathbb{G}_1$ or $\mathbb{G}_T$ | see SPK (4.1), if $y_1 \in \mathbb{G}_T$ then $\hat{g_2}$ is a mandatory input |
| $bsn_E$ | $\{0,1\}^*$ or $\perp$ | basename for generator $\tilde{g} \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn_E)$, if $bsn_E = \perp$ then $\tilde{g} \leftarrow \bar{g}$ |
| $\delta$ | $\mathbb{Z}_p$ | see SPK (4.1), set $\delta = 1$ if not needed |
| $\hat{g_2}$ | $\mathbb{G}_2$ if $y_1 \in \mathbb{G}_T$, or $\perp$ | if $\hat{g_2} \neq \perp$, it moves proof to $\mathbb{G}_T$ by setting $\hat{g} \leftarrow \mathbf{e}(\tilde{g}, \hat{g_2})$; if $\hat{g_2} = \perp$ then $\hat{g} \leftarrow \tilde{g}$ |
| $\gamma$ | $\mathbb{Z}_p$ | see SPK (4.1), set $\gamma = 1$ if not needed |
| $bsn_L$ | $\{0,1\}^*$ or $\perp$ | basename for generator $j \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn_L)$ if $bsn_L \neq \perp$ |
| $y_2$ | $\mathbb{G}_1$ or $\perp$ | see SPK (4.1), if $bsn_L \neq \perp$, then $y_2 \neq \perp$ is mandatory input, else $y_2 = \perp$ |
| $y_3$ | $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, or $\perp$ | see SPK (4.1), set $y_3 = \perp$ if not needed |
| $\alpha_i$ | $\mathbb{Z}_p$ | see SPK (4.1), input given as part of $\{(\alpha_i, b_i, b'_i, b''_i)\}_i$ |
| $b_i$ | same group as $y_1$ | see SPK (4.1), set $b_i = 1_{\mathbb{G}}$ if $\alpha_i$ is not needed in the first equation of (4.1) |
| $b'_i$ | $\mathbb{G}_1$ | see SPK (4.1), set $b'_i = 1_{\mathbb{G}_1}$ if $\alpha_i$ is not needed in the second equation if (4.1) |
| $b''_i$ | same group as $y_3$ | see SPK (4.1), set $b''_i = 1_{\mathbb{G}}$ if $\alpha_i$ is not needed in the third equation of (4.1) |
| $m_h$ | $\{0,1\}^*$ or $\perp$ | message that the host adds to an attestation |
| $m_t$ | $\{0,1\}^*$ or $\perp$ | message the TPM attests to |

**Fig. 2.** Overview of variables used within the TPM and in our Prove protocol.

*Example for Using* Prove. We now give a simple example to show how the Prove protocol must be invoked and give some intuition on how the final proof is assembled by our protocol. Suppose we want to prove:

$$\mathsf{SPK}^*\{(tsk + hsk) : d' = (\mathsf{H}_{\mathbb{G}_1}(bsn_E)^{\delta})^{(tsk+hsk)} \quad \wedge \quad nym = \mathsf{H}_{\mathbb{G}_1}(bsn_L)^{(tsk+hsk)}\}(m_h, m_t),$$

where the TPM holds $tsk$ and the host knows $hsk$. The host will add $hsk$ to the witness for $tsk$, which is the first input to Prove. The second argument is the left hand side of the first equation, which is $d'$. The generator for the witness $tsk + hsk$ is $(\mathsf{H}_{\mathbb{G}_1}(bsn_E)^{\delta})$, which is passed on to the Prove protocol by giving $bsn_E$ and $\delta$ as the next arguments. The protocol has the option to move the proof to $\mathbb{G}_T$ by passing a value $\hat{g}_2$, but as this proof takes place in $\mathbb{G}_1$, we enter $\hat{g}_2 = \perp$. We can prove knowledge of $\gamma \cdot (tsk + hsk)$, but as we want to use witness $tsk + hsk$, we pass $\gamma = 1$. In the second equation, we use $\mathsf{H}_{\mathbb{G}_1}(bsn_L)$ as generator, so we give argument $bsn_L$. Since our proof omits the third equation, we set $y_3 \leftarrow \perp$. The protocol supports an additional list of witnesses with generators in the three equations, but since this equation only uses witness $tsk + hsk$, we pass an empty list as next argument. Finally, we specify $m_t$, the message the TPM attests to, and $m_h$, the additional data added by the host. Therefore, we call

$$\mathsf{Prove}(hsk, d', bsn_E, \delta, \perp, 1, bsn_L, \perp, \emptyset, m_h, m_t).$$

The protocol calls `TPM.Commit` with basenames $bsn_E$ and $bsn_L$ to receive $E = \mathsf{H}_{\mathbb{G}_1}(bsn_E)^{r_{tsk}}$ and $L = \mathsf{H}_{\mathbb{G}_1}(bsn_L)^{r_{tsk}}$ for some $r_{tsk}$, and $K = \mathsf{H}_{\mathbb{G}_1}(bsn_L)^{tsk}$, along with $\bar{n}_t = \mathsf{H}(\text{“}nonce\text{”}, n_t)$, that commits the TPM to TPM nonce $n_t$. The host must change the generator for the first proof equation to $\mathsf{H}_{\mathbb{G}_1}(bsn_E)^{\delta}$ instead of $\mathsf{H}_{\mathbb{G}_1}(bsn_E)$, and add randomness to both values to prevent a malicious TPM from altering the distribution of the resulting proof. It sets $t_1 \leftarrow E^{\delta} \cdot (\mathsf{H}_{\mathbb{G}_1}(bsn_E)^{\delta})^{r_{hsk}} = (\mathsf{H}_{\mathbb{G}_1}(bsn_E)^{\delta})^{r_{tsk}+r_{hsk}}$, and $t_2 \leftarrow L \cdot \mathsf{H}_{\mathbb{G}_1}(bsn_L)^{r_{hsk}} = \mathsf{H}_{\mathbb{G}_1}(bsn_L)^{r_{tsk}+r_{hsk}}$. Next, it hashes the $t$-values along with the proof parameters and messages $m_t$ and $m_h$ using `TPM.Hash`. The TPM inspects $m_t$ and returns $c$, which can only be passed to `TPM.Sign` if the TPM agrees to signing $m_t$. The host now calls `TPM.Sign` with $c$ and a fresh host nonce $n_h$, upon which it receives $n_t$ and $s = r_{tsk} + c' \cdot tsk$. The host checks whether $n_t$ matches the committed TPM nonce, and computes

$\underline{\mathsf{Prove}(hsk, y_1, bsn_E, \delta, \hat{g}_2, \gamma, bsn_L, y_3, \{(\alpha_i, b_i, b'_i, b''_i)\}_i, m_h, m_t):}$

- If $bsn_E \neq \perp$, set $\tilde{g} \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn_E)$, otherwise set $\tilde{g} \leftarrow \bar{g}$.
- If $\hat{g}_2 \neq \perp$, set $\hat{g} \leftarrow \mathbf{e}(\tilde{g}, \hat{g}_2)$, otherwise set $\hat{g} \leftarrow \tilde{g}$.
- If $bsn_L \neq \perp$, set $j \leftarrow \mathsf{H}_{\mathbb{G}_1}(bsn_L)$.
- Call $\underline{\texttt{TPM.Commit}(bsn_E, bsn_L) \to (commitId, \bar{n}_t, E, K, L)}$.
- Take $r_{hsk} \xleftarrow{\$} \mathbb{Z}_p$, set $E' \leftarrow (E \cdot \tilde{g}^{r_{hsk}})^{\gamma \cdot \delta}$. If $bsn_L \neq \perp$, set $K' \leftarrow (K \cdot j^{hsk})^{\gamma}$ and $L' \leftarrow (L \cdot j^{r_{hsk}})^{\gamma}$.
- If $bsn_L \neq \perp$, set $y_2 \leftarrow K' \cdot \prod_i {b'_i}^{\alpha_i}$.
- Take $\{r_{\alpha_i}\}_{i=1}^{l} \xleftarrow{\$} \mathbb{Z}_p^l$. Set $t_1 \leftarrow E' \cdot \prod_i b_i^{r_{\alpha_i}}$ if $b_i \in \mathbb{G}_1$, or $t_1 \leftarrow \mathbf{e}(E', \hat{g}_2) \cdot \prod_i b_i^{r_{\alpha_i}}$ if $b_i \in \mathbb{G}_T$.
- If $bsn_L \neq \perp$, set $t_2 \leftarrow L' \prod_i {b'_i}^{r_{\alpha_i}}$ and $t_2 \leftarrow \perp$ else.
- If $y_3 \neq \perp$, set $t_3 \leftarrow \prod_i {b''_i}^{r_{\alpha_i}}$ and $t_3 \leftarrow \perp$ else.
- Set $m'_h \leftarrow (m_h, y_1, \hat{g}^{\delta}, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3)$.
- Call $\underline{\texttt{TPM.Hash}(m_t, m'_h) \to c}$.[1]
- Take $n_h \xleftarrow{\$} \{0,1\}^{l_n}$.
- Call $\underline{\texttt{TPM.Sign}(commitId, c, n_h) \to (n_t, s)}$.
- Check that $\bar{n}_t = \mathsf{H}(\text{``}nonce\text{''}, n_t)$ and set $n \leftarrow n_h \oplus n_t$, $c' \leftarrow \mathsf{H}(\text{``}FS\text{''}, n, c)$.
- Set $s' \leftarrow \gamma \cdot (s + r_{hsk} + c' \cdot hsk)$ and $s_{\alpha_i} \leftarrow r_{\alpha_i} + c' \cdot \alpha_i$ for $i = 1, \ldots, l$.
- Check $(\hat{g}^{\delta})^{s'} = E' \cdot (y_1 / (\prod_i b_i^{\alpha_i})^{c'}$ and if $bsn_L \neq \perp$, check $j^{s'} = L' \cdot {K'}^{c'}$.
- Set proof $\pi \leftarrow (c', n, s', \{s_{\alpha_i}\})$ and output $(y_2, \pi)$.

$\underline{\mathsf{VerSPK}(\pi, y_1, \hat{g}^{\delta}, y_2, bsn_L, y_3, \{(\alpha_i, b_i, b'_i, b''_i)\}_i, m_h, m_t):}$

- Parse $\pi$ as $(c', n, s', \{s_{\alpha_i}\})$.
- Set $t_1 \leftarrow y_1^{-c'} \cdot (\hat{g}^{\delta})^{s'} \cdot \prod_i {b_i}^{s_{\alpha_i}}$.
- If $bsn_L \neq \perp$, set $t_2 \leftarrow y_2^{-c'} \cdot \mathsf{H}_{\mathbb{G}_1}(bsn_L)^{s'} \cdot \prod_i {b'_i}^{s_{\alpha_i}}$, and $t_2 \leftarrow \perp$ else.
- If $y_3 \neq \perp$, set $t_3 \leftarrow y_3^{-c'} \cdot \prod_i {b''_i}^{s_{\alpha_i}}$ and $t_3 \leftarrow \perp$ else.
- Output 1 if $c' = \mathsf{H}(\text{``}FS\text{''}, n, \mathsf{H}(\text{``}TPM\text{''}, m_t, (m_h, y_1, \hat{g}^{\delta}, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3)))$, and 0 otherwise.

**Fig. 3.** Prove protocol and VerSPK algorithm to create and verify zero-knowledge proofs via the TPM interfaces from Figure 1.

the joint nonce $n \leftarrow n_h \oplus n_t$ and Fiat-Shamir challenge $c' \leftarrow \mathsf{H}(\text{``}FS\text{''}, n, c)$. The host must now add its randomness and $hsk$ to the $s$-value, which it does by setting $s' \leftarrow s + r_{hsk} + c' \cdot hsk$. Finally, it checks whether the resulting proof is valid, to make sure that the TPM contributions did not invalidate the proof. The resulting proof consists of nonce $n$, Fiat-Shamir challenge $c'$, and $s$-value $s'$.

**Security of Prove** We now show that proofs generated by our generic Prove protocol specified in Figure 3 and using the TPM interfaces as described in Figure 1 are unforgeable, device-bound and remain zero-knowledge even if the TPM is subverted.

*Zero-knowledge of SPKs with a Corrupt TPM.* An SPK created with the Prove protocol is zero knowledge in the random oracle model, even when the TPM is corrupt. That is, we prove the absence of any subliminal channel that a malicious TPM could use to break the privacy of the platform. In Section 5 we show that this allows one to devise DAA schemes that guarantee privacy even when the TPM is malicious.

**Lemma 1 (Privacy of SPKs with a TPM).** *The signature proofs of knowledge generated by* Prove *as defined in Figure 4.1, are zero-knowledge, even when the TPM is corrupt.*

[1] Note that sending all these values to the TPM might be slow due to the low bandwidth. Instead, the host could send a hash of $m'_h$ to improve performance without affecting the security. For ease of presentation, we omit this optimization.

*Proof (Sketch).* A corrupt TPM may block the creation of the proof, but if it succeeds, it is zero knowledge. The TPM is involved in proving knowledge of $\gamma \cdot (tsk + hsk)$. The host changes the $r$-value to $\gamma \cdot (r_{tsk} + r_{hsk})$, with $r_{hsk}$ chosen by the host. It takes $r_{hsk} \xleftarrow{\$} \mathbb{Z}_p$, so $r_{tsk} + r_{hsk}$ is uniform in $\mathbb{Z}_p$ regardless of how the TPM chooses $r_{tsk}$. Since $\gamma \neq 0$, $\gamma \cdot (r_{tsk} + r_{hsk})$ is still uniform in $\mathbb{Z}_p$.

The TPM also chooses a nonce $n_t$. It must first commit to this nonce with $\bar{n}_t = \mathsf{H}(\text{"}nonce\text{"}, n_t)$. The host then chooses a nonce $n_h$ uniformly at random in $\{0,1\}^{l_n}$, and the TPM must work with $n = n_h \oplus n_t$, and show that it computed this correctly. Clearly, $n$ is uniform if $n_h$ is uniform.

Since we know the distribution of every part of the zero-knowledge proof, even when the TPM is corrupt, we can simulate proofs of an honest host with a corrupt TPM.

*Unforgeability of SPKs with an Honest TPM.* We now show that proofs generated by Prove are unforgeable with respect to $m_t$, i.e., if the TPM is honest, a corrupt host cannot create a SPK for message $m_t$ that the TPM did not approve to sign.

We consider a corrupt host with oracle access to an honest TPM. The TPM executes `TPM.Create`, outputting $tpk \leftarrow \bar{g}^{tsk}$. The corrupt host cannot create SPKs of structure (4.1) where $tsk$ is protected by the TPM and $\gamma$ and $hsk$ are known and the TPM never signed $m_t$. We require the host to output $\gamma$ and $hsk$ along with his forgery. In a protocol, this means that these values must be fixed (e.g., $\gamma$ always equals 1) or extractable from some proof of knowledge for this lemma to be applicable.

**Lemma 2 (Unforgeability of SPKs with a TPM).** *The signature proofs of knowledge generated by* Prove *as defined in Figure 4.1, are unforgeable w.r.t. $m_t$. More precisely, the host cannot forge a signature proof of knowledge with the structure of (4.1) with a witness $\gamma \cdot (tsk + hsk)$ for known $\gamma, hsk$ if the TPM never signed $m_t$, under the DL assumption in the random oracle model.*

*Proof (Sketch).* We show that if an adversary $\mathcal{A}$ that has access to the TPM interfaces can forge SPK's, we can derive an adversary $\mathcal{B}$ that can solve the discrete logarithm problem. Note that it is crucial that we allow the adversary $\mathcal{A}$ to get full, unconstrained access to the TPM interfaces instead of giving him only indirect access via the Prove protocol, as this correctly models the power a corrupt host will have.

Our reduction $\mathcal{B}$ receives a DL instance $tpk = \bar{g}^{tsk}$ and is challenged to find $tsk$. To do so, we simulate the TPM and the hash function towards $\mathcal{A}$ based on $tpk, \bar{g}$ as follows:

**Hash queries:** For queries $bsn_i$ to $\mathsf{H}_{\mathbb{G}_1}$, take $r_i \xleftarrow{\$} \mathbb{Z}_p$ and return $\mathsf{H}_{\mathbb{G}_1}(bsn_i) = \bar{g}^{r_i}$ and store $(\mathsf{hash}, \mathsf{H}_{\mathbb{G}_1}(bsn_i), r_i)$. Queries to $\mathsf{H}$ and `TPM.Hash` are handled normally.

**Commit query** `TPM.Commit`$(bsn_E, bsn_L)$**:** Take $(s_i, c'_i) \xleftarrow{\$} \mathbb{Z}_p^2$. If $bsn_E \neq \perp$, compute $\mathsf{H}_{\mathbb{G}_1}(bsn_E)$, look up the record $(\mathsf{hash}, \mathsf{H}_{\mathbb{G}_1}(bsn_E), r_E)$, and set $E \leftarrow \bar{g}^s \cdot tpk^{-c'_i \cdot r_E}$. If $bsn_E = \perp$, set $E \leftarrow \bar{g}^s \cdot tpk^{-c'_i}$.
If $bsn_L \neq \perp$, compute $\mathsf{H}_{\mathbb{G}_1}(bsn_L)$, look up the record $(\mathsf{hash}, \mathsf{H}_{\mathbb{G}_1}(bsn_L), r_L)$, and set $K \leftarrow tpk^{r_L} = \mathsf{H}_{\mathbb{G}_1}(bsn_L)^{tsk}$, and $L \leftarrow \bar{g}^s \cdot tpk^{-c'_i \cdot r_L}$. If $bsn_L = \perp$, set $K \leftarrow \perp$ and $L \leftarrow \perp$.
Pick $\bar{n}_t$ uniform in the range of $\mathsf{H}$, store $(commitId, \bar{n}_t, s_i, c'_i)$, increment $commitId$, and output $(commitId, \bar{n}_t, E, K, L)$.

**Sign query** `TPM.Sign`$(commitId, c, n_h)$**:** Look up and remove record $(commitId, \bar{n}_t, s_i, c'_i)$, and output an error if no such record was found. Check that $c$ was marked safe-to-sign in a `TPM.Hash` query. Pick $n_t \xleftarrow{\$} \{0,1\}^{l_n}$ and program the random oracle such that $\mathsf{H}(\text{"}nonce\text{"}, n_t) = \bar{n}_t$. Program the random oracle such that $\mathsf{H}(\text{"}FS\text{"}, n_t \oplus n_h, c) = c'_i$. Since the nonce $n_t$ is fresh and gets only used once, the probability that the random oracle is already defined on that input is negligible. Finally, we output $(n_t, s_i)$.

When $\mathcal{A}$, after having interacted with these oracles, outputs a SPK forgery, i.e., a valid proof with TPM message $m_t$ that the TPM never agreed to sign in `TPM.Hash`, along with values $\gamma, hsk$ such that the proof uses $\gamma \cdot (tsk + hsk)$ as witness, we either have a collision in $\mathsf{H}$ which occurs with negligible probability, or we can rewind to extract $\gamma \cdot (tsk + hsk)$, allowing us to compute $tsk$. $\mathcal{B}$ then outputs $tsk$, solving the DL problem.

*Device Boundedness of SPKs with an Honest TPM.* Finally, we show that proofs generated via $\mathsf{Prove}$ are device bound, i.e., the host cannot create more SPKs than the amount of sign queries the TPM answered. Again, the TPM holds $tsk$ with $tpk = \bar{g}^{tsk}$ created by $\mathtt{TPM.Create}$.

**Lemma 3 (Device Boundedness of SPKs with a TPM).** *The signature proofs of knowledge generated by* $\mathsf{Prove}$ *as defined in Figure 4.1, are device bound. That is, the host cannot create more signature proofs of knowledge with the structure of (4.1) with a witnesses $\gamma \cdot (tsk + hsk)$, where tsk is protected by the TPM and the host knows $\gamma$ and hsk, than the amount of sign queries the TPM answered, under the DL assumption in the random oracle model.*

*Proof (Sketch).* Our reduction receives a DL instance $tpk = \bar{g}^{tsk}$ and must compute $tsk$. The simulation works exactly as in the proof of Lemma 2. If the host made $n$ sign queries and outputs $n + 1$ SPKs and corresponding values $\gamma$ and $hsk$, we look at every $c'$ value of the proofs. If there are two distinct SPKs with the same $c'$ value, there must be a collision in $\mathsf{H}$, which occurs with negligible probability. If all $c'$ values are distinct, one of them must be different from the $c'$ values as created by the TPM. That means the random oracle is not programmed here and we can rewind that proof to extract $\gamma \cdot (tsk + hsk)$. Since we also have $hsk$ and $\gamma$ we can compute $tsk$, which solves the DL problem.

**Proofs Without TPM Contribution** To be able to prove security of our DAA schemes, we must distinguish proofs to which the TPM contributed and proofs that the host created by itself. One way to achieve this is by using different prefixes in the Fiat-Shamir hash computation. Proofs with TPM contribution have a Fiat-Shamir hash $c' \leftarrow \mathsf{H}(\text{“}FS\text{”}, n, \mathsf{H}(\text{“}TPM\text{”}, m_t, m_h))$. Proofs without TPM contribution will use $c' \leftarrow \mathsf{H}(\text{“}FS\text{”}, n, \mathsf{H}(\text{“}NoTPM\text{”}, m_t, m_h))$. We denote TPM contributed proofs by $\mathsf{SPK}^*$, and proofs without TPM contribution $\mathsf{SPK}$.

## 5 Provably Secure DAA Schemes

We now show how to use the proposed TPM interfaces to build provably secure direct anonymous attestation protocols. We start by describing the desired functional and security properties (Section 5.1) and then present two DAA protocols, based on the $q$-SDH assumption and the LRSW assumption (Section 5.2), and argue their security (Section 5.3). We refer to Appendix B for the formal definition of DAA in the form of an ideal functionality and to Appendix D for the detailed security proof.

### 5.1 Definition & Security Model

In a DAA scheme, we have four main entities: a number of TPMs, a number of hosts, an issuer, and a number of verifiers. The scheme comprises a $\mathsf{JOIN}$ and $\mathsf{SIGN}$ protocol, and $\mathsf{VERIFY}$ and $\mathsf{LINK}$ algorithms.

$\underline{\mathsf{JOIN}}$: A TPM and a host together form a platform which performs the join protocol with the issuer who decides if the platform is allowed to become a member. The membership credential of the platform then also certifies a number of attributes $attrs = (a_1, \ldots, a_L)$ given by the issuer. These attributes might include more information about the platform, such as the vendor or model, or other information, such as an expiration date of the credential.

$\underline{\mathsf{SIGN}}$: Once being a member, the TPM and host together can sign messages $m$ with respect to basename $bsn$ resulting in a signature $\sigma$. If a platform signs with a fresh basename, the signature must be anonymous and unlinkable to previous signatures. When signing, the platform can also selectively disclose attributes from its membership credential. For instance, reveal that the signature was created by a TPM of a certain manufacturer, or the expiration date of the credential. We describe the disclosure using a tuple $(D, I)$ where $D \subseteq \{1, \ldots, L\}$ indicates which attributes are disclosed, and $I = (a_1, \ldots, a_L)$ specifies the desired attribute values.

$\underline{\mathsf{VERIFY}}$: Any verifier can check that a signature $\sigma$ on message $m$ stems from a legitimate platform via a deterministic verify algorithm. More precisely, verification gets as input a tuple $(m, bsn, \sigma, (D, I), \mathtt{RL}, \mathtt{SRL})$

and outputs 1 if $\sigma$ is a valid signatures on message $m$ w.r.t. basename $bsn$ and stems from a platform that has a membership credential satisfying the predicate defined via $(D, I)$, and 0 otherwise.

The inputs $\mathtt{RL}$ and $\mathtt{SRL}$ are revocation lists and we support two types of revocation, *private-key-based* revocation and *signature-based* revocation. The first is based on the exposure of a corrupt platform's secret key (or private key) and allows one to recognize and thus reject all signatures generated with this key. That is, the revocation list $\mathtt{RL}$ contains the secret keys of the revoked TPMs. The second type, signature-based revocation, has been proposed by Brickell and Li [BL07,BL11] in their Enhanced Privacy ID (EPID) protocol. It allows one to revoke a platform based on a previous signature from that platform, i.e., here the revocation list $\mathtt{SRL}$ contains the signatures of the revoked TPMs.

LINK: By default, signatures created by an DAA scheme do not leak any information about the identity of the signer. Only when the platform signs repeatedly with the same basename $bsn$, it will be clear that the resulting signatures were created by the same platform, which can be publicly tested via the deterministic LINK algorithm. More precisely, on input two signatures $(\sigma, m, (D, I), \mathtt{SRL})$, $(\sigma', m', (D', I'), \mathtt{SRL}')$, and a basename $bsn$, the algorithm outputs 1 if both signatures are valid and were created by the same platform, and 0 otherwise.

We now describe the desired security properties of DAA schemes in an informal manner. The detailed definition in form of an ideal functionality in the Universal Composability framework [Can00] is given in Appendix B, and closely follows the recent formal models of Camenisch et al. [CDL16c, CDL17].

*Unforgeability.* The adversary can only sign in the name of corrupt TPMs. More precisely, if $n$ corrupt and unrevoked TPMs joined with attributes fulfilling attribute disclosure $(D, I)$, the adversary can create at most $n$ unlinkable signatures for the same basename $bsn$ and attribute disclosure $(D, I)$. In particular, this means that when the issuer and all unrevoked TPMs are honest, no adversary can create a valid signature on a message $m$ w.r.t. basename $bsn$ and attribute disclosure $(D, I)$ when no platform that joined with those attributes signed $m$ w.r.t. $bsn$ and $(D, I)$.

*Non-Frameability.* No adversary can create a signature on a message $m$ w.r.t. basename $bsn$ that links to a signature created by an honest platform, when this honest platform never signed $m$ w.r.t. $bsn$. We require this property to hold even when the issuer is corrupt.

*(Strong) Privacy.* An adversary that is given two signatures $\sigma_1$ and $\sigma_2$ w.r.t. two different basenames $bsn_1 \neq bsn_2$, respectively, cannot distinguish whether both signatures were created by one honest platform, or whether two different honest platforms created the signatures. This property must also hold when the issuer is corrupt.

So far, privacy was conditioned on the honesty of the entire platform, i.e., both the TPM and the host have to be honest. In fact, the previous DAA schemes crucially rely on the honesty of the TPM, and the newly revised TPM interfaces even introduced a subliminal channel that allows a malicious TPM to always encode some identifying information into his signature contribution (see Section 3.1). The latter forestalls any privacy in the presence of a corrupt TPM, even if the DAA protocol built on top of the TPM interfaces would allow for better privacy.

In this work we have proposed TPM interfaces that avoid such subliminal channels and we consequently aim for stronger privacy guarantees for DAA as well. That is, the aforementioned indistinguishability of two signatures $\sigma_1$ and $\sigma_2$ must hold whenever the host is honest, regardless of the corruption state of the TPM. Our notion of *strong privacy* lies between the classical privacy notion (relying also on the honesty of the TPM) and *optimal* privacy that was recently introduced by Camenisch et al. [CDL17]. We discuss the differences between these notions, and to [CDL17] in particular, in Appendix B.

### 5.2 DAA Protocols

We start by presenting the high-level idea of both DAA protocols using our revised TPM 2.0 interfaces, and then describe the concrete instantiations based on the $q$-SDH and the LRSW assumption.

| JOIN : | TPM | $\rightleftharpoons$ | HOST($ipk$) | $\rightleftharpoons$ | ISSUER($isk, attrs = (a_1, \dots, a_L)$) |
|---|---|---|---|---|---|

Host $\xrightarrow{\text{JOIN}}$ Issuer

Issuer: $n \xleftarrow{\$} \{0,1\}^\tau$

Issuer $\xrightarrow{n}$ Host

Host: Request TPM key

Host $\xrightarrow{\texttt{TPM.Create}}$ TPM

TPM: $tsk \xleftarrow{\$} \mathbb{Z}_p$, $tpk \leftarrow \bar{g}^{tsk}$
Store $tsk$

TPM $\xrightarrow{tpk}$ Host

Host: Orchestrate generation of proof $\pi_{tpk}$ by the TPM using the Prove protocol

Host $\xrightarrow{\texttt{TPM.Commit/TPM.Sign}, n}$ TPM

TPM: $tpk' \leftarrow \tilde{g}^{tsk}$ *(optional bridging to a different generator $\tilde{g}$)*
$\pi_{tpk} \leftarrow \mathsf{SPK}^*\{tsk : tpk = \bar{g}^{tsk} \;\wedge\; tpk' = \tilde{g}^{tsk}\}(\text{“}join\text{”}, n)$

TPM $\xrightarrow{tpk',\, \pi_{tpk}}$ Host

Host: Choose host key and generate $gpk$:
$hsk \xleftarrow{\$} \mathbb{Z}_p$, $gpk \leftarrow tpk' \cdot \tilde{g}^{hsk}$
$\pi_{gpk} \leftarrow \mathsf{SPK}\{hsk : gpk/tpk' = \tilde{g}^{hsk}\}(\text{“}join\text{”}, n)$

Host $\xrightarrow{tpk,\, tpk',\, gpk,\, \pi_{tpk},\, \pi_{gpk}}$ Issuer

Issuer: Verify $\pi_{tpk}, \pi_{gpk}$, and sign $gpk$ and attributes $attrs = (a_1, \dots, a_L)$ as $cred \leftarrow \mathsf{PBSign}(isk, (gpk, attrs))$

Issuer $\xrightarrow{cred,\, attrs}$ Host

Host: Verify $cred$ w.r.t. $gpk, attrs, ipk$
Store $(hsk, cred, attrs)$

SIGN : TPM($tsk$) $\rightleftharpoons$ HOST($(hsk, cred, attrs), (ipk, m, bsn, (D, I), \mathtt{RL}, \mathtt{SRL})$)

- The host verifies that its attributes $attrs$ fulfill the predicate $(D, I)$, i.e., it parses $I$ as $(a'_1, \dots, a'_L)$ and $attrs$ as $(a_1, \dots, a_L)$ and checks that $a_i = a'_i$ for every $i \in D$.
- The host and TPM jointly generate the pseudonym $nym \leftarrow \mathsf{H}_{\mathbb{G}}(1||bsn)^{gsk}$ and proof $\pi_{cred}$ of a membership credential on $gsk = tsk + hsk$ and $attrs$:

$$\pi_{cred} \leftarrow \mathsf{NIZK}^*\{(gsk, cred) : nym = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk} \;\wedge\; 1 = \mathsf{PBVf}(ipk, cred, gsk, attrs)\}(\text{“}sign\text{”}, (D, I), m, \mathtt{SRL})$$

- For each tuple $(bsn_i, nym_i) \in \mathtt{SRL}$, the host and TPM jointly create non-revocation proofs $\pi_{\mathtt{SRL},i}$:

$$\pi_{\mathtt{SRL},i} \leftarrow \mathsf{SPK}^*\{gsk : \mathsf{H}_{\mathbb{G}_1}(1||bsn_i)^{gsk} \neq nym_i \;\wedge\; nym = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}\}(\text{“}sign\text{”}).$$

- The host outputs $\sigma \leftarrow (nym, \pi_{cred}, \{\pi_{\mathtt{SRL},i}\})$.

VERIFY($ipk, \sigma, m, bsn, (D, I), \mathtt{RL}, \mathtt{SRL}$) :

- Parse $\sigma = (nym, \pi_{cred}, \{\pi_{\mathtt{SRL},i}\})$.
- Verify $\pi_{cred}, \{\pi_{\mathtt{SRL},i}\}$ w.r.t. $ipk, m, bsn, (D, I), \mathtt{SRL}$,
- For every $gsk_i \in \mathtt{RL}$, check that $\mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk_i} \neq nym$.
- Output 1 if all proofs are correct, and 0 otherwise.

LINK($ipk, bsn, (\sigma, m, (D, I), \mathtt{SRL}), (\sigma', m', (D', I'), \mathtt{SRL}')$) :

- Get $f \leftarrow \mathsf{VERIFY}(ipk, \sigma, m, bsn, (D, I), \emptyset, \mathtt{SRL})$, and $f' \leftarrow \mathsf{VERIFY}(ipk, \sigma', m', bsn, (D', I'), \emptyset, \mathtt{SRL}')$.
- Continue if $f = f' = 1$, else abort with output $\perp$.
- Parse $\sigma = (nym, \pi_{cred}, \{\pi_{\mathtt{SRL},i}\})$, $\sigma' = (nym', \pi'_{cred}, \{\pi'_{\mathtt{SRL},i}\})$.
- If $nym = nym'$, output 1, and 0 otherwise.

**Fig. 4.** High-level overview of the DAA protocols.

Both protocols roughly follow the common structure of previous DAA protocols: the platform, consisting of a TPM and a host, generates a secret key $gsk$ that gets blindly certified by a trusted issuer in a membership credential $cred$. When attributes $attrs = a_1, \dots a_L$ are used, the credential also certifies $attrs$. After that join procedure, the platform can use the key $gsk$ to sign attestations and basenames and prove that it has a valid credential on the underlying key, which certifies the trusted origin of the attestation. The overview of the DAA protocol is depicted in Figure 4.

*Split-Keys for Strong Privacy.* In contrast to existing schemes, we do not set $gsk = tsk$ because solely relying on the secret key $tsk$ of the TPM would not allow for the strong privacy property we are aiming for. Instead, we partially follow the approach of Camenisch et al. [CDL17] and let the host contribute to the platform's secret key. That is, we split the key as $gsk = tsk + hsk$, where $hsk$ is the contribution of the host to the platform secret key. As in previous work, the platform secret key $gsk$ gets blindly signed by the issuer using a partially blind signature $\mathsf{PBSign}$ that certifies the secret key by signing the platform's public key $gpk = \tilde{g}^{gsk}$.

Note that to allow for algorithmic agility, we derive the platform's key from a generator $\tilde{g}$, which can either be a cleared generator created with `TPM.Commit` as $\tilde{g} \leftarrow \mathsf{H}_{\mathbb{G}_1}(0||str)$ for some string $str$, or $\tilde{g} \leftarrow \bar{g}$, i.e. being the standard generator fixed in all TPMs. When using a cleared generator, the input to the hash function will be prepended with a 0-bit to ensure that the same generator will not be used in a signature (where we will prepend a 1-bit when creating generators), as this would break the unlinkability between joining and signing otherwise.

We now have to ensure that $gsk$ is derived from a key $tsk$ held inside a real TPM. To this end, the TPM first has to prove in $\pi_{tpk}$ that its contribution $tpk' = \tilde{g}^{tsk}$ is based on the same secret key $tsk$ as the actual TPM public key $tpk = \bar{g}^{tsk}$. The host then forwards $tpk, tpk'$ and $\pi_{tpk}$ along with a proof $\pi_{gpk}$ that it correctly derived $gpk$ from the TPM's contribution $tpk'$ to the issuer.

Each TPM is equipped by the manufacturer with an endorsement key. This key allows the issuer to verify the authenticity of the TPM provided values in the $\mathsf{JOIN}$ protocol. As this is the standard procedure in all DAA protocols, we omit the details how this authentication is done and implicitly assume that the value $tpk$ in the $\mathsf{JOIN}$ protocol is authenticated with the endorsement key.

After having obtained a membership credential on the joint secret key $gsk$ (and possibly a set of attributes $attrs$), the attestation signatures are then computed jointly by the host and TPM.

*Signature-Based Revocation.* We also want to support signature-based revocation introduced in the EPID protocol by Brickell and Li [BL07, BL11] as it allows one to revoke TPMs without assuming that a secret key held inside the TPM becomes publicly available upon corruption, which improves the standard private-key-based revocation in DAA.

Roughly, for signature-based revocation, a platform would extend its signatures by additional values $(B, nym)$ where $B$ is a random generator for $\mathbb{G}_1$ and $nym \leftarrow B^{gsk}$. The signature revocation list `SRL` contains tuples $\{(B_i, nym_i)\}$ from signatures of the platforms that are revoked. Thus, a platform must also show that it is not among that list by proving $\pi_{SRL,i} \leftarrow \mathsf{SPK}^*\{(gsk) : nym = B^{gsk} \wedge nym_i \neq B_i^{gsk}\}$. Any TPM interface that supports such proofs would raise $B_i$ to the secret key and inevitably provide a static DH oracle.

Camenisch et al. [CDL16a] recently addressed this issue and proposed a $q$-SDH-based DAA scheme with signature-based revocation that avoids this issue. Instead of giving the generator as direct input, it uses $B_i \leftarrow \mathsf{H}_{\mathbb{G}_1}(1||bsn_i)$ computed by the TPM, i.e., the TPM gets $1||bsn_i$ as input and the `SRL` has the form $\{(1||bsn_i, nym_i)\}$. For every $(1||bsn_i, nym_i) \in \mathtt{SRL}$, the platform shows that $\mathsf{H}_{\mathbb{G}_1}(1||bsn_i)^{gsk} \neq nym_i$ by taking a random $\gamma$, setting $C_i = (\mathsf{H}_{\mathbb{G}_1}(1||bsn_i)/nym_i)^{\gamma}$, and proving

$$\pi'_{\mathtt{SRL},i} \leftarrow \mathsf{SPK}^*\{(\gamma \cdot gsk, \gamma) : 1 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{\gamma \cdot gsk}(\frac{1}{nym})^{\gamma} \quad \wedge \quad C_i = \mathsf{H}_{\mathbb{G}_1}(1||bsn_i)^{\gamma \cdot gsk}(\frac{1}{nym_i})^{\gamma}\}(\text{“}sign\text{”}).$$

While the proposed scheme successfully removes the static DH oracle and is provably secure in the UC model, their protocol makes different calls to the TPM to prove non-revocation, and requires the TPM to maintain state $(bsn, nym)$ that it used in the signing procedure to later create the non-revocation proofs. Extra TPM commands would be required to implement this exact behavior in a TPM. In this work, we use the same core idea but slightly change the communication, such that we can leverage the flexible `TPM.Commit` and `TPM.Sign` commands and avoid introducing new TPM commands. In addition, we give the TPM all the input it requires to create the non-revocation proof, such that it does not need to keep any state between signing and creating the non-revocation proof. More precisely, we can construct the non-revocation proof based on our revised TPM interface using the Prove protocol. The host obtains $C_i$ and constructs $\pi_{\mathsf{SRL},i} \leftarrow (C_i, \pi'_{\mathsf{SRL},i})$ by running

$$(C_i, \pi'_{\mathsf{SRL},i}) \leftarrow \mathsf{Prove}(hsk, 1_{\mathbb{G}_1}, 1||bsn, 1, \perp, \gamma, 1||bsn_i, \perp, \{(\gamma, 1/nym, 1/nym_i, \perp)\}, \text{“}sign\text{”}, \perp),$$

To verify $\pi_{\mathsf{SRL},i}$ in the VERIFY algorithm, one parses $\pi_{\mathsf{SRL},i} = (C_i, \pi'_{\mathsf{SRL},i})$, checks that $C_i \neq 1_{\mathbb{G}_1}$, and verifies $\pi'_{\mathsf{SRL},i}$ w.r.t. $(C_i, 1||bsn_i, nym_i, nym)$, where $(1||bsn_i, nym_i) \in \mathsf{SRL}$.

Note that since signature-based revocation is independent of the concrete PBSign scheme used for the membership credential, the above proof instantiation and the revocation checks in VERIFY are the same for the $q$-SDH-based and LRSW-based schemes.

*Concrete Instantiations.* The description of the JOIN and SIGN protocols and the VERIFY and LINK algorithms are given in Figure 4, using an abstract NIZK proof statement for $\pi_{cred}$, and a generic partially-blind signature scheme PBSign for obtaining the membership credential. The concrete instantiation for this proof depends on the instantiation used for the PBSign scheme. In the following two sections we describe how PBSign and $\pi_{cred}$ can be instantiated with a $q$-SDH-based scheme (BBS+ signature [ASM06]) and a LRSW-based scheme (CL-signature [CL04]) respectively. The latter uses a novel way to blindly issue CL signatures, which is significantly more efficient than previous approaches and is of independent interest.

For both concrete instantiations we assume the availability of system parameters consisting of a security parameter $\tau$, a bilinear group $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order $p$ with generators $g_1$ of $\mathbb{G}_1$ and $g_2$ of $\mathbb{G}_2$ and bilinear map $e$, generated w.r.t $\tau$, and with $\bar{g}$ denoting the fixed generator used by the TPMs. Note that we will not repeat the parts of the DAA protocol that are independent of the PBSign instantiation, such as the signature-based revocation, the revocation checks within VERIFY, and the LINK protocol.

**$q$-SDH-based DAA Instantiation** Our $q$-SDH-based scheme is most similar to the scheme by Camenisch et al. [CDL16a], which in turn propose a provably secure version of the scheme by Brickell and Li [BL10], which is standardized as mechanism 3 in ISO/IEC 20008-2 [Int13]. In addition, their and our scheme support membership credentials with selective attribute disclosure, similar to DAA with Attributes as proposed by Chen and Urian [CU15].

We now show how to instantiate PBSign and the affected proofs with $q$-SDH-based BBS+ signatures yielding a provably secure $q$-SDH-based DAA scheme $\Pi_{\mathsf{qSDH-DAA}}$ using the revised TPM 2.0 interfaces proposed in Section 4.

<u>SETUP:</u> The issuer generates its key pair $(ipk, isk)$ as follows:

- Choose $(h_0, \ldots, h_L) \stackrel{\$}{\leftarrow} \mathbb{G}_1^{L+1}$, $x \stackrel{\$}{\leftarrow} \mathbb{Z}_p$, set $X \leftarrow g_2^x$ and $X' \leftarrow g_1^x$, and prove $\pi_{ipk} \stackrel{\$}{\leftarrow} \mathsf{SPK}\{x : X = g_2^x \wedge X' = g_1^x\}(\text{“}setup\text{”})$.
- Set $ipk \leftarrow (h_0, \ldots, h_L, X, X', \pi_{ipk})$, and $isk \leftarrow x$.

Protocol participants, when retrieving $ipk$, will verify $\pi_{ipk}$.

<u>JOIN:</u> Here we show how the host obtains the proof $\pi_{tpk}$ from the TPM and how the issuer computes the membership credential using the BBS+ signature scheme. For this scheme, we set $\tilde{g} = \bar{g}$, so $tpk = tpk'$ and we can simplify $\pi_{tpk}$ to $\pi_{tpk} \leftarrow \mathsf{SPK}^*\{tsk : tpk = \bar{g}^{tsk}\}(\text{“}join\text{”}, n)$.

- The host obtains $\pi_{tpk}$ by calling

$$(*, \pi_{tpk}) \leftarrow \mathsf{Prove}(0, tpk, \perp, 1, \perp, 1, \perp, \perp, \emptyset, \perp, (\text{“}join\text{”}, n)).$$

– The issuer computes the membership credential $cred \leftarrow \mathsf{PBSign}(isk, gpk, attrs)$ on the joint public key $gpk$ and a set of attributes $attrs = (a_1, \ldots, a_L)$ with $isk = x$ as follows: It chooses a random $(e, s) \in \mathbb{Z}_p^2$, and derives

$$A \leftarrow (g_1 \cdot h_0^s \cdot gpk \cdot \prod_{i=1}^{L} h_i^{a_i})^{\frac{1}{e+x}}.$$

That is, the issuer creates a standard BBS+ signature on the message $(gsk, a_1, \ldots, a_L)$, where $gsk = tsk + hsk$ is blindly signed in form of $gpk = \bar{g}^{gsk}$. It sets $cred \leftarrow (A, e, s)$.

– The host upon receiving $(cred, attrs)$ from the issuer, computes $b \leftarrow g_1 \cdot h_0^s \cdot gpk \cdot \prod_{i=1}^{L} h_i^{a_i}$, and checks that $\mathbf{e}(A, Xg_2^e) = \mathbf{e}(b, g_2)$. Finally, it sets $cred' \leftarrow ((A, e, s), b)$.

<u>SIGN:</u> A platform holding a membership credential $cred' = ((A, e, s), b)$ on platform key $gsk$ and attributes $attrs$ can sign message $m$ w.r.t. basename $bsn$, attribute disclosure $(D, I)$, and signature-based revocation list $\mathtt{SRL}$. As shown in Figure 4, each signature $\sigma$ contains a proof of a membership credential $\pi_{cred}$ w.r.t. the pseudonym $nym = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, which are computed as follows:

– The host first randomizes the BBS+ credential $((A, e, s), b)$: Choose $r_1 \xleftarrow{\$} \mathbb{Z}_p^*, r_2 \xleftarrow{\$} \mathbb{Z}_p, r_3 \leftarrow \frac{1}{r_1}$, set $A' \leftarrow A^{r_1}$, $\bar{A} \leftarrow A'^{-e} \cdot b^{r_1} (= A'^x)$, $b' \leftarrow b^{r_1} \cdot h_0^{-r_2}$, and $s' \leftarrow s - r_2 \cdot r_3$. The host and TPM then jointly compute the following proof $\pi'_{cred}$. We denote by $\bar{D} = \{1, \ldots, L\} \backslash D$ the indices of attributes that are not disclosed.

$$\pi'_{cred} \leftarrow \mathsf{SPK}^*\{(gsk, \{a_i\}_{i \in \bar{D}}, e, r_2, r_3, s') : g_1^{-1} \prod_{i \in D} {h_i}^{-a_i} = b'^{-r_3} h_0^{s'} \bar{g}^{gsk} \prod_{i \in \bar{D}} {h_i}^{a_i} \;\wedge$$
$$nym = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk} \;\wedge\; \bar{A}/b' = A'^{-e} \cdot h_0^{r_2}\}((\text{“}sign\text{”}, (D, I), \mathtt{SRL}), m)$$

This proof and pseudonym are computed by running

$$(nym, \pi'_{cred}) \leftarrow \mathsf{Prove}(hsk, d, \perp, 1, \perp, 1, 1||bsn, \bar{A}/b', S, (\text{“}sign\text{”}, (D, I), \mathtt{SRL}), m),$$

with $d \leftarrow g_1^{-1} \prod_{i \in D} h_i^{-a_i}$ and the set of all witnesses for the proof: $S \leftarrow \{(-e, 1_{\mathbb{G}_1}, 1_{\mathbb{G}_1}, A'), (r_2, 1_{\mathbb{G}_1}, 1_{\mathbb{G}_1}, h_0), (-r_3, b', 1_{\mathbb{G}_1}, 1_{\mathbb{G}_1}), (s', h_0, 1_{\mathbb{G}_1}, 1_{\mathbb{G}_1})\} \cup \{(a_i, h_i, 1_{\mathbb{G}_1}, 1_{\mathbb{G}_1})\}_{i \in \bar{D}}$. The host then sets $\pi_{cred} \leftarrow (\bar{A}, A', b', \pi'_{cred})$.

<u>VERIFY:</u> To verify $\pi_{cred} = (\bar{A}, A', b', \pi'_{cred})$ w.r.t. $(ipk, \sigma, m, bsn, (D, I), \mathtt{RL}, \mathtt{SRL})$ and $nym$, parse $ipk = (h_0, \ldots, h_L, X, X', \pi_{ipk})$, check that $A' \neq 1_{\mathbb{G}_1}$ and $\mathbf{e}(A', X) = \mathbf{e}(\bar{A}, g_2)$, and verify $\pi'_{cred}$ with respect to message $m$, basename $bsn$, attribute disclosure $(D, I)$, signature revocation list $\mathtt{SRL}$, randomized credential $(\bar{A}, A', b')$, and pseudonym $nym$.

**LRSW-based DAA Instantiation** We now demonstrate that an LRSW-based DAA scheme can be built on top of the new TPM interface. Our scheme is similar to the scheme by Chen, Page, and Smart [CPS10], standardized as mechanism 4 of ISO/IEC 20008-2 [Int13], but includes the fixes to flaws pointed out by Bernhard et al. [BFG+13] and Camenisch et al. [CDL16c].

Note, for the sake of efficiency we do not include attributes in this scheme. Selective attribute disclosure can be supported using the extension by Chen and Urian [CU15], but it comes with a significant loss in efficiency. When attributes are required, the $q$-SDH-based scheme should be used.

*A New Approach to Issue CL-Signatures.* The main difference to the schemes by Bernhard et al. [BFG+13] and Camenisch et al. [CDL16c] is the way we prevent a static DH oracle when the membership credentials are generated. In LRSW-based schemes, $cred$ is a CL-signature $(a, b, c, d)$ on $gsk$, where for blind signing the issuer chooses $\alpha \xleftarrow{\$} \mathbb{Z}_p^*$ and sets

$$a \leftarrow \bar{g}^{\alpha}, b \leftarrow a^y, c \leftarrow a^x \cdot gpk^{\alpha \cdot xy}, d \leftarrow gpk^{\alpha \cdot y},$$

with $(x, y)$ denoting the issuer's signing key and $gpk = \bar{g}^{gsk}$ the platform public key. The DH oracle arises as the TPM must later prove knowledge of $d = b^{gsk}$, and $b$ is a value chosen by the issuer.

The schemes by Bernhard et al. [BFG+13] and Camenisch et al. [CDL16c] avoid such an oracle by letting the issuer prove $\pi \xleftarrow{\$} \mathsf{SPK}\{(\alpha \cdot y) : b = \bar{g}^{\alpha \cdot y} \wedge d = gpk^{\alpha \cdot y}\}$. Thus, the issuer proves that it correctly computed $d = b^{gsk}$, which shows the TPM that it can use $b$ as a generator without forming a static DH oracle (as the issuer already knows $d$). The TPM must therefore verify $\pi$, store $(b, d)$ along with its key, and only use these values in the subsequent SPKs.

While allowing for a security proof under the standard DL assumption, realizing this approach would require significant changes to the TPM interface to verify and store the additional key material. Further, the TPM 2.0 specification aimed to provide a *generic* interface for a number of protocols, and adding LRSW-DAA specific changes would thwart this effort.

Our goal is to keep the TPM protocol as generic and simple as possible, and we propose a novel and more elegant solution that avoids the DH oracle without requiring the TPM to verify a zero-knowledge proof. For the sake of simplicity we assume $gsk = tsk$ for the exposition of our core idea, and only include the split-key approach $gsk = tsk + hsk$ in the full protocol specification.

The issuer chooses a random nonce $n$ and we derive $b \leftarrow \mathsf{H}_{\mathbb{G}_1}(0||n)$. The TPM receives $n$, derives $b$ and sends $d = b^{gsk}$ to the issuer. Note that $d$ does not leak information about $gsk$ when we model $\mathsf{H}_{\mathbb{G}_1}$ as a random oracle. The issuer then completes the credential by computing

$$a \leftarrow b^{1/y}, \quad c \leftarrow (a \cdot d)^x.$$

It is easy to see that the values $(a, b, c, d)$ derived in that way, form a standard CL signature on $gsk$ as in the existing schemes. Note that we now use $\mathsf{H}_{\mathbb{G}_1}$ in both the join protocol and to create pseudonyms while signing. We prefix the hash computation with a bit to distinguish these cases, to prevent losing privacy when signing with a basename $bsn$ equal to nonce $n$.

This new blind issuance protocol is provably secure under the generalized LRSW assumption as introduced in Section 2, which we prove as one step in our full security proof in Appendix D. We need the generalized LRSW assumption, as the issuer already commits to values $a$ and $b$ before getting the $d$ value and computing $c$ based on $d$. One can easily modify the issuance scheme to be secure under the standard LRSW assumption though, one needs to prepend one extra round between the TPM and the issuer before running the issuance as described above. Therein, the issuer sends a nonce $n'$ to the TPM, and the TPM responds with a proof $\pi \leftarrow \mathsf{SPK}^*\{gsk : gpk = \bar{g}^{gsk}\}(n')$. The issuer verifies $\pi$ and then continues with the issuance as described above. In the security proof this allows to extract $gsk$ from $\pi$ and we can obtain the full signature $(a, b, c)$ on $gsk$ from the LRSW oracle. Note that this extra round can be implemented with our revised TPM interface as well, but slightly reduces the efficiency of the overall $\mathsf{JOIN}$ protocol.

We now describe how this new issuance protocol is used in the LRSW-based instantiation of our DAA protocol. We denote the DAA protocol given in Figure 4 instantiated with the LRSW-based membership credential and the proofs described below as $\Pi_{\mathsf{LRSW-DAA}}$.

<u>SETUP:</u> The issuer generates its key pair $(ipk, isk)$ as follows:

- Choose $x, y \xleftarrow{\$} \mathbb{Z}_p^*$, set $X \leftarrow g_2^x, Y \leftarrow g_2^y$, and compute $\pi_{ipk} \xleftarrow{\$} \mathsf{SPK}\{(x, y) : X = g_2^x \wedge Y = g_2^y\}(\text{“}setup\text{”})$.
- Set $ipk \leftarrow (X, Y, \pi_{ipk})$, and $isk \leftarrow (x, y)$.

When first getting the issuer public key, protocol participants will check $Y \neq 1_{\mathbb{G}_2}$ and verify $\pi_{ipk}$.

<u>JOIN:</u> Opposed to the $q$-SDH-based protocol, we make use of the flexibility for the generator of the platform's key. That is, instead of using $\bar{g}$ we will use $\tilde{g} = \mathsf{H}_{\mathbb{G}_1}(0||n)$ which will also serve as the $b$-value in the improved issuance of CL credentials as described above.

- First, upon receiving $n$ from the issuer, the host and TPM create $gpk$, $tpk'$, $\pi_{tpk}$, $\pi_{gpk}$ based on $\tilde{g} = b = \mathsf{H}_{\mathbb{G}_1}(0||n)$. Recall that the TPM authenticates only the value $tpk = \bar{g}^{tsk}$, so the TPM must prove that $tpk' = \tilde{g}^{tsk}$ uses the same $tsk$ as in its authenticated public key $tpk$:

$$\pi_{tpk} \leftarrow \mathsf{SPK}^*\{tsk : tpk = \bar{g}^{tsk} \ \wedge \ tpk' = \tilde{g}^{tsk}\}(\text{“}join\text{”}, n)$$

The TPM's key contribution $tpk'$ and the proof $\pi_{tpk}$ are created via the $\mathsf{Prove}$ protocol for the following input:

$$(tpk', \pi_{tpk}) \leftarrow \mathsf{Prove}(0, tpk, \perp, 1, \perp, 1, (0||n), \perp, \emptyset, \perp, (\text{``}join\text{''}, n))$$

The host then picks a key $hsk$, computes $gpk = tpk' \cdot \tilde{g}^{hsk}$ and $\pi_{gpk}$ (as described in Figure 4) and finally sends $tpk, tpk', \pi_{tpk}, \pi_{gpk}, gpk$ to the issuer.

– Then, the issuer blindly completes the CL signature on $gsk = tsk + hsk$ as described above: the issuer computes $a \leftarrow \tilde{g}^{1/y}$, $c \leftarrow (a \cdot gpk)^x$, and sets $cred \leftarrow (a, c)$. Note that $gpk = \tilde{g}^{gsk} = b^{gsk}$, so we can use this as the $d$-value of the credential.
– The host upon receiving $cred = (a, c)$ from the issuer verifies that $a \neq 1_{\mathbb{G}_1}$, $\mathbf{e}(a, Y) = \mathbf{e}(\tilde{g}, g_2)$, and $\mathbf{e}(c, g_2) = \mathbf{e}(a \cdot gpk, X)$. Finally, the host sets $cred' = (a, \tilde{g}, c, gpk, n)$.

<u>SIGN:</u> We now describe how to instantiate the membership proof $\pi_{cred}$ for such CL signatures with our TPM methods.

– The host retrieves the join record $(hsk, cred')$ and randomizes the CL credential $cred' = (a, \tilde{g}, c, gpk, n)$ by $r \xleftarrow{\$} \mathbb{Z}_p^*$ and setting $a' \leftarrow a^r, \tilde{g}' \leftarrow \tilde{g}^r, c' \leftarrow c^r, gpk' \leftarrow gpk^r$.
– The host and TPM then jointly compute $nym \leftarrow \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$ for $gsk = tsk + hsk$ and prove knowledge of a CL credential on $gsk$ by creating:

$$\pi'_{cred} \leftarrow \mathsf{SPK}^*\{(gsk) : gpk' = \tilde{g}'^{gsk} \ \wedge \ nym = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}\}((\text{``}sign\text{''}, \mathtt{SRL}), m).$$

This proof and pseudonym $nym$ are computed by

$$(nym, \pi'_{cred}) \leftarrow \mathsf{Prove}(hsk, gpk', (0||n), r, \perp, 1, (1||bsn), \perp, \emptyset, (\text{``}sign\text{''}, \mathtt{SRL}), m).$$

Finally, the host sets $\pi_{cred} \leftarrow (a', \tilde{g}', c', gpk', \pi'_{cred})$.

<u>VERIFY:</u> To verify $\pi_{cred} = (a', \tilde{g}', c', gpk', \pi'_{cred})$ w.r.t. $(ipk, \sigma, m, bsn, \mathtt{RL}, \mathtt{SRL})$ and $nym$, parse $ipk = (X, Y, \pi_{ipk})$, check that $a' \neq 1_{\mathbb{G}_1}$, $\mathbf{e}(a', Y) = \mathbf{e}(\tilde{g}', g_2)$, and $\mathbf{e}(c', g_2) = \mathbf{e}(a' \cdot gpk', X)$, and verify $\pi'_{cred}$ with respect to $(m, bsn, \mathtt{SRL}, \tilde{g}', gpk', nym)$.

### 5.3 Security Properties of our Schemes

In this section we informally discuss the security of our DAA schemes. The formal security proof is postponed to Appendix D.

**Theorem 1 (Informal).** *Protocol $\Pi_{\mathsf{LRSW-DAA}}$ is a secure anonymous attestation scheme under the Generalized LRSW and Decisional Diffie-Hellman assumptions in the random oracle model.*

**Theorem 2 (Informal).** *Protocol $\Pi_{\mathsf{qSDH-DAA}}$ is a secure anonymous attestation scheme under the q-SDH and Decisional Diffie-Hellman assumptions in the random oracle model.*

The proofs of these two theorems are quite similar. In the following we give a proof sketch that treats both schemes at the same time, pointing out the differences when they arise.

*Proof (Sketch).* For each of the properties stated in Section 5.1, we argue why our schemes satisfy them. The actual security proof is structured quite differently as there we prove that an environment cannot distinguish between the interactions with the real world parties and with the ideal specification with a simulator. Nevertheless, the arguments presented here also appear in the full formal proof.

*Unforgeability.* First, we argue that the adversary cannot use a credential from a platform with an honest TPM. In both our schemes, signatures are signature proofs of knowledge of the platform secret key $tsk + hsk$, as defined in (4.1). This means that from Lemma 2 we can directly conclude that the adversary cannot use the credential of a platform with an honest TPM. Second, the adversary cannot use a revoked credential on the key $gsk$ by a corrupt platform. For private-key based revocation, the platform proves that $nym = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$ is correctly constructed, and the revocation check will reject signatures with that pseudonym. If signature-based revocation is used, a pair $(bsn_i, nym_i = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk})$ is included in $\mathsf{SRL}$. In proof $\pi'_{\mathsf{SRL},i}$, the adversary must prove that his $gsk$ is different than the one used in $nym_i$, which contradicts the soundness of the zero knowledge proof.

It remains to show that the adversary cannot create signatures using a forged credential. For $\Pi_{\mathsf{qSDH-DAA}}$, this clearly breaks the existential unforgeability of the BBS+ signature scheme, which is proven under the $q$-SDH assumption. For $\Pi_{\mathsf{LRSW-DAA}}$, we have to show that credentials are unforgeable under the generalized LRSW assumption. For this, we simulate the issuer with a generalized LRSW instance. When the join protocol starts, the issuer asks $\mathcal{O}_X^{\mathsf{a,b}}$ for $(a, b)$. It chooses a fresh nonce $n$ and programs the random oracle $\mathsf{H}_{\mathbb{G}_1}(0||n) = b$. When it receives proofs $\pi_{tpk}$, $\pi_{gpk}$ it extracts $tsk$ and $hsk$ and sets $gsk = tsk + hsk$. It then calls $\mathcal{O}_{X,Y}^{\mathsf{c}}$ on $gsk$ to complete the credential. Now, when the adversary creates a signature with a forged credential, we can extract a credential $(a^*, b^*, c^*)$ on the fresh $gsk^*$ breaking the generalized LRSW assumption.

*Non-Frameability.* An honest platform cannot be framed, under the Discrete Logarithm (DL) assumption (which is implied by the assumptions we make). The host sets $gpk$ and $\tilde{g}$ based on given the DL instance, and must simulate $\pi_{gpk}$ as it does not know $hsk$ such that $gpk = tpk \cdot \tilde{g}^{hsk}$. When signing, the host also simulates the zero-knowledge proofs. Now, if an adversary creates a signature that links to a signature of the honest platform, it must prove knowledge of the discrete logarithm of $gsk$. We rewind to extract and break the DL assumption.

*Strong Privacy.* Our DAA schemes fulfill strong privacy, meaning that privacy is guaranteed as long as the host is honest, i.e., even when the TPM involved in the generation of an attestation is malicious. By Lemma 1, the proofs created together with a (malicious) TPM are zero knowledge. This means we can simulate these proofs without the adversary noticing the difference. Further, note that a platform key $gsk = tsk + hsk$ is uniformly distributed over $\mathbb{Z}_p$ as the host picks $hsk$ uniformly at random from $\mathbb{Z}_p$. To prove that signatures are unlinkable, we let honest hosts pick a fresh key $gsk$ every time they sign with a new basename. This is indistinguishable using a hybrid argument, where in the $i$-th hop, we use a fresh key for $bsn_i$. Every hop is indistinguishable from the previous one under the Decisional Diffie-Hellman (DDH) assumption.

In a nutshell, the latter is proved as follows. Upon receiving a DDH instance $(\alpha, \beta, \gamma)$, program the random oracle so that $\mathsf{H}_{\mathbb{G}_1}(1||bsn_i) \leftarrow \beta$. The host sets $\alpha$ as the $gpk$ value and simulates proof $\pi_{gpk}$. When signing, the host simulates the proof of knowledge and sets $nym \leftarrow \gamma$. If the DDH instance is a DDH tuple, the same key was used to sign, and if it is not a DDH tuple, a fresh key was used.

Signatures are now done using a fresh key for each basename and the proofs are simulated, therefore no adversary can possibly break the anonymity of signatures.

## 6 DAA with Forward Anonymity

An important reason to remove the DH oracle in the TPM interfaces is that such an oracle prevents forward anonymity. As Xi et al. [XYZF14] point out, a host that becomes corrupted can test whether signatures were generated by the embedded TPM using the DH oracle.

Modeling the property of forward anonymity requires one to consider adaptive corruptions, i.e., a signature made by a host should remain anonymous even when at some later point the host becomes corrupted. A property-based notion for this was formally introduced by Xi et al. [XYZF14]. However, extending our ideal specification to also provide this property is nontrivial. First, to enable forward anonymity, the DAA scheme must allow one to create signatures w.r.t. *no* basename, i.e., $bsn = \perp$ and forward anonymity only holds for such signatures. Otherwise, a host that becomes corrupt could trivially link previous signatures generated for some basename $bsn \neq \perp$, by simply requesting a new signature w.r.t. $bsn$ and test for relation via the $\mathsf{link}$ algorithm. This means we would have to remove signature-based revocation from our security model. Second, our formal security proof considers *static* corruptions, whereas forward anonymity is inherently about dynamic corruptions. Indeed, realizing a scheme secure w.r.t. dynamic corruptions would be much less efficient than the scheme we present in this paper.

Despite this, the TPM interfaces we define allow one to build a DAA scheme with forward anonymity (however, the other security properties hold only in presence of static corruptions). That is, if we remove signature-based revocation from our DAA protocols, they fulfill the notion of forward security by Xi et al. For LRSW-based DAA, signing with $bsn = \perp$ means that $nym$ is omitted from the signature and proof $\pi_{cred}$. For $q$-SDH-based DAA, if $bsn = \perp$ then $nym$ is replaced by $j^{gsk}$, where $j$ is taken uniformly at random from $\mathbb{G}_1$ by the TPM, as in the $q$-SDH-based scheme by Brickell and Li [BL10].

Proving the resulting scheme to be forward anonymous would work as follows. The forward anonymity game considers a corrupt issuer. This means $\mathcal{A}$ can instruct platforms to join, and $\mathcal{A}$ runs the issuer side of the protocol. $\mathcal{A}$ can request complete signatures from joined platforms. Next, $\mathcal{A}$ submits the identities of two platforms and a message. The challenger chooses one of the two platforms at random and returns a signature on the given message with basename $bsn = \perp$ on behalf of the chosen platform. The game now models the fact that the host becomes corrupted by giving $\mathcal{A}$ access to the TPM commands of the platforms, and $\mathcal{A}$'s task is to find out which of the two platforms created the signature.

For both schemes, we can prove forward anonymity under the DDH assumption, using a similar proof strategy as for strong privacy. First, the challenger answers all oracles correctly. Next, we modify the game slightly. The challenge signature is now computed under a fresh key, instead of the key of one of the two platforms that $\mathcal{A}$ submitted. In this modified game, no adversary can win with probability better than $\frac{1}{2}$, as the bit that $\mathcal{A}$ has to guess is independent of $\mathcal{A}$'s view. This means $\mathcal{A}$ can only have non-negligible advantage by distinguishing the two games. As argued in the strong privacy proof in Sect. 5.3, the modification in the games is unnoticeable under the DDH assumption. showing that our protocols without signature-based revocation satisfy forward anonymity under the DDH assumption.

## 7 Other Uses of our TPM Interfaces

In many protocols, the user would like to store his keys in secure hardware rather than on a normal computer. This way, the keys are secure and some security is preserved as long as the trusted hardware is not compromised, even when the computer is compromised. This section shows that due to the generic design of our TPM interface, it can be used to secure the keys of other cryptographic protocols. As an example, we consider U-Prove and e-cash with keys stored in a TPM, such that an attacker cannot use a user's U-Prove credential or e-cash wallet without access to the TPM. We discuss these constructions here only informally, i.e., without providing a security proof, as a formal treatment would require a new security model and a detailed proof, which is beyond the scope of this paper. For ease of presentation, we place the full key in the TPM, although we could split the key over the TPM and host as in our DAA schemes.

### 7.1 Device Bound U-Prove

U-Prove [PZ13] is an attribute-based credential system where credential issuance and credential presentation are unlinkable. In the issuance protocol, the user receives a credential with public key $h = (g_0 g_1^{x_1} \dots g_n^{x_n} g_d^{x_d})^\alpha$, where $x_1, \dots, x_n$ are the attribute values of the user, and $x_d$ is the device secret. The device secret makes sure that a secure device must be present to use the credential. To show the credential, the user must prove knowledge of $x_1, \dots, x_n$, $x_d$, and $\alpha$ such that $g_0 = g_1^{x_1} \dots g_n^{x_n} g_d^{x_d} \cdot h^{-1/\alpha}$, with the help of the secure device.

Our proposed changes for TPM 2.0 allow the TPM to be used as secure device for U-Prove. The value $x_d$ will be the TPM secret key, and generator $g_d$ must be the generator $\bar{g}$ known to the TPM. Then, the credential presentation proof $\mathsf{SPK}^*\{(x_1, \dots, x_n, x_d, \alpha) : g_0 = g_1^{x_1} \dots g_n^{x_n} g_d^{x_d} \cdot h^{-1/\alpha}\}$ can be constructed by computing $(nym, \pi) \leftarrow \mathsf{Prove}(0, g_0, \perp, 1, \perp, 1, \perp, \perp, \{(a_1, g_1, \perp, \perp), \dots, (a_n, g_n, \perp, \perp), (1/\alpha, h, \perp, \perp)\}, \perp, \perp)$. By Lemma 3, such proofs can only be made with a contribution from the TPM, so one's credentials cannot be stolen, unless the attacker can access the TPM.

### 7.2 Compact E-Cash

Compact E-Cash [CHL05] allows users to withdraw coins from a bank, and later anonymously spend the coins. The protocol assumes that every user has a key pair $(sk_U, pk_U = g^{sk_U})$ with which it can authenticate towards the bank. To withdraw $2^l$ coins, the user first authenticates towards the bank by proving knowledge of $sk_U$. The user picks wallet secrets $s, l$, where the bank adds randomness to $s$, and the bank places signature $\sigma$ on committed values $sk_U$, $s$, and $l$, using a CL signature. The result of the withdraw protocol is a wallet $(sk_U, s, t, \sigma, J)$, where $J$ is an $l$-bit counter.

To spend a coin at merchant $M$, the user computes $R \leftarrow \mathsf{H}(pk_M, \mathit{info})$, where the merchant provides $\mathit{info}$. Next, the user computes a coin serial number $S \leftarrow g^{\frac{1}{s+J+1}}$ and value $T \leftarrow pk_U \cdot g^{\frac{R}{t+J+1}}$ which is used to detect double spending of coins. Finally, it proves

$$\mathsf{SPK}\{(J, sk_U, s, t, \sigma) : 0 \leq J < 2^l \wedge S = g^{\frac{1}{s+J+1}} \ \wedge\ T \leftarrow pk_U \cdot g^{\frac{R}{t+J+1}} \ \wedge\ \mathsf{Ver}(pk_B, (sk_U, s, t), \sigma) = 1\}$$

We can instantiate Compact E-Cash such that users can securely store their secret key $sk_U$ inside a TPM, using a trick similar as in our LRSW-based DAA scheme. To create its keys, the bank picks secret key $(x, y, z_1, z_2, z_3) \xleftarrow{\$} \mathbb{Z}_p^5$ and sets public key $X \leftarrow g_2^x$, $Y \leftarrow g_2^y$, $Z_1 \leftarrow g_2^{z_1}$, $Z_2 \leftarrow g_2^{z_2}$, and $Z_3 \leftarrow g_2^{z_3}$. The withdrawal of coins start by the bank picking a fresh nonce $n$, and sending $n$, $b \leftarrow \mathsf{H}(n)$, $a \leftarrow b^{1/y}$, $A_i \leftarrow a^{z_i}$ and $B_i \leftarrow b^{z_i}$ for $i = 1, 2, 3$ to the user. The user authenticates by proving $pk_U = g_1^{sk_U} \wedge d = b^{sk_U}$, as in our LRSW-based DAA scheme. In addition, it picks $s'$, $t$, and $r$, and commits to them using generators $B_1$, $B_2$, and $B_3$: $C \leftarrow B_1^{s'} B_2^t B_3^r$. The user sends $C$ with a proof of knowledge of $(s', t, r)$ to the bank. The bank now adds randomness to $s''$ to $s'$ by setting $C' \leftarrow C \cdot B_1^{s''}$ and signs $sk_U$, $s = s' + s''$, $t$, and $r$, by setting $c \leftarrow (a \cdot d \cdot C'^y)^x = a^{x+xy(m+z_1 s+z_2 t+z_3 r)}$. The user now has signature $\sigma = (a, A_1, A_2, A_3, b, B_1, B_2, B_3, c, d)$.

To spend a coin, the user must compute $R$, $S$, and $T$, and prove that everything is correctly computed, as described above. The TPM holding $sk_U$ is only involved in proving that $\sigma$ is a valid CL signature on $(sk_U, s, t, r)$. It randomizes the signature by taking $\rho \leftarrow \mathbb{Z}_p^*$ and setting $a' \leftarrow a^\rho, A'_i \leftarrow A_i^\rho, b' \leftarrow b^\rho, B'_i \leftarrow B_i^\rho, c' \leftarrow c^\rho$. To prove this randomized signature signs $(sk_U, s, t, r)$, the user creates the following proof:

$$\mathsf{SPK}^*\{(sk_U, s, t, r) : \mathbf{e}(c', g_2)/\mathbf{e}(a', X) = \mathbf{e}(b', X)^{sk_U} \mathbf{e}(B'_1, X)^s \mathbf{e}(B'_2, X)^t \mathbf{e}(B'_3, X)^r\}.$$

This proof can be created with the TPM using $(*, \pi) \leftarrow \mathsf{Prove}(0, \mathbf{e}(c', g_2)/\mathbf{e}(a', X), n, \rho, X, 1, \perp, \perp, \{(s, \mathbf{e}(B'_1, X), \perp, \perp), (t, \mathbf{e}(B'_2, X), \perp, \perp)\}, \perp, \perp)$. Now, by Lemma 3, a wallet can only be used if the attacker has access to the TPM holding $sk_U$.

## 8 Conclusion

The TPM is a widely deployed security chip that can be embedded in platforms such that the platform can, among other things, anonymously attest to a remote verifier that it is in a secure state. Unfortunately, the current TPM 2.0 specification for DAA contains several flaws: it contains a static DH oracle towards the host and attestations built on top of this interface cannot be proven to be unforgeable. Fixes proposed in the literature are either impossible to implement within the constraints of the TPM, limit the functionality of the TPM interface, or open a subliminal channel that allows a malicious TPM to embed information in attestations, harming the privacy of the host.

We presented a revised TPM 2.0 interface and a Prove protocol for the host that allows the platform to create provably secure signature proofs of knowledge. The interface does not contain a DH oracle, and a corrupt TPM cannot break the zero-knowledge property of the resulting proofs.

Using the Prove protocol, we constructed two provably secure DAA schemes, one based on the LRSW assumption and one on the $q$-SDH assumption, including DAA extensions featuring signature-based revocation and attributes. Furthermore, we have shown that our TPM interface supports DAA schemes with forward anonymity and can be used to protect keys for other cryptographic schemes, such as e-cash and U-Prove. These latter applications were only shown informally, it remains future work to formally treat these applications.

The Trusted Computing Group has already adopted some of our proposed changes and is currently reviewing the remaining ones. It is our aim to bring these improvements to all the existing attestation standards, such as EPID, ISO/IEC 20008-2, and FIDO attestation, such that all implementations are provably secure and can make use of TPMs.

**Acknowledgments.** The first, third, and fourth author have been supported by the European Research Council (ERC) under Grant #321310 (PERCY).

## References


ANZ13. Tolga Acar, Lan Nguyen, and Greg Zaverucha. A TPM Diffie-Hellman oracle. Cryptology ePrint Archive, Report 2013/667, 2013. `http://eprint.iacr.org/2013/667`.

ASM06. Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA. In Roberto De Prisco and Moti Yung, editors, *SCN 06*, volume 4116 of *LNCS*, pages 111–125. Springer, Heidelberg, September 2006.

BB08. Dan Boneh and Xavier Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *Journal of Cryptology*, 21(2):149–177, April 2008.

BCC04. Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick McDaniel, editors, *ACM CCS 04*, pages 132–145. ACM Press, October 2004.

BFG$^+$13. David Bernhard, Georg Fuchsbauer, Essam Ghadafi, Nigel P. Smart, and Bogdan Warinschi. Anonymous attestation with user-controlled linkability. *Int. J. Inf. Sec.*, 12(3):219–249, 2013.

BG04. Daniel R. L. Brown and Robert P. Gallant. The static Diffie-Hellman problem. Cryptology ePrint Archive, Report 2004/306, 2004. `http://eprint.iacr.org/2004/306`.

BL07. Ernie Brickell and Jiangtao Li. Enhanced privacy id: a direct anonymous attestation scheme with enhanced revocation capabilities. In Peng Ning and Ting Yu, editors, *Proceedings of the 2007 ACM Workshop on Privacy in the Electronic Society, WPES 2007, Alexandria, VA, USA, October 29, 2007*, pages 21–30. ACM, 2007.

BL10. Ernie Brickell and Jiangtao Li. A pairing-based DAA scheme further reducing TPM resources. In Alessandro Acquisti, Sean W. Smith, and Ahmad-Reza Sadeghi, editors, *Trust and Trustworthy Computing, Third International Conference, TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings*, volume 6101 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2010.

BL11. Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *IJIPSI*, 1(1):3–33, 2011.

BN06. Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 319–331. Springer, Heidelberg, August 2006.

BR93. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

Can00. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/2000/067`.

Can04. Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.

CDE$^+$. Jan Camenisch, Manu Drijvers, Alec Edgington, Anja Lehmann, Rolf Lindemann, and Rainer Urian. FIDO ECDAA algorithm, implementation draft. `https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-ecdaa-algorithm-v1.1-id-20170202.html`.

CDL16a. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. In Michael Franz and Panos Papadimitratos, editors, *Trust and Trustworthy Computing - 9th International Conference, TRUST 2016, Vienna, Austria, August 29-30, 2016, Proceedings*, volume 9824 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2016.

CDL16b. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. Cryptology ePrint Archive, Report 2016/663, 2016. `http://eprint.iacr.org/2016/663`.

CDL16c. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 234–264. Springer, Heidelberg, March 2016.

CDL17. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation with subverted tpms. Cryptology ePrint Archive, Report 2017/200, 2017. `http://eprint.iacr.org/2017/200`.

Che06. Jung Hee Cheon. Security analysis of the strong Diffie-Hellman problem. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 1–11. Springer, Heidelberg, May / June 2006.

CHL05. Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 302–321. Springer, Heidelberg, May 2005.

CKY09. Jan Camenisch, Aggelos Kiayias, and Moti Yung. On the portability of generalized Schnorr proofs. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 425–442. Springer, Heidelberg, April 2009.

CL04. Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 56–72. Springer, Heidelberg, August 2004.

CL13. Liqun Chen and Jiangtao Li. Flexible and scalable digital signatures in TPM 2.0. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 37–48. ACM Press, November 2013.

CPS10. Liqun Chen, Dan Page, and Nigel P. Smart. On the design and implementation of an efficient DAA scheme. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, volume 6035 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2010.

CS97. Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups (extended abstract). In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 410–424. Springer, Heidelberg, August 1997.

CS03. Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 126–144. Springer, Heidelberg, August 2003.

CU15. Liqun Chen and Rainer Urian. DAA-A: direct anonymous attestation with attributes. In Mauro Conti, Matthias Schunter, and Ioannis G. Askoxylakis, editors, *Trust and Trustworthy Computing - 8th International Conference, TRUST 2015, Heraklion, Greece, August 24-26, 2015, Proceedings*, volume 9229 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2015.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

GPS08. Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008.

Int13. International Organization for Standardization. ISO/IEC 20008-2: Information technology - Security techniques - Anonymous digital signatures - Part 2: Mechanisms using a group public key, 2013.

LRSW99. Anna Lysyanskaya, Ronald L. Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. In Howard M. Heys and Carlisle M. Adams, editors, *SAC 1999*, volume 1758 of *LNCS*, pages 184–199. Springer, Heidelberg, August 1999.

PZ13. Christian Paquin and Greg Zaverucha. U-prove cryptographic specification v1.1 (revision 3), December 2013.

Sch91. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

Sho97. Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.

Tru04. Trusted Computing Group. TPM main specification version 1.2, 2004.

Tru14. Trusted Computing Group. Trusted platform module library specification, family "2.0", 2014.

XYZF14. Li Xi, Kang Yang, Zhenfeng Zhang, and Dengguo Feng. Daa-related apis in TPM 2.0 revisited. In Thorsten Holz and Sotiris Ioannidis, editors, *Trust and Trustworthy Computing - 7th International Conference, TRUST 2014, Heraklion, Crete, Greece, June 30 - July 2, 2014. Proceedings*, volume 8564 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2014.

## A Generalized LRSW assumption in Generic Groups

We prove that this assumption holds in Shoup's generic group model [Sho97].

**Theorem 3.** *Assumption 3 is hard in the generic group model.*

*Proof.* $\mathcal{B}$ maintains three lists of pairs $L_1 = \{(F_{1,i}, \xi_{1,i}) : i = 0, \ldots, \tau_1 - 1\}$, $L_2 = \{(F_{2,i}, \xi_{2,i}) : i = 0, \ldots, \tau_2 - 1\}$, $L_T = \{(F_{T,i}, \xi_{T,i}) : i = 0, \ldots, \tau_T - 1\}$.

It initializes the lists with $F_{1,0} = 1$, $F_{2,0} = 1$, $F_{2,1} = x$, $F_{2,3} = y$.

**Group Operation:** Given two elements $\xi_{G,i}$, $\xi_{G,j}$ with $G \in \{1, 2, T\}$ and $i, j < \tau_G$, and a bit selecting multiplication or division, $\mathcal{B}$ computes $F_{G,\tau_G} \leftarrow F_{G,i} \pm F_{G,j} \in \mathbb{Z}_p[x, y]$, where the operation depends on the operation selection bit. If $F_{G,\tau_G} = F_{G,l}$ for some $l < \tau_G$, set $\xi_{G,\tau_G} \leftarrow \xi_{G,l}$, otherwise set $\xi_{G,\tau_G}$ to a string in $\{0, 1\}^*$ distinct from all previous $\xi_{G,i}$. Add $(F_{G,\tau_G}, \xi_{G,\tau_G})$ to $L_G$ and increment $\tau_G$ by one. Return $\xi_{G,\tau_G}$ to $\mathcal{A}$.

**Pairing:** Given $\xi_{1,i}$ and $\xi_{2,j}$ with $i < \tau_1$ and $j < \tau_2$, set $F_{T,\tau_T} \leftarrow F_{1,i} \cdot F_{2,j}$. If $F_{T,\tau_T} = F_{T,l}$ for some $l < \tau_T$, set $\xi_{T,\tau_T} \leftarrow \xi_{T,l}$, otherwise set $\xi_{T,\tau_T}$ to a string in $\{0, 1\}^*$ distinct from all previous $\xi_{T,i}$.

**Oracle Queries $\mathcal{O}^{\mathsf{a,b}}$:** In the $u$-th query to $\mathcal{O}^{\mathsf{a,b}}$, set $F_{1,\tau_1} \leftarrow r_u$, $F_{1,\tau_1+1} \leftarrow r_u y$. If $F_{1,\tau_1} = F_{1,l}$ for some $l < \tau_1$, set $\xi_{1,\tau_1} \leftarrow \xi_{1,l}$, otherwise set $\xi_{1,\tau_1}$ to a string in $\{0, 1\}^*$ distinct from all previous $\xi_{1,i}$. Set $\xi_{1,\tau_1}$ in the same way. Add $\{(F_{1,\tau}, \xi_{1,\tau}), (F_{1,\tau+1}, \xi_{1,\tau+1})\}$ to $L_1$. Return $(\xi_{1,\tau_1}, \xi_{1,\tau_1+1})$ to $\mathcal{A}$ and increment $\tau_1$ by two.

**Oracle Queries $\mathcal{O}^{\mathsf{c}}$:** On input $(\xi_{1,i}, \xi_{1,j}, m_u)$, where $(\xi_{1,i}, \xi_{1,j})$ is the output of the $u$-th $\mathcal{O}^{\mathsf{a,b}}$ query and this query has not been input to $\mathcal{O}^{\mathsf{c}}$ before, set $F_{1,\tau_1} \leftarrow F_{1,i}x + F_{1,i} \cdot m \cdot x \cdot y$. If $F_{1,\tau_1} = F_{1,l}$ for some $l < \tau_1$, set $\xi_{1,\tau_1} \leftarrow \xi_{1,l}$, otherwise set $\xi_{1,\tau_1}$ to a string in $\{0, 1\}^*$ distinct from all previous $\xi_{1,i}$.

After making oracle queries, $\mathcal{A}$ outputs $(m, \xi_{1,i}, \xi_{1,j}, \xi_{1,k})$, with $i, j, k < \tau_1$. Only now, we take $(x, y, r_1, \ldots, r_q) \xleftarrow{\$} \mathbb{Z}_p^{q+2}$. What remains to show is that $\mathcal{B}$ simulated the operations and oracles correctly. $\mathcal{B}$ returned different values for values with different polynomials, but now we fixed $(x, y, r_1, \ldots, r_q)$, the polynomials might evaluate to the same point, meaning the simulation was incorrect. We recall the Schwarz-Zippel lemma.

**Lemma 4 (Schwarz-Zippel).** *Let $P \in \boldsymbol{F}[x_1, x_2, \ldots, x_n]$ be a non-zero polynomial of total degree $d \geq 0$ over a field $F$. Let $S$ be a finite subset of $F$ and let $r_1, \ldots, r_n$ be selected at random independently and uniformly from $S$. Then,*

$$\Pr\left[P(r_1, r_2, \ldots, r_n) = 0\right] \leq \frac{d}{|S|}.$$

Elements in $\mathbb{G}_1$ have degree at most 3 and elements in $\mathbb{G}_2$ have degree at most 1, so elements in $\mathbb{G}_T$ have degree at most 4. As we have $q_{\mathbb{G}_1} + 3q_{\mathcal{O}} + 1$ elements in $\mathbb{G}_1$, the probability any two of them evaluate to the same point is less than $3(q_{\mathbb{G}_1} + 3q_{\mathcal{O}} + 1)^2/2q$. We have $q_{\mathbb{G}_2} + 2$ elements in $\mathbb{G}_2$, so the probability of an incorrect simulation is less than $3(q_{\mathbb{G}_2} + 2)^2/2q$. In $\mathbb{G}_T$ there are $q_{\mathbb{G}_T}$ elements, so $\mathcal{B}$ simulated incorrectly with probability less than $4(q_{\mathbb{G}_T})^2/2q$. All these probabilities are negligible, meaning $\mathcal{B}$ simulated correctly with overwhelming probability.

The adversary is successful if $m$ was not queried to $\mathcal{O}^{\mathsf{c}}$, $F_{1,i}(x, y, r_1, \ldots, r_q) \neq 0$, $F_{1,i}(x, y, r_1, \ldots, r_q) \cdot y = F_{1,j}(x, y, r_1, \ldots, r_q)$, and $F_{1,i}(x, y, r_1, \ldots, r_q) \cdot (x + xym) = F_{1,k}(x, y, r_1, \ldots, r_q)$. The last two requirements can hold because the polynomials are the same, $yF_{1,i} = F_{1,j}$ and $(x + xym)F_{1,k}$, or the polynomials can differ but they coincidentally evaluate to the same point on the values $(x, y, r_1, \ldots, r_q)$. First, we show that the polynomials cannot be the same. The adversary has only two options to create elements in $\mathbb{G}_1$: using the group operation and using the oracle. This means that any element in $\mathbb{G}_1$ will be a linear combination of the generator and the oracle results. Therefore, we can write every $F_{1,i}$ as a polynomial over $x, y, r_1, \ldots, r_q$:

$$\begin{aligned} F_{1,i} &= \chi_i + \sum_u \left(\alpha_{i,u} r_u + \beta_{i,u} r_u y + \gamma_{i,u}(r_u x + m_u r_u xy)\right) \\ &= \chi_i + \sum_u \left(\alpha_{i,u} r_u + \gamma_{i,u} r_u x + \beta_{i,u} r_u y + \gamma_{i,u} m_u r_u xy\right) \end{aligned}$$

We can write $yF_{1,i}$ as

$$yF_{1,i} = \chi_i y + \sum_u \big(\alpha_{i,u} r_u y + \gamma_{i,u} r_u xy + \beta_{i,u} r_u y^2 + \gamma_{i,u} m_u r_u xy^2\big).$$

Two polynomials are equal if they contain the same monomials. Because $F_{1,j}$ does not contain monomials $y$, $r_u y^2$, or $r_u xy^2$, the fact that $F_{1,i} = F_{1,j}$ implies that these monomials do not occur in $yF_{1,i}$, i.e., $\chi_i = 0$ and for all $u$, $\beta_{i,u} = 0$ and $\gamma_{i,u} m_u = 0$. With this information, we can write $(x + xym)F_{1,i}$ as

$$(x + xym)F_{1,i} = \sum_u \big(\alpha_{i,u} r_u x + \alpha_{i,u} m r_u xy + \gamma_{i,u} r_u x^2 + \gamma_{i,u} m r_u x^2 y\big)$$

For $(x + xym)F_{1,i}$ to be equal to $F_{1,k}$, we must have $\gamma_{i,u} = 0$, so

$$(x + xym)F_{1,i} = \sum_u \big(\alpha_{i,u} r_u x + \alpha_{i,u} m r_u xy\big)$$

For $(x + xym)F_{1,i}$ to be equal to

$$F_{1,k} = \chi_k + \sum_u \big(\alpha_{k,u} r_u + \gamma_{k,u} r_u x + \beta_{k,u} r_u y + \gamma_{k,u} m_u r_u xy\big)$$

we must have $\chi_k = 0$, and for all $u$, $\alpha_{k,u} = 0$, $\beta_{k,u} = 0$, so we can write

$$F_{1,k} = \sum_u \big(\gamma_{k,u} r_u x + \gamma_{k,u} m_u r_u xy\big)$$

So $F_{1,k} = (x + xym)F_{1,i}$ implies $\alpha_{i,u} = \gamma_{k,u}$ and $\alpha_{i,u} m = \gamma_{k,u} m_u$, i.e., $m = m_u$ for all $u$ with $\alpha_{i,u} \neq 0$. The adversary only wins when $a \neq 1_{\mathbb{G}_1}$, so for some $u$ we must have $\alpha_{i,u} \neq 0$, but then $m$ is equal to some queried message.

What remains to show is that the probability that polynomials $yF_{1,i}$ and $F_{1,j}$ are unequal but $(yF_{1,i})(x, y, r_1, \ldots, r_q) = F_{1,j}(x, y, r_1, \ldots, r_q)$ or that $(x + xym)F_{1,i} \neq F_{1,k}$ but $((x + xym)F_{1,i})(x, y, r_1, \ldots, r_q) = F_{1,k}(x, y, r_1, \ldots, r_q)$ is negligible. Polynomial $yF_{1,i}$ has degree 4, so the probability it evaluates to the same value as $F_{1,j}$ is at most $4/q$ by the Schwarz-Zippel lemma. Similarly, $(x + xym)F_{1,i}$ has degree 6, so the probability it evaluates to 0 is at most $6/q$. □

# B Formal Security Model

This section introduces our formal security model of DAA, which is based on the definition by Camenisch et al. [CDL16c, CDL16a, CDL17]. At the end of this section we also compare the captured privacy guarantees in the presence of subverted TPM with the existing privacy notions, and to optimal privacy [CDL17] in particular.

## B.1 Universal Composability

Our security definition has the form of an ideal functionality $\mathcal{F}_{\mathsf{pdaa+}}$ in the Universal Composability (UC) framework [Can00]. In UC, an environment $\mathcal{E}$ gives inputs to the protocol parties and receives their outputs. In the real world, honest parties execute the protocol, over a network controlled by an adversary $\mathcal{A}$, who can also communicate freely with the environment $\mathcal{E}$. In the ideal world, honest parties forward their inputs to the ideal functionality $\mathcal{F}$. The ideal functionality internally performs the defined task and generates outputs for the honest parties. As $\mathcal{F}$ performs the task at hand in an ideal fashion, i.e., $\mathcal{F}$ is secure by construction.

Informally, a protocol $\Pi$ is said to securely realize an ideal functionality $\mathcal{F}$ if the real world is as secure as the ideal world. To prove that statement one has to show that for every adversary $\mathcal{A}$ attacking the real world, there exists an ideal world attacker or simulator $\mathcal{S}$ that performs an equivalent attack on the ideal world. More precisely, $\Pi$ securely realizes $\mathcal{F}$ if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that no environment $\mathcal{E}$ can distinguish the real world (with $\Pi$ and $\mathcal{A}$) from the ideal world (with $\mathcal{F}$ and $\mathcal{S}$).

### B.2 Session Identifiers and Input/Output

In the UC model, different instances of the protocol are distinguished with session identifiers. Here we use session identifiers of the form $sid = (\mathcal{I}, sid')$ for some issuer $\mathcal{I}$ and a unique string $sid'$. To allow several sub-sessions for the join and sign related interfaces we use unique sub-session identifiers *jsid* and *ssid*.

Every party can give different inputs to the protocol. We distinguish these by adding different labels to these inputs, e.g., the host can give an input labeled with $\mathsf{JOIN}$ to request to join, and an input labeled with $\mathsf{SIGN}$ to start signing a message. Outputs are labeled in a similar way.

### B.3 Ideal Functionality $\mathcal{F}_{\mathsf{pdaa+}}$

This section formally introduces our ideal DAA functionality $\mathcal{F}_{\mathsf{pdaa+}}$, which defines DAA with attributes, signature-based revocation, and strong privacy. It is based on $\mathcal{F}_{\mathsf{pdaa}}$ and $\mathcal{F}^{l}_{\mathsf{daa+}}$ by Camenisch et al. [CDL17, CDL16a]. We now give an informal overview of the interfaces of $\mathcal{F}_{\mathsf{pdaa+}}$, and present the full definition in Figure 5.

*Setup.* The $\mathsf{SETUP}$ interface on input $sid = (\mathcal{I}, sid')$ initiates a new session for the issuer $\mathcal{I}$ and expects the adversary to provide algorithms $(\mathsf{ukgen}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify})$ that will be used inside the functionality. $\mathsf{ukgen}$ creates a new key $gsk$ and a tracing trapdoor $\tau$ that allows $\mathcal{F}_{\mathsf{pdaa+}}$ to trace signatures generated with $gsk$. $\mathsf{sig}$, $\mathsf{ver}$, and $\mathsf{link}$ are used by $\mathcal{F}_{\mathsf{pdaa+}}$ to create, verify, and link signatures, respectively. Finally, $\mathsf{identify}$ allows to verify whether a signature belongs to a certain tracing trapdoor. This allows $\mathcal{F}_{\mathsf{pdaa+}}$ to perform multiple consistency checks and enforce the desired non-frameability and unforgeability properties.

Note that the $\mathsf{ver}$ and $\mathsf{link}$ algorithms assist the functionality only for signatures that are not generated by $\mathcal{F}_{\mathsf{pdaa+}}$ itself. For signatures generated by the functionality, $\mathcal{F}_{\mathsf{pdaa+}}$ will enforce correct verification and linkage using its internal records. While $\mathsf{ukgen}$ and $\mathsf{sig}$ are probabilistic algorithms, the other ones are required to be deterministic. The $\mathsf{link}$ algorithm also has to be symmetric, i.e., for all inputs it must hold that $\mathsf{link}(\sigma, m, \sigma', m', bsn) \leftrightarrow \mathsf{link}(\sigma', m', \sigma, m, bsn)$.

*Join.* A host $\mathcal{H}_j$ can request to join with a TPM $\mathcal{M}_i$ using the $\mathsf{JOIN}$ interface. The issuer is asked to approve the join request, and choose the platform's attributes. $\mathcal{F}_{\mathsf{pdaa+}}$ is parametrized by $L$ and $\{\mathbb{A}_i\}_{0<i\leq L}$, that offer support for attributes. $L$ is the amount of attributes every credential contains and $\mathbb{A}_i$ the set from which the $i$-th attribute is taken. When the issuer approves with attributes $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$, the functionality stores an internal membership record for $\mathcal{M}_i, \mathcal{H}_j, attrs$ in `Members` indicating that from now on that platform is allowed to create attestations.

If the host is corrupt, the adversary must provide $\mathcal{F}_{\mathsf{pdaa+}}$ with a tracing trapdoor $\tau$. This value is stored along in the membership record and allows the functionality to check via the $\mathsf{identify}$ function whether signatures were created by this platform. $\mathcal{F}_{\mathsf{pdaa+}}$ uses these checks to ensure non-frameability and unforgeability whenever it creates or verifies signatures. To ensure that the adversary cannot provide bad trapdoors that would break the completeness or non-frameability properties, $\mathcal{F}_{\mathsf{pdaa+}}$ checks the legitimacy of $\tau$ via the "macro" function $\mathsf{CheckTtdCorrupt}$. This function checks that for all previously generated or verified signatures for which $\mathcal{F}_{\mathsf{pdaa+}}$ has already seen another matching tracing trapdoor $\tau' \neq \tau$, the new trapdoor $\tau$ is not identified as a matching key as well. $\mathsf{CheckTtdCorrupt}$ is defined as follows:

$$
\begin{aligned}
\mathsf{CheckTtdCorrupt}(\tau) = \nexists(\sigma, m, bsn) :& \\
&\Big(\big(\langle \sigma, m, bsn, *, * \rangle \in \mathtt{Signed} \vee \langle \sigma, m, bsn, *, 1 \rangle \in \mathtt{VerResults}\big) \wedge \\
\exists \tau' : \Big(\tau \neq \tau' \;\wedge\; &\big(\langle *, *, \tau' \rangle \in \mathtt{Members} \vee \langle *, *, *, *, \tau' \rangle \in \mathtt{DomainKeys}\big) \;\wedge \\
&\qquad \mathsf{identify}(\sigma, m, bsn, \tau) = \mathsf{identify}(\sigma, m, bsn, \tau') = 1\Big)\Big)
\end{aligned}
$$

*Sign.* After joining, a host $\mathcal{H}_j$ can use the $\mathsf{SIGN}$ interface to request a signature on a message $m$ with respect to basename $bsn$ while proving a certain predicate $p$ holds for his attributes and proving that he is not revoked by signature revocation list $\mathtt{SRL}$. The signature will only be created when the TPM $\mathcal{M}_i$ explicitly agrees to signing $m$, a join record for $\mathcal{M}_i, \mathcal{H}_j, attrs$ in $\mathtt{Members}$ exists such that $attrs$ satisfy $p$ (if the issuer is honest), and the platform is not revoked by $\mathtt{SRL}$.

When a platform wants to sign message $m$ w.r.t. a fresh basename $bsn$, $\mathcal{F}_{\mathsf{pdaa+}}$ generates a new key $gsk$ (and tracing trapdoor $\tau$) via $\mathsf{ukgen}$ and then signs $m$ with that key. The functionality also stores the fresh key $(gsk, \tau)$ together with $bsn$ in $\mathtt{DomainKeys}$, and reuses the same key when the platform wishes to sign repeatedly under the same basename. Using fresh keys for every signature naturally enforces the desired privacy guarantees: the signature algorithm does not receive any identifying information as input, and thus the created signatures are guaranteed to be anonymous (or pseudonymous in case $bsn$ is reused).

To guarantee non-frameability and completeness, our functionality further checks that every freshly generated key, tracing trapdoor and signature does not falsely match with any existing signature or key. More precisely, $\mathcal{F}_{\mathsf{pdaa+}}$ first uses the $\mathsf{CheckTtdHonest}$ macro to verify whether the new key does not match to any existing signature. $\mathsf{CheckTtdHonest}$ is defined as follows:

$$\begin{aligned}\mathsf{CheckTtdHonest}(\tau) = \\ &\forall \langle \sigma, m, bsn, \mathcal{M}, \mathcal{H} \rangle \in \mathtt{Signed} : \mathsf{identify}(\sigma, m, bsn, \tau) = 0 \wedge \\ &\forall \langle \sigma, m, bsn, *, 1 \rangle \in \mathtt{VerResults} : \mathsf{identify}(\sigma, m, bsn, \tau) = 0\end{aligned}$$

Likewise, before outputting $\sigma$, the functionality checks that no one else already has a key which would match this newly generated signature.

Finally, for ensuring unforgeability, the signed message, basename, attribute predicate, signature revocation list, and platform identity are stored in $\mathtt{Signed}$, which will be used when verifying signatures.

*Verify.* Signatures can be verified by any party using the $\mathsf{VERIFY}$ interface. $\mathcal{F}_{\mathsf{pdaa+}}$ uses its internal $\mathtt{Signed}$, $\mathtt{Members}$, and $\mathtt{DomainKeys}$ records to enforce unforgeability and non-frameability. It uses the tracing trapdoors $\tau$ stored in $\mathtt{Members}$ and $\mathtt{DomainKeys}$ to find out which platform created this signature. If no match is found and the issuer is honest, the signature is a forgery and rejected by $\mathcal{F}_{\mathsf{pdaa+}}$. If the signature to be verified matches the tracing trapdoor of some platform with an honest host, but the signing records do not show that they signed this message w.r.t. the basename, attribute predicate, and signature revocation list, $\mathcal{F}_{\mathsf{pdaa+}}$ again considers this to be a forgery and rejects. If the platform has an honest TPM, only checks on the message and basename are made. If the records do not reveal any issues with the signature, $\mathcal{F}_{\mathsf{pdaa+}}$ uses the $\mathsf{ver}$ algorithm to obtain the final result.

The verify interface also supports verifier-local revocation. The verifier can input a revocation list $\mathtt{RL}$ containing tracing trapdoors, and signatures matching any of those trapdoors are no longer accepted.

*Link.* Using the $\mathsf{LINK}$ interface, any party can check whether two signatures $(\sigma, \sigma')$ on messages $(m, m')$ respectively, generated with the same basename $bsn$ originate from the same platform or not. $\mathcal{F}_{\mathsf{pdaa+}}$ again uses the tracing trapdoors $\tau$ stored in $\mathtt{Members}$ and $\mathtt{DomainKeys}$ to check which platforms created the two signatures. If they are the same, $\mathcal{F}_{\mathsf{pdaa+}}$ outputs that they are linked. If it finds a platform that signed one, but not the other, it outputs that they are unlinked, which prevents framing of platforms with an honest host.

*Conventions.* The full definition of $\mathcal{F}_{\mathsf{pdaa+}}$ is presented in Figure 5. We use a number of conventions to simplify the definition of $\mathcal{F}_{\mathsf{pdaa+}}$. First, we require that $\mathsf{identify}(\sigma, m, bsn, \tau) = 0$ if $\sigma$ or $\tau$ is $\perp$. Second, whenever we need approval from the adversary to proceed, $\mathcal{F}_{\mathsf{pdaa+}}$ sends an output to the adversary and waits for a response. This means that in that join or sign session, no other inputs are accepted except the expected response from the adversary. Third, if any check that $\mathcal{F}_{\mathsf{pdaa+}}$ makes fails, the sub-session is invalidated and $\perp$ is output to the caller.

1. $\texttt{Issuer Setup.}$ On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
  – Verify that $sid = (\mathcal{I}, sid')$.
  – Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
  – Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
  – Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

---

**Join**

2. $\texttt{Join Request.}$ On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
  – Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
  – Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \bot, status\rangle$ with $status \leftarrow delivered$.
  – Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \texttt{Members}$ already exists.
  – Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ $\texttt{Join Proceed.}$ On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
  – Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
  – Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
  – If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
  – Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
  – Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into $\texttt{Members}$ and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

---

**Sign**

4. $\texttt{Sign Request.}$ On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
  – If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\texttt{Members}$, abort.
  – If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \bot$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
    • Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \texttt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\texttt{DomainKeys}$.
    • Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
    • Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor $\tau'$ registered in $\texttt{Members}$ or $\texttt{DomainKeys}$ with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
  – Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
  – Output $(\mathsf{SIGNPROCEED}, sid, ssid, m)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
5. $\texttt{Sign Proceed.}$ On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
  – Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
  – If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\texttt{Members}$.
  – For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}'_i, \mathcal{H}'_j)$ from $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i, *\rangle \in \texttt{Members}$ and $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i\rangle \in \texttt{DomainKeys}$ where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
    • Check that there are no two distinct $\tau$ values matching $\sigma'$.
    • Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
  – Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in $\texttt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

---

**Verify & Link**

6. $\texttt{Verify.}$ On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
  – Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i, *\rangle \in \texttt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in \texttt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
    • More than one $\tau_i$ was found.
    • $\mathcal{I}$ is honest and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found for which an entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle \in \texttt{Members}$ exists with $p(attrs) = 1$.
    • $\mathcal{M}_i$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, *, *\rangle \in \texttt{Signed}$ exists.
    • $\mathcal{H}_j$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle \in \texttt{Signed}$ exists.
    • There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$ and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ for an honest $\mathcal{H}_j$ was found.
    • For some matching $\tau_i$ and $(\sigma', m', bsn') \in \mathtt{SRL}$, $\mathsf{identify}(\sigma', m', bsn', \tau_i) = 1$.
  – If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
  – Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to $\texttt{VerResults}$ and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. $\texttt{Link.}$ On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
  – Output $\bot$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the $\mathsf{VERIFY}$ interface with $\mathtt{RL} = \emptyset$).
  – For each $\tau_i$ in $\texttt{Members}$ and $\texttt{DomainKeys}$ compute $b_i \leftarrow \mathsf{identify}(\sigma, m, bsn, \tau_i)$ and $b'_i \leftarrow \mathsf{identify}(\sigma', m', bsn, \tau_i)$ and do the following:
    • Set $f \leftarrow 0$ if $b_i \neq b'_i$ for some $i$.
    • Set $f \leftarrow 1$ if $b_i = b'_i = 1$ for some $i$.
  – If $f$ is not defined yet, set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
  – Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 5.** Our ideal DAA functionality with strong privacy $\mathcal{F}_{\mathsf{pdaa+}}$

### B.4 Comparison of $\mathcal{F}_{\mathsf{pdaa+}}$ with Previous Definitions

Our functionality $\mathcal{F}_{\mathsf{pdaa+}}$ is based on previous UC-based DAA functionalities $\mathcal{F}^{l}_{\mathsf{daa}}$ [CDL16c], $\mathcal{F}^{l}_{\mathsf{daa+}}$ [CDL16a] which extends $\mathcal{F}^{l}_{\mathsf{daa}}$ with attributes and signature-based revocation, and $\mathcal{F}_{\mathsf{pdaa}}$ [CDL17], which strengthens the privacy guarantees of $\mathcal{F}^{l}_{\mathsf{daa}}$. We now show how our functionality compares to these other DAA functionalities.

*Attributes and Signature-based Revocation.* Our functionality $\mathcal{F}_{\mathsf{pdaa+}}$ supports adding attributes to the membership credentials, and selectively disclosing attributes when signing, as well as signature-based revocation. $\mathcal{F}_{\mathsf{pdaa+}}$ can be seen as $\mathcal{F}_{\mathsf{pdaa}}$ extended with attributes and signature based revocations, in the same way that $\mathcal{F}^{l}_{\mathsf{daa+}}$ adds these features to $\mathcal{F}^{l}_{\mathsf{daa}}$.

*Realistic TPM Interfaces.* Contrary to the approach of $\mathcal{F}^{l}_{\mathsf{daa+}}$, in our definition $\mathcal{F}_{\mathsf{pdaa+}}$ the TPM is agnostic of attributes, predicates or the $\mathtt{SRL}$. That is, when signing it neither explicitly sees or approves the attributes or $\mathtt{SRL}$. This reflects that the actual TPM interfaces do not provide any such outputs or approvals either, and in fact, there is no practical reason to do so and would only make the TPM interfaces more complicated. Thus, we opted for adapting the functionality accordingly.

Similarly, the previous UC-based definitions [CDL16c, CDL16a, CDL17] let the TPM approve both the message and basename for which the hosts requests as signature. In this definition, the TPM is only responsible for approving the message being signed, but does no longer receive (and approve) the basename. Again, this is done to better capture the actual TPM interfaces that provide such checks only for the message.

The resulting unforgeability and non-frameability guarantees are as follows. No adversary can sign a message $m$ w.r.t. basename $bsn$, attribute predicate $p$, and signature revocation list $\mathtt{SRL}$, if the host did not sign exactly that. If the TPM is honest but the host is corrupt, the unforgeability is a bit weaker, as the TPM only checks the message. Therefore, if the TPM signed message $m$, the adversary is allowed to create signatures on $m$ w.r.t. any $p$ and $\mathtt{SRL}$ that hold for the platform (i.e., the platform has the attributes to fulfill $p$ and is not revoked by $\mathtt{SRL}$). The TPM does not explicitly approve $bsn$, but we force the (possibly corrupt) host to choose one $bsn$ when signing, and signatures can only be valid if the message-basename combination was signed. Because the TPM does not explicitly approve the basename, our unforgeability with an honest TPM and corrupt host is slightly weaker than previous UC-based definitions [CDL16c, CDL16a, CDL17] where the TPM must explicitly approve the basename.

When the host is honest but the TPM is corrupt, our definition also assures unforgeability and non-frameability like $\mathcal{F}_{\mathsf{pdaa}}$, which provides stronger guarantees than [CDL16c] and [CDL16a], where both properties are not ensured when the TPM is corrupt.

*Strong Privacy (vs. Optimal Privacy).* Previous DAA schemes and definitions condition their privacy property on the honesty of the entire platform, i.e., as soon as either the TPM or host is corrupt, no privacy is guaranteed anymore. Whereas the honesty of the host is indeed necessary (a corrupt host can always break privacy by outputting identifying information), relying on the honesty of the TPM as well is an unnecessarily strong assumption. In fact, it even contradicts the original goal of DAA, namely to provide anonymous attestations without having to trust the hardware. This mismatch was recently discussed by Camenisch et al. [CDL17] who propose the notion of DAA with *optimal* privacy which must hold even in the presence of corrupted or subverted TPMs. In contrast to $\mathcal{F}^{l}_{\mathsf{daa}}$ and $\mathcal{F}^{l}_{\mathsf{daa+}}$ where the adversary provides the signature whenever the host or TPM are corrupt, the functionality with optimal privacy $\mathcal{F}_{\mathsf{pdaa}}$ outputs anonymous signatures as long as the host is honest. As the signatures are given directly to the host, the adversary learns nothing about them, even if the TPM is corrupt.

Unfortunately, the authors also show that optimal privacy cannot be achieved using constructions where the TPM and host together create a Fiat-Shamir proof of knowledge, which rules out the most efficient DAA schemes. The DAA protocol with optimal privacy proposed in [CDL17] comes with a significant re-design, shifting most of the computations from the TPM to the host and would also require new operations to be implemented on the TPM.

The goal of this work is to obtain the best privacy properties with as minimal changes to the existing TPM and DAA specifications as possible. We therefore relax their notion of optimal privacy, and show how

| corrupt TPM | $\mathcal{F}^{l}_{\mathsf{daa}}$, $\mathcal{F}^{l}_{\mathsf{daa+}}$ | $\mathcal{F}_{\mathsf{pdaa+}}$ (this work) | $\mathcal{F}_{\mathsf{pdaa}}$ |
|---|---|---|---|
| *standard* | - | - | + |
| *isolated* | - | + | ++ |

**Fig. 6.** Comparison of privacy guarantees for an honest host in the presence of a corrupt TPM (either corrupt in the standard UC or isolated model of [CDL17]).

this can be achieved with modest modifications to the current DAA specifications and using our proposed TPM interfaces. Roughly, our new privacy notion – which we term *strong privacy* – allows the TPM to see the anonymous signature that is generated by the functionality and consequently also condition its behavior on the signature value. Thus, while the actual signature shown to the TPM is still guaranteed to be anonymous, the TPM can influence the final distribution of the signatures by blocking certain signature values (a signature is only output to the host when the TPM explicitly approved it). A TPM performing such a "blocking attack" to alter the signature distribution can clearly be noticed by the host though, and thus, this attack has rather limited impact in practice.

The main reason why exposing the signature value to the TPM reduces the privacy guarantees stems from the way UC models corruption: In the standard UC corruption model, the adversary is allowed to see all inputs to the party he corrupts. That is, he will see the signatures given for approval to the TPM and can later re-identify the platform from the signature. However, as Camenisch et al. [CDL17] argue, in case of the TPM this standard UC corruption model gives the adversary much more power than in reality. In the real world, the TPM is embedded inside a host who controls all communication with the outside world, i.e., the adversary cannot communicate directly with the TPM but only via the (honest) host. To model such subversion more accurately, [CDL17] introduces *isolated corruptions*, where the adversary can specify the code that the isolated, yet subverted TPM will run, but cannot directly interact with the isolated TPM.

Applying this concept of isolated corruptions to our notion of strong privacy then yields significantly stronger privacy guarantees than with the standard corruption model: In signing the adversary no longer sees the signature which is only given to the isolated corrupt TPM. That is, when considering isolated TPM corruptions, the only difference to the optimal privacy notion of [CDL17] is the aforementioned "blocking attack" which allows a corrupt TPM to influence the signature distribution, but with the risk of being caught by the host. Thus, w.r.t. isolated corruption, our notion of strong privacy is almost equivalent to optimal privacy, yet allows for significantly more efficient instantiation. An overview of the different privacy guarantees of this and the previous works is given in Figure 6.

# C Detailed Description of our DAA Protocols

We now formallly introduce our DAA protocols $\Pi_{\mathsf{qSDH-DAA}}$ and $\Pi_{\mathsf{LRSW-DAA}}$ that we will prove secure. A couple of additions to our generic scheme as presented in Figure 4 are made. First, we explicitly define the inputs and outputs of the protocol. In the `TPM.Hash` command, the TPM must decide whether it considers a message "safe to sign". This is now an explicit output to the environment. Second, we add session identifiers to the TPM's inputs. These session identifiers are required for universal composability and strengthen our security result by guaranteeing that our security is preserved by protocol composition. However, if one is only concerned with standalone security, the session identifiers can be ommitted. We let parties implicitly query $\mathcal{F}_{\mathsf{ca}}$ [Can04] when they need the issuer public key. To model the authentication of the TPM towards the issuer, we use $\mathcal{F}_{\mathsf{auth*}}$ as defined by Camenisch et al. [CDL16c].

The modified TPM interface is depicted in Figure 7. The rest of the protocols are defined as follows, where certain parts of the protocol differ between $\Pi_{\mathsf{qSDH-DAA}}$ and $\Pi_{\mathsf{LRSW-DAA}}$.

## C.1 Setup

1. On input $(\mathsf{SETUP}, sid)$, the issuer generates its keys as described in sections 5.2 and 5.2, and registers the key with $\mathcal{F}_{\mathsf{ca}}$. Output $(\mathsf{SETUPDONE}, sid)$.

Session system parameters: $\mathbb{G}_1 = \langle \bar{g} \rangle$ of prime order $q$, nonce bit length $l_n$, random oracles $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_p$ and $\mathsf{H}_{\mathbb{G}_1} : \{0,1\}^* \to \mathbb{G}_1$. Initialize $\mathtt{Committed} \leftarrow \emptyset$ and $\mathit{commitId} \leftarrow 0$. Let TPM $\mathcal{M}_i$ be embedded in host $\mathcal{H}_j$.

---

**Init.** On input $\mathtt{TPM.Create}(\mathit{sid}, \mathit{jsid})$:
- If this is the first invocation of $\mathtt{TPM.Create}$, choose a fresh secret key $\mathit{tsk} \xleftarrow{\$} \mathbb{Z}_p$ and compute public key $\mathit{tpk} \leftarrow \bar{g}^{\mathit{tsk}}$.
- Parse $\mathit{sid}$ as $(\mathcal{I}, \mathit{sid}')$, store $\mathit{tsk}$, and send $\mathit{tpk}$ to issuer $\mathcal{I}$ over $\mathcal{F}_{\mathsf{auth}*}$.

**Hash.** On input $\mathtt{TPM.Hash}(\mathit{sid}, \mathit{ssid}, m_t, m_h)$:
- If $m_t \neq \perp$, output $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid}, m_t)$.
- On input $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid})$, compute $c \leftarrow \mathsf{H}(\text{“}\mathit{TPM}\text{”}, m_t, m_h)$.
- Mark $c$ as “safe to sign” and output $c$.

**Commit.** On input $\mathtt{TPM.Commit}(\mathit{sid}, \mathit{ssid}, \mathit{bsn}_E, \mathit{bsn}_L)$:
- If $\mathit{bsn}_E \neq \perp$, set $\tilde{g} \leftarrow \mathsf{H}_{\mathbb{G}_1}(\mathit{bsn}_E)$, otherwise set $\tilde{g} \leftarrow \bar{g}$.
- Choose $r \xleftarrow{\$} \mathbb{Z}_p$, $n_t \xleftarrow{\$} \{0,1\}^{l_n}$ and store $(\mathit{sid}, \mathit{ssid}, \mathit{commitId}, r, n_t)$ in $\mathtt{Committed}$.
- Set $\bar{n}_t \leftarrow \mathsf{H}(\text{“}\mathit{nonce}\text{”}, n_t)$, $E \leftarrow \tilde{g}^r$, and $K, L \leftarrow \perp$.
- If $\mathit{bsn}_L \neq \perp$, set $j \leftarrow \mathsf{H}_{\mathbb{G}_1}(\mathit{bsn}_L)$, $K \leftarrow j^{\mathit{tsk}}$ and $L \leftarrow j^r$.
- Output $(\mathit{commitId}, \bar{n}_t, E, K, L)$ and increment $\mathit{commitId}$.

**Sign.** On input $\mathtt{TPM.Sign}(\mathit{sid}, \mathit{ssid}, \mathit{commitId}, c, n_h)$:
- Retrieve record $(\mathit{sid}, \mathit{ssid}, \mathit{commitId}, r, n_t)$ and remove it from $\mathtt{Committed}$, output an error if no such record was found.
- If $c$ is safe to sign, set $c' \leftarrow \mathsf{H}(\text{“}\mathit{FS}\text{”}, n_t \oplus n_h, c)$ and $s \leftarrow r + c' \cdot \mathit{tsk}$ and output $(n_t, s)$.

**Fig. 7.** Our proposed modified TPM 2.0 interface with the required UC session identifiers and inputs/outputs.

### C.2 Join

1. $\mathcal{H}_j$ on input $(\mathsf{JOIN}, \mathit{sid}, \mathit{jsid}, \mathcal{M}_i)$ performs the following tasks:
   - It sends $(\mathit{sid}, \mathit{jsid}, \mathsf{JOIN})$ to the issuer.
   - When it receives $(\mathit{sid}, \mathit{jsid}, n)$ from $\mathcal{I}$, it calls $\mathtt{TPM.Create}(\mathit{sid})$ to receive $\mathit{tpk}$. It creates proof $\pi_{\mathit{tpk}} \leftarrow \mathsf{SPK}^*\{\mathit{tsk} : \mathit{tpk} = \bar{g}^{\mathit{tsk}} \;\wedge\; \mathit{tpk}' = \tilde{g}^{\mathit{tsk}}\}(\text{“}\mathit{join}\text{”}, n)$ using the $\mathsf{Prove}$ protocol. The issuer must receive $\mathit{tpk}$ in an authenticated manner, which can be realized in multiple ways. We model this as the TPM using $\mathcal{F}_{\mathsf{auth}*}$ to send $\mathit{tpk}$ to the issuer via the host.
   - The host notices $\mathcal{M}_i$ sending $\mathit{tpk}$ via $\mathcal{F}_{\mathsf{auth}*}$ to $\mathcal{I}$. It takes $\mathit{hsk} \xleftarrow{\$} \mathbb{Z}_p$, sets $\mathit{gpk} \leftarrow \mathit{tpk}' \cdot \tilde{g}^{\mathit{hsk}}$, and proves $\pi_{\mathit{gpk}} \leftarrow \mathsf{NIZK}\{\mathit{hsk} : \mathit{gpk}/\mathit{tpk}' = \tilde{g}^{\mathit{hsk}}\}(\text{“}\mathit{join}\text{”}, n)$.
   - The host appends $\mathit{tpk}', \mathit{gpk}, \pi_{\mathit{tpk}}, \pi_{\mathit{gpk}}$ to the message $\mathit{tpk}$ being sent to the issuer over $\mathcal{F}_{\mathsf{auth}*}$.
   - The issuer upon receiving $\mathit{tpk}, \mathit{tpk}', \mathit{gpk}, \pi_{\mathit{tpk}}, \pi_{\mathit{gpk}}$ from $\mathcal{F}_{\mathsf{auth}*}$, where $\mathit{tpk}$ is authenticated by TPM $\mathcal{M}_i$, verifies proofs $\pi_{\mathit{tpk}}$ and $\pi_{\mathit{gpk}}$ and outputs $(\mathsf{JOINPROCEED}, \mathit{sid}, \mathit{jsid}, \mathcal{M}_i)$.
2. $\mathcal{I}$ on input $(\mathsf{JOINPROCEED}, \mathit{sid}, \mathit{jsid}, \mathit{attrs})$:
   - $\mathcal{I}$ creates credential $\mathit{cred} \leftarrow \mathsf{PBSign}(\mathit{isk}, (\mathit{gpk}, \mathit{attrs}))$, where the instantiation of $\mathsf{PBSign}$ differs between $\Pi_{\mathsf{qSDH-DAA}}$ and $\Pi_{\mathsf{LRSW-DAA}}$, and sends $(\mathit{sid}, \mathit{jsid}, \mathit{cred}, \mathit{attrs})$ to $\mathcal{H}_j$.
   - $\mathcal{H}_j$, upon receiving $\mathit{cred}$ and $\mathit{attrs}$, verifies $\mathit{cred}$ w.r.t. $\mathit{gpk}$, $\mathit{attrs}$, and $\mathit{ipk}$.
   - It stores that it joined with $\mathcal{M}_i$, stores $(\mathit{hsk}, \mathit{cred}, \mathit{attrs})$, and outputs $(\mathsf{JOINED}, \mathit{sid}, \mathit{jsid}, \mathit{attrs})$.

### C.3 Sign

1. When a host $\mathcal{H}_j$ receives input $(\mathsf{SIGN}, \mathit{sid}, \mathit{ssid}, \mathcal{M}_i, m, \mathit{bsn}, p, \mathtt{SRL})$:
   - The host checks that it joined with $\mathcal{M}_i$. If so, it looks up $(\mathit{hsk}, \mathit{cred}, \mathit{attrs})$ from the join protocol and verifies that $\mathit{attrs}$ fulfill predicate $(D, I)$, i.e., it parses $I$ as $(a'_1, \ldots, a'_L)$ and $\mathit{attrs}$ as $(a_1, \ldots, a_L)$ and checks that $a_i = a'_i$ for every $i \in D$.
   - The host and TPM jointly generate the pseudonym $\mathit{nym} \leftarrow \mathsf{H}_{\mathbb{G}}(1||\mathit{bsn})^{\mathit{gsk}}$ and proof $\pi_{\mathit{cred}}$ of a membership credential on $\mathit{gsk} = \mathit{tsk} + \mathit{hsk}$ and $\mathit{attrs}$ using the $\mathsf{Prove}$ protocol.
   - For each tuple $(\mathit{bsn}_i, \mathit{nym}_i) \in \mathtt{SRL}$, the host and TPM jointly create non-revocation proofs $\pi_{\mathtt{SRL},i}$:
     $$\pi_{\mathtt{SRL},i} \leftarrow \mathsf{SPK}^*\{\mathit{gsk} : \mathsf{H}_{\mathbb{G}_1}(1||\mathit{bsn}_i)^{\mathit{gsk}} \neq \mathit{nym}_i \;\wedge\; \mathit{nym} = \mathsf{H}_{\mathbb{G}_1}(1||\mathit{bsn})^{\mathit{gsk}}\}(\text{“}\mathit{sign}\text{”}).$$
     If a non-revocation proof fails, the host aborts.
   - The host sets $\sigma \leftarrow (\mathit{nym}, \pi_{\mathit{cred}}, \{\pi_{\mathtt{SRL},i}\})$ and outputs $(\mathsf{SIGNATURE}, \mathit{sid}, \mathit{ssid}, \sigma)$.

### C.4 Verify

1. A party $\mathcal{V}$ upon input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$:
   – Parse $\sigma = (nym, \pi_{cred}, \{\pi_{\mathtt{SRL},i}\})$.
   – Verify $\pi_{cred}, \{\pi_{\mathtt{SRL},i}\}$ w.r.t. $ipk, m, bsn, (D, I), \mathtt{SRL}$.
   – For every $gsk_i \in \mathtt{RL}$, check that $\mathsf{H}_{\mathbb{G}_1}(bsn)^{gsk_i} \neq nym$.
   – Set $f = 1$ if all proofs are correct, and $f = 0$ otherwise. Output $(\mathsf{VERIFIED}, sid, f)$.

### C.5 Link

1. A party $\mathcal{V}$ upon input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$:
   – Check $\mathsf{VERIFY}(ipk, \sigma, m, bsn, (D, I), \mathtt{RL}, \mathtt{SRL}) = 1$ and $\mathsf{VERIFY}(ipk, \sigma', m', bsn, (D', I'), \mathtt{RL}', \mathtt{SRL}') = 1$. If either does not hold, output $\perp$.
   – If both signatures are valid, parse $\sigma = (nym, \pi_{cred}, \{\pi_{\mathtt{SRL},i}\})$ and $\sigma' = (nym', \pi'_{cred}, \{\pi'_{\mathtt{SRL},i}\})$. If $nym = nym'$, set $f = 1$, otherwise, set $f = 0$.
   – Output $(\mathsf{LINK}, sid, f)$.

## D Security of our DAA Schemes

**Theorem 1.** *Protocol $\Pi_{\mathsf{LRSW-DAA}}$ as defined in Section C securely realizes $\mathcal{F}_{\mathsf{pdaa+}}$ (without support for attributes, i.e., $L = 0$) under the Generalized LRSW and Decisional Diffie-Hellman assumptions in the random oracle model.*

**Theorem 2.** *Protocol $\Pi_{\mathsf{qSDH-DAA}}$ as defined in Section C securely realizes $\mathcal{F}_{\mathsf{pdaa+}}$ (for any amount of attributes $L$, $\mathbb{A}_i = \mathbb{Z}_p$, and selective disclosure as attribute predicates $\mathbb{P}$) under the q-Strong Diffie-Hellman and Decisional Diffie-Hellman assumptions in the random oracle model.*

We have to prove that our scheme realizes $\mathcal{F}_{\mathsf{pdaa+}}$, which means proving that for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every environment $\mathcal{E}$ we have $\mathrm{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} \approx \mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{E}}$.

To show that no environment $\mathcal{E}$ can distinguish the real world, in which it is working with our DAA protocols and adversary $\mathcal{A}$, from the ideal world, in which it uses $\mathcal{F}_{\mathsf{pdaa+}}$ with simulator $\mathcal{S}$, we use a sequence of games. We start with the real world protocol execution. In the next game we construct one entity $\mathcal{C}$ that runs the real world protocol for all honest parties. Then we split $\mathcal{C}$ into two pieces, a functionality $\mathcal{F}$ and a simulator $\mathcal{S}$, where $\mathcal{F}$ receives all inputs from honest parties and sends the outputs to honest parties. We start with a dummy functionality, and gradually change $\mathcal{F}$ and update $\mathcal{S}$ accordingly, to end up with the full $\mathcal{F}_{\mathsf{pdaa+}}$ and a satisfying simulator. First we define all intermediate functionalities and simulators, and then we prove that they are all indistinguishable from each other.

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SETUP}, sid), \mathcal{I})$ to $\mathcal{S}$.

---

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid, attrs), \mathcal{I})$ to $\mathcal{S}$.

---

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}), \mathcal{H}_j)$ to $\mathcal{S}$.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

---

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL}), \mathcal{V})$ to $\mathcal{S}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn), \mathcal{V})$ to $\mathcal{S}$.

**Fig. 8.** $\mathcal{F}$ for GAME 3

When a simulated party "$\mathcal{P}$" outputs $m$ and no specific action is defined, send $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$.
**Forwarded Input**

– On input $(\mathsf{FORWARD}, m, \mathcal{P})$.
  • Give "$\mathcal{P}$" input $m$.

**Fig. 9.** Simulator for Game 3

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid, attrs), \mathcal{I})$ to $\mathcal{S}$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}), \mathcal{H}_j)$ to $\mathcal{S}$.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL}), \mathcal{V})$ to $\mathcal{S}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn), \mathcal{V})$ to $\mathcal{S}$.

**Fig. 10.** $\mathcal{F}$ for Game 4

When a simulated party "$\mathcal{P}$" outputs $m$ and no specific action is defined, send $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$.

**Setup**

Honest $\mathcal{I}$

- On input $(\mathsf{SETUP}, sid)$ from $\mathcal{F}$.
  - Parse $sid$ as $\mathcal{I}, sid'$.
  - Give "$\mathcal{I}$" input $(\mathsf{SETUP}, sid)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{SETUPDONE}, sid)$, $\mathcal{S}$ takes the issuer key pair. Note that the simulator also knows the issuer secret key, as it is simulating "$\mathcal{I}$".
  - Define $\mathsf{sig}(gsk, m, bsn, p, \mathsf{SRL})$ as follows: First, create a BBS+ or CL signature (depending on the instantiation) using the issuer key on $gsk$ and attribute values where the disclosed attributes are taken from $p$ and the undisclosed attributes are set to dummy values. Next, the algorithm performs the real world signing algorithm (performing both the tasks from the host and the TPM).
  - Define $\mathsf{ver}(\sigma, m, bsn, p, \mathsf{SRL})$ as the real world verification algorithm, except that the private-key revocation check is ommitted.
  - Define $\mathsf{link}(\sigma, m, \sigma', m', bsn)$ as follows: Parse the signatures as $(A', nym, \pi, \{\pi_i\}) \leftarrow \sigma$, $(A'', nym', \pi', \{\pi'_i\}) \leftarrow \sigma'$, and output 1 iff $nym = nym'$.
  - Define $\mathsf{identify}(\sigma, m, bsn, \tau)$ as follows: parse $\sigma$ as $(nym, \pi_{cred}, \{\pi_{\mathsf{SRL},i}\})$ and check $\tau \in \mathbb{Z}_p$ and $nym = \mathsf{H}_1(bsn)^\tau$. If so, output 1, otherwise 0.
  - Define $\mathsf{ukgen}$ as follows: take $gsk \xleftarrow{\$} \mathbb{Z}_p$ and output $(gsk, gsk)$.
  - $\mathcal{S}$ sends $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.

Corrupt $\mathcal{I}$

- $\mathcal{S}$ notices this setup as it notices $\mathcal{I}$ registering a public key with "$\mathcal{F}_{\mathsf{ca}}$" with $sid = (\mathcal{I}, sid')$.
  - If the registered key is of the expected form and $\pi_{ipk}$ is valid, $\mathcal{S}$ extracts the issuer secret key from $\pi_{ipk}$.
  - $\mathcal{S}$ defines the algorithms $\mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen}$ as before, but now depending on the extracted key.
  - $\mathcal{S}$ sends $(\mathsf{SETUP}, sid')$ to $\mathcal{F}$ on behalf of $\mathcal{I}$.
- On input $(\mathsf{SETUP}, sid)$ from $\mathcal{F}$.
  - $\mathcal{S}$ sends $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.
- On input $(\mathsf{SETUPDONE}, sid)$ from $\mathcal{F}$
  - $\mathcal{S}$ continues simulating "$\mathcal{I}$".

**Forwarded Input**

- On input $(\mathsf{FORWARD}, m, \mathcal{P})$.
  - Give "$\mathcal{P}$" input $m$.

**Fig. 11.** Simulator for Game 4

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid, attrs), \mathcal{I})$ to $\mathcal{S}$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}), \mathcal{H}_j)$ to $\mathcal{S}$.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f \rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 12.** $\mathcal{F}$ for Game 5

When a simulated party "$\mathcal{P}$" outputs $m$ and no specific action is defined, send $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$.

**Setup**

Honest $\mathcal{I}$

- On input $(\mathsf{SETUP}, sid)$ from $\mathcal{F}$.
  - Parse $sid$ as $\mathcal{I}, sid'$.
  - Give "$\mathcal{I}$" input $(\mathsf{SETUP}, sid)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{SETUPDONE}, sid)$, $\mathcal{S}$ takes the issuer key pair. Note that the simulator also knows the issuer secret key, as it is simulating "$\mathcal{I}$".
  - Define $\mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$ as follows: First, create a BBS+ or CL signature (depending on the instantiation) using the issuer key on $gsk$ and attribute values where the disclosed attributes are taken from $p$ and the undisclosed attributes are set to dummy values. Next, the algorithm performs the real world signing algorithm (performing both the tasks from the host and the TPM).
  - Define $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$ as the real world verification algorithm, except that the private-key revocation check is ommitted.
  - Define $\mathsf{link}(\sigma, m, \sigma', m', bsn)$ as follows: Parse the signatures as $(A', nym, \pi, \{\pi_i\}) \leftarrow \sigma$, $(A'', nym', \pi', \{\pi'_i\}) \leftarrow \sigma'$, and output 1 iff $nym = nym'$.
  - Define $\mathsf{identify}(\sigma, m, bsn, \tau)$ as follows: parse $\sigma$ as $(nym, \pi_{cred}, \{\pi_{\mathtt{SRL},i}\})$ and check $\tau \in \mathbb{Z}_p$ and $nym = \mathsf{H}_1(bsn)^\tau$. If so, output 1, otherwise 0.
  - Define $\mathsf{ukgen}$ as follows: take $gsk \xleftarrow{\$} \mathbb{Z}_p$ and output $(gsk, gsk)$.
  - $\mathcal{S}$ sends $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.

Corrupt $\mathcal{I}$

- $\mathcal{S}$ notices this setup as it notices $\mathcal{I}$ registering a public key with "$\mathcal{F}_{\mathsf{ca}}$" with $sid = (\mathcal{I}, sid')$.
  - If the registered key is of the expected form and $\pi_{ipk}$ is valid, $\mathcal{S}$ extracts the issuer secret key from $\pi_{ipk}$.
  - $\mathcal{S}$ defines the algorithms $\mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen}$ as before, but now depending on the extracted key.
  - $\mathcal{S}$ sends $(\mathsf{SETUP}, sid')$ to $\mathcal{F}$ on behalf of $\mathcal{I}$.
- On input $(\mathsf{SETUP}, sid)$ from $\mathcal{F}$.
  - $\mathcal{S}$ sends $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.
- On input $(\mathsf{SETUPDONE}, sid)$ from $\mathcal{F}$
  - $\mathcal{S}$ continues simulating "$\mathcal{I}$".

**Verify & Link**

Nothing to simulate.

**Forwarded Input**

- On input $(\mathsf{FORWARD}, m, \mathcal{P})$.
  - Give "$\mathcal{P}$" input $m$.

**Fig. 13.** Simulator for GAME 5

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \perp, status\rangle$ with $status \leftarrow delivered$.
   - Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into `Members` and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}), \mathcal{H}_j)$ to $\mathcal{S}$.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 14.** $\mathcal{F}$ for Game 6

When a simulated party "$\mathcal{P}$" outputs $m$ and no specific action is defined, send $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$.

**Setup**

Unchanged.

**Join**

Honest $\mathcal{H}$, $\mathcal{I}$

- $\mathcal{S}$ receives $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - It simulates the real world protocol by giving "$\mathcal{H}_j$" input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ and waits for output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ from "$\mathcal{I}$".
  - $\mathcal{S}$ sends $(\mathsf{JOINSTART}, sid, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
  - $\mathcal{S}$ continues the simulation by giving "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$, and waits for output $(\mathsf{JOINED}, sid, jsid, attrs)$ from "$\mathcal{H}_j$".
  - Output $(\mathsf{JOINCOMPLETE}, sid, jsid, \perp)$ to $\mathcal{F}$.

Honest $\mathcal{H}$, Corrupt $\mathcal{I}$

- $\mathcal{S}$ receives $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINSTART}, sid, jsid)$ to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ as $\mathcal{I}$ is corrupt.
  - It simulates the real world protocol by giving "$\mathcal{H}_j$" input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ and waits for output $(\mathsf{JOINED}, sid, jsid, attrs)$ from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ on $\mathcal{I}$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINCOMPLETE}, sid, jsid, \perp)$ to $\mathcal{F}$.

Honest $\mathcal{M}$, $\mathcal{I}$, Corrupt $\mathcal{H}$

- $\mathcal{S}$ notices this join as "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$.
  - $\mathcal{S}$ knows the identity of the host involved in this join session as it is simulating "$\mathcal{M}_i$", let this be $\mathcal{H}_j$. For corrupt platforms, the exact identity of the host does not matter.
  - $\mathcal{S}$ extracts takes $tsk$ from simulating "$\mathcal{M}_i$" and extracts $hsk$ from $\pi_{gpk}$, and sets $gsk = tsk + hsk$.
  - $\mathcal{S}$ sends $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - $\mathcal{S}$ sends $(\mathsf{JOINSTART}, sid, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINCOMPLETE}, sid, jsid, gsk)$ to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{JOINED}, sid, jsid, attrs)$ as $\mathcal{H}_j$ is corrupt.
  - $\mathcal{S}$ continues the simulation by giving "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$.

Honest $\mathcal{I}$, Corrupt $\mathcal{M}$, $\mathcal{H}$

- $\mathcal{S}$ notices this join as "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$.
  - $\mathcal{S}$ does not know the identity of the host involved in this join session. It sets $\mathcal{H}_j$ as an aribitrary corrupt host. For corrupt platforms, the exact identity of the host does not matter.
  - $\mathcal{S}$ extracts $tsk$ from $\pi_{tpk}$ $hsk$ from $\pi_{gpk}$, and sets $gsk = tsk + hsk$.
  - $\mathcal{S}$ sends $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - $\mathcal{S}$ sends $(\mathsf{JOINSTART}, sid, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINCOMPLETE}, sid, jsid, gsk)$ to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{JOINED}, sid, jsid, attrs)$ as $\mathcal{H}_j$ is corrupt.
  - $\mathcal{S}$ continues the simulation by giving "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$, $\mathcal{I}$

- $\mathcal{S}$ notices this join as "$\mathcal{M}_i$" receives messages from a host $\mathcal{H}_j$ running the join protocol with $sid$ and $jsid$.
  - As $\mathcal{F}$ guarantees no security properties for platforms with a corrupt host when the issuer is corrupt, and $\mathcal{M}$ does not receive any output in the join protocol, $\mathcal{S}$ does not need to involve $\mathcal{F}$ and can simply continue simulating $\mathcal{M}_i$.

**Verify & Link**

Nothing to simulate.

**Forwarded Input**

- On input $(\mathsf{FORWARD}, m, \mathcal{P})$.
  - Give "$\mathcal{P}$" input $m$.

**Fig. 15.** Simulator for Game 6

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \perp, status\rangle$ with $status \leftarrow delivered$.
   - Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into `Members` and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`, abort.
   - If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \perp$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     - Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`.
   - For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}'_i, \mathcal{H}'_j)$ from $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     - Check that there are no two distinct $\tau$ values matching $\sigma'$.
     - Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 16.** $\mathcal{F}$ for Game 7

**Setup**

Unchanged.

**Join**

Unchanged.

**Sign**

Honest $\mathcal{H}$, $\mathcal{M}$

$\mathcal{S}$ not notice this signing taking place.

Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

- $\mathcal{S}$ receives $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$. After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends $(\mathsf{SIGNPROCEED}, sid, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

- $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m)$.
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and $\mathtt{SRL}$ of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^\delta, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

**Verify & Link**

Nothing to simulate.

**Fig. 17.** Simulator for GAME 7

1. `Issuer Setup`. On input ($\mathsf{SETUP}, sid$) from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output ($\mathsf{SETUP}, sid$) to $\mathcal{A}$ and wait for input ($\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen}$) from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store ($sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen}$) and output ($\mathsf{SETUPDONE}, sid$) to $\mathcal{I}$.

**Join**

2. `Join Request`. On input ($\mathsf{JOIN}, sid, jsid, \mathcal{M}_i$) from host $\mathcal{H}_j$.
   - Output ($\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j$) to $\mathcal{A}$ and wait for input ($\mathsf{JOINSTART}, sid, jsid$) from $\mathcal{A}$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \bot, status\rangle$ with $status \leftarrow delivered$.
   - Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in$ `Members` already exists.
   - Output ($\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i$) to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed`. On input ($\mathsf{JOINPROCEED}, sid, jsid, attrs$) from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output ($\mathsf{JOINCOMPLETE}, sid, jsid$) to $\mathcal{A}$ and wait for input ($\mathsf{JOINCOMPLETE}, sid, jsid, \tau$) from $\mathcal{A}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
   - Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into `Members` and output ($\mathsf{JOINED}, sid, jsid, attrs$) to $\mathcal{H}_j$.

**Sign**

4. `Sign Request`. On input ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}$) from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`, abort.
   - If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \bot$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     - Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   - Output ($\mathsf{SIGNPROCEED}, sid, ssid, m$) to $\mathcal{M}_i$ when it is honest, and ($\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma$) when $\mathcal{M}_i$ is corrupt.
5. `Sign Proceed`. On input ($\mathsf{SIGNPROCEED}, sid, ssid$) from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`.
   - For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}'_i, \mathcal{H}'_j)$ from $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i, *\rangle \in$ `Members` and $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i\rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     - Check that there are no two distinct $\tau$ values matching $\sigma'$.
     - Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in `Signed` and output ($\mathsf{SIGNATURE}, sid, ssid, \sigma$) to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify`. On input ($\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL}$) from some party $\mathcal{V}$.
   - Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to `VerResults` and output ($\mathsf{VERIFIED}, sid, f$) to $\mathcal{V}$.
7. `Link`. On input ($\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn$) from a party $\mathcal{V}$.
   - Output $\bot$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output ($\mathsf{LINK}, sid, f$) to $\mathcal{V}$.

**Fig. 18.** $\mathcal{F}$ for Game 8

**Setup**
Unchanged.
**Join**
Unchanged.
**Sign**
Honest $\mathcal{H}$, $\mathcal{M}$
$\mathcal{S}$ not notice this signing taking place.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

- $\mathcal{S}$ receives $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$. After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends $(\mathsf{SIGNPROCEED}, sid, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

- $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m)$.
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and `SRL` of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^\delta, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

**Verify & Link**
Nothing to simulate.

**Fig. 19.** Simulator for Game 8

1. `Issuer Setup`. On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
 – Verify that $sid = (\mathcal{I}, sid')$.
 – Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
 – Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
 – Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request`. On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
 – Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
 – Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \perp, status\rangle$ with $status \leftarrow delivered$.
 – Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in$ `Members` already exists.
 – Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed`. On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
 – Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
 – Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
 – If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
 – Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
 – Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into `Members` and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request`. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
 – If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`, abort.
 – If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \perp$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
 • Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathtt{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
 • Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
 – Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
 – Output $(\mathsf{SIGNPROCEED}, sid, ssid, m)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
5. `Sign Proceed`. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
 – Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
 – If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`.
 – For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}'_i, \mathcal{H}'_j)$ from $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i, *\rangle \in$ `Members` and $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i\rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
 • Check that there are no two distinct $\tau$ values matching $\sigma'$.
 • Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
 – Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify`. On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
 – Set $f \leftarrow 0$ if at least one of the following conditions hold:
 • There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
 – If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
 – Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link`. On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
 – Output $\perp$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
 – Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
 – Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 20.** $\mathcal{F}$ for Game 9

**Setup**
Unchanged.
**Join**
Unchanged.
**Sign**
Honest $\mathcal{H}$, $\mathcal{M}$
$\mathcal{S}$ not notice this signing taking place.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

– $\mathcal{S}$ receives ($\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma$) from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}$). After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output ($\mathsf{SIGNATURE}, sid, ssid, \sigma$) from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends ($\mathsf{SIGNPROCEED}, sid, ssid$) on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

– $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs ($\mathsf{SIGNPROCEED}, sid, ssid, m$).
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and $\mathtt{SRL}$ of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^\delta, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}$) on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.

– $\mathcal{S}$ receives ($\mathsf{SIGNATURE}, sid, ssid, \sigma$) from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input ($\mathsf{SIGNPROCEED}, sid, ssid$).

**Verify & Link**
Nothing to simulate.

**Fig. 21.** Simulator for Game 9

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \perp, status\rangle$ with $status \leftarrow delivered$.
   - Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   - Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into `Members` and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`, abort.
   - If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \perp$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     - Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor $\tau'$ registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`.
   - For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}_i', \mathcal{H}_j')$ from $\langle \mathcal{M}_i', \mathcal{H}_j', \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i', \mathcal{H}_j', \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     - Check that there are no two distinct $\tau$ values matching $\sigma'$.
     - Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 22.** $\mathcal{F}$ for GAME 10

**Setup**
Unchanged.
**Join**
Unchanged.
**Sign**
Honest $\mathcal{H}$, $\mathcal{M}$
$\mathcal{S}$ not notice this signing taking place.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

- $\mathcal{S}$ receives $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$. After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends $(\mathsf{SIGNPROCEED}, sid, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

- $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m)$.
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and `SRL` of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^{\delta}, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

**Verify & Link**
Nothing to simulate.

**Fig. 23.** Simulator for GAME 10

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \bot, status\rangle$ with $status \leftarrow delivered$.
   - Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
   - Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$, abort.
   - If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \bot$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     - Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\mathtt{DomainKeys}$.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor $\tau'$ registered in $\mathtt{Members}$ or $\mathtt{DomainKeys}$ with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$.
   - For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}_i', \mathcal{H}_j')$ from $\langle \mathcal{M}_i', \mathcal{H}_j', \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i', \mathcal{H}_j', \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     - Check that there are no two distinct $\tau$ values matching $\sigma'$.
     - Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in $\mathtt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one $\tau_i$ was found.
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to $\mathtt{VerResults}$ and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $\bot$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 24.** $\mathcal{F}$ for GAME 11

**Setup**
Unchanged.
**Join**
Unchanged.
**Sign**
Honest $\mathcal{H}$, $\mathcal{M}$
$\mathcal{S}$ not notice this signing taking place.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

- $\mathcal{S}$ receives ($\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma$) from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}$). After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output ($\mathsf{SIGNATURE}, sid, ssid, \sigma$) from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends ($\mathsf{SIGNPROCEED}, sid, ssid$) on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

- $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs ($\mathsf{SIGNPROCEED}, sid, ssid, m$).
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and $\mathtt{SRL}$ of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^\delta, \{(b_i, b_i', b_i'')\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}$) on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives ($\mathsf{SIGNATURE}, sid, ssid, \sigma$) from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input ($\mathsf{SIGNPROCEED}, sid, ssid$).

**Verify & Link**
Nothing to simulate.

**Fig. 25.** Simulator for Game 11

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   – Verify that $sid = (\mathcal{I}, sid')$.
   – Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   – Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   – Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

---

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   – Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   – Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \perp, status\rangle$ with $status \leftarrow delivered$.
   – Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
   – Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   – Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   – Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   – If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   – Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   – Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

---

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   – If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$, abort.
   – If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \perp$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     • Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\mathtt{DomainKeys}$.
     • Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
     • Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor $\tau'$ registered in $\mathtt{Members}$ or $\mathtt{DomainKeys}$ with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   – Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   – Output $(\mathsf{SIGNPROCEED}, sid, ssid, m)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   – Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   – If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$.
   – For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}'_i, \mathcal{H}'_j)$ from $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     • Check that there are no two distinct $\tau$ values matching $\sigma'$.
     • Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   – Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in $\mathtt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

---

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   – Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     • More than one $\tau_i$ was found.
     • $\mathcal{I}$ is honest and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found for which an entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle \in \mathtt{Members}$ exists with $p(attrs) = 1$.
     • There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   – If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   – Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to $\mathtt{VerResults}$ and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   – Output $\perp$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   – Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   – Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 26.** $\mathcal{F}$ for Game 12

**Setup**
Unchanged.
**Join**
Unchanged.
**Sign**
Honest $\mathcal{H}$, $\mathcal{M}$
$\mathcal{S}$ not notice this signing taking place.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

– $\mathcal{S}$ receives $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$. After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends $(\mathsf{SIGNPROCEED}, sid, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

– $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m)$.
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and `SRL` of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^{\delta}, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.

– $\mathcal{S}$ receives $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

**Verify & Link**
Nothing to simulate.

**Fig. 27.** Simulator for Game 12

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \perp, status\rangle$ with $status \leftarrow delivered$.
   - Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in$ `Members` already exists.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   - Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into `Members` and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`, abort.
   - If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \perp$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     - Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor $\tau'$ registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`.
   - For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}'_i, \mathcal{H}'_j)$ from $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i, *\rangle \in$ `Members` and $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i\rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     - Check that there are no two distinct $\tau$ values matching $\sigma'$.
     - Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i, *\rangle \in$ `Members` and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one $\tau_i$ was found.
     - $\mathcal{I}$ is honest and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found for which an entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle \in$ `Members` exists with $p(attrs) = 1$.
     - **$\mathcal{M}_i$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, *, *\rangle \in$ `Signed` exists.**
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 28.** $\mathcal{F}$ for Game 13

**Setup**
Unchanged.
**Join**
Unchanged.
**Sign**
Honest $\mathcal{H}$, $\mathcal{M}$
$\mathcal{S}$ not notice this signing taking place.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

- $\mathcal{S}$ receives ($\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma$) from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}$). After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output ($\mathsf{SIGNATURE}, sid, ssid, \sigma$) from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends ($\mathsf{SIGNPROCEED}, sid, ssid$) on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

- $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs ($\mathsf{SIGNPROCEED}, sid, ssid, m$).
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and `SRL` of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^\delta, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}$) on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives ($\mathsf{SIGNATURE}, sid, ssid, \sigma$) from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input ($\mathsf{SIGNPROCEED}, sid, ssid$).

**Verify & Link**
Nothing to simulate.

**Fig. 29.** Simulator for Game 13

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \bot, status\rangle$ with $status \leftarrow delivered$.
   - Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
   - Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$, abort.
   - If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \bot$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     - Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\mathtt{DomainKeys}$.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor $\tau'$ registered in $\mathtt{Members}$ or $\mathtt{DomainKeys}$ with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$.
   - For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}_i', \mathcal{H}_j')$ from $\langle \mathcal{M}_i', \mathcal{H}_j', \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i', \mathcal{H}_j', \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     - Check that there are no two distinct $\tau$ values matching $\sigma'$.
     - Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in $\mathtt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one $\tau_i$ was found.
     - $\mathcal{I}$ is honest and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found for which an entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle \in \mathtt{Members}$ exists with $p(attrs) = 1$.
     - $\mathcal{M}_i$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, *, *\rangle \in \mathtt{Signed}$ exists.
     - **$\mathcal{H}_j$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle \in \mathtt{Signed}$ exists.**
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to $\mathtt{VerResults}$ and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $\bot$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 30.** $\mathcal{F}$ for Game 14

**Setup**

Unchanged.

**Join**

Unchanged.

**Sign**

Honest $\mathcal{H}$, $\mathcal{M}$

$\mathcal{S}$ not notice this signing taking place.

Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

- $\mathcal{S}$ receives $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$. After calling $\mathtt{TPM.Commit}$, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends $(\mathsf{SIGNPROCEED}, sid, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

- $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m)$.
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and $\mathtt{SRL}$ of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a $\mathtt{TPM.Hash}$ query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^{\delta}, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed $\mathtt{TPM.Commit}$ with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

**Verify & Link**

Nothing to simulate.

**Fig. 31.** Simulator for GAME 14

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \bot, status\rangle$ with $status \leftarrow delivered$.
   - Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \texttt{Members}$ already exists.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
   - Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into `Members` and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`, abort.
   - If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \bot$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     - Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \texttt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor $\tau'$ registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}), \mathcal{H}_j)$ to $\mathcal{S}$.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in `Members`.
   - For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}_i', \mathcal{H}_j')$ from $\langle \mathcal{M}_i', \mathcal{H}_j', \tau_i, *\rangle \in \texttt{Members}$ and $\langle \mathcal{M}_i', \mathcal{H}_j', \tau_i\rangle \in \texttt{DomainKeys}$ where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     - Check that there are no two distinct $\tau$ values matching $\sigma'$.
     - Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i, *\rangle \in \texttt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in \texttt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one $\tau_i$ was found.
     - $\mathcal{I}$ is honest and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found for which an entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle \in \texttt{Members}$ exists with $p(attrs) = 1$.
     - $\mathcal{M}_i$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, *, *\rangle \in \texttt{Signed}$ exists.
     - $\mathcal{H}_j$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle \in \texttt{Signed}$ exists.
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$ **and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ for an honest $\mathcal{H}_j$ was found.**
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $\bot$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 32.** $\mathcal{F}$ for Game 15

**Setup**
Unchanged.
**Join**
Unchanged.
**Sign**
Honest $\mathcal{H}$, $\mathcal{M}$
$\mathcal{S}$ not notice this signing taking place.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

- $\mathcal{S}$ receives $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$. After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends $(\mathsf{SIGNPROCEED}, sid, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

- $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m)$.
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and `SRL` of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^\delta, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- $\mathcal{S}$ receives $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

**Verify & Link**
Nothing to simulate.

**Fig. 33.** Simulator for GAME 15

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   - Verify that $sid = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   - Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \perp, status\rangle$ with $status \leftarrow delivered$.
   - Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   - Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$, abort.
   - If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \perp$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     - Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\mathtt{DomainKeys}$.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor $\tau'$ registered in $\mathtt{Members}$ or $\mathtt{DomainKeys}$ with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}), \mathcal{H}_j)$ to $\mathcal{S}$.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$.
   - For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}'_i, \mathcal{H}'_j)$ from $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     - Check that there are no two distinct $\tau$ values matching $\sigma'$.
     - Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in $\mathtt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one $\tau_i$ was found.
     - $\mathcal{I}$ is honest and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found for which an entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle \in \mathtt{Members}$ exists with $p(attrs) = 1$.
     - $\mathcal{M}_i$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, *, *\rangle \in \mathtt{Signed}$ exists.
     - $\mathcal{H}_j$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle \in \mathtt{Signed}$ exists.
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$ and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ for an honest $\mathcal{H}_j$ was found.
     - For some matching $\tau_i$ and $(\sigma', m', bsn') \in \mathtt{SRL}$, $\mathsf{identify}(\sigma', m', bsn', \tau_i) = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to $\mathtt{VerResults}$ and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 34.** $\mathcal{F}$ for GAME 16

**Setup**
Unchanged.
**Join**
Unchanged.
**Sign**
Honest $\mathcal{H}$, $\mathcal{M}$
$\mathcal{S}$ not notice this signing taking place.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

– $\mathcal{S}$ receives $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$. After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{“}nonce\text{”}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends $(\mathsf{SIGNPROCEED}, sid, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

– $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m)$.
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and $\mathtt{SRL}$ of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^{\delta}, \{(b_i, b'_i, b''_i)\}, t_1, y_2, bsn_L, t_2, y_3, t_3))$. For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.

– $\mathcal{S}$ receives $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

**Verify & Link**
Nothing to simulate.

**Fig. 35.** Simulator for Game 16

1. `Issuer Setup.` On input $(\mathsf{SETUP}, sid)$ from issuer $\mathcal{I}$.
   – Verify that $sid = (\mathcal{I}, sid')$.
   – Output $(\mathsf{SETUP}, sid)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   – Check that $\mathsf{ver}$, $\mathsf{link}$, and $\mathsf{identify}$ are deterministic.
   – Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

**Join**

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   – Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{A}$.
   – Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, \perp, status\rangle$ with $status \leftarrow delivered$.
   – Abort if $\mathcal{I}$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
   – Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. `$\mathcal{I}$ Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid, attrs)$ from $\mathcal{I}$, with $attrs \in \mathbb{A}_1 \times \ldots \times \mathbb{A}_L$.
   – Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   – Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   – If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   – Else, verify that the provided tracing trapdoor $\tau$ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   – Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau, attrs\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid, attrs)$ to $\mathcal{H}_j$.

**Sign**

4. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL})$ from $\mathcal{H}_j$ with $p \in \mathbb{P}$.
   – If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$, abort.
   – If $\mathcal{H}_j$ is corrupt, set $\sigma \leftarrow \perp$. If $\mathcal{H}_j$ is honest, generate the signature for a fresh or established key:
     • Retrieve $(gsk, \tau)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\mathtt{DomainKeys}$.
     • Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn, p, \mathtt{SRL})$, check $\mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL}) = 1$.
     • Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor $\tau'$ registered in $\mathtt{Members}$ or $\mathtt{DomainKeys}$ with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   – Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status \leftarrow request$.
   – Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$ when it is honest, and $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma)$ when $\mathcal{M}_i$ is corrupt.
   – Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}), \mathcal{H}_j)$ to $\mathcal{S}$.
5. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   – Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, p, \mathtt{SRL}, \sigma, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   – If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle$ with $p(attrs) = 1$ exists in $\mathtt{Members}$.
   – For every $(\sigma', m', bsn') \in \mathtt{SRL}$, find all $(\tau_i, \mathcal{M}'_i, \mathcal{H}'_j)$ from $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}'_i, \mathcal{H}'_j, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma', m', bsn', *, \tau_i) = 1$.
     • Check that there are no two distinct $\tau$ values matching $\sigma'$.
     • Check that no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
   – Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle$ in $\mathtt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

**Verify & Link**

6. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, p, \mathtt{RL}, \mathtt{SRL})$ from some party $\mathcal{V}$.
   – Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i, *\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     • More than one $\tau_i$ was found.
     • $\mathcal{I}$ is honest and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found for which an entry $\langle \mathcal{M}_i, \mathcal{H}_j, *, attrs\rangle \in \mathtt{Members}$ exists with $p(attrs) = 1$.
     • $\mathcal{M}_i$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, *, *\rangle \in \mathtt{Signed}$ exists.
     • $\mathcal{H}_j$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j, p, \mathtt{SRL}\rangle \in \mathtt{Signed}$ exists.
     • There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$ and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ for an honest $\mathcal{H}_j$ was found.
     • For some matching $\tau_i$ and $(\sigma', m', bsn') \in \mathtt{SRL}$, $\mathsf{identify}(\sigma', m', bsn', \tau_i) = 1$.
   – If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn, p, \mathtt{SRL})$.
   – Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to $\mathtt{VerResults}$ and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.
7. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, p, \mathtt{SRL}, \sigma', m', p', \mathtt{SRL}', bsn)$ from a party $\mathcal{V}$.
   – Output $\perp$ to $\mathcal{V}$ if at least one signature $(\sigma, m, bsn, p, \mathtt{SRL})$ or $(\sigma', m', bsn, p', \mathtt{SRL}')$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   – For each $\tau_i$ in $\mathtt{Members}$ and $\mathtt{DomainKeys}$ compute $b_i \leftarrow \mathsf{identify}(\sigma, m, bsn, \tau_i)$ and $b'_i \leftarrow \mathsf{identify}(\sigma', m', bsn, \tau_i)$ and do the following:
     • Set $f \leftarrow 0$ if $b_i \neq b'_i$ for some $i$.
     • Set $f \leftarrow 1$ if $b_i = b'_i = 1$ for some $i$.
   – If $f$ is not defined yet, set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   – Output $(\mathsf{LINK}, sid, f)$ to $\mathcal{V}$.

**Fig. 36.** $\mathcal{F}$ for Game 17

**Setup**
Unchanged.
**Join**
Unchanged.
**Sign**
Honest $\mathcal{H}$, $\mathcal{M}$
$\mathcal{S}$ not notice this signing taking place.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

– $\mathcal{S}$ receives ($\mathsf{SIGNPROCEED}, sid, ssid, m, bsn, \mathtt{SRL}, \sigma$) from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}$). After calling `TPM.Commit`, "$\mathcal{H}_j$" will receive $\bar{n}_t \leftarrow \mathsf{H}(\text{"}nonce\text{"}, n_t)$, where the simulator knows $n_t$ as it simulates the random oracle. It sets $n_h$ such that $n_t \oplus n_h$ equals the nonce $n$ from $\sigma$. It performs the same procedure for every nonce in $\pi_{\mathtt{SRL},i}$. Wait for output ($\mathsf{SIGNATURE}, sid, ssid, \sigma$) from "$\mathcal{H}_j$".
  - $\mathcal{S}$ sends ($\mathsf{SIGNPROCEED}, sid, ssid$) on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

– $\mathcal{S}$ notices this signing session as "$\mathcal{M}_i$" outputs ($\mathsf{SIGNPROCEED}, sid, ssid, m$).
  - Note that $\mathcal{S}$ must make a signing query on $\mathcal{H}_j$'s behalf but does not know the $bsn$, $p$, and $\mathtt{SRL}$ of this signing session. If $\mathcal{I}$ is corrupt, $\mathcal{F}$ does not make any checks on those values, so we can use arbitrary values. If $\mathcal{I}$ is honest, $\mathcal{F}$ does perform checks on $bsn$, so we must find the correct value. The host has made a `TPM.Hash` query, and for this signing session to produce a valid signature, the message to be hashed has structure $m, (m_h, y_1, \hat{g}^{\delta}, \{(b_i, b_i', b_i'')\}, t_1, y_2, bsn_L, t_2, y_3, t_3)$). For all basenames that "$\mathcal{M}_i$" performed `TPM.Commit` with, it checks $y_2 = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$, where it knows $gsk$ from the join protocol. If such a $bsn$ is found, we have the correct basename, and if no such $bsn$ is found, this session will not yield a valid signature and we can continue to use a dummy $bsn$.
  - $\mathcal{S}$ sends ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn, p, \mathtt{SRL}$) on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.

– $\mathcal{S}$ receives ($\mathsf{SIGNATURE}, sid, ssid, \sigma$) from $\mathcal{F}$ as "$\mathcal{H}_j$" is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{M}_i$" input ($\mathsf{SIGNPROCEED}, sid, ssid$).

**Verify & Link**
Nothing to simulate.

**Fig. 37.** Simulator for GAME 17

We now show that every game hop is indistinguishable from the previous. Note that although we separate $\mathcal{F}$ and $\mathcal{S}$, in reductions we can consider them to be one entity, as this does not affect $\mathcal{A}$ and $\mathcal{E}$.

**Game 1:** This is the real world.

**Game 2:** We let the simulator $\mathcal{S}$ receive all inputs and generate all outputs. It does so by simulating all honest parties honestly. It simulates the oracles honestly, except that it chooses encryption keys in the $\mathsf{crs}$ of which it knows corresponding secret keys, allowing it to decrypt messages encrypted to the $\mathsf{crs}$. Clearly, this is equal to the real world.

**Game 3:** We now start creating a functionality $\mathcal{F}$ that receives inputs from honest parties and generates the outputs for honest parties. It works together with a simulator $\mathcal{S}$. In this game, we simply let $\mathcal{F}$ forward all inputs to $\mathcal{S}$, who acts as before. When $\mathcal{S}$ would generate an output, it first forwards it to $\mathcal{F}$, who then outputs it. This game hop simply restructures Game 2, we have Game 3 = Game 2.

**Game 4:** $\mathcal{F}$ now handles the setup queries, and lets $\mathcal{S}$ enter algorithms that $\mathcal{F}$ will store. $\mathcal{F}$ checks the structure of $sid$, and aborts if it does not have the expected structure. This does not change the view of $\mathcal{E}$, as $\mathcal{I}$ in the protocol performs the same check, giving Game 4 = Game 3.

**Game 5:** $\mathcal{F}$ now handles the verify and link queries using the algorithsm that $\mathcal{S}$ defined in Game 4. In Game 4, $\mathcal{S}$ defined the $\mathsf{ver}$ algorithm as the real world with the private key revocation check ommitted. As $\mathcal{F}$ performs this check separately. The link algorithm is equal to the real world algorithm, showing that using these algorithms does not change the verification or linking outcome, so Game 5 = Game 4.

**Game 6:** We now let $\mathcal{F}$ handle the join queries. $\mathcal{S}$ receives enough information from $\mathcal{F}$ to correctly simulate the real world protocol. Only when a join query with honest issuer and corrupt TPM and host takes place, $\mathcal{S}$ misses some information. It must make a join query with $\mathcal{F}$ on the host's behalf, but it does not know the identity of the host. However, it is sufficient to choose an arbitrary corrupt host. This results in a different host registered in `Members`, but $\mathcal{F}_{\mathsf{pdaa+}}$ will not use this information when the registered host is corrupt. Since $\mathcal{S}$ can always simulate the real world protocol, we have Game 6 = Game 5.

**Game 7:** $\mathcal{F}$ now handles the sign queries. There is no network traffic in the signing protocol (as we assume a perfectly secure channel between the TPM and host), so the simulation only has to worry about inputs and outputs. If both the host and TPM are honest, the adversary would not be activated in the real world, and therefore $\mathcal{S}$ does not have to simulate anything. If the TPM is corrupt but the host is honest, the adversary runs the TPM part of the signing protocol. The simulator simulates an honest host towards the adversary and can prevent $\mathcal{F}$ from outputting a signature if the simulated real world would not yield a signature. However, if the simulated real world outputs a signature, Lemma 1 shows that the signature will be anonymous, as the host rerandomizes the contributions from the adversary. We now argued that $\mathcal{F}$ will not output a signature if the simulated real world would not output a signature. However, $\mathcal{F}$ may prevent a signature from being output, when the TPM and host did not yet join, or when the signature generated by $\mathcal{F}$ does not pass verification. If the TPM and host did not join, and the host is honest, the simulated real world would also not output a signature, as the host performs this check. The signatures $\mathcal{F}$ generate will always pass verification, as the algorithms that $\mathcal{S}$ set in Game 4 will only create valid signatures. This shows that $\mathcal{F}$ outputs a signature if and only if the real world would outputs a signature.

$\mathcal{S}$ can simulate the real world protocol and block any signatures that would not be successfully generated in the real world. $\mathcal{F}$ may prevent a signature from being output, when the TPM and host did not yet join, or when the signature generated by $\mathcal{F}$ does not pass verification. If the TPM and host did not join, and the host is honest, the real world would also not output a signature, as the host performs this check. The signatures $\mathcal{F}$ generate will always pass verification, as the algorithms that $\mathcal{S}$ set in Game 4 will only create valid signatures. This shows that $\mathcal{F}$ outputs a signature if and only if the real world would outputs a signature.

What remains to show is that the signatures that $\mathcal{F}$ outputs are indistinguishable from the real world signatures. First, notice that the simulator takes care that the nonces in signatures in the real world match the nonces in the ideal world signatures. In addition, Lemma 1 shows that the zero knowledge proof of the signature is always distributed equally, like in the ideal world. The only difference is the exact statement thats being proved. When one party creates two signatures with different basenames, the real world protocol would use the same $gsk$, whereas $\mathcal{F}$ signs with different keys to show that the signatures are unlinkable. We make this change gradually. First, all signatures come from the real world, and then we let $\mathcal{F}$ gradually create more signatures, until all signatures come from $\mathcal{F}$. Let Game 7.$i.j$ denote the game in which $\mathcal{F}$ creates all signatures for platforms with TPMs $\mathcal{M}_{i'}$ with $i' < i$, lets $\mathcal{S}$ create the signatures if $i' > i$, and for the platform with TPM $\mathcal{M}_i$, the first $j$ distinct basenames are signed. We show that Game 7.$i.j$ is indistinguishable from Game 7.$i.(j+1)$, and by repeating this argument, we have Game 7 $\approx$ Game 6.

*Proof of* Game 7.$i.j \approx$ Game 7.$i.(j+1)$*:* We show that distinguishing Game 7.$i.j$ and Game 7.$i.(j+1)$ can be reduced to the DDH problem. The real world and the ideal world output signatures in the same cases: if and only if the platform joined and is not revoked by the signature revocation list $\mathtt{SRL}$, a signature is generated.

The difference between the two is that in the first game, when the platform with $\mathcal{M}_i$ signs w.r.t. $bsn_{j+1}$, the signature is made like in the real world protocol, using the key $gsk$ that it joined with. In the latter game, a credential is created on a fresh key $gsk$ and the signature uses that to construct signature $\sigma = (nym, \pi_{cred}, \{\pi_{\mathtt{SRL},i}\})$. We can reduce noticing this difference to the DDH problem.

The reduction takes as input a DDH instance $\bar{g}, \alpha, \beta, \gamma \in \mathbb{G}_1$ and must answer whether $log_{\bar{g}}(\alpha) \cdot log_{\bar{g}}(\beta) = log_{\bar{g}}(\gamma)$. We will simulate the platform with $\mathcal{M}_i$ using the unknown discrete log $log_{\bar{g}}(\alpha)$ as $gsk$ when joining and signing, except for signatures with $bsn_{j+1}$: there we use the unknown $log_{\beta}(\gamma)$ as $gsk$. Note that if the DDH instance is a DDH tuple, this is equivalent to game Game 7.$i.j$, whereas if it's not, this is equivalent to Game 7.$i.(j+1)$, showing that the two games are indistinguishable under the DDH assumption.

The simulation works as follows. Random oracle $\mathsf{H}_{\mathbb{G}_1}$ is simulated by returning $\bar{g}^r$ for $r \stackrel{\$}{\leftarrow} \mathbb{Z}_p$ while maintaining consistency, except for $bsn_{j+1}$, then it returns $\beta$. It simulates the host corresponding to TPM $\mathcal{M}_i$ as follows. Instead of choosing a value $hsk$ and computing $gpk$ using the TPM's $tsk$, we let $gsk$ be the (unknown) discrete log of $\alpha$: $gsk = log_{\bar{g}}(\alpha)$. We need to compute the platform public key $gpk = \tilde{g}^{gsk}$. For $\Pi_{\mathsf{qSDH-DAA}}$, $\tilde{g} = \bar{g}$ so we can set $gpk \leftarrow \alpha \ \ (= \tilde{g}^{gsk})$. For $\Pi_{\mathsf{LRSW-DAA}}$, $\tilde{g} = \mathsf{H}_{\mathbb{G}_1}(0||n) = \bar{g}^r$ for some $r$ known by simulating the random oracle, so we set $gpk \leftarrow \alpha^r \ \ (= \mathsf{H}_{\mathbb{G}_1}(0||n)^{gsk} = \tilde{g}^{gsk}$.

The host does not know the correspoding $hsk$, but it can still create $\pi_{gpk}$ as this proof is simulated. This means that $\mathcal{F}$ can no longer run $\mathsf{identify}$ as the simulator cannot extract $hsk$ to find the complete $gsk = tsk + hsk$. However, as we know $r$ such that $\mathsf{H}_{\mathbb{G}_1}(1||bsn) = \bar{g}^r$ for every $bsn$, we can identify signatures matching $gsk$ by checking whether $nym = \alpha^r$.

To sign with $bsn_l$, the reduction proceeds as follows:

- If $l \leq j$, $\mathcal{F}$ handles the signing.
- If $l = j+1$, the reduction must sign using $log_{\beta}(\gamma)$ as secret key. For $\Pi_{\mathsf{qSDH-DAA}}$, the proof of knowledge of the credential $\pi_{cred}$ can be simulated perfectly and we only need to worry about the pseudonym $nym$, which we set as $\gamma \ \ (= \beta^{log_{\beta}(\gamma)} = \mathsf{H}_{\mathbb{G}_1}(1||bsn_{j+1})^{log_{\beta}(\gamma)}$. For $\Pi_{\mathsf{LRSW-DAA}}$, we must in addition pay attention to the simulation of $\pi_{cred}$. As we know the issuer secret key, we can create a credential on $log_{\beta}(\gamma)$ by using $\beta, \gamma$ as $b, d$ values respectively. Then, we continue with the standard proof of knowledge, while simulating $\pi'_{cred}$ as we do not know $log_{\beta}(\gamma)$.
- If $l > j+1$, the reduction signs using the credential obtained in the join process, but it does not know $gsk = log_{\bar{g}}(\alpha)$. We can still sign, by simulating $\pi'_{cred}$ and setting $nym \leftarrow \alpha^r \ \ (= \mathsf{H}_{\mathbb{G}_1}(1||bsn_l)^{gsk})$ where $r$ is taken from simulating the random oracle.

**Game 8:** $\mathcal{F}$ now runs the $\mathsf{CheckTtdCorrupt}$ algorithm when $\mathcal{S}$ gives the extracted $\tau$ from platforms with a corrupt host. This checks that $\mathcal{F}$ has not seen valid signatures yet that match both this key and existing key. The $\mathsf{identify}$ algorithm checks whether a $\tau$ matches a pseudonym by checking $nym \stackrel{?}{=} \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{\tau}$. Note $\mathsf{identify}$ only accepts values in $\mathbb{Z}_p$, and that with overwhelming probability, $\mathsf{H}_{\mathbb{G}_1}(1||bsn) \neq 1_{\mathbb{G}_1}$, so there exists only one $\tau \in \mathbb{Z}_p$. Therefore we have Game 8 $\approx$ Game 7.

**Game 9:** When $\mathcal{F}$ creates fresh domain keys when signing for platforms with an honest host, it checks that there are no signatures that match this key. As argued in the previous game, with overwhelming probability there is one unique key matching a signature, which means that there already is a signature valid under the freshly created key. As $\mathsf{ukgen}$ takes the values from $\mathbb{Z}_p$ which is exponentially large, the probability that there already is a signature with this key is negligible, so GAME 9 $\approx$ GAME 8.

**Game 10:** $\mathcal{F}$ now performs additional tests on the signatures it creates, and if any fails, it aborts. First, it checks whether the generated signature matches the key it was generated with. With the algorithms $\mathcal{S}$ defined in GAME 4, this always holds. Second, $\mathcal{F}$ checks that there is no other platform with a key that matches this signature. If this check would happen with nonnegligible probability, we can break the DL assumption. Note that the DL assumption does not appear in the theorem statements, as for Theorem 1 it is implied by the generalized LRSW assumption, and for Theorem 2, it is implied by the $q$-SDH assumption.

We make this check for one platform and one basename at a time. The reduction receives a DL instance $\alpha = \bar{g}^{gsk}$ for some unknown $gsk$, and the functionality now uses this unknown $gsk$ as the domain key for the platform and basename under consideration. It simulates the signatures by programming random oracle $\mathsf{H}_{\mathbb{G}_1}$ to compute $nym = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{gsk}$ without knowing $gsk$ and simulating the zero-knowledge proofs $\pi_{cred}$, as in GAME 7. If the functionality now finds a value $\tau$ such that $nym = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^{\tau}$, with overwhelming probability we have $\mathsf{H}_{\mathbb{G}_1}(1||bsn) \neq 1_{\mathbb{G}_1}$, and therefore $\tau = gsk$, solving the DL problem.

**Game 11:** In verification, $\mathcal{F}$ now checks whether it knows multiple tracing keys that match one signature. With overwhelming probability, there will be no $bsn$ such that $\mathsf{H}_{\mathbb{G}_1}(1||bsn) = 1_{\mathbb{G}_1}$, meaning that there is a unique key tracing every signature, and showing that this check will never change the verification outcome.

**Game 12:** When $\mathcal{I}$ is honest, $\mathcal{F}$ verifying a signature now checks whether the signature matches some key of a platform that joined, and if not, rejects the signature. We prove that this check will trigger with negligible probability for $\Pi_{\mathsf{LRSW-DAA}}$ and $\Pi_{\mathsf{qSDH-DAA}}$ individually.

$\Pi_{\mathsf{LRSW-DAA}}$. For $\Pi_{\mathsf{LRSW-DAA}}$, we reduce this hop to the generalized LRSW assumption. As we described this protocol without using attributes, we are only concerned with membership, there are no attribute predicates possible. The reduction receives the issuer public key from the generalized LRSW problem, and registers this as its public key, along with a simulated proof. When running the join protocol, the issuer first queries $\mathcal{O}_X^{\mathsf{a,b}}(\cdot)$ to get $a$ and $b$. Then, it picks a fresh nonce $n$ and programs $\mathsf{H}_{\mathbb{G}_1}(0||n) = b$. When the join protocol proceeds and reaches the point where the issuer would compute $a, c$, it extracts $gsk$ from $\pi_{tpk}$ and $\pi_{gpk}$ and queries $\mathcal{O}_{X,Y}^{\mathsf{c}}(a, b, gsk)$ to receive $c$, forming a valid credential. The algorithm $\mathsf{sig}$ that is used by the functionality can no longer depend on the issuer secret key, as this key is unknown. The algorithm now uses the oracles to create a credential and simulates the proof.

Note that we only call $\mathcal{O}_{X,Y}^{\mathsf{c}}(a, b, gsk)$ on $gsk$-values that are put in `Members` and `DomainKeys`, and that for each of those $gsk$-values, the corresponding identities of the TPM and host are stored in `Members`. Therefore, from a signature that does not match any of the signed $gsk$ values, we can extract a new LRSW credential, breaking the generalized LRSW assumption.

$\Pi_{\mathsf{qSDH-DAA}}$. For $\Pi_{\mathsf{qSDH-DAA}}$, we reduce this hop to the $q$-SDH assumption, where $q - 1$ is the amount of platforms that the issuer allows to join. Camenisch et al. [CDL16b] show that with the $q$-SDH assumption $q-1$ BBS+ signatures can be simulated, and a forgery allows one to break the $q$-SDH assumption. Simultaneously, the $q$-SDH can be used to create a pair $g_1, g_1^x$, where $x$ is the BBS+ signing key. For readability we will phrase the reduction as playing the unforgeability game of the BBS+ signature, while we also use the pair $g_1, g_1^x$, so technically we reduce directly to the $q$-SDH assumption using the Camenisch et al. proof.

The reduction now receives a BBS+ public key. We use the pair $(g_1, g_1^x)$ by setting $X' \leftarrow g_1^x$, and simulate the proof $\pi_{ipk}$. When the issuer wants to issue a credential in the join protocol, it extracts $gsk$ from $\pi_{tpk}$ and $\pi_{gpk}$ and uses BBS+ signing oracle to sign $gsk$ and the attribute values. The algorithm $\mathsf{sig}$ that is used by the functionality now does not issue a credential, but simulates $\pi_{cred}$. Proof $\pi_{cred}$ consists of $(\bar{A}, A', b', \pi'_{cred})$. Note that for every honest proof, $A'$ is uniformly at random in $\mathbb{G}_1^*$ and $\bar{A} = A'^x$, where $x$ is the issuer secret key. We can simulate this by setting $\rho \overset{\$}{\leftarrow} \mathbb{Z}_p^*$, $\mathcal{A}' \leftarrow g_1^{\rho}$, $\bar{A} \leftarrow (g_1^x)^{\rho}$. As $b'$ is uniform in $\mathbb{G}_1$ in honest proofs, we can simply take a random element to simulate this. Finally, we simulate $\pi'_{cred}$ to perfectly simulate the proof of knowledge of the credential. When the functionality notices a valid signature that does not match any signed $gsk$ value of a platform that joined with satisfying attributes, we can extract a BBS+ forgery from $\pi_{cred}$, and break the $q$-SDH assumption.

**Game 13:** $\mathcal{F}$ now rejects signatures on a message $m$ w.r.t basename $bsn$ that match the key of a platform with an honest TPM but record in $\mathcal{F}$'s `Signed` that this TPM signed $m$ w.r.t. $bsn$. If this check triggers with nonnegligible probability, we can break the DL assumption.

First, we rule out forgeries with a message $m$ that the TPM did not sign, This follows directly from Lemma 2. In this reduction we simulate the TPM without knowing $tsk$, meaning we cannot extract $gsk$ to run $\mathsf{identify}$. However, we can replace $\mathsf{identify}$ with checking $nym = gpk^r$ where $r$ is taken from the random oracle: $\mathsf{H}_{\mathbb{G}_1}(1, bsn) = \bar{g}^r$. This means $\mathcal{F}$ can still identify the forgery from which we need to extract. Next, we rule out forgeries where message $m$ and basename $bsn$, where the TPM signed $m$ but not w.r.t. $bsn$. When signing $m$, the simulator simulating the TPM does not directly see the basename, but it can find out the only basename that could yield a valid signature: It sees $nym$ when performing `TPM.Hash`. When performing `TPM.Commit` commands, it computed values $K = \mathsf{H}_{\mathbb{G}_1}(1, bsn)^{tsk}$ for a number of basenames. For each of those basenames, it now computes $K^{hsk}$ and sees if this equals $nym$. The simulator then registers this basename in list `Signed` of $\mathcal{F}$. Note that when verifying the proof, a verifier will check that $nym$ is correctly hashed in the Fiat-Shamir hash, so a signature using this Fiat-Shamir hash cannot yield a valid signature with a different basename. This means that if $\mathcal{F}$ identifies a signature on $m, bsn$ where the TPM never signed these values, the proof is not simulated and we can extract $hsk$ by rewinding, breaking the DL assumption.

**Game 14:** $\mathcal{F}$ now rejects signatures on message $m$ w.r.t. a basename $bsn$, attribute predicate $p$, and signature revocation list `SRL`, that match the key of a platform with an honest host, but that host never signed this. We reduce this check triggering with nonnegligible probability to breaking the DL assumption.

The reduction receives DL instance $\alpha$ and must find $log_{\bar{g}}(\alpha)$. We simulate the host using this unknown discrete logarithm as $gsk$. For $\Pi_{\mathsf{qSDH-DAA}}$, we have $\bar{g} = \tilde{g}$, so we can set $\alpha$ as $gpk$ in the join protocol, and simulate $\pi_{gpk}$. For $\Pi_{\mathsf{LRSW-DAA}}$, we have $\tilde{g} = \mathsf{H}_{\mathbb{G}_1}(0||n) = \bar{g}^r$ for some $r$ known by simulating the random oracle, so we set $gpk \leftarrow \alpha^r \;\; (= \mathsf{H}_{\mathbb{G}_1}(0||n)^{gsk} = \tilde{g}^{gsk}$.

When signing, the host now simulates $\pi_{cred}$ using its power over the random oracles. The pseudonym $nym$ for a basename $bsn$ is computed by looking up $r$ such that $\mathsf{H}_{\mathbb{G}_1}(1||bsn) = \bar{g}^r$ from simulating $\mathsf{H}_{\mathbb{G}_1}$ and setting $nym \leftarrow \alpha^r$. Proof $\pi_{cred}$ can be simulated.

Note that $\mathcal{F}$ can no longer run $\mathsf{identify}$ as it does not know $gsk$. However, as we know $r$ such that $\mathsf{H}_{\mathbb{G}_1}(1||bsn) = \bar{g}^r$ for every $bsn$, we can identify signatures matching $gsk$ by checking whether $nym = \alpha^r$.

When $\mathcal{F}$ now receives a signature on $m$ w.r.t. $bsn, p, \mathtt{SRL}$ that matches $gsk$ while it never signed, it means the proof is not simulated and it proves knowledge of $gsk$. We can then extract $gsk$ to break the DL assumption.

**Game 15:** $\mathcal{F}$ now prevents private key revocation of platforms with an honest host. We show that if this happens with nonnegligible probability, we can break the DL assumption.

We simulate the platform with a discrete logarithm instance in the same way as Game 14. Clearly, if a value $\tau$ that matches one of this platform's signatures is found on the private key revocation list `RL`, we have found the desired discrete logarithm.

**Game 16:** $\mathcal{F}$ now enforces signature based revocation when verifying a signature. Let $nym$ be the pseudonym in this signature, and $bsn$ the basename. It checks that there is no $\tau$ with $nym = \mathsf{H}_{\mathbb{G}_1}(1||bsn)^\tau$ such that for some $(nym', bsn') \in \mathtt{SRL}$, $\mathsf{H}_{\mathbb{G}_1}(1||bsn')^{gsk} \neq nym$. The platform proves this using Camenisch-Shoup inequality proofs [CS03], so by soundness of the proofs, this check will only trigger with negligible probability.

**Game 17:** $\mathcal{F}$ now puts requirements on the $\mathsf{link}$ algorithm. These requirements do not change the output.

With overwhelming probability, we have $\mathsf{H}_{\mathbb{G}_1}(\cdot) \neq 1_{\mathbb{G}_1}$, so there is one unique $gsk \in \mathbb{Z}_p$ that matches the signature. If one $gsk$ matches one of the signatures but not the other, then $nym \neq nym'$ and $\mathsf{link}$ would also output 0. If both signatures match some $gsk$, then by soundness of the proof, we have $nym = nym'$ and $\mathsf{link}$ would also output 1. Therefore we have Game 17 = Game 16.

The functionality in Game 17 is equal to $\mathcal{F}_{\mathsf{pdaa+}}$, completing our security proof. $\square$

| This document | TPM Spec | |
|---|---|---|
| $tsk$ | $d_s$ | TPM secret key |
| $\bar{g}$ | $G$ | Fixed generator of $\mathbb{G}_1$ |
| $\tilde{g}$ | $P1$ | Generator used in `TPM.Commit` |
| $bsn$ | $s2, y2$ | The basename controlling linkability |
| $j$ | $(x2, y2)$ | Base of pseudonym computation |
| $q$ | $n$ | Order of $\mathbb{G}_1$ |
| $c$ | $P$ | Digest entered in `TPM.Sign` |
| $c'$ | $T$ | Fiat-Shamir hash |
| $m_t$ | TPM State | The data the TPM attests to |

**Table 1.** The meaning of variables and their name in the TPM specification.

## E Specification of our TPM 2.0 Commands

In this section we describe our revised TPM 2.0 commands used in Section 3 and Section 4 in the TPM specification notation [Tru14]. We again highlight our proposed changes in blue.

Note that the notation used in Section 3 and Section 4 differs from the notation used in the TPM specification. Table 1 shows how the variables used in the body of the paper correspond to the TPM specification.

The `TPM.Hash` is already part of the TPM 2.0 specification. Due to the limited storage, the TPM cannot store a list of $c$ values that are safe to sign. Instead, this behavior is implemented by outputting a MAC on $c$ when it is safe to sign. Whenever it receives a $c$ value with a MAC on it, it is treated as safe to sign. As we propose no changes to `TPM.Hash`, we do not show the command in full detail here.

### E.1 Generate a DAA key: TPM2_Create()

An ECDAA key can be generated by using an existing TPM 2.0 command, TPM2_Create(). The command refers to a parent key that is a storage key and is created by the TPM in advance. The command creates a fresh ECDAA key pair $tk = (tpk, tsk)$, and outputs a wrapped key blob. The key blob includes the following information: the private part of the key $tsk$ encrypted under the parent key, the public part of the key $tpk$, and the corresponding tag which allows the TPM to verify integrity and authenticity of the key.

### E.2 Make a Commitment: TPM2_Commit()

This is a modification of the existing TPM2_Commit() command in the current TPM 2.0 specification. This command performs the first part of a split operation of the ECDAA signature operation. We use a different way to define the value $P1$. In the current specification, the host may enter any point, which is why a malicious host can use the TPM as a static DH oracle. In this modification, we replace the point $P1$ with a base point $(x1, y1)$ where $x1 := HnameAlg(s1) \mod p$ in the LRSW-DAA scheme and let the TPM verify this point is generated from a hash function. If no $s1$ is given, the TPM uses standard generator $G$, which is what will be used in $q$-SDH-based DAA.

The *signHandle* parameter refers to the ECDAA key. TPM2_Commit() has the following parameters, all of which are optional.

- $s1$: octet array used to derive $x$-coordinate of a base point.
- $y1$: $y$-coordinate of the point associated with $s1$.
- $s2$: octet array used to derive $x$-coordinate of a base point.
- $y2$: $y$-coordinate of the point associated with $s2$.

In the algorithm below, the following additional values are used in addition to the command parameters:

- HnameAlg: hash function using the *nameAlg* of the key associated with *signHandle*.

– $p$: field modulus of the curve associated with $signHandle$.
– $n$: order of the curve associated with $signHandle$.
– $ds$: private key associated with $signHandle$.
– $G$: generator of the curve associated with $signHandle$.
– $Q$: public key associated with $signHandle$ and corresponding to $ds$, i.e., $Q = [ds]G$.
– $c$: counter that increments each time TPM2_Commit() is executed.

The Commit algorithm performs as follows:

1. Validate that $s1$ and $y1$ are either both Empty Buffers or both not Empty Buffers. If this does not hold, return an error message and abort.
2. If $s1$ is an Empty Buffer, skip to step 5.
3. Compute $x1 := HnameAlg(s1) \mod p$.
4. If $(x1, y1)$ is not a point on the curve of $signHandle$, return an error message and abort.
5. Validate that $s2$ and $y2$ are either both Empty Buffers or both not Empty Buffers. If this does not hold, return an error message and abort.
6. If $s2$ is an Empty Buffer, skip to step 9.
7. Compute $x2 := HnameAlg(\text{“}nonce\text{”}, s2) \pmod p$.
8. If $(x2, y2)$ is not a point on the curve of $signHandle$, return an error message and abort.
9. Set $K$, $L$, and $E$ to be Empty Buffers.
10. Generate $r$ or derive $r$ from an existing secret, dependent on the algorithm, which is specified in the existing TPM 2.0 specification.
11. Set $r := r \mod n$.
12. If $s1$ is an Empty Buffer, set $E := [r]G$.
13. If $s1$ is not an Empty Buffer, set $E := [r](x1, y1)$.
14. If $s2$ is not an Empty Buffer, set $K := [ds](x2, y2)$ and $L := [r](x2, y2)$.
15. If $K$, $L$, or $E$ is the point at infinity, return an error message and abort (negligible probability).
16. Follow the same mechanism of giving the value $r$, obtain $nonceT$.
17. Compute $nT = \mathsf{HverifyAlg}(nonceT)$.
18. Set $counter := commitCount$.
19. Set $commitCount := commitCount{+}1$. NOTE 1: Depending on the method of generating $r$ and $nonceT$, it may be necessary to update the tracking array here.
20. Output $K$, $L$, $E$, $nT$, and $counter$.
    NOTE 2: Depending on the input parameters, $K$, $L$ or $E$ may be Empty Points. There are the following different cases:
    (a) $E := [r](x1, y1)$ and both $K$ and $L$ are Empty Points - the Sign process without linkability and without signature-based revocation in the LRSW-based DAA scheme.
    (b) $E := [r](x1, y1)$, $K := [ds](x2, y2)$ and $L := [r](x2, y2)$ - either the Sign process with linkability in the LRSW-based DAA scheme or a proof of signature-based revocation.
    (c) $E := [r]G$ and both $K$ and $L$ are Empty Points - the Schnorr signature.
    (d) $E := [r]G$, $K := [ds](x2, y2)$ and $L := [r](x2, y2)$ - the sDH-based DAA scheme.

### E.3 Complete a Signature: TPM2_Sign()

This is a modification of the existing TPM2_Sign() command in the current TPM 2.0 specification. To complete the ECDAA sign operation, the TPM uses the same random or pseudo-random values $r$ and $nonceT$ as used in TPM2_Commit(). These values are determined by the counter field in the scheme parameter of the signing command. We add an input $nonceH$, which is a nonce chosen by the host. This prevents the TPM from embedding information in $nonce$, which would harm the privacy of the host.

TPM2_Sign() has the following parameters.

– $c$: counter associated with the corresponding TPM2_Commit() execution.
– $P$: hash value associated with the corresponding to TPM2_Commit() execution.

- $nonceH$: nonce from the host.
- HschemeAlg: hash function used to compute a signature.

The signature is created using a modified Schnorr signature and the operation is as follows:

1. Retrieve the values $r$ and $nonceT$ based on $c$. The mechanism has been specified in the existing TPM 2.0 specification.
2. Set $T :=$ HschemeAlg("$TPM$" $||$ $nonceT \oplus nonceH$ $||$ $P$). NOTE: The symbol $||$ denotes the concatenation operation.
3. Compute integer $s := (r + T \cdot ds) \pmod{n}$.
4. If $s = 0$, output failure (negligible probability).

The signature is the tuple $(nonceT, s)$.