Direct Anonymous Attestation with Optimal TPM Signing Efficiency

Kang Yang¹, Liqun Chen², Zhenfeng Zhang³, Christopher J.P. Newton², Bo Yang³, and Li Xi³

¹ State Key Laboratory of Cryptology, Beijing 100878, China
yangk@skl.ac.cn

² University of Surrey, UK
{liqun.chen, c.newton}@surrey.ac.uk

³ Trusted Computing and Information Assurance Laboratory, SKLCS,
Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
{zfzhang, yangbo, xili}@tca.iscas.ac.cn

Abstract. Direct Anonymous Attestation (DAA) is an anonymous signature scheme, which is designed to allow the Trusted Platform Module (TPM), a small chip embedded in a host computer, to attest to the state of the host system, while preserving the privacy of the user. DAA provides two signature modes: fully anonymous signatures and pseudonymous signatures. To generate a DAA signature, the calculations are divided between the TPM and the host. One main goal towards designing new DAA schemes is to reduce the signing burden on the TPM side as much as possible, since the TPM has only limited resources when compared to the host and the computational overhead of the TPM dominates the whole signing performance. In an optimal DAA scheme, the signing workload on the TPM will be no more than that required for a normal signature. DAA has developed about seventeen years, but no scheme has achieved this optimal signing efficiency for both signature modes.

In this paper, we propose the first DAA scheme which achieves this optimal TPM signing efficiency for both signature modes. In particular, the TPM takes only a single exponentiation in a prime-order group when generating a DAA signature. Additionally, this single exponentiation can be precomputed, which enables our scheme to achieve fast online signing time. Our DAA scheme is provably secure under the DDH, DBDH and $q$-SDH assumptions in the Universally Composable (UC) framework. Our scheme can be implemented using the existing TPM 2.0 commands, and thus is compatible with the TPM 2.0 specification. There are three important use cases for DAA: quoting platform configuration register values, certifying a key and signing a message. We have benchmarked the TPM 2.0 commands needed for these use cases on an Infineon TPM 2.0 chip, which is also useful to evaluate the TPM signing efficiency of other DAA schemes. We also implemented the host signing and verification algorithm for our DAA scheme on a laptop with 1.80GHz Intel Core i7-8550U CPU. Our experimental results show that our DAA scheme obtains the total signing time of roughly 140 ms and the online signing time of roughly 60 ms for both signature modes. Based on our benchmark results, our scheme is about 2× (resp., 5×) faster than existing DAA schemes supported by TPM 2.0 in terms of total signing efficiency (resp., online signing efficiency for the pseudonymous signature mode).

In addition, our DAA scheme supports selective attribute disclosure, which can satisfy more application requirements. We also extend our DAA scheme to support signature-based revocation and to guarantee privacy against subverted TPMS. The two extended DAA schemes keep the TPM signing efficiency optimal for both signature modes, and outperform existing related schemes in terms of signing performance.

Keywords: Direct anonymous attestation · TPM 2.0 implementation · Anonymous signatures · Provable security

1 Introduction

With the rapid growth of devices connected to the internet, it is becoming increasingly difficult to secure the devices [CCD⁺17]. To achieve better security, one approach is to place a root of trust such as a Trusted Platform Module (TPM) into such devices and use this to attest to the current state of the device. It is crucial that such attestations are privacy-preserving. On the one hand, anonymous attestation protects the privacy of owners of the devices, which adhere to one of the essential elements of privacy-enhancing systems (i.e., disassociability) developed by NIST [NIS15]. On the other hand, it minimizes the information available to the adversary and satisfies the so-called data minimization principle [PH10]. In addition, the privacy protection of users receives more attentions, due to the introduced General Data Protection Regulation (GDPR) [GD16] (Europe’s new privacy regulation). More than a billion devices use the TPM technology;
virtually all enterprise PCs, many servers and embedded systems include TPMs [Tru20], which makes DAA one of the most complex cryptographic schemes that has been widely implemented.

The Trusted Computing Group (TCG), an industry standardization group, has developed Direct Anonymous Attestation (DAA) to realize such attestations in a privacy-preserving manner. DAA schemes have been included in both of the TPM 1.2 and the TPM 2.0 specifications [Tru03, Tru16] and these have been adopted as ISO/IEC 11889 international standards [ISO09, ISO15]. While the TPM 1.2 supports a RSA-based DAA scheme [BCC04], the more recent TPM 2.0 supports multiple ECDAA schemes which are built on pairing-friendly elliptic curves, which have the better performance. The ISO/IEC have also standardized three DAA schemes [BCC04, CPS10, BL10b] in the ISO/IEC 20008-2 standard [Int13]. In this paper, we will only focus on ECDAA schemes. Chen and Li [CL13] defined the TPM 2.0 commands needed to implement two alternative DAA schemes [CPS10, BL10b]. The flexibility of these commands means that they could be used to implement further ECDAA schemes. For example, the scheme presented by Camenisch et al. [CDL16a] (in the full version of their paper) can also be implemented using these TPM 2.0 commands. However, for this scheme the session identifiers used for the UC security proof need to be eliminated.

DAA allows a TPM, which is a small chip embedded in a host, to attest either to the current state of the host system or to some other message, while preserving the privacy of the user owning the TPM. DAA provides two signature modes so that a user can decide whether a signature should be linkable to other signatures or not. Specifically, signatures w.r.t. the empty basename $bsn = \bot$ are fully anonymous (i.e., un-linkable). Alternatively, signatures w.r.t. a non-empty basename $bsn \neq \bot$ are pseudonymous, meaning that signatures under the same basename are linkable, while signatures under different basenames are unlinkable. In some applications such as anonymous subscription [LDK+13, KLL+18] and vehicular communication (V2X) [PSFK15, WCG+17], pseudonymous signatures may be preferable or required for system operations. Pseudonymous signatures provide an advantage that allowing users to create pseudonyms at a service provider and obtaining value-added services from the service provider.

While TPM is a small discrete chip and has only limited resources, the host has much more powerful computational capability (e.g., the host is about a factor of $300 \times$ faster than the TPM according to the experimental results [CL13, BCN14]). However, the host provides less security tolerance than the TPM. As pointed out by Camenisch et al. [CDL16b], the main challenge in designing a DAA scheme is to distribute the computational work between the TPM and host such that the workload of the TPM is as small as possible, while this does not affect the security in the case that the host is corrupted. A crucial feature of DAA is that the TPM and host cooperatively create a signature via executing a sign protocol. In an optimal DAA scheme, the signing workload on the TPM will be no more than that required for a normal signature such as ECSchnorr [Sch91, Tru16]. Specifically, only one exponentiation is required for the TPM when generating a signature, where one exponentiation is necessary to prevent the corrupted host from forging signatures without interacting with the TPM. Informally, we say that the TPM signing efficiency of a DAA scheme is fully optimal if the TPM takes only a single exponentiation per signature generation for both two signature modes, and partially optimal if one exponentiation holds for only one signature mode.

The original DAA scheme was introduced by Brickell, Camenisch and Chen [BCC04], but requires the TPM to compute exponentiations over a large RSA modulus, which leads to the costly computational burden for the TPM. Later, researchers resorted to bilinear pairings in order to construct more efficient ECDAA schemes. The ECDAA schemes fall into two categories: 1) LRSW-DAA schemes [BCL08, CMS08, CPS10, BFG+13, BCL12, CDL16b] based on the LRSW assumption [LRSW99] or its variants; and 2) SDH-DAA schemes [CF08, Che10, BL10b, CDL16a] based on the $q$-SDH assumption [BB08]. DAA schemes have developed about seventeen years, and improved gradually the signing efficiency of the TPM. However, only the LRSW-DAA schemes [BFG+13, BCL12] achieves the partially optimal TPM signing efficiency for fully anonymous signature mode. Furthermore, the best known SDH-DAA scheme [CDL16a] requires three exponentiations for the TPM to generate a signature for both two signature modes.
1.1 Our Contributions

In this paper, we propose the first DAA scheme with fully optimal TPM signing efficiency and denote it by DAA_{OPT}. That is, DAA_{OPT} only requires the TPM to carry out a single exponentiation in a prime-order group $\mathbb{G}_1$ when creating a signature for both signature modes. Moreover, the single exponentiation can be pre-computed, which allows our scheme DAA_{OPT} to obtain fast on-line signing time. Additionally, we present a simple implementation trick of parallel computation to reduce the signing time needed by the host side. Our ECDAA scheme DAA_{OPT} is provably secure under the DDH, DBDH and $q$-SDH assumptions in the Universally Composable (UC) security model [CDL16b, CDL16a] and the random oracle model [BR93].

Our scheme DAA_{OPT} is compatible with the TPM 2.0 specification, i.e., DAA_{OPT} can be implemented using the existing TPM 2.0 commands. For the first time, we consider TPM 2.0 implementations of three important DAA use cases, i.e., quoting PCR values, certifying a TPM key, and signing an arbitrary message given by the host, where these DAA use cases are corresponding to three types of DAA applications that will be shown in Section 1.2. We have implemented and benchmarked several TPM 2.0 commands on an Infineon TPM 2.0 chip with vendor ID IFXSLB9670, which is installed on a module designed for the Raspberry Pi. Our benchmark results allow us to evaluate the TPM signing performance for these three DAA use cases.

We implemented the host signing and the verification algorithm for our scheme DAA_{OPT} on a laptop with 1.80GHz Intel Core i7-8550U CPU over the BN P256 curve using the AMCL library. Together with the benchmark result on an Infineon TPM 2.0 chip, we obtain that DAA_{OPT} needs about 138 ms for total signing, 50 ms for online signing and 5.9 ms for verification in the fully anonymous signature mode. In the pseudonymous signature mode, the running time is about 144 ms, 64.6 ms and 8.1 ms respectively. Specifically, our scheme DAA_{OPT} is about $2 \times$ faster for total signing time and $5 \times$ faster for on-line signing time than the existing DAA schemes supported by TPM 2.0, in the pseudonymous signature mode. When generating a fully anonymous signature, DAA_{OPT} is about $2 \times$ more efficient than the known SDH-DAA schemes supported by TPM 2.0. Our scheme DAA_{OPT} has the same efficiency as the state-of-the-art LRSW-DAA scheme compatible with TPM 2.0 in terms of generating fully anonymous signatures, but is more efficient than this scheme in terms of the verification efficiency.

In addition, our DAA scheme DAA_{OPT} supports selective attribute disclosure, which can satisfy more application requirements. We also extend DAA_{OPT} to support signature-based revocation and to guarantee privacy in presence of subverted TPMs. The two extended DAA schemes keep the TPM signing efficiency fully optimal, and provide significantly better signing performance than known related schemes.

1.2 Applications of Our DAA Scheme

We outline three types of applications for our DAA scheme DAA_{OPT} depending on what DAA signatures are used for. In Section 5, we present how to use the TPM 2.0 commands to implement our DAA scheme with three use cases in order to support these types of applications.

**APPLICATION I**: We can apply DAA_{OPT} to the remote attestation of trusted computing by quoting the Platform Configuration Register (PCR) values recording the current state of the host system in order to protect the users’ privacy, when a signature is used to quote the PCR values. Additionally, our scheme DAA_{OPT} with the pseudonymous signature mode can be applied to V2X [WCG+17] via attesting to the current status of the vehicle which is recorded in the PCR values. In these cases, our scheme DAA_{OPT} provides the advantage of fast attestation/authentication.

**APPLICATION II**: We can apply DAA_{OPT} to the Fast IDentity Online (FIDO) authentication framework [FID17] to eliminate the unacceptably high risk in the FIDO basic attestation scheme that an attestation key is shared across a set of authenticators with identical characteristics, where a signature is used to certify a key created by the TPM. In this application, the TPM creates a new authentication key, and generates a fully anonymous signature (by cooperating with the host) to certify that the key is stored properly in the
TPM. The FIDO alliance is in the process of standardizing a specification called FIDO ECDAA [CDE+17], which requires three exponentiations for the TPM to generate a signature. When applying \( \text{DAA}_{\text{OPT}} \) with a fully anonymous signature mode to the FIDO authentication framework, we can reduce the signing cost of the TPM from three exponentiations in FIDO ECDAA to only one exponentiation.

\textbf{APPLICATION III:} We can also use \( \text{DAA}_{\text{OPT}} \) to construct an anonymous authentication scheme by combining it with TLS [CLR+10], to support the anonymous public transportation system [ALT+15], or to realize anonymous subscription [KLL+18]. In these applications, a signature is used to sign an arbitrary message given by the host such as a nonce from a verifier, a timestamp and a public key created by the host. For these cases, our scheme \( \text{DAA}_{\text{OPT}} \) not only prevents the sharing of credentials under the assumption that users cannot extract secret keys from TPMs, but also provides fast authentication.

In addition, for mobile devices, Raj et al. [RSW+16] presented the implementation of a firmware-based TPM (fTPM) using ARM TrustZone [ARM], which supports the TPM 2.0 specification. As a result, we can also apply \( \text{DAA}_{\text{OPT}} \) to mobile devices with ARM TrustZone by using fTPM to perform the TPM operations, and provide the advantage of better on-line signing performance and smaller Trusted Computing Base (TCB), compared to known DAA schemes supported by TPM 2.0.

1.3 Related Work

\textbf{LRSW-DAA.} Brickell et al. [BCL08] proposed the first LRSW-DAA scheme over symmetric bilinear groups. This scheme is further improved in [CMS08, CPS10] over asymmetric bilinear groups. Later, Bernhard et al. [BFG+13] utilized the special algebraic structure of randomized credentials, which implicitly contain unlinkable tags, to minimize the TPM’s signing overhead for fully anonymous signature mode. However, their LRSW-DAA scheme still requires three exponentiations in \( G_1 \) for the TPM to create a pseudonymous signature. Brickell et al. [BCL12] uses the batch proof and verification technique to construct the most efficient LRSW-DAA scheme for now, which reduces the signing overhead of the TPM to two exponentiations in \( G_1 \) for the generation of a pseudonymous signature. However, this scheme is not compatible with the TPM 2.0 specification [Tru16]. Canard et al. [CPS14] proposed an efficient approach to delegate the computations of the TPM to the host in the interactive zero-knowledge proofs of knowledge. Using their method to the proofs of knowledge for pseudonymous signatures in Bernhard et al.’s DAA scheme [BFG+13], they show that the TPM could pre-compute one exponentiation in \( G_2 \), and compute on-line one exponentiation in \( G_1 \). Although the signing efficiency of the TPM is not improved, the TPM’s on-line signing cost is reduced by two exponentiations in group \( G_1 \). However, the group operations in \( G_2 \) for the TPM are not supported by TPM 2.0. All the above LRSW-DAA schemes do not support attributes.

\textbf{SDH-DAA.} Chen and Feng [CF08] introduced the first SDH-DAA scheme. Chen [Che10] improved the signing efficiency of the TPM via removing an element of the credential. Later, Brickell and Li [BL10b] further improved the signing efficiency of the TPM by changing the way of delegation computation between the TPM and host, such that the TPM takes three exponentiations in \( G_1 \) per sign protocol run. Recently, Camenisch et al. [CDL16a] proposed an efficient proof of knowledge for BBS+ signatures [ASM06], and then constructed a SDH-DAA scheme which improves the signing efficiency on the host side. Their scheme is the most efficient SDH-DAA scheme for now, but still requires three exponentiations in \( G_1 \) for the TPM to generate a signature for both modes of signatures.

\textbf{DAA with Attributes.} Chen and Urian [CU15] introduced DAA with attributes, which extends DAA to support attributes such as the manufacturer and model version of the platform and an expiration date of the credential etc. DAA with attributes supports selective attribute disclosure, i.e., a user can choose a part of attributes to disclose in a signature but other undisclosed attributes keep hidden. They proposed two DAA schemes with attributes by extending the LRSW-DAA scheme [CPS10] and the SDH-DAA scheme [BL10b] respectively, where their schemes allow the TPM to protect multiple attributes. While their SDH-DAA
scheme with attributes [CU15] has $O(1)$ credential size, their LRSW-DAA scheme with attributes [CU15] requires $O(n)$ credential size, where $n$ is the number of attributes. Later, Camenisch et al. [CDL16a] proposed a $q$-SDH-based DAA scheme with attributes, which stores all attributes on the host to obtain better efficiency. All these DAA schemes with attributes [CU15, CDL16a] can still be implemented using the TPM 2.0 commands.

**Signature-Based Revocation.** Brickell and Li [BL07, BL10a] introduced Enhanced Privacy ID (EPID), which extends DAA with signature-based revocation. This revocation extension allows one to revoke a platform, based on a previous signature from the platform, even if the signature is fully anonymous. While private key revocation in DAA allows to revoke a platform by adding the platform’s secret key to the revocation list, signature-based revocation allows for revocation without knowing the secret key of the platform and is an improvement over private key revocation. The pairing-based EPID scheme [BL10a] is recommended by Intel to serve as the industry standard for privacy-preserving authentication in Internet of Things (IoTs). These EPID schemes [BL07, BL10a] require $6n_r$ exponentiations for the TPM to prove that the platform has not been revoked, where $n_r$ is the size of the signature revocation list. This is too expensive for a TPM with limited resources. Recently, Camenisch et al. [CDL16a] showed how to delegate the TPM’s partial computations to the host in the signature-based revocation, which reduces the overhead of the TPM to $3n_r$ exponentiations. However, it is still too expensive for the TPM with limited resources.

**DAA with Subverted TPMs.** Recently, Camenisch et al. [CDL17] considered the setting that TPMs are possible to be subverted, i.e., the TPMs are created by a compromised manufacturer. A subverted TPM may create a subliminal channel (i.e., embedding some information into a signature) to compromise the privacy of a user. They proposed a DAA scheme with subverted TPMs, which requires two (resp., one) exponentiations for the TPM to produce a pseudonymous (resp., fully anonymous) signature. However, their DAA scheme requires that the TPM performs group operations in $G_2$, and thus cannot be implemented by the TPM 2.0 commands, even if one can modify the TPM 2.0 commands with small changes. Later, Camenisch et al. (S&P’17) [CCD+17] modified the TPM 2.0 commands with minimal changes to the current TPM 2.0 commands, and showed that the modified TPM 2.0 commands can avoid a subliminal channel. Then, they used the modified TPM 2.0 commands to implement two ECDAA schemes with subverted TPMs, where one is based on the $q$-SDH assumption [BB08] and the other is based on a generalized variant of the LRSW assumption. Both the two schemes support signature-based revocation, but only the $q$-SDH-based scheme considers the support of attributes. Their signature-based revocation mechanism [CCD+17] still requires $3n_r$ exponentiations for the TPM when proving the platform has not been revoked.

### 1.4 Organization

We present the preliminaries in Section 2. We recall the definitions of DAA schemes in Section 3. In Section 4, we present the construction of our DAA scheme $\text{DAA}_{\text{OPT}}$ and two ways to further improve the efficiency of $\text{DAA}_{\text{OPT}}$, and also give an informal security analysis of $\text{DAA}_{\text{OPT}}$. In Section 5, we present the TPM 2.0 implementation of our DAA scheme involving three use cases of DAA. We evaluate the performance of our DAA scheme via comparing it with known DAA schemes supported by TPM 2.0 in Section 6. Signature-based revocation extension of our DAA scheme is shown in Appendix B.1, and we extend our DAA scheme to guarantee privacy against subverted TPMs in Appendix B.2. We provide an alternative description of our DAA scheme for UC security in Appendix C, and give a full formal security proof in Appendix D.
2 Preliminaries

2.1 Notation

Throughout this paper, we denote the security parameter by $\lambda$. We use $x \overset{\$}{\leftarrow} S$ to denote that sampling $x$ uniformly at random from a finite set $S$. For a group $G$, $G^*$ denotes the set $G \setminus \{1_G\}$, where $1_G$ is the identity element of $G$. We use $[n]$ to denote the set $\{1, \ldots, n\}$. We say that a function $f : \mathbb{N} \rightarrow [0, 1]$ is negligible if for every positive polynomial $\text{poly}(\cdot)$ and all sufficiently large $\lambda$ such that $f(\lambda) < 1/\text{poly}(\lambda)$. We say that a function $f$ is overwhelming if $1 - f$ is negligible.

2.2 Bilinear Groups

Let $G$ be a probabilistic polynomial time (PPT) bilinear-group generator that on input a security parameter $1^\lambda$, outputs a bilinear group $A = (p, G_1, G_2, G_T, e, g_1, g_2)$, where $G_1$, $G_2$ and $G_T$ are groups of prime order $p$, $g_1$ and $g_2$ are the generators of $G_1$ and $G_2$ respectively, and $e : G_1 \times G_2 \rightarrow G_T$ is a bilinear map.

We say that $e : G_1 \times G_2 \rightarrow G_T$ is a bilinear map (pairing) if it is efficiently computable and satisfies the following properties: 1) bilinearity, i.e., $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$ for all $a, b \in \mathbb{Z}_p$; 2) non-degeneracy, i.e., $e(g_1, g_2) \neq 1_{G_T}$ for all generators $g_1 \in G_1$ and $g_2 \in G_2$. Following [GPS08], pairings are categorized into three types: 1) Type-1 pairings (a.k.a. symmetric pairings) have $G_1 = G_2$; 2) Type-2 pairings require $G_1 \neq G_2$, but there exists an efficiently computable isomorphism $\psi : G_2 \rightarrow G_1$ such that $g_1 = \psi(g_2)$; 3) Type-3 pairings provide $G_1 \neq G_2$, but now there is no efficiently computable isomorphisms between $G_1$ and $G_2$. Type-2 and Type-3 pairings are called asymmetric pairings. Throughout this paper, we only consider Type-3 pairings.

2.3 Signature Proofs of Knowledge

We will use the notation introduced by Camenisch and Stadler [CS97] to abstract Signature Proofs of Knowledge (SPKs) on proving knowledge of discrete logarithms and statements about them. The SPKs can be obtained using Fiat-Shamir heuristic [FS86] to transform the corresponding Sigma protocols. For instance, $\pi \leftarrow \text{SPK}\{(x) : y = g^x\}(m)$ denotes a signature proof of knowledge $\pi$ on a message $m$, which proves knowledge of a witness $x$ such that $y = g^x$, where $G = \langle g \rangle$ is a group of prime order $p$. The SPKs are zero-knowledge via programming the random oracle and knowledge extractable in the random oracle model [PS00].

2.4 Assumptions

**Assumption 1 (DBDH).** We say that the Decisional Bilinear Diffie-Hellman (DBDH) assumption [BB04] holds for $G$ if any PPT adversary $A$ and $A = (p, G_1, G_2, G_T, e, g_1, g_2) \leftarrow G(1^\lambda)$, there exists a negligible function $\nu(\cdot)$ such that

\[
\Pr[a, b, c \overset{\$}{\leftarrow} \mathbb{Z}_p : A(A, g_1^a, g_2^b, g_1^c, g_2^d, e(g_1, g_2)^{abc}) = 1] - \\
\Pr[a, b, c, d \overset{\$}{\leftarrow} \mathbb{Z}_p : A(A, g_1^a, g_2^b, g_1^c, g_2^d, e(g_1, g_2)^{d}) = 1] \leq \nu(\lambda).
\]

In fact, the above assumption is an asymmetric version of the original DBDH assumption [BB04] for symmetric bilinear pairings. Recently, Desmoulins et al. [DLST14] used an analogous asymmetric version of the original DBDH assumption, where the adversary is given an additional element $g_1^a$ as input. Freire et al. [FHKP13] used an asymmetric version of the original DBDH assumption over Type-2 pairings (DBDH-2) as introduced in [Gal05], where the adversary is given $(g_2, g_1^a, g_2^b, g_2^c)$ as input. For Type-2 pairings, the elements $g_1^a$ and $g_2^c$ can be computed via $\psi(g_2^b)$ and $\psi(g_2^c)$ respectively. Thus, the adversary is actually given $(g_1, g_2, g_1^a, g_1^b, g_2^b, g_1^c, g_2^c)$ as input in the DBDH-2 assumption.
Assumption 2 (DDH$_{G_1}$). We say that the Decisional Diffie-Hellman (DDH) assumption holds in group $G_1$ if for any PPT adversary $A$ and $\Lambda = (p, G_1, G_2, G_T, e, g_1, g_2) \leftarrow \mathcal{G}(1^\lambda)$, there exists a negligible function $\nu(\cdot)$ such that

$$\left| \Pr[a, b \leftarrow Z_p : A(a, g_1^a, g_1^{ab}) = 1] - \Pr[a, b, c \leftarrow Z_p : A(a, g_1^{a+b}, g_1^c) = 1] \right| \leq \nu(\lambda).$$

Assumption 3 ($q$-SDH). We say that the $q$-Strong Diffie-Hellman ($q$-SDH) assumption [BB08] holds for $G$ if for any PPT adversary $A$ and $\Lambda = (p, G_1, G_2, G_T, e, g_1, g_2) \leftarrow \mathcal{G}(1^\lambda)$, there exists a negligible function $\nu(\cdot)$ such that

$$\Pr[\gamma \leftarrow Z_p^* : (g_1^{1/(\gamma+c)}, c) \leftarrow A(\Lambda, g_1^\gamma, \ldots, g_1^{\gamma^q}, g_2^\gamma)] \leq \nu(\lambda),$$

where $c \in Z_p \setminus \{-\gamma\}$.

3 Definitions of DAA Schemes

In this section, we review the syntax of DAA schemes and the desired security properties for DAA, i.e., anonymity, unforgeability and non-frameability. We adopt the security model for DAA by Camenisch et al. [CDL16b], which is defined as an ideal functionality $F_{\text{daa}}^l$ in the Universal Composability (UC) framework [Can01]. We extend this model to support the functionality of attributes by following the extension [CDL16a]. We refer the reader to Appendix A (or [CDL16b, CDL16a]) for the formal security definition of DAA in the form of an ideal functionality.

3.1 Syntax of DAA Schemes

In a DAA scheme, there are four types of parties: TPM $\mathcal{M}_i$ and host $\mathcal{H}_j$ constituting a platform, issuer $\mathcal{I}$ and verifier $\mathcal{V}$. The DAA scheme consists of three algorithms Setup, Verify and Link, and two protocols Join and Sign.

Setup. Given a set of system parameters params on a security parameter $\lambda$, an issuer $\mathcal{I}$ generates its public key $ipk$ and secret key $isk$, where params and ipk are publicly available. We assume that params and ipk are implicit inputs for the following protocols and algorithms.

Join. This is an interactive protocol between a platform $(\mathcal{M}_i, \mathcal{H}_j)$ and the issuer $\mathcal{I}$ who decides whether the platform is allowed to become a member. By executing the join protocol, the platform creates a secret key $gsk$, and receives a number of attributes $attrs = (a_1, \ldots, a_n)$ and a credential $cre$ given by $\mathcal{I}$. The credential $cre$ certifies the secret key $gsk$ and attributes $attrs$, where the attributes include more information about the platform such as the manufacturer and model version and an expiration date of the credential etc.

Sign. After being a member, a TPM $\mathcal{M}_i$ and a host $\mathcal{H}_j$ can jointly sign a message $m$ w.r.t. a basename $bsn$ resulting in a signature $\sigma$, where $bsn$ is either the name string of a verifier or a special symbol $\bot$. We refer to $\sigma$ as a fully anonymous signature if $bsn = \bot$ and a pseudonymous signature otherwise. The platform can also selectively disclose a part of attributes from its credential $cre$, e.g., disclosing that the signature was created by a TPM of a certain manufacturer or the expiration date of the credential. We denote the disclosure of attributes by $(D, I)$, where $D \subseteq \{1, \ldots, n\}$ is a set indicating which attributes are disclosed, $I = (a_1, \ldots, a_n)$ is a tuple specifying the disclosed attribute values, and $a_i$ is set as $\bot$ if the $i$-th attribute is not disclosed. We also denote by $\bar{D}$ the set of the indices of undisclosed attributes, i.e., $\bar{D} = \{1, \ldots, n\} \setminus D$. 
Verify. Given a message $m$, a basename $bsn$, a signature $\sigma$, an attribute disclosure $(D, I)$ and a revocation list $RL$ consisting of the secret keys of corrupted platforms, a verifier $V$ can run a deterministic verification algorithm to check that $\sigma$ is valid on $m$ w.r.t. $bsn$ and stems from a platform that holds a credential satisfying the predicate defined by $(D, I)$. The verification algorithm outputs 1 if the check passes and 0 otherwise.

The revocation list $RL$ is used to support private key revocation. When a secret key (private key) of a corrupted platform is exposed, the secret key would be added to $RL$, which allows a verifier to recognize and thus reject all the signatures created by the secret key.

Link. On input two message/signature pairs $(m_0, \sigma_0)$ and $(m_1, \sigma_1)$, attribute disclosure $(D_0, I_0)$ and $(D_1, I_1)$ and a basename $bsn \neq \bot$, a verifier $V$ can run a deterministic link algorithm to decide whether the two signatures link or not. If both $\sigma_0$ and $\sigma_1$ are valid on respective $(m_0, (D_0, I_0))$ and $(m_1, (D_1, I_1))$ w.r.t. the same $bsn \neq \bot$ and were produced by the same secret key, the link algorithm outputs 1 (linked). Otherwise, the link algorithm outputs $\bot$ if one of $\sigma_0$ and $\sigma_1$ is invalid and 0 (unlinked) otherwise.

3.2 Desired Security Properties for DAA

Following the work [CDL16b], a DAA scheme should satisfy the following desired security properties:

**Anonymity.** Given two signatures with respect to different basenames or $bsn = \bot$, no adversary can distinguish whether both signatures were generated by the same honest platform, or whether they were created by two different honest platforms. The property requires that the entire platform (TPM+host) is honest, and should hold even if the issuer is corrupted.

**Unforgeability.** This property requires that the issuer is honest, and should hold even if some or all hosts are corrupted.

1. If all unrevoked TPMs are honest, no adversary can produce a signature on a message $m$ w.r.t. a basename $bsn$ and attribute disclosure $(D, I)$, when no platform that joined with those attributes signed $m$ w.r.t. $bsn$ and $(D, I)$.

2. An adversary can only sign in the name of corrupted TPMs. More precisely, if $k$ corrupted and unrevoked TPMs joined with attributes fulfilling attribute disclosure $(D, I)$ for some integer $k$, the adversary can create at most $k$ unlinkable signatures w.r.t. the same basename $bsn \neq \bot$ and attribute disclosure $(D, I)$.

**Non-frameability.** No adversary can create a signature on a message $m$ w.r.t. a basename $bsn$ which links to a signature created by an honest platform, when the platform never signed $m$ w.r.t. $bsn$. The property requires that the entire platform is honest, and should hold even if the issuer is corrupted.

4 Our DAA Scheme

We present the construction of our DAA scheme (denoted by $DAA_{OPT}$). Our scheme $DAA_{OPT}$ supports selective attribute disclosure, and would be degraded as a standard DAA scheme when removing the attributes (i.e., $n = 0$). Following [CDL16a], we consider that only the secret key is protected by the TPM and all attributes are stored on the host in order to obtain better efficiency. We will further improve the computational efficiency of $DAA_{OPT}$ by presenting online/offline DAA signatures and a simple implementation trick of parallel computation. We prove that protocol $DAA_{OPT}$ securely realizes functionality $F^l_{daa}$ with static corruption and attributes defined in [CDL16b, CDL16a] under the DBDH, DDH$_{G_1}$ and $q$-SDH assumptions in the random oracle model, based on the proofs by Camenisch et al. [CDL16b, CDL16a]. We informally argue the security of $DAA_{OPT}$ in this section, and give the detailed security proof in Appendix D. First of all, we describe the high-level ideas underlying the construction of $DAA_{OPT}$.
4.1 High Level Description

We adopt the BBS+ signature to issue DAA credentials, where the BBS+ signature scheme was proposed in [ASM06] based on the schemes [BBS04, CL04]. This means that a platform consisting of a TPM and a host will obtain a credential \((A, x, u)\) on a secret key \(gsk\) and attributes \(attrs = (a_1, \ldots, a_n)\) such that

\[
A = (g_1 \cdot gsk^u \cdot h_0^{\prod_{i=1}^{n} h_i^{a_i}})^{1/(\gamma + x)}
\]

in the join protocol, where \(n\) is the number of attributes and \(\gamma\) is the secret key of the issuer. We use the proof of knowledge for BBS+ signatures in the full version of [CDL16a] to prove possession of such a credential.\(^4\) In particular, the credential \(A\) is randomized as \(T_1\) and the validity of \(T_1\) is proved using a signature proof of knowledge.

Except for the proof of knowledge of a credential, a pseudonym and its proof are included in a DAA signature for \(bsn \neq \bot\). Furthermore, an unlinkable tag and its proof are also involved in a signature to support private-key revocation for \(bsn = \bot\). In the existing DAA schemes, a pseudonym is set as \(K = H_G(bsn)^{gsk}\), and an unlinked tag is set as \((B, K = B^{gsk})\) for a random \(B \in G\) or \(B = H_G(str)\) with a random string \(str\), where \(H_G : \{0, 1\}^* \rightarrow G\) is a random oracle, \(G\) is a cyclic group such as \(G_1\) and \(gsk\) is the TPM’s secret key. This results in that the TPM needs to cost two exponentiations to compute \(K\) and prove the validity of \(K\) in known DAA schemes. In this paper, we propose two techniques to achieve the fully optimal TPM signing efficiency.

For pseudonymous signature mode, we propose a technique of delegable pseudonyms, which is inspired by Canard et al.’s method [CPS14] on delegation of zero-knowledge proofs of knowledge. Specifically, a pseudonym on a basename \(bsn\) is computed as \(K = e(g, H_G(bsn))^{gsk}\), where \(H_G : \{0, 1\}^* \rightarrow G_2\) is a random oracle. The new construction of pseudonyms allows the TPM to delegate the computations of a pseudonym \(K\) and a commitment \(L = e(g, H_G(bsn))^{r}\) to the host, where \(L\) is used to prove knowledge of \(gsk\) such that \(K = e(g, H_G(bsn))^{gsk}\) and \(r\) is chosen at random by the TPM. Concretely, the host stores a public key \(gpk = g^{gsk}\) created by the TPM in the join protocol and receives a commitment \(E = g^r\) from the TPM in the sign protocol. Then, the host can compute \(K\) and \(L\) via \(K \leftarrow e(gpk, H_G(bsn))\) and \(L \leftarrow e(E, H_G(bsn))\) respectively.

From the construction of pseudonyms, we can see that \(gpk = g^{gsk}\) must keep hidden, and otherwise \(gpk\) can be used to identify the signatures. Thus, the platform cannot directly send \(gpk\) to the issuer in the join protocol. A possible way is to let the platform send a Pedersen commitment \(C = g^{gsk} \cdot h_0^{u'}\) to the issuer. However, this way is not compatible with the TPM 2.0 specification. That is, by the existing TPM 2.0 commands, neither the TPM could create a Pedersen commitment \(C\) nor the TPM could check whether the commitment \(C'\) received by the issuer is created correctly using the TPM public key \(gpk\) when the host chooses the randomness \(u'\). To be compatible with the TPM 2.0 specification, we split the key \(gsk\) into a secret key \(tsk\) chosen by the TPM and a secret key \(hsk\) picked by the host via \(gsk \leftarrow tsk + hsk\), where the technique of splitting keys was previously used for guaranteeing privacy against subverted TPMs in [CDL17]. Specifically, the TPM sends a public key \(tpk = g^{tsk}\) to the host, and the host stores \(gpk = tpk \cdot g^{hsk}\). In the join protocol, the host picks \(u' \leftarrow Z_p\) and computes \(C \leftarrow g^{hsk} \cdot h_0^{u'}\), and then sends \(tpk\) and \(C\) to the issuer for requesting a credential. Now, a commitment \(L\) can be computed by the host via picking \(\hat{r} \leftarrow Z_p\) and computing \(L \leftarrow e(E \cdot g^r, H_G(bsn))\).

For fully anonymous signature mode, we present a technique of delegable unlinkable tags to delegate the computations of an unlinkable tag \((B, K = B^{gsk})\) and the corresponding commitment \(L = B^{r+\hat{r}}\) to the host, where \(r, \hat{r}\) are chosen at random by the TPM and host respectively. Specifically, the host picks \(b \leftarrow Z_p\), and then can compute \((B, K)\) (resp., \(L\)) via randomizing \((\bar{g}, gpk)\) (resp., \(E\)) as \((B = \bar{g}^b, K = gpk^b)\) (resp., \(L = (E \cdot \bar{g}^\hat{r})^b\)).

\(^4\)This proof of knowledge is based on the proofs of knowledge for the weak Boneh-Boyen signature in [ALT+15, CDH16]. Concurrently, Barki et al. [BBDT17] gave a similar proof of knowledge for BBS+ signatures but slightly less efficient.
<table>
<thead>
<tr>
<th>TPM $\mathcal{M}_i$</th>
<th>Host $\mathcal{H}_j$</th>
<th>Issuer $\mathcal{I}$ (isk = $\gamma$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{tsk} \leftarrow Z_p^*$</td>
<td>$\text{tpk} \leftarrow g^{\text{tsk}}$</td>
<td>$\text{tpk} \leftarrow \text{tpk} \cdot \text{hpk}$</td>
</tr>
<tr>
<td>$\text{tpk} \leftarrow g^{\text{tsk}}$</td>
<td>$\text{hpk} \leftarrow g^\text{sk}$</td>
<td>$C \leftarrow \text{hpk} \cdot h_0^\text{i}$</td>
</tr>
<tr>
<td>$\lambda \leftarrow H_{\gamma}$</td>
<td>$\bar{z} \leftarrow r \cdot g^r$</td>
<td>$N_i \leftarrow {0,1}^\lambda$</td>
</tr>
<tr>
<td>$r \leftarrow Z_p$</td>
<td>$E \leftarrow \bar{g}^r$</td>
<td>$\text{JOIN}$</td>
</tr>
</tbody>
</table>

Fig. 1: The join protocol of DAAOPT. The notation TPM.Create, TPM.Commit and TPM.Sign represent the TPM requests of the following procedures: creating a TPM key, generating a commitment and generating a signature respectively. Note that they are not real TPM 2.0 commands.

### 4.2 Detailed Construction

We assume the public availability of system parameters $\text{params} = (\lambda, p, G_1, G_2, G_T, e, g_1, g_2, \bar{g}, \ell_n)$, where $\lambda$ is a security parameter, $(p, G_1, G_2, G_T, e, g_1, g_2)$ is a set of bilinear group parameters generated by $G_1^\lambda$, $\bar{g} \in G_1$ is a fixed generator and $\ell_n$ denotes the bit length of nonce picked by TPMs. We will use four independent hash functions $H_i : \{0,1\}^* \rightarrow Z_p$ for $i \in \{1,2,3\}$ and $H_{G_2} : \{0,1\}^* \rightarrow G_2$ modeled as random oracles. Note that $H_{G_2}$ can be implemented fast using the hashing algorithms [FCKRH12, BP17], and the speed of calculating $H_{G_2}$ is doubled in the case of BN curves [FCKRH12].

**Setup.** Given system parameters $\text{params}$, an issuer $\mathcal{I}$ creates its public/private key pair ($ipk$, $isk$) as follows:
1. Choose $h_0, h_1, \ldots, h_n \in G_1^*$.
2. Pick $\gamma \leftarrow Z_p^*$, and compute $w \leftarrow g_2^\gamma$. 

The sign protocol of Fig. 2: represent the TPM requests rather than real TPM 2.0 commands.

\[ R_1 \) recover a commitment \( c_1 \) for each \( i \in \bar{D} \),
\[ \hat{E} \leftarrow \overline{g}_1^g, \hat{R}_1 \leftarrow \overline{Y}^{r-\gamma} \cdot h_0^c \cdot \prod_{i \in \bar{D}} h_i^{r_i}, \]
\[ R_2 \leftarrow T_1^{-r_1} \cdot h_0^{r_2}, \hat{E} \leftarrow E \cdot \hat{E}, R_1 \leftarrow E \cdot \hat{R}_1 \]

If \( bsn \neq \bot \), \( B \leftarrow \bot \), \( K \leftarrow e(gpk, H_{\bar{D}_2}(bsn)) \), \( L \leftarrow e(\bar{E}, H_{\bar{D}_2}(bsn)) \).
\[ c_{ih} \leftarrow H_2("sign", \bar{g}, g_1, \{h_{i1}\}_{i=0}^n, T_1, T_2, Y', B, K, R_1, R_2, L) \]

Fig. 2: The sign protocol of DAA\textsubscript{OPT}. For the case that \( bsn \neq \bot \), \( B \) is set as \( \bot \), as \( e(\bar{g}, H_{\bar{D}_2}(bsn)) \) can be computed offline by the verifier. The elements marked in the dashed box can be computed offline by the TPM and host. Again, TPM.Commit and TPM.Sign represent the TPM requests rather than real TPM 2.0 commands.

3. Prove knowledge of secret key \( \gamma \) on public key \( w \) by

\[ \pi_1 \leftarrow \text{SPK}_1\{\gamma\} : w = g_2^x \text{("setup")}. \]

4. Set ipk \( \leftarrow \{\{h_{i1}\}_{i=0}^n, w, \pi_1\} \) and isk \( \leftarrow \gamma \).

\( \text{SPK}_1 \) can be constructed in the following standard way: 1) pick \( r \leftarrow \mathbb{Z}_p \) and compute a commitment \( R \leftarrow g_2^r ; 2) \) generate a challenge \( c \leftarrow H_3\text("setup", g_2, w, R) ; 3) \) produce a response \( s \leftarrow r + c \cdot \gamma \mod p \); 4) output a proof \( \pi_1 \leftarrow (c, s) \). The proof \( \pi_1 = (c, s) \) can be easily verified publicly by doing the following: 1) recover a commitment \( R' \leftarrow g_2^c \cdot w^{-c} ; 2) \) compute \( c' \leftarrow H_3\text("setup", g_2, w, R') ; 3) \) accept the proof if \( c = c' \) and reject it otherwise. The issuer \( I \) also registers its public key ipk at a Certification Authority (CA) such that anyone can get the public key ipk correctly.

Join. The join protocol executed between the TPM \( M_i \), host \( H_j \) and issuer \( I \) is shown in Figure 1, where JOIN denotes a join request. We assume that \( M_i \) can authenticate itself to \( I \) and convince \( I \) that \( tpk \) is created by a legitimate TPM. This can be realized by enabling \( M_i \) and \( I \) to communicate over a semi-authenticated channel, meaning that a message sent to the issuer consists of an authenticated part (i.e., \( tpk \)) and an unauthenticated part (i.e., \( (C, \pi_t, \pi_h) \)). Multiple methods can be adopted to establish the semi-authenticated channel using the TPM’s endorsement key [Tru16], where an overview is provided in [BFG+13]. We can adopt the method in [CW10] to establish the semi-authenticated channel, where the method has been adopted by the TCG in the TPM 2.0 specification [CL13, Tru16]. Besides, by using this method [CW10], \( I \) can send a credential \( (A, x, u'' \rangle \) and the attributes \( \text{attrs} \) to the platform in a confidential manner via encrypting them with an encryption scheme.
In the join protocol, $\cal M_i$ creates a public key $tpk$ and $\cal H_j$ produces a Pedersen commitment [Ped92] $C = g^hsk h_0^{u'}$. Then, $\cal M_i$ proves knowledge of secret key $tsk$ with the help of $\cal H_j$, i.e., they cooperatively produce a signature proof of knowledge

$$\pi_t \leftarrow \text{SPK}_t\{(tsk) : tpk = g^{tsk}\} ("\text{TPM}.join", N_t).$$

The host $\cal H_j$ also proves knowledge of secret key $hsk$ and randomness $u'$ via independently generating

$$\pi_h \leftarrow \text{SPK}_h\{(hsk, u') : C = g^{hsk} h_0^{u'}\} ("\text{Host}.join", N_t).$$

Upon receiving a tuple $(tpk, C, \pi_t, \pi_h, \mathcal{I})$, $\mathcal{I}$ checks the validity of proofs $\pi_t$ and $\pi_h$, and then blindly issues a BBS+ signature $(A, x, u'')$ on key $gsk$ and attributes $attrs = \{a_i\}_{i=1}^n$ to platform $(\cal M_i, \cal H_j)$, where $gsk = tsk + hsk \mod p$ is the secret key of this platform. Except for the BBS+ signature $(A, x, u'')$ and secret key $hsk$, $\cal H_j$ stores $\mathcal{P} = tpk \cdot g^{hsk}$ and $Y = g_1 \cdot gpk \cdot h_0^u \cdot \prod_{i=1}^n h_i^{a_i}$ in credential $cre$ for fast signing. To be compatible with the TPM 2.0 specification, the TPM does not output a digest $c$, and instead the host re-computes $c$ from $N_t$ and $ch$.

**Sign.** A TPM $\cal M_i$ and a host $\cal H_j$ can cooperatively sign a message $m$ w.r.t. basename $bsn$ and attribute disclosure $(D, I)$ by executing the sign protocol shown in Figure 2, where $(D, I)$ is selectively disclosed by $\cal H_j$. To generate a signature, $\cal H_j$ randomizes $A$ and $Y$ as $T_1 = A^{t_1}$ and $Y' = Y^{t_1} h_0^{t_2}$ respectively, and computes $T_2 = Y^{t_1} T_1^{-x}$ such that $T_2 = T_1^y$. Then, $\cal H_j$ generates an unlinkable-tag/pseudonym $(B, K = B^{gsk})$, where either $B = g^b$ or $B = e(g, H_{G2}(bsn))$. Next, $\cal M_i$ cooperates with $\cal H_j$ to produce a signature proof of knowledge

$$\pi_2 \leftarrow \text{SPK}_2\{(gsk, \{a_i\}_{i \in D}, x, \bar{u}, t_2, t_3) : g_1^{-1} \prod_{i \in D} h_i^{-a_i} = Y'^{t_2 - t_3} g^{gsk} h_0^{u} \prod_{i \in D} h_i^{a_i} \land T_2/Y' = T_1^{x} h_0^{t_2} \land K = B^{gsk}\} ("\text{sign}", m, bsn, D, I),$$

where $t_3 = t_2^{-1} \mod p$ and $\bar{u} = u - t_2 t_3 \mod p$.

In the process of generating a proof $\pi_2$, host $\cal H_j$ calls TPM $\cal M_i$ to produce a signature proof of knowledge

$$\pi_t \leftarrow \text{SPK}_t\{(tsk) : tpk = g^{tsk}\} ("\text{sign}", m, bsn, D, I).$$

**Verify.** On input a message $m$, a basename $bsn$, a signature $\sigma$, attribute disclosure $(D, I)$ and a revocation list $RL$, a verifier $V$ can verify the signature as follows:

1. Parse signature $\sigma$ as $(T_1, T_2, Y', B, K, \pi_2)$ and proof $\pi_2$ as $(c, \bar{s}, s_x, s_y, s_t, t, \{s_i\}_{i \in D}, N_t)$.
2. Check that $B \neq 1_G$, if $bsn = \bot$ and $B = \bot$ otherwise. If $bsn \neq \bot$, compute $B = e(g, H_{G2}(bsn))$.
3. Check that $e(T_1, w) = e(T_2, g_2)$.
4. Verify the validity of proof $\pi_2$ as follows:
   
   (a) Compute the following three commitments:
   $$R_1' \leftarrow Y'^{-s_{t_3}} \cdot g^\bar{s} \cdot h_0^{s_y} \cdot \prod_{i \in D} h_i^{s_{a_i}} \cdot g_1^c \cdot \prod_{i \in D} h_i^{c-a_i},$$
   $$R_2' \leftarrow T_1^{-s_x} \cdot h_0^{s_y} \cdot (T_2/Y')^{-c} L' \leftarrow B^\bar{s} \cdot K^{-c}.$$  
   
   (b) Calculate $c' h \leftarrow H_2("\text{sign}", g, g_1, \{h_i\}_{i=0}^n, T_1, T_2, Y', B, K, R_1', R_2', L')$, where $\hat{B} = B$ if $bsn = \bot$ and $\hat{B} = \bot$ otherwise.
   
   (c) Compute $c' \leftarrow H_1(N_t, m, bsn, D, I, c'_h)$.
4.3 Efficiency Improvement

In this section, we present online/offline DAA signatures and a simple implementation trick of parallel computation to improve the computational efficiency of DAA_{\text{OPT}}. We only describe the efficiency improvement of the sign protocol, but the two methods can be also applied to the join protocol.

**Online/Offline DAA Signatures.** The notion of online/offline signatures was introduced by Even, Goldreich and Micali [EGM96]. We apply the online/offline signing idea into DAA_{\text{OPT}} to obtain fast online signing time. In particular, we transform the sign protocol of DAA_{\text{OPT}} into an online/offline sign protocol, based on the fact that a basename bsn is submitted online by a verifier and a message \( m \) to be signed is determined online (e.g., \( m \) may include the PCR values of the current state of the host system or a nonce \( N_v \) from the verifier).

In the offline phase, TPM \( \mathcal{M}_i \) can pre-compute a commitment \( E = \bar{g}' \), and host \( \mathcal{H}_j \) can pre-compute the following elements: \( T_1, T_2, Y', \bar{E}, R_1, R_2, t_3, \bar{u} \), as they are independent of \( m \) and bsn. That is, the elements marked in the dashed box of Figure 2 can be computed offline. In the online phase, \( \mathcal{H}_j \) firstly computes \( B, K, L \) and \( c_h \), and then \( \mathcal{M}_i \) can generate \( (N_t, s) \) without any costly computation. Finally, \( \mathcal{H}_j \) can rapidly complete the computation of a signature by re-computing a digest \( c \) and generating a proof \( \pi_2 = (c, s, s_x, s_\bar{u}, s_{t_2}, s_{t_3}, \{s_{a_i}\})_{i \in D}, N_i \)) fast. By default, \( \mathcal{M}_i \) and \( \mathcal{H}_j \) would securely delete the intermediate pre-computation results after the signatures are produced. For the case of bsn = \( \perp \), \( \mathcal{H}_j \) could further pre-compute \( B, K, L \) and \( c_h \), at the cost of differentiating that pre-computation results are used to create which type of signatures. In the above online/offline DAA signatures, we assume that the host is allowed to select offline the attribute disclosure \( (D, I) \). If some applications only allow that the attribute disclosure is determined online, then the host has to compute \( \prod_{i \in D} \bar{h}_{i}^{r_{a_i}} \) and \( c_h \) online.

In a straightforward way, a randomness \( r \) is stored inside the TPM \( \mathcal{M}_i \) after a TPM.Commit request and deleted after a TPM.Sign request. However, such implementation is too expensive for the TPM with limited storage, when multiple pre-computations are required. TPM 2.0 [Tru16] provides an alternative efficient implementation without storing the random numbers, which allows us to prepare the pre-computation values for multiple signatures. Roughly, the TPM generates a randomness \( r \) via a Key Derivation Function (KDF) with a secret seed and a counter, and maintains a bit table of fixed size to mark which random numbers have been used. We refer the reader to [Tru16, CL13] for the details.

**Implementation Trick of Parallel Computation.** The TPM is a small discrete chip with independent CPU and memory, and has much less resources than the host. Therefore, when TPM \( \mathcal{M}_i \) is computing a commitment \( E \), host \( \mathcal{H}_j \) can compute in parallel the following elements: \( t_3, \bar{u}, T_1, T_2, Y', \bar{E}, \bar{R}_1, R_2, B, K \), if the number \( u = |D| \) of undisclosed attributes is not very large. By using this implementation trick, the signing time consuming at the host side can be reduced significantly. This trick can be also applied to other DAA schemes. Although this implementation trick is simple, it has not been considered in all existing DAA schemes as best as we know.
4.4 Security Properties of Our DAA Scheme

In this section, we give an informal security analysis to argue the security of our protocol DAA_{OPT}. For every security property as described in Section 3.2, we argue why DAA_{OPT} satisfies it. Note that this is structurally quite different from the actual security proof. In the actual proof, we prove that no environment \mathcal{Z} can distinguish the real world where it is interacting with protocol DAA_{OPT} and adversary \mathcal{A}, from the ideal world where it is interacting with ideal functionality \mathcal{F}_{DAA}^l and simulator \mathcal{S}. Nevertheless, the arguments described here are also involved in the formal security proof.

**Theorem 1 (informal).** The protocol DAA_{OPT} is secure under the DBDH, DDH_{G_1} and q-SDH assumptions in the random oracle model.

**Proof (Sketch).** We argue that DAA_{OPT} is anonymous, unforgeable and non-frameable as follows.

**Anonymity.** The SPK_2 constructed in the sign protocol of DAA_{OPT} is zero-knowledge by programming random oracles H_1 and H_2. Thus, there exists a simulator that can simulate a proof \pi_2 of SPK_2 for any statement, and no adversary can notice the difference. To prove that signatures are unlinkable, we pick a fresh key gsk \leftarrow Z_p for bsn = \perp or a new basename bsn \neq \perp, compute an unlinkable-tag/pseudonym (B, K) with gsk, and simulate a proof \pi_2 of SPK_2 in every signature generation of honest platforms. This is indistinguishable using a hybrid argument, where in the i-th game hop we use a fresh key gsk_i every time that the honest platform signs with bsn_i = \perp (or a new basename bsn_i \neq \perp).

We prove that the i-th game hop is indistinguishable from the (i-1)-th one under the DDH_{G_1} assumption if bsn_i = \perp and the DBDH assumption otherwise. For the case of bsn_i = \perp, given a DDH_{G_1} instance \langle \bar{g}, \bar{g}^\alpha, \bar{g}^\beta, \bar{g}^\chi \rangle with either \chi = \alpha \beta or \chi \leftarrow Z_p, we simulate as follows. We set \bar{g}^\alpha as the TPM’s public key tpk and simulate a proof \pi_t due to the zero-knowledge property of SPK_t, and choose hsk \leftarrow Z_p as the host’s secret key. When signing with bsn_i = \perp, we simulate a proof \pi_2 of SPK_2, and set B = \bar{g}^\beta and K = \bar{g}^\chi \cdot (\bar{g}^\beta)^hsk. If \chi = \alpha \beta, the same key was used to sign, and if \chi \leftarrow Z_p, a fresh key was used. For the case that bsn_i \neq \perp and bsn_i is a new basename, given a DBDH instance \langle g_1, g_2, g_1^\delta, g_2^\beta, C, e(g_1, g_2) \chi \rangle with either \chi = \alpha \beta \delta or \chi \leftarrow Z_p, we simulate as follows. We set g_1^\delta as \bar{g} and the unknown \alpha as the key gsk of the platform. We can choose tsk \leftarrow Z_p as the TPM’s secret key, and pick C \leftarrow G_1 and simulate a proof \pi_h, as SPK_h is zero-knowledge. We also program the random oracle such that H_{G_1}(bsn_i) = g_2^\beta. When signing with bsn_i, we simulate a proof \pi_2 and set K = e(g_1, g_2)^\chi as the pseudonym. If \chi = \alpha \beta \delta, the same key was used to sign, and if \chi \leftarrow Z_p, a fresh key was used. Now, for any signature of honest platforms, an unlinkable-tag/pseudonym is computed using a fresh key, a proof \pi_2 is simulated. Besides, T_1 is uniformly random in G_1\ast, T_2 = T_1^\ast and Y' is uniformly random in G_1, and thus they do not involve any information about the honest platform. Therefore, no adversary could break the anonymity of DAA_{OPT}.

**Unforgeability.** First, we argue that no adversary could forge signatures using a credential cre from a platform with an honest TPM even if the host is corrupted. Signatures in our protocol DAA_{OPT} include the proofs of SPK_2 which prove knowledge of secret key gsk = tsk + hsk. Then, the adversary must know secret key tsk if it uses the credential cre, as SPK_2 is a proof of knowledge. This is infeasible under the Discrete-Logarithm (DL) assumption implied by the assumptions in Theorem 1, where the security analysis is very similar to the one in the non-frameability and omitted here. Second, a platform proves that K = B^{gsk} is constructed correctly using the same key from its credential via SPK_2. If key gsk is added to the revocation list RL, the private revocation check would reject all signatures created by gsk.

Next, we only need to show that no adversary could forge signatures using a credential that were not issued by the honest issuer. We can reduce this to existential unforgeability against adaptive chosen message attacks (EUF-CMA) of the BBS+ signature, which has been proved under the q-SDH assumption [ASM06,
Specifically, for the issuance of a credential, we can extract a platform secret key $gsk$ and a randomness $u'$ from proofs $\pi_t$ and $\pi_h$ of SPK$_t$ and SPK$_h$, and then make a query $gsk$ to the signing oracle and obtain a BBS+ signature $(A, x, u)$. Then we can issue the corresponding credential $(A, x, u - u')$ to the platform. When we extract a platform secret key and credential from a forged signature, the key was not signed by the issuer, then the key and credential must be a forgery of the BBS+ signature scheme.

**Non-frameability.** We argue that an honest platform cannot be framed under the DL assumption, even though the issuer is corrupted. Given a DL instance $(\bar{g}, \bar{g}^\alpha)$, we set $\bar{g}^\alpha$ as the TPM’s public key $tpk$ and pick $hsk \overset{\$}{\leftarrow} Z_p$ as the host’s secret key. Then, we simulate a proof $\pi_t$ of SPK$_t$ by programming the random oracle $H_1$ in every execution of the join or sign protocol associated with the honest platform. If the adversary forges a signature which links to a signature of the honest platform, it must prove knowledge of the secret key $gsk$ of the platform. We can extract the key $gsk$ from the proof $\pi_2$ in the forged signature, and output $gsk - hsk$ as the discrete logarithm $\alpha$ which breaks the DL assumption.

In the full formal security proof as described in Appendix D, we rewind to extract the witnesses from the proofs of SPK$_1$, SPK$_t$, SPK$_h$ and SPK$_2$, which is in line with the security proofs of recent DAA schemes with Fiat-Shamir proofs in the UC model [CDL16a, CCD+17]. Camenisch et al. [CDL16b, CDL16a] also consider that instantiating the SPKs to be online extractable via combining Paillier encryption [Pai99] with Fiat-Shamir proofs [FS86]. However, the instantiation is considerably more expensive, and is not compatible with TPM 2.0. As in [CDL16a, CCD+17], we prove that DAA$_{OPT}$ satisfies the stand-alone security instead of UC security when instantiating the underlying SPKs by Fiat-Shamir proofs and rewinding for extraction. As a result, we require that the join protocol is executed sequentially for the security proof.

## 5 TPM 2.0 Implementation of Our DAA Scheme

We show how to implement our DAA scheme DAA$_{OPT}$ using the TPM commands specified in the TPM 2.0 specification [Tru16]. Specifically, we first give a brief description of the TPM 2.0 commands that will be used to implement DAA$_{OPT}$, and refer the reader to TPM 2.0 [Tru16] for details. Then, we show how to implement DAA$_{OPT}$ using these TPM 2.0 commands.

TPM 2.0 allows different types of signatures (e.g., ECDAA, ECSchnorr and U-Prove) to be obtained by using the same TPM commands. This is achieved by splitting the signing procedure into two TPM commands: the first one is TPM2.Commit() that produces a commitment and the second one is a signing command. The signing command has several versions, dependent on what the signature is used for. As examples, we consider three DAA use cases as follows:

- **Use Case I** (corresponding to APPLICATION I): a DAA signature is used to quote PCR values, and a TPM 2.0 command TPM2.Quote() should be invoked.
- **Use Case II** (corresponding to APPLICATION II): a DAA signature is used to certify a key created by the TPM, and a TPM 2.0 command TPM2.Certify() should be invoked.
- **Use Case III** (corresponding to APPLICATION III): a DAA signature is used to sign an arbitrary message provided by the host, and a TPM 2.0 command TPM2.Sign() should be invoked.

In Use Cases I and II, a message $m$ to be signed consists of two parts: a TPM message $m_t$ (i.e., either PCR values or a TPM key) and a host message $m_h$ (e.g., a nonce from a verifier). In Use Case III, a message $m$ to be signed is totally provided by the host.

### 5.1 Outline of TPM 2.0 Commands

Following the TPM 2.0 specification [Tru16], cryptographic keys are stored in a key hierarchy, which includes a root key, an arbitrary number of layers of storage keys and one layer of leaf keys. Usually, only the
Create

root key is stored inside the TPM. Each other key has a parent key in one layer above this key, and each storage key protects at least one child key. A leaf key is used for encryption/decryption, signing/verification or key exchange. A TPM makes use of a key with the following three items:

- **Key handle**: A key handle is a 32-bit value issued by the TPM when a key is loaded into the TPM. When the key is subsequently used in a command, the handle is taken as input to this command. If more than one key is involved in a command, all handles of these keys are taken as input to the command. The key can be used for multiple commands and when no longer required it can be unloaded and its handle released. After a key handle is released, the key needs to be re-loaded if it needs to be used again.

- **Key name**: The name of an asymmetric key is used for identifying the key externally, and it is a hash digest of the public portion of the key. We use $tpk.name$ to denote the key name of a public key $tpk$.

- **Key blob**: A key stored outside of the TPM is in a format of a key blob that is associated with its parent key PKEY. For an asymmetric key pair, written as $tk = (tpk, tsk)$ with the public and private portions, the key blob is

$$
(tk)^* = ((tsk)_{sk}, tpk, MAC_{MK}((tsk)_{sk}||tpk.name)),
$$

where $(tsk)_{sk}$ is a symmetric-encryption ciphertext on plaintext $tsk$ under the key $sk$, $MAC_{MK}()$ is a message authentication code (MAC) under a key $MK$, and $(sk, MK)$ is derived from the parent key PKEY using a key derivation function, i.e., $(SK, MK) \leftarrow KDF(PKEY, SALT)$; $SALT$ is used to make $PKEY$ reusable. For simplicity, we will omit the salt from $KDF(PKEY, SALT)$ in the rest part of this section.

When $tk$ is used as a TPM DAA signing key or any other TPM signing keys, it has a property named as restricted or unrestricted. A restricted signing key is used to quote PCR values, to certify a TPM key, or to sign a TPM computed hash digest. An unrestricted key can be used to sign any given message. Therefore, a message signed under an unrestricted key cannot be claimed that this is a set of PCR values, a key created by the TPM etc. A TPM key $tk$ must be restricted for Use Cases I and II, and it is either restricted or unrestricted for Use Case III.

In the following description of TPM 2.0 commands, we continue using $G_1 = \langle g \rangle$ to denote a group of prime order $p$ with a fixed generator $g$. Let $H : \{0, 1\}^* \rightarrow \mathbb{Z}_p$ be a cryptographic hash function used by a TPM. To implement DAA$_{OPT}$, we recommend using the following TPM 2.0 commands:

- Both TPM2_Create() and TPM2_CreatePrimary() are used to create a TPM key $tk = (tpk, tsk)$. For TPM2_Create() : the TPM does the following:
  1. Choose a fresh secret key $tsk \xleftarrow{\$} \mathbb{Z}_p$ and compute a public key $tpk \leftarrow g^{tsk}$.
  2. Set a restricted or unrestricted attribute for the key.
  3. Generate and output a key blob $(tk)^*$.

  For TPM2_CreatePrimary() instead of creating a key from a random number, it is created from a TPM secret seed using a KDF. To simplify the writing, we will use TPM2_Create() only in the remaining part of this paper.

- TPM2_Load() is used to load a key into the TPM. On input TPM2_Load($(tk)^*$) : the TPM takes as input a key blob $(tk)^*$ and its parent key handle, from that the TPM finds the parent key PKEY, which must have already been loaded into the TPM. The TPM then generates $(sk, MK) \leftarrow KDF(PKEY)$, computes $tpk.name$ from $tpk$ and checks the validity of $MAC_{MK}((tsk)_{sk}||tpk.name)$. The TPM decrypts $(tsk)_{sk}$, and checks if $(tpk, tsk)$ forms a valid key pair. If the checks pass, the TPM outputs a key handle $tk.handle$ along with the key name $tpk.name$. Now $tk$ is stored inside the TPM and can be used for future operations.
- Several TPM 2.0 hash commands allow a TPM to compute a hash digest with different message lengths. If the message is not longer than one hash block, use TPM2_Hash(). Otherwise, use a set of commands to handle sequences. In this paper, we use TPM2_Hash() only to implement DAAOPT. On input TPM2_Hash(msg) : with a message msg given by the host, the TPM does the following:
  1. Check that the first octets of message msg are not “TPM_GENERATED_VALUE”.
  2. If the check passes, compute a digest \( c_t \leftarrow H(msg) \) and a “TPM_TK_HASHCHECK” ticket \( \tau \) which is a MAC on message \( c_t \).
  3. Output \((c_t, \tau)\).

The TPM also has an internal hash operation that can handle a message \( m_t \) generated by the TPM, such as PCR values or a TPM key. In this case, the message will start with the label “TPM_GENERATED_VALUE”.

- TPM2_Commit() is the first TPM command in the TPM signing procedure.
  On input TPM2_Commit\((P_1, s_2, y)\) : the TPM executes as follows:
  1. If \( P_1 \neq \bot \), check whether \( P_1 \in G_1 \) or not.
  2. If \((s_2, y) \neq \bot \), compute \( x = H(s_2) \) for a cryptography hash function \( H \), and then set \( B \leftarrow (x, y) \) and check whether \( B \in G_1 \) or not. The string \( s_2 \) may contain a basename bsn for DAA.
  3. If the above checks fail, output an error and abort.
  4. Set \( E, K, L \leftarrow \bot \).
  5. Pick \( r \overset{\$}{\leftarrow} \mathbb{Z}_p \) and store \((ctr, r)\) in a list \( Committed \), where \( ctr \) is a counter used to retrieve \( r \). Here, we assume that \( Committed \) and \( ctr \) are initialized as \( \emptyset \) and \( 0 \) respectively.
  6. If \( P_1 \neq \bot \), compute \( E \leftarrow P_1^r \).
  7. If \((s_2, y) \neq \bot \), compute \( K \leftarrow B_{tsk} \) and \( L \leftarrow B^r \).
  8. If \( P_1 = \bot \) and \((s_2, y) = \bot \), compute \( E \leftarrow \overline{g}^r \).
  9. Increment \( ctr \) and output \((E, K, L, ctr)\).

The second TPM command in the TPM signing procedure, as we discussed before, has three cases: TPM2_Sign(), TPM2_Certify() and TPM2_Quote(), dependent on what the signature is used for.

- On input TPM2_Sign\((c_t, \tau, ctr)\) : the TPM executes as follows:
  1. If the TPM key is unrestricted and \( \tau = \bot \), check that the size of \( c_t \) is equal to the output length of \( H \).
  2. Otherwise, check the validity of ticket \( \tau \).
  3. If the above check passes, execute the following CryptSign\((c_t, ctr)\) function: \(^5\)
     (a) Retrieve a pair \((ctr, r)\) and remove it from list \( Committed \), output an error if no such pair was found.
     (b) Pick \( N_t \overset{\$}{\leftarrow} \{0, 1\}^{l_n} \) and compute \( c \leftarrow H(N_t, c_t) \).
     (c) Compute \( s \leftarrow r + c \cdot tsk \mod p \) and output \((N_t, s)\).

- On input TPM2_Certify\((qualifyData, keyhandle, ctr)\) : Given an extra data \( qualifyData \), a \( keyhandle \) and a counter \( ctr \), the TPM retrieves a public key \( m_t \) using the key handle \( keyhandle \), and does the following:
  1. Compute a hash digest \( c_t \leftarrow H(qualifyData, H("TPM_GENERATED_VALUE", m_t)) \).
  2. Execute the CryptSign\((c_t, ctr)\) function as described in TPM2_Sign() to obtain \((N_t, s)\).
  3. Output \((N_t, s)\).

- On input TPM2_Quote\((qualifyData, PCRselect, ctr)\) : the TPM executes as follows:
  1. Select the corresponding PCR values \( m_t \) from the PCR according to \( PCRselect \), and compute a hash digest of \( m_t \) denoted by \( pcrDigest \).
  2. Compute a hash digest \( c_t \leftarrow H(qualifyData, H("TPM_GENERATED_VALUE", pcrDigest)) \).
  3. Execute the CryptSign\((c_t, ctr)\) function as described in TPM2_Sign() to obtain \((N_t, s)\).
  4. Output \((N_t, s)\) and \( pcrDigest \).

\(^5\) Note that a nonce \( N_t \) has been added to the CryptSign function in the revision 01.38 of TPM 2.0 specification [Tru16].
TPM2_ActivateCredential() is used to allow the DAA issuer to authenticate the public key $tpk$ of a TPM and to issue a credential $cre'$ and a number of attributes $attrs$ confidentially in the join protocol by using the endorsement key $ek = (epk, esk)$ of the TPM. Given an endorsement public key $epk$ and a TPM public key $tpk$, the issuer generates a fresh secret seed $seed$ and a fresh symmetric encryption key $k$, and then computes an encryption blob $(ct)^*$ as follows:

$$(ct)^* = ENC_{epk}(tpk, k) = ((seed)_{epk}, (k)_{sk}, MAC_{MK}((k)_{sk}||tpk.name))$$

where $(seed)_{epk}$ is a public-encryption ciphertext on message $seed$ under public key $epk$, $(sk, mk) \leftarrow KDF(seed)$ and $(k)_{sk}$ is a symmetric-encryption ciphertext on message $k$ under secret key $sk$. Additionally, the issuer generates a symmetric-encryption ciphertext $(cre'||attrs)_k$ on message $cre'||attrs$ under key $k$.

On input TPM2_ActivateCredential$(ek.handle, tk.handle, (ct)^*)$: the TPM executes as follows:

1. Retrieve a secret key $esk$ using a handle $ek.handle$, and decrypt $(seed)_{epk}$ with $esk$ to obtain $seed$.
2. Derive a symmetric key $sk$ and a MAC key $mk$, i.e., $(sk, mk) \leftarrow KDF(seed)$.
3. Retrieve a key name $tpk.name$ using a handle $tk.handle$ and compute $MAC_{sk}((k)_{sk}||tpk.name)$.
4. Check whether the computed MAC value matches the one in encryption blob $(ct)^*$.
5. If the check fails, output an error. Otherwise, decrypt $(k)_{sk}$ with $sk$ and output $k$.

When the TPM releases $k$, the host can decrypt $(cre'||attrs)_k$ with key $k$ to obtain a credential $cre'$ and its attributes $attrs$ from the issuer.

5.2 The TPM 2.0 Implementation of Our DAA Scheme

For the sake of simplicity, we consider that a key $tk = (tpk, tsk)$ is always loaded into the TPM via TPM2_Load(), before it would be used. Thus, we could omit the invocation of TPM2_Load() in the description of implementing our scheme DAAOPT.

Below, we present how to use the TPM 2.0 commands described in Section 5.1 to implement the TPM.Create, TPM.Commit and TPM.Sign procedures in DAAOPT.

- For the TPM.Create procedure, the host $H_j$ calls a TPM command TPM2_Create(), and the TPM $M_i$ outputs a key blob $(tk)^*$ including a public key $tpk$.
- For the TPM.Commit procedure, $H_j$ calls a TPM command TPM2_Commit(⊥, ⊥), and $M_i$ outputs a commitment $E = g^r$ and a counter $ctr$.
- For the TPM.Sign procedure in the join protocol, we consider two cases relying on whether a signing key $tsk$ is restricted or not.
  1. If the TPM secret key $tsk$ is restricted, host $H_j$ calls a TPM command TPM2_Hash($c_h$), and TPM $M_i$ outputs a digest $c_t$ and a ticket $\tau$. Then, $H_j$ calls TPM2_Sign($c_t, \tau, ctr$), and $M_i$ outputs $(N_i, s)$.
  2. If the TPM secret key $tsk$ is unrestricted, host $H_j$ calls TPM2_Sign($c_h$, ⊥, $ctr$), and TPM $M_i$ outputs $(N_i, s)$.
- For the TPM.Sign procedure in the sign protocol, we consider three DAA use cases as follows.
  1. For Use Case I, host $H_j$ first computes a hash digest $d_h \leftarrow H(\text{"qualifyingData"}, m_h, bsn, D, I, c_h)$, and then calls TPM2_Quote($d_h, PCRselect, ctr$). TPM $M_i$ outputs $(N_i, s)$ along with $pcrDigest$.
  2. For Use Case II, the host $H_j$ loads the key to be certified into the TPM by calling a TPM command TPM2_Load() to receive a key handle $keyhandle$. Then, $H_j$ computes a hash digest $d_h \leftarrow H(\text{"qualifyingData"}, m_h, bsn, D, I, c_h)$ and calls a TPM command TPM2_Certify($d_h, keyhandle, ctr$). The TPM $M_i$ outputs $(N_i, s)$.
  3. For Use Case III, we distinguish which type the TPM secret key $tsk$ belongs to.
(a) If $tsk$ is restricted, host $H_j$ computes $d_h \leftarrow H(\text{"hostMessage"}, m, bsn, D, I, c_h)$, and then calls a TPM command TPM2_Hash$(d_h)$. TPM $M_i$ outputs a digest $c_t$ and a ticket $\tau$. Then, $H_j$ calls a TPM command TPM2_Sign$(c_t, \tau, ctr)$ and $M_i$ outputs $(N_t, s)$.

(b) If $tsk$ is unrestricted, host $H_j$ can compute $c_t \leftarrow H(m, bsn, D, I, c_h)$ by itself. Then, $H_j$ calls a TPM command TPM2_Sign$(c_t, 1, ctr)$ and $M_i$ outputs $(N_t, s)$.

In the above TPM 2.0 implementation, we let the host compute the hash digest of $m_h$ (or $m$) and $bsn$, $D, I, c_h$ to achieve better performance. This has no impact for the security even if the host is corrupted, since the simulator controls the random oracle $H$, can extract a tuple $(m_h/m, bsn, D, I, c_h)$ from the $H$-list maintained by itself, and send the tuple to the ideal functionality in the security proof. Depending on the use case and the type of the protocol, the hash function $H_1$ used by the TPM in the construction of our scheme $\text{DAA}_{\text{OPT}}$ has different ways of implementation, which would be explicit from the application scenario and that either the join protocol or the sign protocol is executed by a platform.

Below, we show how to use $\text{TPM2}_{\text{ActivateCredential}}()$ to establish a semi-authenticated channel between the TPM and issuer in the join protocol, by following the description in [CL13].

1. A host $H_j$ sends an endorsement public key $\text{epk}$ and a public key $\text{tpk}$ to an issuer $I$ as the $\text{JOIN}$ request.
2. Upon receiving $\text{epk}$ and $\text{tpk}$, $I$ checks the validity of $\text{epk}$ via validating the certificate of $\text{epk}$. If the check passes, $I$ picks a nonce $N_I \leftarrow \{0, 1\}^\lambda$ and generates an encryption blob $(ct_1)^* \leftarrow \text{ENC}_{\text{epk}}(\text{tpk}, N_I)$. Then $I$ sends $(ct_1)^*$ to $H_j$.
3. $H_j$ calls $\text{TPM2}_{\text{ActivateCredential}}(\text{ek.handle}, \text{tk.handle}, (ct_1)^*)$ and the TPM $M_i$ outputs $N_I$, where endorsement key $\text{ek}$ and TPM key $\text{tk}$ are assumed to have been loaded into $M_i$ via $\text{TPM2}_{\text{Load}}()$.
4. Upon receiving a tuple $(C, \pi_t, \pi_h)$ and a nonce $N_I$, $I$ checks the validity of $N_I$ and proofs $\pi_t, \pi_h$. If the check passes, $I$ generates a credential $\text{cre}' \leftarrow (A, x, u^n)$ and a number of attributes $\text{attrs} = (a_1, \ldots, a_n)$. Then $I$ creates a fresh key $k$, and generates an encryption blob $(ct_2)^* \leftarrow \text{ENC}_{\text{epk}}(\text{tpk}, k)$ and a symmetric-encryption ciphertext $sc \leftarrow (\text{cre}'||\text{attrs})_k$. $I$ sends $((ct_2)^*, sc)$ to $H_j$.
5. $H_j$ calls $\text{TPM2}_{\text{ActivateCredential}}(\text{ek.handle}, \text{tk.handle}, (ct_2)^*)$ and the TPM $M_i$ outputs $k$. $H_j$ decrypts ciphertext $sc$ with key $k$ and obtains $\text{cre}' = (A, x, u^n)$ and $\text{attrs} = (a_1, \ldots, a_n)$.

### 6 Performance Evaluation

In this section, we first provide the benchmark results on an Infineon TPM 2.0 chip, which can be used to evaluate the TPM signing performance for three use cases of DAA considered by us. Then, we give the experimental results for the host signing and verification efficiency on a laptop for our DAA scheme without considering attributes.

Next, we compare the efficiency of our scheme $\text{DAA}_{\text{OPT}}$ with the existing DAA schemes supported by the TPM 2.0 specification [Tru16]. We use CPS, BL and CDL to denote these DAA schemes, where CPS is based on the LRSW-DAA scheme [CPS10], BL is based on the SDH-DAA scheme [BL10b], and CDL is the SDH-DAA scheme in the full version of [CDL16a] but removes the session identifiers for UC security. In particular, we evaluate the efficiency of BL when considering the efficiency improvement of this scheme using this optimization in [CU15]. We also compare the efficiency of these DAA schemes with the functionality extension of attributes, where CPS and BL can be extended to support attributes following [CU15], and CDL provides the support of attributes by itself. For fairness, we consider that all the DAA schemes let the host store all attributes and the TPM protect the secret key only. We refer the reader to [CL13, CU15] for the implementation details of CPS and BL using the TPM 2.0 commands. In all our comparisons, we can directly obtain the efficiency of standard DAA schemes (without attributes) when setting both the number of attributes $n$ and the number of undisclosed attributes $u$ as zero.

We also give the comparison of concrete sizes of credentials and signatures over two kinds of BN curves recommended by the TCG. We omit the efficiency comparison of the join protocol, since the join protocol is executed much less frequently than the sign protocol or the verification algorithm.
<table>
<thead>
<tr>
<th>TPM2_Commit()†</th>
<th>TPM2_Qoute()</th>
<th>TPM2_Certify()</th>
</tr>
</thead>
<tbody>
<tr>
<td>87.4</td>
<td>50.2</td>
<td>23.0</td>
</tr>
<tr>
<td>87.6</td>
<td>50.1</td>
<td></td>
</tr>
<tr>
<td>217.1</td>
<td>49.8</td>
<td></td>
</tr>
<tr>
<td>152.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>217.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† The running time is in milliseconds (ms) and averaged over 150 random instances.

6.1 Benchmark Results and Performance of Our DAA Scheme

We present the benchmark results for the following TPM 2.0 commands:

TPM2_Commit(), TPM2_Qoute(), TPM2_Certify(), TPM2_Sign(), TPM2_Hash(),

by implementing them on an Infineon TPM 2.0 chip with vendor ID IFXSLB9670. The TPM 2.0 chip is installed on a module designed for the Raspberry Pi. The program used to obtain the timings was running on a Raspberry Pi 3 fitted with the Infineon TPM module, and was compiled using g++ 6.3.0. The Raspberry Pi 3 is equipped with a 64-bit ARMv7 processor, but the operating system Raspbian (version 4.14.30) runs in 32-bit mode. We adopt SHA256 to implement the hash function \( H \) used by the TPM 2.0 chip.

The TCG recommended two types of Barreto-Naehrig (BN) curves [BN06] (i.e., BN_P256 and BN_P638) to support bilinear pairings. These BN elliptic curves have the form \( y^2 = x^3 + b \) with embedding degree 12, where \( b = 3 \) for BN_P256 and \( b = 257 \) for BN_P638. According to the state-of-art analysis results [KB16, BD18], the BN_P256 curve only achieves about 100-bit security level, and the BN_P638 curve will provide more than 128-bit security level. Currently, only the BN_P256 curve is implemented on the TPM 2.0 chips, and the implementation of the BN_P638 curve has not been available for TPM 2.0 chips. Therefore, we only consider the BN_P256 curve to evaluate the computational efficiency of our DAA scheme. However, we will adopt both the BN_P256 and BN_P638 curves to evaluate the sizes of credentials and signatures. In particular, when considering the point compression technique, the size (in bits) of an element in group \( Z_p \), \( G_1 \) and respective \( G_T \) is shown as follows: \( |Z_p| = 256 \), \( |G_1| = 257 \) and \( |G_T| = 3072 \) over the BN_P256 curve; and \( |Z_p| = 638 \), \( |G_1| = 639 \) and \( |G_T| = 7656 \) over the BN_P638 curve.

We consider five cases for the implementation of the TPM2_Commit() command:

Case 1. No input, i.e., \( P_1 = \bot \) and \( (s_2, y) = \bot \). Our scheme DAA_{OPT} uses TPM2_Commit() in this case.

Case 2. A single elliptic curve point \( P_1 \neq \bot \) is input, but \( (s_2, y) = \bot \). The LRSW-DAA scheme CPS uses this case with a random \( P_1 \) to generate fully anonymous signatures.

Case 3. Both a curve point \( P_1 \neq \bot \) and \( (s_2, y) \neq \bot \) are input, and \( P_1 \) is a random point. The LRSW-DAA scheme CPS uses this case to produce pseudonymous signatures.

Case 4. Only \( (s_2, y) \neq \bot \) is input and \( P_1 = \bot \). In this case, only \( K = B^{tsk}, L = B^r \) are output. As far as we know, no DAA schemes uses this case.

Case 5. Both a curve point \( P_1 \neq \bot \) and \( (s_2, y) \neq \bot \) are input, and \( P_1 \) is a fixed base point. The SDH-DAA schemes BL and CDL use this case to generate signatures for both bsn = \( \bot \) and bsn \( \neq \bot \).

Our benchmark results for these TPM 2.0 commands are shown in Table 1. For TPM2_Qoute(), only one PCR value is selected. The running time of TPM2_Certify() does not include the time creating a public key to be signed, where the public key is assumed to be created offline. These benchmark results will be helpful to evaluate the TPM performance of other DAA schemes.
Table 2: Performance of our DAA scheme without attributes

<table>
<thead>
<tr>
<th>DAAOPT</th>
<th>TPM signing</th>
<th>Host signing</th>
<th>Platform signing</th>
<th>Verify (ms)</th>
<th>Signature size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Online</td>
<td>Total</td>
<td>Opt.*</td>
<td>Online</td>
</tr>
<tr>
<td>bsn = ⊥</td>
<td>137.2</td>
<td>49.8</td>
<td>14.1</td>
<td>1.1</td>
<td>0.2</td>
</tr>
<tr>
<td>bsn ≠ ⊥</td>
<td>137.2</td>
<td>49.8</td>
<td>25.9</td>
<td>6.8</td>
<td>14.8</td>
</tr>
</tbody>
</table>

* The running time in the columns of “Opt.” considers the optimization of parallel computation.

Table 3: Efficiency comparison of the sign protocol and verification algorithm among DAA schemes

<table>
<thead>
<tr>
<th>DAA Scheme¹</th>
<th>Sign protocol</th>
<th>Verification†</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TPM signing</td>
<td>Host signing</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>Online</td>
</tr>
<tr>
<td>CPS</td>
<td>1E_{G_1}</td>
<td>H + mul</td>
</tr>
<tr>
<td></td>
<td>1E_{G_1}^{2+n} + 1E_{G_1^{opt}} + 1E_{G_2^{opt}} + 4P + [2P]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3E_{G_1}</td>
<td>3E_{G_1}</td>
</tr>
<tr>
<td></td>
<td>1E_{G_1}^{2+n} + 1E_{G_1^{opt}} + 1E_{G_2^{opt}} + 4P + [2P]</td>
<td></td>
</tr>
<tr>
<td>BL</td>
<td>3E_{G_1}</td>
<td>H + mul</td>
</tr>
<tr>
<td></td>
<td>1E_{G_1}^{2+n} + 1E_{G_1^{opt}} + 2P</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3E_{G_1}</td>
<td>3E_{G_1}</td>
</tr>
<tr>
<td></td>
<td>1E_{G_1}^{2+n} + 1E_{G_1^{opt}} + 2P</td>
<td></td>
</tr>
<tr>
<td>CDL</td>
<td>3E_{G_1}</td>
<td>H + mul</td>
</tr>
<tr>
<td></td>
<td>1E_{G_1}^{2+n} + 1E_{G_1^{opt}} + 2P</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3E_{G_1}</td>
<td>3E_{G_1}</td>
</tr>
<tr>
<td></td>
<td>1E_{G_1}^{2+n} + 1E_{G_1^{opt}} + 2P</td>
<td></td>
</tr>
<tr>
<td>DAAOPT</td>
<td>1E_{G_1}</td>
<td>H + mul</td>
</tr>
<tr>
<td></td>
<td>1E_{G_1}^{2+n} + 1E_{G_1^{opt}} + 2P</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1E_{G_1}</td>
<td>H + mul</td>
</tr>
<tr>
<td></td>
<td>1E_{G_1}^{2+n} + 1E_{G_1^{opt}} + 2P</td>
<td></td>
</tr>
</tbody>
</table>

¹ E_{G_i} (i ∈ {1, 2, T}); the cost of the product of m powers in G_i; E_{G_i}: the cost of one exponentiation in G_i; P: the cost of a bilinear pairing; E_{G_i^{opt}} (i ∈ {1, 2}): the cost of one m-multi-exponentiations in group G_i with the size of the exponents being t such as a half of the size of p. n is the total number of attributes and u denotes the number of undisclosed attributes.

† The row for bsn = ⊥ (resp., bsn ≠ ⊥) represents the cost of a fully anonymous (resp., pseudonymous) signature.

[| X | denotes the incremental computational cost X when considering the support of attributes.

We use the Apache Milagro Cryptographic Library (AMCL) [SMBA19] with the BN_P256 curve and an ate pairing to evaluate the performance of the host signing and the verification algorithm for our DAA scheme DAAOPT without considering attributes. We obtained the running time on a laptop with 1.80GHz Intel Core i7-8550U CPU averaged over 150 random instances. We also measured the online signing time and the signing time with an optimization of parallel computation on the host side. The performance of our scheme DAAOPT is described in Table 2. From this table, we can see that DAAOPT provides an attractive signing efficiency and a reasonable signature size as a trade-off of faster signing.

6.2 Efficiency Comparison of DAA Schemes Supported by TPM 2.0

We first give a theoretical comparison by counting the number of costly operations in each DAA scheme, since the costly operations dominate the performance of DAA schemes. In Table 3, we compare the efficiency of the signing protocol and verification algorithm of the DAA schemes supported by TPM 2.0, where the online signing cost for the host is obtained by assuming that attribute disclosure (D, I) is allowed to be selected offline. We count the computational costs of a hash function and a modular multiplication r + c · tsk mod p for the TPM (denoted by H and mul) in Table 3, since they are still expensive for the TPM. In contrast, these computational costs are ignored for the host signing and verification algorithm, as they are very fast and much more efficient than exponentiations for the host and verifier with much more powerful computational capability.
From Table 3, we can see that our scheme DAA\textsubscript{OPT} is the only scheme achieving the fully optimal TPM signing efficiency. The (online) signing efficiency of the TPM for the pseudonymous signature mode in DAA\textsubscript{OPT} significantly outperforms other DAA schemes. The verification cost in Table 3 does not include private key revocation. In terms of the efficiency of private key revocation, DAA\textsubscript{OPT} has the same efficiency as other DAA schemes for fully anonymous signatures, and provides the same on-line efficiency as other schemes for pseudonymous signatures, as the verifier can pre-compute $e(\bar{g}, H_{G_2}(\text{bsn}))^{gsk}$ for each $gsk \in \mathcal{RL}$.

We measured the speed of AMCL [SMBA19] with the BN\textsubscript{P256} curve and an ate pairing on a laptop with 1.80GHz Intel Core i7-8550U CPU. We found that $1E_{G_1}$, $1E_{G_2}$, $1E_{G_T}$ and $1P$ take about 0.23 ms, 0.51 ms, 0.72 ms and 2.0 ms respectively. Using these benchmarks along with the running time for the TPM 2.0 commands in Table 1, we compare the computational efficiency of our DAA scheme with the known DAA schemes supported by TPM 2.0 in Figure 3. Our comparison does not consider attributes, but considers the optimization of parallel computation between the TPM and host. In Figure 3, we estimate the running time for the host signing and verification algorithm. The estimated time is not exact, but is enough to compare the efficiency of DAA schemes. This is reasonable for comparison as the fast operations (e.g., hash function and modular multiplication) have little impact on the running time over the host and verifier platforms with powerful computational capabilities. Specifically, Camenisch et al. [CDL17] used benchmark results to estimate the efficiency of the host signing and verification algorithm for their DAA scheme. Using the benchmark results to estimate the performance of a scheme has also appeared in [AKS12, CDD17].

For the comparison in Figure 3, we set the TPM key as unrestricted, and thus need not to invoke the TPM\textsubscript{2,Hash()} command when signing a message in Use Case III. From Figure 3, it can be seen that our scheme DAA\textsubscript{OPT} is about 2× more efficient than other DAA schemes for the pseudonymous signature mode, and is about 2× faster than the SDH-DAA schemes BL and CDL for the fully anonymous signature mode. In terms of online signing efficiency for the pseudonymous signature mode, DAA\textsubscript{OPT} is about 5× faster than other DAA schemes. DAA\textsubscript{OPT} has the same signing efficiency as CPS for the fully anonymous signature mode, but is more efficient than CPS in terms of the verification efficiency. For the verification efficiency, DAA\textsubscript{OPT} is comparable to the SDH-DAA schemes BL and CDL in both signature modes.

In Table 4, we compare the sizes of credentials and signatures, where the bit-length $\ell_n$ of a nonce $N_t$ is counted as $|Z_p|$. While the SDH-DAA schemes DAA\textsubscript{OPT}, BL and CDL have $O(1)$ credential size, the LRSW-DAA scheme CPS has $O(n)$ credential size. Moreover, when supporting attributes, the incremental size of signatures in the SDH-DAA schemes is much less than CPS. While CDL and DAA\textsubscript{OPT} are provably

---

Our estimation does not consider the optimizations of multi-exponentiations and batch pairings, where they can be applied to all the DAA schemes and further reduce the running time of the host signing and the verification algorithm.
Table 4: Theoretical comparison of the sizes of credentials and signatures*

<table>
<thead>
<tr>
<th>DAA Scheme</th>
<th>Credential Size</th>
<th>Signature Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bsn = \perp</td>
<td>bsn \neq \perp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPS</td>
<td>4</td>
<td>G_1</td>
</tr>
<tr>
<td>BL</td>
<td>1</td>
<td>G_1</td>
</tr>
<tr>
<td>CDL</td>
<td>2</td>
<td>G_1</td>
</tr>
<tr>
<td>DAA_{OPT}</td>
<td>3</td>
<td>G_1</td>
</tr>
</tbody>
</table>

* [G]: the bit-length of an element in group G. n is the total number of attributes and u is the number of undisclosed attributes.

Table 5: Comparison of concrete sizes of credentials and signatures*

<table>
<thead>
<tr>
<th>DAA Scheme</th>
<th>Credential Size (Bytes)</th>
<th>Signature Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bsn = \perp</td>
<td>bsn \neq \perp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPS</td>
<td>129 + 33n</td>
<td>225 + 33n + 32u</td>
</tr>
<tr>
<td>BL</td>
<td>65</td>
<td>289 + 32u</td>
</tr>
<tr>
<td>CDL</td>
<td>129</td>
<td>385 + 32u</td>
</tr>
<tr>
<td>DAA_{OPT}</td>
<td>193</td>
<td>385 + 32u</td>
</tr>
</tbody>
</table>

* In each section, the left column denotes the size over the BN_P256 curve and the right column represents the size over the BN_P638 curve.

secure in the UC security model, no rigorous security proof is known for CPS and BL with/without attributes in a valid security model [BFG+13, CU15, CDL16b, CDL16a].

In Table 5, we compare the concrete sizes of credentials and signatures, where the sizes involving n and u are the incremental sizes when the support of attributes is required. The LRSW-DAA scheme CPS has the smallest size for signatures without considering attributes, but the largest overhead to support attributes. For the SDH-DAA schemes providing a better support of attributes, BL has the smallest sizes for credentials and signatures, but has not a rigorous security proof in a valid security model. In terms of the signature size, our scheme DAA_{OPT} is the same as CDL for a fully anonymous signature mode, but larger than CDL for a pseudonymous signature mode. This is a trade-off from the faster signing time demonstrated in Figure 3. The signature size in DAA_{OPT} is acceptable, especially for the applications that only one signature is sent in every transaction. The applications include remote attestation, anonymous subscription services, anonymous V2X, FIDO authentication etc. In the applications, the signing time is more crucial than the signature size, as only one signature needs to be sent, where a pseudonymous signature in DAA_{OPT} has at most 0.7KB/1.7KB (resp., 1KB/2.5KB) when u = 0 (resp., u = 10).

7 Conclusion

We have proposed the first DAA scheme with fully optimal TPM signing efficiency. The full optimization means that in both the fully anonymous mode and pseudonymous mode the TPM’s signing cost is equal to the cost of generating a traditional digital signature, e.g. an ECSchnorr signature. We have proved that our DAA scheme is secure under the DDH, DBDH and q-SDH assumptions in the UC security model [CDL16b, CDL16a] and the random oracle model. To demonstrate the performance of our DAA scheme, we implemented three DAA use cases using existing TPM 2.0 commands. Our scheme provides significantly better signing efficiency than other known DAA schemes supported by TPM 2.0. We have also extended our DAA scheme to support signature-based revocation and to guarantee privacy in the presence of subverted TPMs.
Acknowledgements

Kang Yang is supported by the National Natural Science Foundation of China (Nos. 61932019, 61802021). Liqun Chen is supported by the EU Horizon 2020 research and innovation program under grant agreement No. 779391 (FutureTPM). Zhenfeng Zhang is supported by the National Key Research and Development Program of China (No. 2017YFB0802504), and by the National Natural Science Foundation of China (No. U1536205). The authors would like to thank Jiang Zhang and anonymous reviewers for their helpful comments.

References


A Formal Security Model for DAA

In this section, we define the UC security model [CDL16b, CDL16a]. 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 $A$, who may communicate freely with $Z$. In the ideal world, honest parties forward their inputs to an ideal functionality $F$, which then internally performs the defined task and generates the parties’ outputs that are forwarded to $Z$ by them.

Informally, we say that a protocol $Π$ securely realizes an ideal functionality $F$, if the real world in which $Π$ is used is as secure as the ideal world where $F$ is used. To prove the statement, one needs to show that for every adversary $A$ mounting an attack in the real world, there exists an ideal world adversary (often called simulator) $S$ that performs an equivalent attack in the ideal world. More precisely, $Π$ securely realizes $F$ if for every adversary $A$, there exists a simulator $S$, such that no environment $Z$ can distinguish interacting with the real world with $Π$ and $A$ from interacting with the ideal world with $F$ and $S$.

Now, we review the formal definition [CDL16b] of ideal functionality $F_{\text{daa}}^l$ with static corruption, meaning that the adversary decides beforehand which parties are corrupted and makes the information known to the ideal functionality. We further extend the definition to support the functionality of attributes following the modification [CDL16a].

In the UC model, different instances of the protocol are distinguished with session identifiers. Following [CDL16b], we use session identifiers of the form $\text{sid} = (\mathcal{I}, \text{sid}')$ for some issuer $\mathcal{I}$ and a unique string $\text{sid}'$. To allow multiple sub-sessions for the join and sign related interfaces, we use unique sub-session identifiers $\text{jsid}$ and $\text{ssid}$. $F_{\text{daa}}^l$ is parametrized by a leakage function $l : \{0, 1\}^* \rightarrow \{0, 1\}^\ast$, which models the information leakage that occurs in the communication between a TPM $M_i$ and a host $H_j$. As $F_{\text{daa}}^l$ is extended to support attributes, we have parameters $n$ and $\{\mathcal{A}_i\}_{1 \leq i \leq n}$, where $n$ is the number of attributes that every membership credential includes and $\mathcal{A}_i$ is the set from which the $i$-th attribute is taken. Following [CDL16a], a parameter $P$ is used to describe which proofs over the attributes a platform can make. Using this generic method, the ideal functionality capture both simple protocols that only support selective attribute disclosure and more advanced protocols that support arbitrary predicates. Every value $\hat{p} \in P$ is a predicate over the attributes, i.e., $\hat{p} : \mathcal{A}_1 \times \cdots \times \mathcal{A}_n \rightarrow \{0, 1\}$.

Below, we show several algorithms $(\text{ukgen}, \text{sig}, \text{ver}, \text{link}, \text{idem})$ which are provided by the simulator and will be used in the ideal functionality.

- $gsk \leftarrow \text{ukgen}()$ will be used to generate a secret key $gsk$ for an honest platform.
- $\sigma \leftarrow \text{sig}(gsk, m, \text{bsn}, \hat{p})$ takes as input $gsk$, a message $m$, a basename $\text{bsn}$ and a predicate $\hat{p}$, and outputs a signature $\sigma$. The algorithm will be used for honest platforms.
- $f \leftarrow \text{ver}(m, \text{bsn}, \sigma, \hat{p})$ takes as input a message $m$, a basename $\text{bsn}$, a signature $\sigma$ and a predicate $\hat{p}$, and then outputs $f = 1$ if $\sigma$ is valid on $m$ w.r.t. $\text{bsn}$ and $\hat{p}$ and $f = 0$ otherwise. This algorithm will be used in the VERIFY interface.
- $f \leftarrow \text{link}(m_0, \sigma_0, m_1, \sigma_1, \text{bsn})$ takes as input two message/signature pairs $(m_0, \sigma_0)$ and $(m_1, \sigma_1)$ and a basename $\text{bsn}$, and outputs $f = 1$ if both signatures were created by the same platform and $f = 0$ otherwise. This algorithm will be used in the LINK interface.
- $f \leftarrow \text{idem}(m, \text{bsn}, \sigma, gsk)$ takes as input a message $m$, a basename $\text{bsn}$, a signature $\sigma$ and a secret key $gsk$, and outputs $f = 1$ if $\sigma$ is a signature on $m$ w.r.t. basename $\text{bsn}$ under key $gsk$ and $f = 0$ otherwise. This algorithm will allow $F_{\text{daa}}^l$ to perform multiple consistency checks whenever a new key $gsk$ is created or provided by the simulator.

While ukgen and sig are probabilistic, the other three algorithms are deterministic. Besides, the link algorithm has to be symmetric, i.e., for all inputs it must hold that

$$\text{link}(m_0, \sigma_0, \hat{p}_0, m_1, \sigma_1, \hat{p}_1, \text{bsn}) = \text{link}(m_1, \sigma_1, \hat{p}_1, m_0, \sigma_0, \hat{p}_0, \text{bsn}).$$

Note that algorithms ver and link only assist the ideal functionality for signatures which are not produced by $F_{\text{daa}}^l$ itself. For signatures generated by the functionality, $F_{\text{daa}}^l$ enforces correct verification and linkage using its internal records.
Setup

1. Issuer Setup. On input (SETUP, sid) from issuer I.
   - Verify that sid = (I, sid') and output (SETUP, sid) to S.
2. Set Algorithms. On input (ALG, sid, ukgen, sig, ver, link, identify) from S.
   - Check that ver, link and identify are deterministic (i).
   - Store (sid, ukgen, sig, ver, link, identify) and output (SETUPDONE, sid) to I.

Join

3. Join Request. On input (JOIN, sid, jsid, M_i) from host H_j.
   - Create a join session record (jsid, M_i, H_j, ∅, status) with status ← request.
   - Output (JOINSTART, sid, jsid, M_i) to S.
4. Join Request Delivery. On input (JOINSTART, sid, jsid) from S.
   - Update the session record (jsid, M_i, H_j, ∅, status) to status ← delivered.
   - Abort if I or M_i is honest and a record (M_i, ∅, ∅, status) ∈ Members already exists (ii).
   - Output (JOINPROCEED, sid, jsid, M_i) to I.
5. Join Proceed. On input (JOINPROCEED, sid, jsid, attrs) from I with attrs ∈ A_1 × · · · × A_n.
   - Update the session record (jsid, M_i, H_j, ∅, status) to ∅ ← attrs and status ← complete.
   - Output (JOINCOMPLETE, sid, jsid, attrs') to S, where attrs' ← ∅ if M_i and H_j are honest
     and attrs' ← attrs otherwise.
6. Platform Key Generation. On input (JOINCOMPLETE, sid, jsid, gsk) from S.
   - Look up record (jsid, M_i, H_j, attrs, status) with status = complete.
   - If M_i and H_j are honest, set gsk ← ∅.
   - Else verify that the provided gsk is eligible via checking
     - CheckGskHonest(gsk) = 1 (iii) if M_i is honest and H_j is corrupted, or
     - CheckGskCorrupt(gsk) = 1 (iv) if M_i is corrupted.
   - Add (M_i, H_j, gsk, attrs) into Members and output (JOINED, sid, jsid) to H_j.

Fig. 4: The Setup and Join Related Interfaces of Ideal Functionality \( F^I_{\text{daa}} \). The roman numbers are the labels for the different checks made within the ideal functionality and will be used as reference in the security proof.

We provide the detailed definition of ideal functionality \( F^I_{\text{daa}} \) in Figure 4 and Figure 5, and refer the reader to [CDL16b, CDL16a] for the explanations of \( F^I_{\text{daa}} \) and the argument of why \( F^I_{\text{daa}} \) realizes the desired security properties. The ideal functionality will use two “macros” to decide whether a key gsk is consistent with its internal records or not, where the two macros are used relying on whether a TPM is honest or corrupted. Both macros output 1 indicating a new key gsk is consistent with the internal records and 0 that signals an invalid key. The two macros are defined as below:

\[
\text{CheckGskHonest}(gsk) = \forall \langle m, bsn, \sigma, *, * \rangle \in \text{Signed} : \text{identify}(m, bsn, \sigma, gsk) = 0 \land \\
\forall \langle m, bsn, \sigma, *, 1 \rangle \in \text{VerResults} : \text{identify}(m, bsn, \sigma, gsk) = 0
\]

\[
\text{CheckGskCorrupt}(gsk) = \exists m, bsn, \sigma : \\
\big( (\langle m, bsn, \sigma, *, * \rangle \in \text{Signed} \lor \langle m, bsn, \sigma, *, 1 \rangle \in \text{VerResults} ) \land \exists gsk' : (gsk \neq gsk' \land (\langle *, *, gsk', * \rangle \in \text{Members} \lor \langle *, *, gsk' \rangle \in \text{DomainKeys}) \land \text{identify}(m, bsn, \sigma, gsk) = \text{identify}(m, bsn, \sigma, gsk') = 1) \big).
\]

To simplify the definition of \( F^I_{\text{daa}} \), the following conventions are made: 1) all requests other than the SETUP are ignored until one setup phase is completed; 2) when \( F^I_{\text{daa}} \) performs any check that fails, it outputs ⊥ directly to the caller; 3) whenever \( F^I_{\text{daa}} \) runs one of the algorithms ukgen, sig, ver, link, identify, it does so without maintaining states.
Sign
7. Sign Request. On input (SIGN, sid, ssid, Mᵢ, m, bsn, ˙p) from host Hⱼ with ˙p ∈ P.
   - If Hᵢ is honest and no entry (Mᵢ, Hⱼ, *, attr) with ˙p(attr) = 1 exists in Members, abort.
   - Create a sign session record (ssid, Mᵢ, Hⱼ, m, bsn, ˙p, status) with status ← request.
   - Output (SIGNSTART, sid, ssid, l(m, bsn, ˙p), Mᵢ, Hⱼ) to S.
8. Sign Request Delivery. On input (SIGNSTART, sid, ssid) from S.
   - Update the session record (ssid, Mᵢ, Hⱼ, m, bsn, ˙p, status) to status ← delivered.
   - Output (SIGNPROCEED, sid, ssid, m, bsn, ˙p) to Mᵢ.
   - Look up record (ssid, Mᵢ, Hⱼ, m, bsn, ˙p, status) with status = delivered.
   - Output (SIGNCOMPLETE, sid, ssid) to S.
10. Signature Generation. On input (SIGNCOMPLETE, sid, ssid, σ) from S.
    - If I is honest, check that (Mᵢ, Hⱼ, *, attr) with ˙p(attr) = 1 exists in Members.
    - If Mᵢ and Hⱼ are honest, ignore σ from S and internally generate a signature for a fresh or established gsk:
      - If bsn ̸= ⊥, retrieve gsk from (Mᵢ, bsn, gsk) ∈ DomainKeys for (Mᵢ, bsn). If no such gsk exists or bsn = ⊥, generate gsk ← ukgen(). Check that CheckGskHonest(gsk) = 1 (∗) and store (Mᵢ, bsn, gsk) in DomainKeys.
      - Compute signature σ ← sig(gsk, m, bsn, ˙p) and check ver(m, bsn, σ, ˙p) = 1 (∗i).
      - Check that identify(m, bsn, σ, gsk) = 1 (∗ii) and check that there is no Mᵢ’ ̸= Mᵢ with key gsk’ registered in Members or DomainKeys with identify(m, bsn, σ, gsk’) = 1 (∗iii).
    - If Mᵢ is honest, store (m, bsn, σ, Mᵢ, ˙p) in Signed.
    - Output (SIGNATURE, sid, ssid, σ) to Hⱼ.
Verify
11. Verify. On input (VERIFY, sid, m, bsn, σ, ˙p, RL) from some party V.
    - Retrieve all pairs (Mᵢ, gskᵢ) from (Mᵢ, *, gskᵢ) ∈ Members and (Mᵢ, *, gskᵢ) ∈ DomainKeys such that identify(m, bsn, σ, gskᵢ) = 1. Set f ← 0 if at least one of the following conditions hold:
      - More than one key gskᵢ was found (∗x).
      - I is honest and no pair (Mᵢ, gskᵢ) was found for which an entry (Mᵢ, *, *, attr) ∈ Members with ˙p(attr) = 1 exists (∗x).
      - There is an honest Mᵢ but no entry ⟨m, bsn, *, Mᵢ, ˙p⟩ ∈ Signed exists (∗xi).
      - There is a gsk’ ∈ RL such that identify(m, bsn, σ, gsk’) = 1 and no pair (Mᵢ, gskᵢ) for an honest Mᵢ was found (∗xii).
    - If f ̸= 0, set f ← ver(m, bsn, σ, ˙p) (∗xii).
    - Add ⟨m, bsn, σ, RL, f⟩ to VerResults and output (VERIFIED, sid, f) to V.
Link
12. Link. On input (LINK, sid, m₀, σ₀, ˙p₀, m₁, σ₁, ˙p₁, bsn) from some party V with bsn ̸= ⊥.
    - Output ⊥ to V if at least one signature tuple (m₀, bsn, σ₀, ˙p₀) or (m₁, bsn, σ₁, ˙p₁) is not valid, which is verified via the VERIFY interface with RL = ∅ (∗xiv).
    - For each key gskᵢ in Members and DomainKeys, compute bᵢ ← identify(m₀, bsn, σ₀, gskᵢ) and bᵢ’ ← identify(m₁, bsn, σ₁, gskᵢ), and then do the following:
      - Set f ← 0 if bᵢ ̸= bᵢ’ for some i (∗xv).
      - Set f ← 1 if bᵢ = bᵢ’ = 1 for some i (∗xvi).
    - If f is not defined yet, set f ← link(m₀, σ₀, m₁, σ₁, bsn).
    - Output (LINK, sid, f) to V.

Fig. 5: The Sign, Verify, and Link Related Interfaces of Ideal Functionality $F^{\text{I}}_{\text{d}}$.

B Two Extensions of Our DAA Schemes

B.1 Signature-Based Revocation Extension

In this section, we extend our scheme DAA_OPT to support signature-based revocation. Our DAA scheme with signature-based revocation keeps compatible with the TPM 2.0 specification [Tru16]. While known
DAA schemes with signature-based revocation [BL07, BL10a, CDL16a, CCD17] require at least 3$n_r E_{G_1}$ for the TPM to prove that the platform has not been revoked, our DAA scheme provides the fully optimal TPM signing efficiency, where $n_r$ denotes the number of revoked platforms.

We present a signature-based revocation mechanism, following the basic revocation idea in the EPID scheme [BL07]. We propose an efficient method to delegate most computations of the TPM to its host and keep the fully optimal signing efficiency for the TPM.

Now, we use $K = e(g, H_{G_2}(str))^gsk$ for both $bsn \neq \bot$ and $bsn = \bot$, where $str = bsn$ if $bsn \neq \bot$ and $str \leftarrow \{0, 1\}^{l_r}$ otherwise. A verifier $V$ locally maintains a signature revocation list $SRL = \{\langle str_i, K_i \rangle\}_{i=1}^{n_r}$, where $K_i = e(g, H_{G_2}(str_i))^gsk_i$ for some $gsk_i \in Z_p$. To prove non-revocation towards $V$, a platform with secret key $gsk$ needs to prove in zero-knowledge $K_i = e(g, H_{G_2}(str_i))^gsk$ for all $i \in [n_r]$. The proof can be done using the zero-knowledge proof of inequality of discrete logarithms by Camenisch and Shoup [CS03]: choose $v_i \leftarrow Z_p$ and compute $V_i \leftarrow (e(g, H_{G_2}(str_i))^{gsk}/K_i)^{v_i}$; and then generate the following proof of knowledge:

$$SPK_r \left\{ \left\{ (v_i \cdot gsk, v_i)_{i=1}^{n_r} \right\} : V_i = e(g, H_{G_2}(str_i))^{v_i \cdot gsk} \cdot K_i^{-v_i} \wedge 1_{G_r} = e(g, H_{G_2}(str))^{v_i \cdot gsk} \cdot K^{-v_i} \text{ for each } i \in [n_r] \right\},$$

where $K = e(g, H_{G_2}(str))^gsk$.

We can extend DAAOPT to support signature-based revocation by extending the signing operations of the host and the verification algorithm, where the operations executing by the TPM keep unchanged. Concretely, the host $H_j$ is further given a signature revocation list SRL, and additionally performing the following operations in the sign protocol to prove that the platform has not been revoked.

1. For each $i \in [n_r]$, $H_j$ chooses $v_i \leftarrow Z_p^*$ and computes $V_i \leftarrow (gpk^{v_i}, H_{G_2}(str_i)) \cdot K_i^{-v_i}$. $H_j$ sends a request TPM. Commit to the TPM and receives as response.
2. If $bsn \neq \bot$, $H_j$ sets $str \leftarrow bsn$ and $B \leftarrow \bot$, and computes $K \leftarrow e(gpk, H_{G_2}(bsn))$. If $bsn = \bot$, $H_j$ changes the computational manner of unlinkable tags as: pick $v_i \leftarrow \{0, 1\}^{l_r}$, and set $B \leftarrow \bot$ and compute $K \leftarrow e(gpk, H_{G_2}(str))$.
3. For each $i \in [n_r]$, $H_j$ picks $\alpha_i, \beta_i \leftarrow Z_p$ and computes $F_i \leftarrow E^{v_i} \cdot g^{\alpha_i}$; and then calculates $W_i \leftarrow e(F_i, H_{G_2}(str_i)) \cdot K_i^{-\beta_i}$ and $Z_i \leftarrow e(F_i, H_{G_2}(str)) \cdot K^{-\beta_i}$, where $E = G^g$.
4. $H_j$ computes $s_{ch} \leftarrow H_2(\text{sign''}, g, g_1, \{h_i\}_{i=0}^{n_r}, T_1, T_2, Y', B, K, R_1, R_2, L, SRL, \{V_i, W_i, Z_i\}_{i=1}^{n_r})$. Then $H_j$ sends $TPM.Sig(\text{sign}, m, str, D, I, s_{ch})$ to the TPM and receives $(N_t, s)$ as response.
5. For each $i \in [n_r]$, $H_j$ computes $s_i \leftarrow \alpha_i + s \cdot v_i \mod p$ (i.e., $s_i = (\alpha_i + r \cdot v_i + \hat{r} \cdot v_i) + c \cdot gsk \cdot v_i \mod p$) and $s_i' \leftarrow \beta_i + c \cdot v_i \mod p$, where $(c, s)$ is computed by $H_j$ as in Figure 2.
6. $H_j$ sets $\pi_r \leftarrow \{\langle s_i, s_i' \rangle_{i=1}^{n_r}\}$, and outputs a signature $\sigma \leftarrow (T_1, T_2, Y', B, K, \{V_i\}_{i=1}^{n_r}, \pi_2, \pi_r, str)$, where $str = bsn$ is unnecessary to be included in the signature if $bsn \neq \bot$.

For the support of signature-based revocation, a verifier $V$ is given a signature revocation list SRL and additionally checks the validity of $\{V_i\}_{i=1}^{n_r}$ and proof $\pi_r$. The changes of the verification algorithm are described as follows:

1. Ignore the check of $B$ and compute $B \leftarrow e(g, H_{G_2}(str))$ where $str$ is taken from the signature if $bsn = \bot$ and $str = bsn$ otherwise.
2. For every $i \in [n_r]$, check that $V_i \neq 1_{G_r}$.
3. For each $(str_i, K_i) \in SRL$, compute the following commitments $W_i' \leftarrow e(g^{s_i}, H_{G_2}(str_i)) \cdot K_i^{-s_i'} \cdot V_i^{-c}$ and $Z_i' \leftarrow e(g^{s_i}, H_{G_2}(str)) \cdot K^{-s_i'}$.
4. Compute $c_{ch}' \leftarrow H_2(\text{sign''}, g, g_1, \{h_i\}_{i=0}^{n_r}, T_1, T_2, Y', B, K, R_1, R_2, L, SRL, \{V_i, W_i', Z_i'\}_{i=1}^{n_r})$ and $c' \leftarrow H_1(N_t, m, str, D, I, s_{ch})$. 
B.2 Privacy Extension against Subverted TPMs

We extend our DAA scheme DAADOPT to guarantee privacy against subverted TPMs. Our DAA scheme with subverted TPMs keeps the TPM signing efficiency fully optimal, and outperforms the existing DAA schemes with subverted TPMs [CDL17, CCD17] in terms of signing performance. Recently, Camenisch et al. [CCD17] modified the TPM 2.0 commands with minimal changes and used them to implement two ECDAA schemes with subverted TPMs. We can use their modified TPM 2.0 commands to implement our DAA scheme with subverted TPMs.

Following the techniques in [CCD17], we can extend DAADOPT to guarantee privacy in the presence of subverted TPMs, and thus avoid a subliminal channel that may be created by a subverted TPM. The extended DAA scheme with subverted TPMs is the same as DAADOPT, except that the join and sign protocols are changed as follows:

- For TPM.Commit request, the TPM \( \mathcal{M}_i \) picks \( N_t \leftarrow \{0,1\}^\ell \) and computes \( \bar{N}_t \leftarrow H(\text{“nonce”}, N_t) \) using a hash function \( H : \{0,1\}^* \rightarrow \mathbb{Z}_p \) modeled as a random oracle, and then outputs \( (E, \bar{N}_t) \) where \( E = g^r \).
- The host \( \mathcal{H}_j \) chooses \( N_h \leftarrow \{0,1\}^\ell \), and does the following:
  - In the join protocol, send TPM.\text{Sign}, \((c_h, N_h)\) to \( \mathcal{M}_i \).
  - In the sign protocol, send TPM.\text{Sign}, \((m, \text{bsn}, D, I, c_h, N_h)\) to \( \mathcal{M}_i \).
- On input TPM.\text{Sign}, \((msg, N_h)\), \( \mathcal{M}_i \) computes \( c \leftarrow H_1(N_t \oplus N_h, msg) \), and outputs \( (N_t, s) \), where \( msg \) is either \( c_h \) or \( (m, \text{bsn}, D, I, c_h) \) and \( s = r + c \cdot tsk \mod p \).
- \( \mathcal{H}_j \) checks whether \( \bar{N}_t = H(\text{“nonce”}, N_t) \) or not. If the check passes, \( \mathcal{H}_j \) computes \( N \leftarrow N_t \oplus N_h \), and then re-computes \( c \leftarrow H_1(N, msg) \) where \( msg \) is defined as in previous step. \( \mathcal{H}_j \) checks whether \( g^s = E \cdot tpk^c \) or not. If the equality holds, \( \mathcal{H}_j \) sends \( (tpk, C, \pi_t, \pi_h) \) to the issuer in the join protocol; or completes the computation of a signature \( \sigma \) and puts a nonce \( N \) instead of \( N_t \) to \( \sigma \) in the sign protocol.

Since the TPM commits to a nonce \( N_t \) before seeing the nonce \( N_h \), and \( N_t \) is randomized as \( N = N_t \oplus N_h \) by the host, the subverted TPM cannot embed any information into the nonce \( N \). In the random oracle model, \( c = H_1(N, msg) \) will be a random value, which cannot be controlled by the subverted TPM. A platform secret key \( gsk \) is split into a TPM secret key \( tsk \) and a host secret key \( hsk \) in a modular addition manner. The host proves knowledge of \( hsk \), which randomizes the \( E \) and \( s \) from the TPM. Furthermore, the validity of \( (E, c, s) \) is also verified by the host. As a result, a subverted TPM cannot embed any information into a signature, and our DAA scheme guarantees privacy against subverted TPMs.

C Alternative Description of Our DAA Protocol for UC Security

In this section, we provide an alternative description of our DAA protocol DAADOPT for UC security. We add session identifiers to DAADOPT, which is required for universal composability.

We assume that a common reference string functionality \( F_{\text{crs}}^{\text{OPT}} \) and a certification authority functionality \( F_{\text{ca}} \) are available for all parties. The former will be used to provide the parties with the system parameters \( \text{parms} \), and the latter will allow the issuer to register its public key \( \text{ipk} \). The communication between the TPM and host is modeled using the secure message transmission functionality \( F_{\text{smt}}^{\text{l}} \) which enables confidential and authenticated communication. In fact, \( F_{\text{smt}}^{\text{l}} \) is naturally guaranteed by the physical proximity of the TPM and host forming a platform [CDL16b]. We refer the reader to [Can01, Can04] for the definitions of the standard ideal functionalities \( F_{\text{crs}}^{\text{OPT}}, F_{\text{ca}} \) and \( F_{\text{smt}}^{\text{l}} \). For the sake of readability, we will not explicitly write that the parties call \( F_{\text{crs}}^{\text{OPT}}, F_{\text{ca}} \) and \( F_{\text{smt}}^{\text{l}} \) to retrieve the system parameters \( \text{parms} \) and the issuer’s public key \( \text{ipk} \), nor explicitly describe that the TPM and host call \( F_{\text{smt}}^{\text{l}} \) for communication between them, which is in line with previous work [CDL16b, CDL16a, CCD17]. We use the ideal functionality \( F_{\text{auth}}^{\text{l}} \), introduced in [CDL16b] to model the semi-authenticated channel between the TPM and issuer. In particular, the TPM can use \( F_{\text{auth}}^{\text{l}} \) to send its public key \( tpk \) to the issuer via the host.

An alternative description of our protocol DAADOPT for UC security is shown as follows.
Setup. On input (SETUP, sid), the issuer $I$ checks that \( \text{sid} = (I, \text{sid}') \) for some $\text{sid}'$, and then creates its public key \( \text{ipk} = \{(h_i)_{i=0}^n, w, \pi_i \} \) and secret key \( \text{isk} = \gamma \) as described in §4.2. Then $I$ registers ipk with $F_{\text{ca}}$, and outputs (SETUPDONE, sid).

Join. A platform consisting of a TPM $\mathcal{M}_i$ and a host $H_j$ executes the join protocol with $I$ as follows:

1. Upon input (JOIN, sid, jsid, $\mathcal{M}_i$), $H_j$ parses \( \text{sid} = (I, \text{sid}') \), and sends a message (JOIN, sid, jsid) to $I$.

2. Upon receiving (JOIN, sid, jsid) from a party $H_j$, $I$ chooses a fresh nonce $N_i \leftarrow \{0,1\}^\lambda$ and sends (sid, jsid, $N_i$) back to $H_j$.

3. Upon receiving (sid, jsid, $N_i$) from issuer $I$, $H_j$ sends (TPM.Create, sid, jsid) to $\mathcal{M}_i$, $\mathcal{M}_i$ checks that no key record exists,\(^7\) chooses $\text{tsk} \leftarrow \mathbb{Z}_p$, and stores a key record $(\text{sid}, H_j, \text{tsk})$. Then $\mathcal{M}_i$ sends a TPM public key $\text{tpk}$ back to $H_j$, $\mathcal{M}_i$ and $H_j$ jointly generate $\pi_t \leftarrow \text{SPK}_i(\text{tsk}) = (\text{"TPM.join"}, N_i)$ via running a protocol described in Figure 1, where the only difference is that $H_j$ additionally sends (sid, jsid) to $\mathcal{M}_i$ when sending TPM.Commit or TPM.Sign requests.

4. $H_j$ notices $\mathcal{M}_i$, sending $\text{tpk}$ over $F_{\text{auth}_j}$ to $I$. $H_j$ computes a commitment $C \leftarrow g^{\text{hsk}h_{0}^{\prime}}$ and a platform public key $\text{gpk} \leftarrow \text{tpk} \cdot g^{\text{hsk}}$. Then $H_j$ generates $\pi_h \leftarrow \text{SPK}_j((\text{hsk}, u') : C = g^{\text{hsk}h_{0}^{\prime}})$ ("Host join", $N_i$) as in Figure 1.

5. $H_j$ appends $\pi_t, \pi_h$ to the message $\text{tpk}$, which is sent to $I$ over $F_{\text{auth}_j}$.

6. Upon receiving (tpk, $C, \pi_t, \pi_h$) from $F_{\text{auth}_j}$ where tpk is authenticated by TPM $\mathcal{M}_i$, issuer $I$ verifies the validity of proofs $\pi_t$ and $\pi_h$ as in Figure 1, and checks that $\mathcal{M}_i$ did not join before. $I$ stores (jsid, tpk, $C, \mathcal{M}_i, H_j$) and outputs (JOINPROCEED, sid, jsid, $\mathcal{M}_i$).

The join session is completed, when the issuer receives an explicit input that tells it to proceed with join session $\text{jsid}$ and issue attributes $\text{attrs} = (a_1, \ldots, a_n)$.

1. Upon input (JOINPROCEED, sid, jsid, $\text{attrs}$), $I$ retrieves the record $(\text{jsid}, \text{tpk}, C, \mathcal{M}_i, H_j)$ and marks $\mathcal{M}_i$ as "joined". Then $I$ creates a credential $A \leftarrow (g_1 \cdot \text{tpk} \cdot C \cdot h_0^{u''} \cdot \prod_{i=1}^n h_i^{a_i})^{1/(\gamma + x)}$ for two randomnesses $u', x \in \mathbb{Z}_p$. $I$ sends (sid, jsid, ($A, x, u''$), $\text{attrs}$) to $H_j$ over $F_{\text{auth}_j}$. We assume that $F_{\text{auth}_j}$ also provides the confidentiality of ($(A, x, u'')$, $\text{attrs}$). This assumption holds when we use the method [CW10] to realize functionality $F_{\text{auth}_j}$.

2. Upon receiving (sid, jsid, ($A, x, u''$), $\text{attrs}$) from $I$, $H_j$ computes $u \leftarrow u' + u'' \mod p$ and $Y \leftarrow g_1 \cdot g^{\text{hsk}} \cdot h_0^{u''} \cdot \prod_{i=1}^n h_i^{a_i}$, and then checks that $e(A, w \cdot g_2^\lambda) = e(Y, g_2)$. $H_j$ stores $(\text{sid}, \mathcal{M}_i, \text{cre} = (A, x, u, Y, \text{gpk}, \text{hsk}), \text{attrs})$ and outputs (JOINED, sid, jsid).

Sign. The sign protocol runs between a TPM $\mathcal{M}_i$ and a host $H_j$. By executing the protocol, they can jointly sign a message $m$ w.r.t. a basename bsn and attribute predicate $(D, I)$.

1. Upon input (SIGN, sid, ssid, $\mathcal{M}_i, m, \text{bsn}, (D, I))$, host $H_j$ retrieves the join record $(\text{sid}, \mathcal{M}_i, \text{cre} = (A, x, u, Y, \text{gpk}, \text{hsk}), \text{attrs})$. Then $H_j$ checks if his attributes fulfill the predicate, i.e., it parses $\text{attrs}$ as $(a_1, \ldots, a_n)$ and $I$ as $(a_1', \ldots, a_n')$, and checks that $a_i = a_i'$ for each $i \in D$. Next, $H_j$ randomizes the credential as $T_1 \leftarrow A^{t_1}, Y' \leftarrow Y^{t_1} h_0^{t_2}$ and computes $T_2 \leftarrow Y^{t_1} T_1^{-x}$. $H_j$ computes an unlinkable tag $(B, K = B^{\text{hsk}})$ for a random $B \in \mathbb{G}_1$ if bsn = ∅ and a pseudonym $(B = \perp, K = e(g, H_{G_2}(\text{bsn}))^{\text{hsk}})$ otherwise using public key $\text{gpk}$ as in Figure 2. $H_j$ sends (sid, ssid, m, bsn, (D, I)) to $\mathcal{M}_i$.

2. Upon receiving (sid, ssid, m, bsn, (D, I)) from $H_j$, $\text{TPM}_i$ asks for permission to proceed. Then $\mathcal{M}_i$ checks that a join record (sid, $H_j$, tsk) exists, and stores (sid, ssid, m, bsn, (D, I)) and outputs (SIGNPROCEED, sid, ssid, m, bsn, (D, I)).

The signature is completed when $\mathcal{M}_i$ gets permission to proceed for ssid.

\(^7\) If we consider a TPM with different keys as multiple different "TMPS" with a single key, the check of key record can be omitted.

\(^8\) As such, the DAA schemes [CDL16a, CCD\'17] need to keep the credential and attributes of a platform confidential in the join protocol.
Upon input \((\text{SIGNPROCEED}, \text{sid}, \text{ssid})\), \(M_i\) retrieves a join record \((\text{sid}, H_j, \text{tsk})\) and a sign record \((\text{sid}, \text{ssid}, m, \text{bsn}, (D, I))\). Then \(M_i\) cooperates with \(H_j\) to generate

\[ \pi_2 \leftarrow \text{SPK}_2\{\{gsk; \{a_i\}_{i \in D}, x, \tilde{u}, t_2, t_3\} : g_1^{-1} \prod_{i \in D} h_i^{a_i - a_i} = Y' = t_3 \bar{g}^{gsk} h_0 \prod_{i \in D} h_i^{a_i} \land T_2/Y' = T_1^{-x} h_0 \land K = B^{gsk}\} \]

This is completed via executing the sign protocol described in Figure 2, except that \(H_j\) additionally sends \((\text{sid}, \text{ssid})\) to \(M_i\) when sending TPM.Commit or TPM.Sign requests.

2. \(H_j\) sets \(\sigma \leftarrow (T_1, T_2, Y', B, K, \pi_2)\) and outputs \((\text{SIGNATURE}, \text{sid}, \text{ssid}, \sigma)\).

**Verify.** Upon input \((\text{VERIFY}, \text{sid}, m, \text{bsn}, \sigma, (D, I), \text{RL})\), a party \(V\) verifies the signature as follows:

1. Parse \(\sigma\) as \((T_1, T_2, Y', B, K, \pi_2)\).
2. Check that \(B \neq 1_{G_1}\) if \(\text{bsn} = \bot\) and \(B = \bot\) otherwise. If \(\text{bsn} \neq \bot\), compute \(B \leftarrow e(g, H_{G_2}(\text{bsn}))\).
3. Check that \(e(T_1, w) = e(T_2, g_2)\).
4. Verify the validity of proof \(\pi_2\) on message \(\{\text{"sign"}, m, \text{bsn}, D, I\}\) following the description in §4.2.
5. For every \(gsk_i \in \text{RL}\), check that \(K \neq B^{gsk_i}\).
6. If all the checks pass, set \(f \leftarrow 1\), otherwise \(f \leftarrow 0\).
7. Output \((\text{VERIFIED}, \text{sid}, f)\).

**Link.** Upon input \((\text{LINK}, \text{sid}, m_0, \sigma_0, D_0, I_0, m_1, \sigma_1, D_1, I_1, \text{bsn})\) with \(\text{bsn} \neq \bot\), a party \(V\) verifies the two signatures and decides whether they are linked or not.

1. Verify that both \(\sigma_0\) and \(\sigma_1\) are valid with respect to \((m_0, \text{bsn}, D_0, I_0)\) and \((m_1, \text{bsn}, D_1, I_1)\) respectively.
   
   Output \(\bot\) if one of them is not valid.
2. Parse \(\sigma_0\) and \(\sigma_1\) as \((T_1, 0, T_2, 0, Y'_0, B_0, K_0, \pi_{2,0})\) and \((T_1, 1, T_2, 1, Y'_1, B_1, K_1, \pi_{2,1})\).
3. If \(K_0 = K_1\), set \(f \leftarrow 1\), otherwise \(f \leftarrow 0\).
4. Output \((\text{LINK}, \text{sid}, f)\).

## D Security Proof of Our DAA Scheme

In this section, we formally state Theorem 1, and give the proof of Theorem 1 based on the security proofs by Camenisch et al. [CDL16b, CDL16a]. As pointed out by Camenisch et al. [CCD+17], the session identifiers for UC security can be omitted, if one is only concerned with stand-alone security. Thus, the security of \(\text{DAA}_{\text{OPT}}\) as described in §4.2 straightforwardly follows the one of the same protocol with an addition of session identifiers as described in Appendix C, which would be proved in the following theorem.

**Theorem 1.** The protocol \(\text{DAA}_{\text{OPT}}\) as described in Section C securely realizes \(\mathcal{F}_{\text{daa}}^l\) with static corruption (for any polynomial number of attributes \(n\), \(k_i = \mathbb{Z}_p\) and selective attribute disclosure as attribute predicates \(\mathbb{P}\)) under the DBDH, DDH_{G_1} and \(q\)-SDH assumptions in the \((\mathcal{F}_{\text{auth}}, \mathcal{F}_{\text{sp}}, \mathcal{F}_{\text{sm}}, \mathcal{F}_{\text{cb}})\)-hybrid model and the random oracle model.

**Proof.** In this proof, we use \(\equiv\) to denote the computational indistinguishability. We also use \(\text{EXEC}_{\text{DAA}_{\text{OPT}}, A, Z}\) to denote the real world ensemble in which environment \(Z\) is interacting with protocol \(\text{DAA}_{\text{OPT}}\) and adversary \(A\); \(\text{IDEAL}_{\mathcal{F}_{\text{daa}}, \mathcal{S}, Z}\) to denote the ideal world ensemble in which \(Z\) is interacting with ideal functionality \(\mathcal{F}_{\text{daa}}\) and simulator \(\mathcal{S}\). Our proof uses the known result that the BBS+ signature scheme is EUF-CMA secure under the \(q\)-SDH assumption [ASM06, CDL16a]. Thus, we can directly reduce the security of \(\text{DAA}_{\text{OPT}}\) to the EUF-CMA security of the BBS+ signature scheme.

We need to prove that for every PPT adversary \(A\), there exists a PPT simulator \(\mathcal{S}\), such that for every PPT environment \(Z\)

\[
\text{EXEC}_{\text{DAA}_{\text{OPT}}, A, Z} \equiv \text{IDEAL}_{\mathcal{F}_{\text{daa}}, \mathcal{S}, Z}
\]
We use a sequence of games based on the ones in [CDL16b, CDL16a] to proceed the proof, and prove that it is computationally indistinguishable between two successive games. We start with the real world protocol execution. In the next game, we construct an entity $C$ who runs the real world protocol for all honest parties. Then, we split $C$ into a functionality $F$ and a simulator $S$, where $F$ receives all inputs from honest parties and sends the outputs to honest parties. We start with a “dummy functionality”, then gradually change $F$ and $S$ accordingly, and finally end up with the full functionality $F^l_{daa}$ and a satisfying simulator.

Prior to describing the games, we prove that the signature proofs of knowledge $SPK_1$, $SPK_t$, $SPK_h$ and $SPK_2$ are zero-knowledge by constructing a simulator and showing that the simulation is perfect unless the simulator aborts with negligible probability.

- For $SPK_1\{\{\gamma\} : w = g_2^\gamma\}$ (“setup”), a simulator $Sim_1$ is constructed as follows: 1) pick $c, s \leftarrow \mathbb{Z}_p$ and compute $R \leftarrow g_2^c \cdot w^{-c}$; 2) program the random oracle such that $H_3(\text{“setup”}, g_2, w, R) = c$ and abort if encountering a collision, i.e., $H_3(\text{“setup”}, g_2, w, R)$ has already been defined; 3) output $\pi_1 \leftarrow (c, s)$.

The simulated proof has the same distribution as the real proof unless $Sim_1$ aborts with probability $\leq q_{h_3}/p$, which is negligible, where $q_{h_3}$ is the number of queries to random oracle $H_3$.

- For $SPK_t\{\{\text{tsk}\} : tpk = g^{\text{tsk}}\}$ (“TPM.\text{join}”, $N_t$) with an honest host, a simulator $Sim_t'$ is constructed as follows: 1) pick $c, s \leftarrow \mathbb{Z}_p$ and compute $E \leftarrow g^c \cdot tpk^{-c}$; 2) make a query (“TPM.\text{join}”, $g$, $tpk$, $E$, $N_t$) to random oracle $H_2$ and obtain $c_h; 3)$ choose $N_t \leftarrow \{0, 1\}^{\ell_n}$, program random oracle such that $H_1(N_t, c_h) = c$, and abort if encountering a collision; 4) output $\pi_t \leftarrow (c, s, N_t)$.

The simulation is perfect, unless $H_1(N_t, c_h)$ has already been defined with probability at most $q_{h_1}/2^{\ell_n} \cdot (q_{h_2}/p + q_{h_3}/p)$, where $q_{h_1}$ is the number of $H_1$ queries and $q_{h_2}$ denotes the number of $H_2$ queries associated with label “TPM.\text{join}”.

- For $SPK_t\{\{\text{tsk}\} : tpk = g^{\text{tsk}}\}(\text{msg})$ with a corrupted host, a simulator $Sim_t''$ is constructed as follows: 1) on input a request $\text{TPM.\text{Commit}}$, choose $c, s \leftarrow \mathbb{Z}_p$ and compute $E \leftarrow g^c \cdot tpk^{-c}$, and then output $E$; 2) on input $\text{TPM.\text{Sign}}(\text{msg})$, pick $N_t \leftarrow \{0, 1\}^{\ell_n}$ and program the random oracle such that $H_1(N_t, \text{msg}) = c$ and abort if encountering a collision; 3) output $(N_t, s)$.

The simulation is perfect, unless $H_2(\text{“Host.\text{join}”}, g, h_0, C, R, N_t)$ has already been defined with probability at most $q_{h_2}'/2^{\ell_n}$.

- For $SPK_h\{\{hsk, u'\} : C = g^{hsk} h_0^{u'}\}$ (“Host.\text{join}”, $N_t$), a simulator $Sim_h$ is constructed as follows: 1) pick $z, s, s' \leftarrow \mathbb{Z}_p$ and compute $R \leftarrow g^z \cdot h_0^{s'} \cdot C^{-c}$; 2) program the random oracle such that $H_2(\text{“Host.\text{join}”}, g, h_0, C, R, N_t) = z$ and abort if encountering a collision; 3) output $\pi_h \leftarrow (z, s, s')$.

The simulation is perfect, unless $H_2(\text{“Host.\text{join}”}, g, h_0, C, R, N_t)$ has already been defined with probability at most $q_{h_2}'/2^{\ell_n}$.

- For $SPK_2\{\{gsk, \{a_i\}_{i \in D}, x, \tilde{u}, t_2, t_3\} : g_1^{-1} \prod_{i \in D} h_i^{-a_i} = Y_{-t_3} g^{gsk} h_0^{\tilde{u}} \prod_{i \in D} h_i^{a_i} \wedge T_2/Y' = T_1^{-x} h_0^{2} \wedge K = B^{gsk} \}$ (“sign”, $m$, bsn, $D, I$), a simulator $Sim_2$ is constructed as follows: 1) pick $c, s, s_x, s_{\tilde{u}}, s_{t_2}, s_{t_3}, \{s_{a_i}\}_{i \in D} \leftarrow \mathbb{Z}_p$ and $N_t \leftarrow \{0, 1\}^{\ell_n}$; 2) compute $R_1 \leftarrow Y_{-s_{t_3}} \cdot g\tilde{u} \cdot \prod_{i \in D} h_i^{a_i} \cdot g_i \cdot \prod_{i \in D} h_i^{c_{a_i}} \cdot R_2 \leftarrow T_1^{-s_x} \cdot h_{t_2}^{s_{t_2}} \cdot (T_2/Y')^{-c} \wedge L \leftarrow B^{s_l} \cdot K^{-c}$; 3) make a query (“sign”, $g, g_1, \{h_i\}_{i = 0}^{1}, T_1, T_2, Y', B, K, R_1, R_2, L$) to random oracle $H_2$ and get $c_h$ as the answer; 4) program the random oracle such that $H_1(N_t, m, bsn, D, I, c_h) = c$ and abort if encountering a collision; 5) output $\pi_2 \leftarrow (c, s, s_x, s_{\tilde{u}}, s_{t_2}, s_{t_3}, \{s_{a_i}\}_{i \in D}, N_t)$.

The simulated proof has the same distribution as the real proof, unless $H_1(N_t, m, bsn, D, I, c_h)$ has been already defined with probability $\leq q_{h_1}/2^{\ell_n}(q_{h_1}/p + q_{h_2}'/p^3)$, where $q_{h_2}'$ is the number of queries to random oracle $H_2$ associated with label “sign”. By rewinding and programming the random oracles, we can construct the knowledge extractors $Ext_1$, $Ext_t$, $Ext_h$ and $Ext_x$ for $SPK_1$, $SPK_t$, with honest hosts, $SPK_h$ and $SPK_2$ respectively.

We define all intermediate functionalities and simulators in Appendix D.1, and then prove that they are indistinguishable by following the games constructed.
**Game 1.** This is the real world protocol. We have Game 1 = EXEC$\text{DAA}_{\text{OPT}}$.4.$\mathcal{Z}$.

**Game 2.** $\mathcal{C}$ receives all inputs for honest parties and simulates the real world protocol for honest parties via simply running the protocol DAA$_{\text{OPT}}$ honestly. Furthermore, $\mathcal{C}$ simulates all hybrid functionalities $\mathcal{F}^i_{\text{auth}}, \mathcal{F}^i_{\text{ca}}, \mathcal{F}^i_{\text{sm}}, \mathcal{F}^i_{\text{crs}}$ honestly.

By construction, Game 2 is equivalent to Game 1.

**Game 3.** Now, we split $\mathcal{C}$ into a dummy functionality $\mathcal{F}$ and a simulator $\mathcal{S}$. $\mathcal{F}$ behaves as an ideal functionality, and so the messages that it sends and receives are confidential and authenticated. Thus, the adversary $\mathcal{A}$ will not notice them. Functionality $\mathcal{F}$ receives all the inputs, and forwards them to simulator $\mathcal{S}$. $\mathcal{S}$ simulates the real world protocol DAA$_{\text{OPT}}$ for all honest parties, and sends the outputs to $\mathcal{F}$, who forwards them to environment $\mathcal{Z}$. The outputs generated by the honest parties simulated by $\mathcal{S}$ are not sent anywhere, and only $\mathcal{S}$ notices them. $\mathcal{S}$ sends the equivalent outputs to $\mathcal{F}$ using an OUTPUT interface such that $\mathcal{F}$ can use the same outputs.

Game 3 is simply game 2 except for structuring differently. Thus Game 3 = Game 2.

**Game 4.** Now, $\mathcal{F}$ uses the procedure specified in $\mathcal{F}^{i}_{\text{daa}}$ to deal with the setup related interfaces. As a result, $\mathcal{S}$ will send the algorithms $\text{ukgen}, \text{sig}, \text{ver}, \text{link}, \text{identify}$ to $\mathcal{F}$. $\mathcal{F}$ stores the algorithms from $\mathcal{S}$, and checks whether $\text{sid}$ is the expected form or not. For corrupt issuer, $\mathcal{S}$ can extract the issuer’s secret key from SPK$_1$. Note that the check of $\mathcal{F}$ for $\text{sid}$ does not change the view of $\mathcal{Z}$, as honest issuer $\mathcal{I}$ does the same check upon receiving $\text{sid}$ and $\mathcal{S}$ calls the SETUP interface on behalf of corrupt issuer $\mathcal{I}$. Thus, to the soundness of SPK$_1$, the view of $\mathcal{Z}$ is not changed. Thus, Game 4 $\approx$ Game 3.

**Game 5.** Now, $\mathcal{F}$ responds the queries for VERIFY and LINK interfaces using the provided algorithms $\text{ver}$ and $\text{link}$, instead of forwarding them to $\mathcal{S}$. Note that $\mathcal{F}$ has not to perform the additional checks (i.e., Check (ix)-Check(xii)) which will be added in later games. For Check (xv), $\mathcal{F}$ rejects a signature if a matched $\text{gsk}^i \in RL$ is found, but does not eliminate honest TPMs from this check yet.

There are no message flows for the verify and link algorithms, and so we only need to show that the outputs are equal. The verification algorithm that $\mathcal{F}$ uses is the same as the one of real-world protocol DAA$_{\text{OPT}}$, except that private key revocation check is omitted. $\mathcal{F}$ performs this revocation check separately, and thus the outputs for verify queries are equal. The real-world link algorithm outputs $\perp$ if one of two signatures is invalid. $\mathcal{F}$ does the same. The algorithm compares the equality of two pseudonyms, which is exactly what $\mathcal{F}$ does. Thus, the outputs for link queries are equal. In all, Game 5 = Game 4.

**Game 6.** In this game, $\mathcal{F}$ is changed to handle the join related interfaces by using the same procedure as $\mathcal{F}^{i}_{\text{daa}}$, but omit the additional checks (i.e., Check (iii)-Check (iv)). If at least one of the TPM and host is honest, $\mathcal{S}$ knows the identities $\mathcal{M}$ and $\mathcal{H}$, and can correctly use them towards $\mathcal{F}$ and its simulation. If both TPM and host are corrupted but the issuer is honest, $\mathcal{S}$ cannot determine the identity of the host, since the host does not authenticate itself to the issuer in the real-world join protocol. In this case, $\mathcal{S}$ has to choose an arbitrary corrupt host $\mathcal{H}$ to invoke the JOIN interface. In the JOINCOMPLETE interface, $\mathcal{S}$ needs to provide the secret key of the platform $\text{gsk}$. When the TPM (resp., host) is honest, $\mathcal{S}$ simulates the party and knows the secret key $\text{tsk}$ (resp., $\text{hsk}$). When the TPM (resp., host) is corrupted but the issuer is honest, $\mathcal{S}$ can extract the secret key $\text{tsk}$ (resp., $\text{hsk}$) from the proof $\pi_t$ (resp., $\pi_h$). Then $\mathcal{S}$ can compute $\text{gsk} \leftarrow \text{tsk} + \text{hsk} \mod p$. For the case that both the host and the issuer are corrupted but the TPM is honest, $\mathcal{S}$ does not need to involve $\mathcal{F}$ and simply continues the simulation of the TPM, since $\mathcal{F}$ guarantees no security properties for the case, and the TPM does not receive inputs or send outputs in the join related interfaces.

We must guarantee $\mathcal{F}$ outputs the same values as the real-world protocol. Since the join related interfaces do not output any crypto value, but only messages like start and complete, we just need to assure that whenever the real-world protocol would reach a certain output, $\mathcal{F}$ also allows the output, and vice versa. From the real world to the functionality, this is clearly satisfied, as $\mathcal{F}$ does not perform additional checks and thus will always proceed for any input that it receives from $\mathcal{S}$. For all outputs triggered by $\mathcal{F}$, $\mathcal{S}$ has to
give an explicit approval, which enables $S$ to block any output by $F$ if the real-world protocol would not proceed at a certain point. Thus, from the functionality to the real world, this can also be satisfied.

When both the TPM and host are corrupted but the issuer is honest, $S$ uses an arbitrary corrupt host when calling the JOIN interface, which will result in a different host being stored in Members list of $F$. However, $F$ never uses the identity of the host in the case that both the TPM and host are corrupted. Although $F$ sets $gsk \leftarrow \bot$ when both the TPM and host are honest, this has no impact, since the signatures are still generated by $S$ and the VERIFY and LINK interfaces of $F$ do not perform additional checks that make use of the internal records and secret keys.

We have to argue that $F$ does not prevent an execution which was allowed in the previous game. $F$ only aborts if $M$ has already registered and $I$ is honest. Since $I$ checks whether $M$ has already registered or not before outputting JOINPROCEED in the real-world protocol, $F$ keeps consistent in Game 5 and Game 6.

If $S$ can extract the secret keys from proofs $\pi_t$ and $\pi_h$ successfully, $F$ stores the keys consistent with the real-world protocol when the TPM and host are not both honest. Furthermore, $S$ can simulate the real-world protocol and keep everything in sync with $F$. Due to the soundness of SPK$_t$ and SPK$_h$, we have Game 6 $\approx$ Game 5.

**Game 7.** For signing with $bsn = \bot$, $F$ now generates signatures for honest platforms using fresh keys and the ukgen and sig algorithms defined in the setup phase. The procedure for signing with $bsn \neq \bot$ has not changed. One difference is that the signature created by $F$ will use a credential containing dummy attribute values for the undisclosed attributes. This change is not noticeable, since only $(T_1, T_2, Y')$ and proof $\pi_2$ are affected. $F$ use sig to generate uniformly random $Y'$ and $T_1, T_2$ under the constraint that $T_2 = T_1^\alpha$, which have the same distribution as the elements in the signatures created by the sign protocol of DAAGPT. The proof $\pi_2$ created by $F$ is indistinguishable from the one generated by the real-world sign protocol due to the zero-knowledge property of SPK$_2$. Besides, the unlinkable-tag/pseudonym $(B, K)$ created by $F$ has the same distribution as the one from the real-world sign protocol, since the only difference is to generate $(B, K)$ using directly secret key $gsk$ rather than using public key $gpk$.

We will use a hybrid argument to prove that environment $Z$ cannot notice the change that the signatures w.r.t. $bsn = \bot$ are now produced by $F$ using fresh keys rather than the same key. We make this change for signing inputs with $bsn = \bot$ gradually. In Game $7.k.k'$, $F$ forwards all signing inputs with $\mathcal{M}_i, i > k$ to $S$, who creates signatures as in Game 6. Signing inputs with $\mathcal{M}_i, i < k$ are handled by $F$ via using fresh keys and the ukgen and sig algorithms. For signing inputs with $\mathcal{M}_k$, the first $k'$ signing inputs are handled by $F$, and later signing inputs will be forwarded to $S$. Clearly, we have Game $7.1.0 = Game 6$. Let $v$ be the number of honest platforms and $\rho_k$ be the number of signing inputs with $\mathcal{M}_k$ and $bsn = \bot$. Clearly, we have Game $7.k.\rho_k = Game 7.k + 1.0$ for any $k \in \{1, \ldots, v - 1\}$ and Game $7.v.\rho_v = Game 7$. Thus, to prove that no environment can distinguish Game 7 from Game 6, it is enough to show that anyone cannot distinguish Game $7.k.k' - 1$ from Game $7.k.k'$ for any $k \in [v]$ and $k' \in [\rho_k]$.

We bound the difference between Game $7.k.k' - 1$ and Game $7.k.k'$ using a reduction from the DDH$_{G_1}$ assumption. In this reduction, we allow $S$ and $F$ to share information, since in the reduction the separation of $S$ and $F$ is irrelevant. $S$ is given a DDH$_{G_1}$ instance $(p, G_1, G_2, g_T, e, g_2, \bar{g}, \bar{g}^\alpha, \bar{g}^\beta, \bar{g}^\chi)$ for unknown $\alpha, \beta \in \mathbb{Z}_p$ and aims to decide whether $\chi = \alpha\beta$ or not. We modify $S$ working with $F$ parametrized by $k, k'$ to obtain an intermediate game $G_{7,k,k'}$, which is the same as Game $7.k.k' - 1$ except for the following exceptions:

- $S$ picks $g_1 \leftarrow G_1$ and sets $(p, G_1, G_2, g_T, e, g_1, g_2, \bar{g})$ as the system parameters params, as it simulates $F_{crs}$.
- $S$ sets $\bar{g}^\alpha$ as the public key $tpk$ of TPM $\mathcal{M}_k$. $S$ runs $Sim'_t$ to generate a proof $\pi_t$ in the join protocol. $S$ chooses $hsk \leftarrow \mathbb{Z}_p$ as the secret key of host $T_h$.
- For the $k'$-th signature w.r.t. $bsn = \bot$ for $\mathcal{M}_k$, we modify $F$ to output a signature as follows:
  1) Choose $T_1 \leftarrow G_1^*$ and compute $T_2 \leftarrow T_1^\gamma$, and then pick $Y' \leftarrow G_1$. 


2) Set $B \leftarrow g^\beta$ and $K \leftarrow g^\lambda \cdot (g^\beta)^{h \cdot sk}$.

3) Send $(T_1, T_2, Y', B, K)$ to $\mathcal{S}$, who runs $Sim_2$ to generate a proof $\pi_2$ and sends $\pi_2$ back to $\mathcal{F}$.

4) Output a signature $\sigma = (T_1, T_2, Y', B, K, \pi_2)$.

- Signing queries with $\mathcal{M}_k$, which are related to $bsn = \perp$ but occur after the $k'$-th one, or are with respect to $bsn \neq \perp$, are handled by $\mathcal{S}$. To create a signature for $\mathcal{M}_k$, $\mathcal{S}$ runs $Sim'_k$ to generate the outputs of “$\mathcal{M}_k$” in the sign protocol and executes the operations at the host side following the specification of $\text{DAA}^{\text{OPT}}$.

Due to the zero-knowledge property of $\text{SPK}_t$, proof $\pi_t$ generated by $Sim'_t$ is computationally indistinguishable from the real proof created by the witness $tsk$. By the zero-knowledge property of $\text{SPK}_t$, we have that the signatures produced by $\mathcal{S}$ using $Sim'_k$ are computationally indistinguishable from the ones created via executing the real-world sign protocol. The elements $T_1, T_2, Y'$ in the $k'$-th signature has the same distribution as the ones generated by the sig algorithm as well as the real-world sign protocol. By the zero-knowledge property of $\text{SPK}_t$ and $\text{SPK}_2$, we have that $G^{'k} = (g_1^{\alpha}, g_2^{\beta}, g_3^{\gamma})$ is computationally indistinguishable to $G^{8} = (g_1^{\alpha}, g_2^{\beta}, g_3^{\gamma})$.

Overall, we have $\text{Game } 7 \approx \text{Game } 6$.

**Game 8.** For signing with $bsn \neq \perp$, $\mathcal{F}$ now generates signatures for honest platforms using fresh keys and the ukgen and sig algorithms defined in the setup phase.

Again, we will use a hybrid argument to prove that environment $\mathcal{Z}$ cannot notice the change that the signatures w.r.t. fresh basename $bsn \neq \perp$ are now generated by $\mathcal{F}$ using fresh keys instead of the same key. We make this change for signing inputs with $bsn \neq \perp$ gradually. In Game $8.k.k'$, $\mathcal{F}$ forwards all signing inputs with $\mathcal{M}_i$, $i > k$ to $\mathcal{S}$, who creates signatures as in Game $7$. Signing inputs with $\mathcal{M}_i$, $i < k$ are handled by $\mathcal{F}$ via using fresh keys and the ukgen and sig algorithms. For signing inputs with $\mathcal{M}_k$, the first $k'$ non-empty basenames are handled by $\mathcal{F}$, and later signing inputs will be forwarded to $\mathcal{S}$. Clearly, we have Game $8.1.0 = \text{Game } 7$. Let $v$ be the number of honest platforms and $\rho_k$ be the number of different basenames with $\mathcal{M}_k$ and $bsn \neq \perp$. Clearly, we have Game $8.k.\rho_k = \text{Game } 8.k + 1.0$ for any $k \in \{1, \ldots, v - 1\}$ and Game $8.\nu.\rho_v = \text{Game } 8$. Thus, to prove that no environment can distinguish Game $8$ from Game $7$, it is enough to show that anyone cannot distinguish Game $8.k.k' - 1$ from Game $8.k.k'$ for any $k \in [v]$ and $k' \in [\rho_k]$.

We bound the difference between Game $8.k.k' - 1$ and Game $8.k.k'$ using a reduction from the DBDH assumption. $\mathcal{S}$ is given a DBDH instance $(p, G_1, G_2, G_T, e, g_1, g_2, g_1^\alpha, g_1^\beta, g_2^\gamma, e(g_1, g_2)^\chi)$ for unknown $\alpha, \beta, \gamma \in \mathbb{Z}_p$, and aims to decide whether $\chi = \alpha \beta \gamma$ or not. We modify $\mathcal{S}$ working with $\mathcal{F}$ parametrized by $k, k'$ to obtain an intermediate game $G^{8} = (g_1^{\alpha}, g_2^{\beta}, g_3^{\gamma})$, which is the same as Game $8.k.k' - 1$ except for the following exceptions:

- $\mathcal{S}$ sets $\bar{g} = g_1^\delta$ and $(p, G_1, G_2, G_T, e, g_1, g_2, g)$ as the system parameters params, as it simulates $\mathcal{F}_{\text{crs}}$.

- $\mathcal{S}$ sets the unknown discrete logarithm $\alpha$ as the key $gsk$ for the honest platform with $\mathcal{M}_k$. $\mathcal{S}$ chooses $C \leftarrow G_1$ and runs $Sim_h$ to generate a proof $\pi_h$ in the join protocol. $\mathcal{S}$ simulates the TPM “$\mathcal{M}_k$” honestly, i.e., choosing a key $tsk \leftarrow \mathbb{Z}_p$ and generates $\text{SPK}_t$ with witness $tsk$ etc. Note that the platform public key $gpk = g_1^\alpha$ which is unknown for $\mathcal{S}$.

- $\mathcal{S}$ chooses $j^* \leftarrow [Q]$ as the guess that the $j^*$-th query $bsn$- to random oracle $H_{G_2}$ is used as the $k'$-th basename $bsn^*$ in the signing queries, where $Q$ is the number of $H_{G_2}$ queries. Without loss of generality, we assume that $\mathcal{S}$ guesses correctly with probability $1/Q$.

- $\mathcal{S}$ maintains a $H_{G_2}$-List which is initially empty. For the $j$-th query $bsn_j$ to random oracle $H_{G_2}$, $\mathcal{S}$ responds as follows:
  - If $bsn_j$ has already been queried, retrieve $(bsn_j, W_j, \ast)$ from $H_{G_2}$-List and return $W_j$.
  - Otherwise, if $j = j^*$, respond with $g_2^\gamma$ and adds $(bsn^*_j, g_2^\gamma, \ast)$ to $H_{G_2}$-List.
  - If $j \neq j^*$, pick $r \leftarrow \mathbb{Z}_p$, compute $W_j \leftarrow g_2^r$, adds $(bsn_j, W_j, r)$ to $H_{G_2}$-List, and respond with $W_j$.\)
– When generating signatures w.r.t. the $k'$-th basename $bsn^*$ for $\mathcal{M}_k$, we modify $\mathcal{F}$ to output a signature as follows:

1. Choose $T_1 \leftarrow G_1^*$ and compute $T_2 \leftarrow T_1^*$, and then pick $Y' \leftarrow G_1$.
2. Set $B \leftarrow \bot$ and $K \leftarrow e(g_1, g_2)^\lambda$.
3. Send $(T_1, T_2, Y', B, K)$ to $\mathcal{S}$, who runs Sim$_2$ to generate a proof $\pi_2$ and sends $\pi_2$ back to $\mathcal{F}$.
4. Output a signature $\sigma = (T_1, T_2, Y', B, K, \pi_2)$.

– Signing queries with $\mathcal{M}_k$ and later basenames are handled by $\mathcal{S}$. To generate a signature w.r.t. $bsn \neq \bot$, $\mathcal{S}$ does the following:

1. Choose $T_1 \leftarrow G_1^*$ and compute $T_2 \leftarrow T_1^*$, and then pick $Y' \leftarrow G_1$.
2. Set $B \leftarrow \bot$ and retrieve $(bsn, *, r)$ from $H_{G_2}$-List.
3. Compute a pseudonym $K \leftarrow e(g_1^\alpha, (g_2^\delta)^r)$. Thus, $K = e(g_1^\delta, g_2^\gamma)^\alpha = e(g, H_{G_2}(bsn))^{gsk}$.
4. Run Sim$_2$ to generate a proof $\pi_2$ and output $\sigma = (T_1, T_2, Y', B, K, \pi_2)$.

By the zero-knowledge property of SPK$_h$, proof $\pi_h$ generated by Sim$_h$ is computationally indistinguishable from the real proof generated in the real-world join protocol. The elements $T_1, T_2, Y'$ has the same distribution as the ones generated by the sig algorithm as well as the real-world sign protocol. Due to the zero-knowledge property of SPK$_2$, proof $\pi_2$ created by Sim$_2$ is computationally indistinguishable from the one generated with the witnesses. If $\mathcal{S}$ guesses correctly, $H_{G_2}(bsn^*) = g_2^\delta$, and thus $K = e(g, H_{G_2}(bsn^*))^{gsk} = e(g_1, g_2)^{\alpha\beta\delta}$ will be a pseudonym computed in the real-world sign protocol.

Thus, we have that $G_{k,k'}^8$ is computationally indistinguishable to Game $8.k.k' - 1$ (resp., Game $8.k.k'$) if $\chi = \alpha\beta\delta$ (resp., $\chi \mathrel{\bot} \mathbb{Z}_p$) and thus signatures with the $k'$-th basename $bsn^*$ are based on the key $gsk = \alpha$ from the join phase (resp., a fresh key). Thus, no polynomial-time distinguisher can distinguish Game $8.k.k'$ from Game $8.k.k' - 1$. In all, Game $8 \approx$ Game $7$.

**Game 9.** Now, $\mathcal{F}$ checks whether the platform’s attributes fulfill the attribute predicate or not when the host is honest, and no longer reveals $(m, bsn, \tilde{p})$ to $\mathcal{S}$, but only the leakage $l(m, bsn, \tilde{p})$. All the adversary notices are the leakage of the secure channel between the TPM and host. $\mathcal{S}$ can still simulate this by taking dummy messages, basenames and attribute predicates that result in the same leakage and using the values to simulate the real-world protocol.

In the real-world sign protocol, the host checks if his attributes fulfill the given attribute predicate. $\mathcal{F}$ does the same check in the SIGN interface. Thus, no adversary can notice the change for $\mathcal{F}$. As simulator $\mathcal{S}$ guarantees that the dummy attribute predicate still holds for the platform’s attributes, any signing query that would previously succeed will still succeed. Thus, we have Game $9 = $ Game $8$.

**Game 10.** $\mathcal{F}$ no longer informs the simulator about the attributes of an honest platform in the join phase. $\mathcal{S}$ now uses dummy attributes in the join protocol. Moreover, $\mathcal{F}$ now only allows platforms that joined with attributes fulfilling the attribute predicate to sign, when $\mathcal{I}$ is honest (denoting this check by $joinatt$).

Although dummy attributes are used by $\mathcal{S}$ in the join protocol, this does not change the view of the adversary, as the credential and attributes of an honest platform are sent confidentially over $\mathcal{F}_{auth}$ by using an encryption scheme. Functionality $\mathcal{F}$ checks whether the attribute predicate holds for the platform’s attributes, and only then will $\mathcal{S}$ be notified. Thus, $\mathcal{S}$ knows that it has to simulate with some dummy attribute predicate that holds for the dummy attributes that it chooses in the join protocol. When both the TPM and host are honest, a signature is generated by $\mathcal{F}$, and thus contains the correct attributes and attribute predicate.

We show that this check $joinatt$ does not change the view of environment $\mathcal{E}$. Before signing with TPM $\mathcal{M}_i$ in the real-world protocol, an honest host $\mathcal{H}_j$ always checks whether it has joined with $\mathcal{M}_i$ and aborts otherwise. So there is no difference for honest hosts. An honest TPM $\mathcal{M}_i$ only signs, if it has joined with some host $\mathcal{H}_j$. Thus, there is no difference for honest TPMs. When an honest TPM $\mathcal{M}_i$ executes the join protocol with a corrupted host $\mathcal{H}_j$ and the honest issuer $\mathcal{I}$, $\mathcal{S}$ will make a join query with $\mathcal{F}$ on behalf of $\mathcal{H}_j$, which guarantees that $\mathcal{M}_i$ and $\mathcal{H}_j$ are in list $\mathcal{M}_j$. Thus, $\mathcal{F}$ still allows any signing that could take place in the real sign protocol.
Overall, we have Game 10 = Game 9.

**Game 11.** In this game, $F$ additionally checks the validity of every new key $gsk$, which is received in the join interface or generated in the sign interface, i.e., Check (iii), Check (iv) and Check (v).

We show that these checks will fail with negligible probability. We only consider valid signatures from VerResults and Signed, where list Signed only contains valid signatures added for honest TPMs and hosts, and $\bot$ added for honest TPMs and corrupt hosts. Note that $\text{identify}(m, bsn, \bot, gsk) = 0$. Thus, we only need to consider valid signatures.

When the TPM is corrupted, $F$ checks that $\text{CheckGskCorrupt}(gsk) = 1$ for the key $gsk$ which is obtained by combining the key $t_{sk}$ extracted from proof $\pi_t$ with the key $hsk$ extracted from proof $\pi_h$. This check prevents the adversary $A$ from choosing a key $gsk \neq gsk'$ such that both keys fit to the same signature. It is impossible, since there exists only a single key $gsk$ for each valid signature such that $\text{identify}(m, bsn, \sigma, gsk) = 1$, where $B \neq 1_{G_1}$ if $bsn = \bot$ and $H_{G_2}(bsn) \neq 1_{G_2}$ with overwhelming probability otherwise. Thus, this check will fail with only negligible probability.

When the TPM is honest, $F$ checks that $\text{CheckGskHonest}(gsk) = 1$ whenever it receives or creates a new key $gsk$. If the host is corrupted, $S$ extracts a key $hsk$ from proof $\pi_h$ and adds this key to a simulated key $t_{sk}$ such that obtaining the key $gsk$. By this check, we avoid the registration of platform keys such that matching signatures already exist. Again, there is one unique key $gsk$ matching a valid signature as $B \neq 1_{G_1}$ if $bsn = \bot$ and $H_{G_2}(bsn) \neq 1_{G_2}$ with overwhelming probability otherwise. Moreover, a key $gsk$ chosen by the ukgen algorithm is uniformly random in an exponentially large group $Z_p$, and this also holds for a simulated key $t_{sk}$ (and thus $gsk$). Thus, the probability that there already is a signature under the key $gsk$ is negligible.

In all, we have Game 11 $\approx$ Game 10.

**Game 12.** Now, after creating a signature, $F$ additionally checks whether the signature passes the verification and matches the correct key, i.e., Check (vi) and Check (vii). Besides, with the help of internal key records Members and DomainKeys, $F$ checks that no platform has already a key matching the newly generated signature, i.e., Check (viii).

Check (vi) will always succeed, since the sig algorithm generates valid signatures. $F$ runs the sig algorithm to set $K = B^{gsk}$ for either a random $B \in G_1^*$ or $B = e(\tilde{g}, H_{G_2}(bsn))$, and thus Check (vii) will also always succeed.

We reduce that Check (viii) fails to the Discrete-Logarithm (DL) assumption in $G_1$, which is implied by the assumptions claimed in Theorem 1. $S$ is given a DL instance $(\tilde{g}, \tilde{g}^\alpha)$ in $G_1$ and attempts to output $\alpha$. $F$ working with $S$ chooses one of signing queries with honest platforms at random, as there are only polynomial many signing queries. For this chosen signing query, $F$ sharing information with $S$ does the following:

1) Set the unknown $\alpha$ as the key $gsk$ and $\tilde{g}^\alpha$ as $gpk$.
2) Run the sig algorithm to create a signature $\sigma$ with the only difference that using Sim2 to simulate a proof $\pi_2$.
3) Output a signature $\sigma$.

When $F$ re-uses the unknown key $\alpha$, it repeats the above same procedure. By the zero-knowledge property of SPK2, $\pi_2$ generated by Sim2 is computationally indistinguishable from the ones created by the sig algorithm. Since $B \neq 1_{G_1}$ for valid signatures and $H_{G_2}(bsn) = 1_{G_2}$ with negligible probability, there is one unique key matching a valid signature with overwhelming probability. If $F$ finds a key $gsk$ matching any of the signatures created by the above process in Members or DomainKeys, it must be the discrete logarithm $\alpha$, and $S$ outputs $gsk$.

Overall, we have Game 12 $\approx$ Game 11.

**Game 13.** In the VERIFY interface, $F$ now additionally checks whether it finds multiple platform keys identifying this signature, i.e., Check (ix). If so, $F$ rejects the signature.
We show that this check does not change the outputs of the VERIFY interface, since any signature that would pass the verification in Game 12 will still pass the verification in this game with overwhelming probability. If a signature $\sigma$ on $m$ w.r.t. $bsn$ and $\hat{p}$ would pass the verification in the previous game, we have $\text{ver}(m, bsn, \sigma, \hat{p}) = 1$. Thus, $B \neq 1_G$ when $bsn = \bot$. The probability that $B = e(\tilde{g}, H_{G_2}(bsn)) = 1_G$ is negligible, as $H_{G_2}$ is a random oracle. A key $gsk$ matching a signature means that $K = B^{gsk}$, and there is only a single key matching the signature when $B \neq 1$. Therefore, the event that multiple keys match a valid signature only occurs if $e(\tilde{g}, H_{G_2}(bsn)) = 1_G$ that happens with negligible probability. Thus, Game 13 $\approx$ Game 12.

**Game 14.** If $I$ is honest, $F$ now only accepts signatures on platform keys and attribute values on which $I$ issued credentials.

This check changes the verification outcome with negligible probability under the assumption that the BBS+ signature is EUF-CMA secure. This assumption holds under the $q$-SDH assumption [ASM06, CDL10a]. $S$ is given a BBS+ public key ($\{h_i\}_{i=0}^n, w$). In the following reduction, $F$ and $S$ share information, and behave exactly as in Game 14 with the following exceptions:

- $S$ runs $\text{Sim}_1$ to generate a proof $\pi_1$ and registers a public key ($\{h_i\}_{i=0}^n, w, \pi_1$).
- When $I$ needs to issue a credential in the join protocol, $S$ runs $\text{Ext}_t$ to extract a TPM key $t_{sk}$ from proof $\pi_t$ if the TPM is corrupted, and runs $\text{Ext}_b$ to extract a witness $(h_{sk}, u')$ from proof $\pi_b$ if the host is corrupted. If the TPM or the host are honest, $S$ knows the related key as it simulates the party. Then, $S$ computes $gsk \leftarrow t_{sk} + h_{sk} \mod p$. Next, $S$ makes a query $(gsk, \text{attrs})$ to its signing oracle and receives a BBS+ signature $(A, x, u)$. Finally, $S$ calculates $u' \leftarrow u - u' \mod p$ and sends $(A, x, u', \text{attrs})$ on behalf of $I$ over $\mathcal{F}_{auth}$.  

- When signing for honest platforms, $F$ uses the signing oracle to generate BBS+ signatures on fresh keys and attributes that the platform joined with. Note that all the platform keys queried to the signing oracle are stored in Members or DomainKeys, and the attributes of platforms are stored in Members.

- When $F$ finds a valid signature $\sigma = (T_1, T_2, Y', B, K, \pi_2)$ w.r.t. attribute predicate $\hat{p} = (D, I)$ such that no matching key $gsk$ has been found for a certain platform with attributes fulfilling $\hat{p}$, $S$ uses $\text{Ext}_2$ to extract a witness $(gsk, \{a_i\}_{i \in D}, x, \tilde{u}, t_2, t_3)$ from $\pi_2$ such that $g_{t_1}^{-1} \prod_{i \in D} h_i^{-a_i} = Y'^{-t_3} \tilde{g}^{gsk} h_0^{a_i} \prod_{i \in D} h_i^{a_i}, T_2 / Y' = T_1^{-x} h_0^{t_2}$ and $K = B^{gsk}$. Then $S$ sets $\text{attrs}$ according the attribute disclosure $(D, I)$ and the extracted attributes $\{a_i\}_{i \in D}$, and computes $A \leftarrow T_1^{t_3}$ and $u \leftarrow \tilde{u} + t_2 \cdot t_3 \mod p$. $S$ outputs $((gsk, \text{attrs}), (A, x, u))$ as a forgery of the BBS+ signature scheme.

Due to the zero-knowledge property of SPK$_{1}$, environment $Z$ cannot distinguish a simulated proof $\pi_1$ from a real proof. By the soundness of SPK$_{h}$ and SPK$_{h}$, $S$ can simulate the executions of join protocol successfully. Below, we show that the extracted credential $(A, x, u)$ is a valid BBS+ signature on a message $(gsk, \text{attrs})$. Since $\sigma$ is a valid signature, the equation $e(T_1, w) = e(T_2, g_2)$ holds, and thus $T_2 = T_1^\gamma$. From the equalities $g_{t_1}^{-1} \prod_{i \in D} h_i^{-a_i} = Y'^{-t_3} \tilde{g}^{gsk} h_0^{a_i} \prod_{i \in D} h_i^{a_i}$ and $T_2 / Y' = T_1^{-x} h_0^{t_2}$, we have the following relation holds: $T_1^{t_3} T_2 = g_{t_1} \tilde{g}^{gsk} h_0^{a_i} \prod_{i = 1}^n h_i^{a_i}$ and $T_1 / Y' = T_1^{-x} h_0^{t_2}$. Replacing $T_1^{t_3}$, $T_2$ and $\tilde{u} + t_2 t_3$ with $A, T_1^\gamma$ and $u$ respectively, we have $A^{\gamma + x} = g_{t_1} \tilde{g}^{gsk} h_0^{a_i} \prod_{i = 1}^n h_i^{a_i}$. Since no matching key $gsk$ has been found for a certain platform with attributes fulfilling $(D, I)$, $F$ and $S$ never make a query $(gsk, \text{attrs})$ to the signing oracle. Overall, we have Game 14 $\approx$ Game 13.

**Game 15.** Now, $F$ rejects any signature $\sigma$ on message $m$ w.r.t. basename $bsn$ and predicate $\hat{p}$ such that $\sigma$ matches the key $gsk$ of a platform with an honest TPM, but the TPM never signed $m$ w.r.t. $bsn$ and $\hat{p}$, i.e., additionally performing Check (xi).

We use a hybrid argument to prove that environment $Z$ cannot notice this change under the DL assumption. We distinguish two cases depending whether the host is honest or not.

For the case that the TPM is honest but the host is corrupt, we proceed the following hybrid argument. Game 15.i is the same as Game 14, except that $F$ performs this check (xi) for the first $i$ platforms with an
honest TPM and a corrupt host. We use a reduction from the DL assumption to bound the difference between Game 15.1 and Game 15.2. \( S \) is given a DL instance \((\hat{g}, \hat{g}^\alpha)\) in \( \mathbb{G}_1 \) and simulates as follows:

- \( S \) sets the unknown \( \alpha \) and \( \hat{g}^\alpha \) as the secret key \( tsk \) and respective public key \( tpk \) of TPM \( M_i \).
- For the join session and sign sessions with \( M_i \), \( S \) uses \( \text{Sim}'' \) to generate the proofs of SPKt.
- As the corresponding host \( H_j \) is corrupted, \( S \) uses \( \text{Ext}_h \) to extract \( hsk \) from proof \( \pi_h \). Then, \( S \) computes \( gpk \leftarrow tpk \cdot \hat{g}^{hsk} \) as the public key of the platform.
- For any verification query with a signature \( \sigma \) w.r.t. \( bsn \neq \bot \), \( F \) can check that \( K = e(gpk, H_{\mathbb{G}_2}(bsn)) \) to decide if \( \sigma \) matches the key \( gsk = \alpha + hsk \). When \( F \) finds a valid signature \( \sigma \) on message \( m \) w.r.t. basename \( bsn \neq \bot \) and attribute predicate \( \tilde{p} \) matching key \( gsk \) but \( M_i \) never signed \( m \) w.r.t. \( bsn \) and \( \tilde{p} \), \( S \) runs \( \text{Ext}_2 \) to extract key \( gsk \) from proof \( \pi_2 \) in signature \( \sigma \). Then \( S \) outputs \( \alpha \leftarrow gsk - hsk \mod p \) as the solution of the DL problem.
- For verification queries with signatures w.r.t. \( bsn = \bot \), \( F \) now skips the check that one pair \((M_i, gsk)\) is found, as it does not know the key \( gsk \). Since there are only polynomial many verification queries, \( F \) chooses one verification query at random as the guess that this is the first verification query with a signature \( \sigma \) on message \( m \) w.r.t. basename \( bsn = \bot \) and predicate \( \tilde{p} \) such that \( \sigma \) is valid and matches the key \( gsk \) but \( M_i \) never signed \( m \) w.r.t. \( bsn \) and \( \tilde{p} \). If \( F \) guesses successfully, \( S \) uses \( \text{Ext}_2 \) to extract the key \( gsk \) from the proof \( \pi_2 \) in signature \( \sigma \). Then \( S \) outputs \( \alpha \leftarrow gsk - hsk \mod p \) as the solution of the DL problem.

For the case that both the TPM and host are honest, we use a reduction from the assumption to bound the difference between Game 14 and Game 15. \( S \) is given a DL instance \((\hat{g}, \hat{g}^\alpha)\) in \( \mathbb{G}_1 \), shares information with \( F \), and simulates as follows:

- Whenever \( F \) would choose a new key \( gsk_i \) to sign for an honest platform, \( F \) picks \( r_i \leftarrow \mathbb{Z}_p^* \) and sets the unknown \( \alpha r_i \) as \( gsk_i \) and computes \( gpk_i \leftarrow (\hat{g}^\alpha)^{r_i} \). Then, \( F \) generates a signature using the sig algorithm and public key \( gpk_i \), except for using \( \text{Sim}_2 \) to simulate a proof \( \pi_2 \).
- For any verification query with a signature \( \sigma \) w.r.t. \( bsn \neq \bot \), \( F \) can check that \( K = e(gpk_i, H_{\mathbb{G}_2}(bsn)) \) to decide if \( \sigma \) matches the key \( gsk_i \). When \( F \) finds a valid signature \( \sigma \) on message \( m \) w.r.t. basename \( bsn = \bot \) and attribute predicate \( \tilde{p} \) matching some key \( gsk_i \) but the platform never signed \( m \) w.r.t. basename \( bsn \) and \( \tilde{p} \), \( S \) runs \( \text{Ext}_2 \) to extract key \( gsk_i \) from proof \( \pi_2 \) in signature \( \sigma \), and outputs \( gsk_i/r_i \mod p \) as the solution of the DL problem.
- For verification queries with signatures w.r.t. \( bsn = \bot \), \( F \) now skips the check that one pair \((*, gsk_i)\) for an honest platform is found, as it cannot know the key \( gsk_i = \alpha r_i \). Since there are only polynomial many verification queries, \( F \) chooses one verification query at random as the guess that the signature \( \sigma \) on message \( m \) w.r.t. basename \( bsn = \bot \) and \( \tilde{p} \) in this query is the first valid signature such that matching some key \( gsk_i \) for an honest platform but the platform never signed \( m \) w.r.t. basename \( bsn = \bot \) and \( \tilde{p} \). If \( F \) guesses correctly, \( S \) uses \( \text{Ext}_2 \) to extract key \( gsk_i \) from the proof \( \pi_2 \) in signature \( \sigma \), and outputs \( gsk_i/r_i \mod p \) as the solution of the DL problem.

By the zero-knowledge of SPKt and SPK2 and the soundness of SPKh and SPK2, Game 15 \( \approx \) Game 14.

**Game 16.** Now \( F \) prevents private key revocation of platforms with an honest TPM.

If an environment \( Z \) can put a key \( gsk \) into the revocation list \( RL \) such that \( gsk \) matches a signature from a platform with an honest TPM, we can construct an algorithm breaking the DL assumption. We show this in two steps: first \( F \) prevents this for pairs \((M_i, gsk)\) from \( \text{Members} \); and then \( F \) prevents this also for pairs \((M_i, gsk)\) from \( \text{DomainKeys} \). Note that for honest platforms there are only pairs \((M_i, gsk)\) in \( \text{DomainKeys} \) such that \( gsk \neq \bot \); for honest TPMs with corrupt hosts, there are only pairs \((M_i, gsk)\) in \( \text{Members} \).

For the case that this check aborts for a pair found in \( \text{Members} \), we can solve the DL problem. \( S \) is given a DL instance \((\hat{g}, \hat{g}^\alpha)\) in \( \mathbb{G}_1 \), \( S \) chooses one platform with honest TPM \( M_i \) and corrupt host \( H_j \) at random as the guess that \((M_i, \hat{g})\) is the first pair such that this check aborts. \( S \) sets \( \hat{g}^\alpha \) as the public key \( tpk \) of \( M_i \), and extracts \( hsk \) from proof \( \pi_h \). Then \( S \) uses \( \text{Sim}'' \) to simulate the proofs of SPKt in the join and sign protocols for \( M_i \). When \( F \) finds a key \( gsk \) in the revocation list \( RL \) matching a signature from the platform with honest
\[ M_i, S \text{ outputs } gsk - hsk \mod p \text{ as the solution of the DL problem, since there is only one key matching a signature.} \]

For the case that this check aborts for a pair found in DomainKeys, we can solve the DL problem. \( S \) is given a DL instance \((\overline{g}, \overline{g}^\alpha)\) in \( G_1 \). Whenever \( F \) would choose a new key \( gsk_i \) to sign for an honest platform, \( F \) picks \( r_i \leftarrow Z_p^* \) and sets the unknown \( \alpha r_i \) as \( gsk_i \) and computes \( gpk_i \leftarrow (\overline{g}^\alpha)^r_i \). Then, \( F \) generates a signature using the \text{sig} algorithm and public key \( gpk_i \), except for using \text{Sim}_2 \) to simulate a proof \( \pi_2 \). When \( F \) finds a key \( gsk \) matching one signature created by \( gsk_i \) in the revocation list RL, \( S \) outputs \( gsk/r_i \mod p \) as the solution of the DL problem.

In all, we have Game 16 \( \approx \) Game 15.

**Game 17.** \( F \) performs all the additional checks done by \( F^l_{\text{daa}} \) for the LINK interface, i.e., Check (xv) and Check (xvi).

We show that these checks do not change the output of the link queries. If a platform key matching one of two signatures but not the other, \( F \) outputs \( f = 0 \). If one key matches both signatures, \( F \) outputs \( f = 1 \). For the signatures that have already been verified, we have \( B \neq 1_{G_1} \) if \( bsn = \perp \) and \( e(\overline{g}, H_{G_2}(bsn)) \neq 1_{G_T} \) with overwhelming probability otherwise. Thus, there is one unique key \( gsk \in Z_p \) such that \( \text{identify}(m, bsn, \sigma, gsk) = 1 \) with overwhelming probability. If there is a key \( gsk \) that matches one of two signatures but not the other, we have \( K_0 \neq K_1 \) and the link algorithm would also output 0 by the soundness of \( \text{SPK}_2 \). If there is some key \( gsk \) matching both signatures, we have \( K_0 = K_1 \) and the link algorithm would also output 1 by the soundness of \( \text{SPK}_2 \).

Overall, we have Game 17 \( \approx \) Game 16.

The functionality in Game 17 is equal to \( F^l_{\text{daa}} \), i.e., Game 17 = \text{IDEAL}_{F^l_{\text{daa}}, S, Z}, \) which completes our proof. \( \square \)

D.1 Functionalities and Simulators
Setup
1. On input (SETUP, sid) from issuer I.
   - Output (FORWARD, (SETUP, sid), I) to S.

Join
2. On input (JOIN, sid, jsid, M_i) from host H_j.
   - Output (FORWARD, (JOIN, sid, jsid, M_i, H_j) to S.
3. On input (JOINPROCEED, sid, jsid, attrs) from I with attrs \in H_1 \times \cdots \times H_n.
   - Output (FORWARD, (JOINPROCEED, sid, jsid, attrs), I) to S.

Sign
4. On input (SIGN, sid, ssid, M_i, m, bsn, \tilde{p}) from host H_j with \tilde{p} \in P.
   - Output (FORWARD, (SIGN, sid, ssid, M_i, m, bsn, \tilde{p}), H_j) to S.
5. On input (SIGNPROCEED, sid, ssid) from M_i.
   - Output (FORWARD, (SIGNPROCEED, sid, ssid), M_i) to S.

Verify
6. On input (VERIFY, sid, m, bsn, \sigma, \hat{p}, RL) from some party V.
   - Output (FORWARD, (VERIFY, sid, m, bsn, \sigma, \hat{p}, RL), V) to S.

Link
7. On input (LINK, sid, m_0, \sigma_0, \tilde{p}_0, m_1, \sigma_1, \tilde{p}_1, bsn) from some party V with bsn \neq \bot.
   - Output (FORWARD, (LINK, sid, m_0, \sigma_0, \tilde{p}_0, m_1, \sigma_1, \tilde{p}_1, bsn), V) to S.

Output
8. On input (OUTPUT, P, m) from S.
   - Output (m) to P.

Fig. 6: Functionality \( \mathcal{F} \) for Game 3

Setup
- Upon receiving (FORWARD, (SETUP, sid), I) from \( \mathcal{F} \), S provides I with input ( SETUP, sid).

Join
- Upon receiving (FORWARD, (JOIN, sid, jsid, M_i), H_j) from \( \mathcal{F} \), S provides H_j with input (JOIN, sid, jsid, M_i).
- Upon receiving (FORWARD, (JOINPROCEED, sid, jsid, attrs), I) from \( \mathcal{F} \), S provides “I” with input (JOINPROCEED, sid, jsid, attrs).

Sign
- Upon receiving (FORWARD, (SIGN, sid, ssid, M_i, m, bsn, \tilde{p}), H_j) from \( \mathcal{F} \), S provides “H_j” with input (SIGN, sid, ssid, M_i, m, bsn, \tilde{p}).
- Upon receiving (FORWARD, (SIGNPROCEED, sid, ssid), M_i) from \( \mathcal{F} \), S provides “M_i” with input (SIGNPROCEED, sid, ssid).

Verify
- Upon receiving (FORWARD, (VERIFY, sid, m, bsn, \sigma, \hat{p}, RL), V) from \( \mathcal{F} \), S provides “V” with input (VERIFY, sid, m, bsn, \sigma, \hat{p}, RL).

Link
- Upon receiving (FORWARD, (LINK, sid, m_0, \sigma_0, \tilde{p}_0, m_1, \sigma_1, \tilde{p}_1, bsn), V) from \( \mathcal{F} \), S provides “V” with input (LINK, sid, m_0, \sigma_0, \tilde{p}_0, m_1, \sigma_1, \tilde{p}_1, bsn).

Output
- When any party “P” simulated by S outputs a message m, S sends (OUTPUT, P, m) to functionality \( \mathcal{F} \).

Fig. 7: Simulator for Game 3
Setup
1. Issuer Setup. On input (SETUP, sid) from issuer $I$.
   – Verify that $sid = (I, sid')$ and output (SETUP, sid) to $S$.
   – Check that ver, link and identify are deterministic.
   – Store (sid, ukgen, sig, ver, link, identify) and output (SETUPDONE, sid) to $I$.

Join
3. On input (JOIN, sid, jsid, $M_i$) from host $H_j$.
   – Output (FORWARD, (JOIN, sid, jsid, $M_i$), $H_j$) to $S$.
4. On input (JOINPROCEED, sid, jsid, attrs) from $I$ with attrs $\in A_1 \times \cdots \times A_n$.
   – Output (FORWARD, (JOINPROCEED, sid, jsid, attrs), $I$) to $S$.

Sign
5. On input (SIGN, sid, ssid, $M_i$, m, bsn, $\hat{p}$) from host $H_j$ with $\hat{p} \in P$.
   – Output (FORWARD, (SIGN, sid, ssid, $M_i$, m, bsn, $\hat{p}$), $H_j$) to $S$.
6. On input (SIGNPROCEED, sid, ssid) from $M_i$.
   – Output (FORWARD, (SIGNPROCEED, sid, ssid), $M_i$) to $S$.

Verify
7. On input (VERIFY, sid, m, bsn, $\sigma$, $\hat{p}$, RL) from some party $V$.
   – Output (FORWARD, (VERIFY, sid, m, bsn, $\sigma$, $\hat{p}$, RL), $V$) to $S$.

Link
8. On input (LINK, sid, $m_0$, $\sigma_0$, $\hat{p}_0$, $m_1$, $\sigma_1$, $\hat{p}_1$, bsn) from some party $V$ with $bsn \neq \perp$.
   – Output (FORWARD, (LINK, sid, $m_0$, $\sigma_0$, $\hat{p}_0$, $m_1$, $\sigma_1$, $\hat{p}_1$, bsn), $V$) to $S$.

Output
9. On input (OUTPUT, $P$, m) from $S$.
   – Output (m) to $P$.

Fig. 8: Functionality $\mathcal{F}$ for Game 4
Setup

Honest Issuer $I$
- On input (SETUP, sid) from $F$.
  - Try to parse sid as $(I, sid')$ and output $\perp$ to $I$ if that fails.
  - Provide “I” with input (SETUP, sid).
  - Upon receiving an output (SETUPDONE, sid) from “I”, $S$ creates its public key $ipk = (\{h_i\}_{i=0}^{n}, w, \pi_1)$ and secret key $isk = \gamma$ following the specification of $DA_OPT$.
  - Define ukgen() as follows: choose $gsk \leftarrow Z_p$ and output $gsk$.
  - Define sig($gsk, m, bsn, p$) as follows:
    1. Create a BBS+ credential $(A, x, u)$ on $gsk$ and attributes $attrs = (a_1, \ldots, a_n)$ where the disclosed attributes are taken from predicate $\hat{p} = (D, I)$ and the undisclosed attributes are set as dummy values.
    2. Compute $gpk \leftarrow g^{a_k} \mod p$ and $Y = g_1 \cdot gpk \cdot h_0^1 \cdot \prod_{i=1}^{n} h_i^{a_i}$.
    3. Following the computational operations at the host side in the real-world sign protocol, randomize $A, Y$ and compute $B, K$ as follows:
       a) Choose $t_1 \leftarrow Z_p^*$ and $t_2 \leftarrow Z_p$, and compute $T_1 \leftarrow A^{t_1}, T_2 \leftarrow Y^{t_1} \cdot T_1^{-x}$ and $Y' \leftarrow Y^{t_1} \cdot h_0^{-t_2}$.
       b) If $bsn = \perp$, pick $b \leftarrow Z_p^*$ and compute $B \leftarrow g^b, K \leftarrow gpk^b$; Otherwise, set $B \leftarrow \perp$ and compute $K \leftarrow e(gpk, H_{o2}(bsn))$.
    4. Compute $t_3 = t_1^{-1} \mod p$ and $\bar{u} = u - t_2 \cdot t_3 \mod p$. Without the necessity of distributing the computations between the TPM and host, straightforwardly generate a proof $\pi \leftarrow SPK_2\{\langle gsk, \{a_i\}_{i \in D}, x, \bar{u}, t_2, t_3\rangle : g_1^{-1} \prod_{i \in D} h_i^{a_i} = Y^{-r_3} g^{a_k} h_0^{1} \prod_{i \in D} h_i^{a_i} \land T_2 / Y' = T_1^{-r_3} h_0^{r_3} \land K = B^{a_k}\}$ (“sign”, $m, bsn, D, I$) as below:
       a) Choose $\bar{r}, r_s, r_0, r_1, r_3 \leftarrow Z_p, N_1 \leftarrow \{0, 1\}^{t_2}$, and $r_a \leftarrow Z_p$ for each $i \in D$.
       b) Compute $R_1 \leftarrow Y^{-r_3} \cdot g^\bar{r}, h_0^{r_0} \cdot \prod_{i \in D} h_i^{r_a} / R_2 \leftarrow T_1^{-r_3} \cdot h_0^{r_2}$.
       c) Calculate $L \leftarrow B^\bar{r}$ if $bsn = \perp$ and $L \leftarrow e(g_0, H_{o2}(bsn))^\bar{r}$ otherwise.
       d) Compute $c_0 \leftarrow H_2(\langle \text{"sign"}, \bar{g}, g, \{h_i\}_{i=0}^{n}, T_1, T_2, Y', B, K, R_1, R_2, L \rangle)$.
       e) Compute $c \leftarrow H_1(N_1, m, bsn, D, I, c_0)$.
       f) Compute $\bar{s} \leftarrow \bar{r} + c \cdot gsk \mod p, s_0 \leftarrow r_s + c \cdot x \mod p, s_a \leftarrow r_a + c \cdot \bar{u} \mod p, s_{t_2} \leftarrow r_{t_2} + c \cdot t_2 \mod p, s_{t_3} \leftarrow r_{t_3} + c \cdot t_3 \mod p$, and $s_a \leftarrow r_a + c \cdot a_i \mod p$ for each $i \in D$ where $a_i$ is a dummy value.
       g) Set $\pi_2 \leftarrow (c, \bar{s}, s_0, s_a, s_{t_2}, s_{t_3}, \{s_{a_i}\}_{i \in D}, N_1)$.
    5. Output a signature $\sigma \leftarrow (T_1, T_2, Y', B, K, \pi_2)$.
       - Define $\text{ver}(m, bsn, \sigma, \hat{p})$ as the real world verification algorithm except that the private key revocation check is omitted.
       - Define link($m_{0}, \sigma_0, \hat{p}_0, m_{1}, \sigma_1, \hat{p}_1, bsn$) as follows: 1) parse the signatures $\sigma_0$ and $\sigma_1$ as $(T_{1,0}, T_{2,0}, Y_{1}', B_0, K_0, \pi_{2,0})$ and $(T_{1,1}, T_{2,1}, Y_{1}', B_1, K_1, \pi_{2,1})$ respectively; 2) output 1 if $K_0 = K_1$ and 0 otherwise.
       - Define $\text{identify}(m, bsn, \sigma, gsk)$ as follows: 1) parse $\sigma$ as $(T_1, T_2, Y', B, K, \pi_2)$; 2) compute $B \leftarrow e(\bar{g}, H_{o2}(bsn))$ if $bsn \neq \perp$; 3) check $gsk \in Z_q$ and $K = B^{a_k}$; 4) output 1 if the check passes and 0 otherwise.
- $S$ sends $(\text{ALG}, sid, \text{ukgen}, \text{sig}, \text{ver}, \text{link}, \text{identify})$ to $F$.

Corrupt Issuer $I$
- $S$ notices this setup as it notices $I$ registering a public key with “$F_{eq}$” with $sid = (I, sid')$.
  - If the registered key $ipk$ is of the form $h_0, h_1, \ldots, h_n, w, \pi_1$ and the proof $\pi_1$ is valid, $S$ uses Ext$_1$ to extract a secret key $\gamma$ from proof $\pi_1$.
  - $S$ defines the algorithms $\text{ukgen}$, $\text{sig}$, $\text{ver}$, $\text{link}$ as before, but now relying on the extracted secret key.
  - $S$ sends (SETUP, sid) to $F$ on behalf of $I$.
- $S$ sends $(\text{ALG}, sid, \text{ukgen}, \text{sig}, \text{ver}, \text{link}, \text{identify})$ to $F$.

Join, Sign, Verify, Link
- Unchanged.

Output
- When any simulated party “$P$” outputs a message $m$ which is not explicitly handled by $S$ yet, $S$ sends (OUTPUT, $P$, $m$) to $F$.

Fig. 9: Simulator for Game 4
Setup
Unchanged

Join
3. On input (JOIN, sid, jsid, M_i) from host H_j,
   – Output (FORWARD, (JOIN, sid, jsid, M_i), H_j) to S.
4. On input (JOINPROCEED, sid, jsid, attrs) from I with $attrs \in A_1 \times \cdots \times A_n$.
   – Output (FORWARD, (JOINPROCEED, sid, jsid, attrs), I) to S.

Sign
5. On input (SIGN, sid, ssid, M_i, m, bsn, \hat{p}) from host H_j with $\hat{p} \in \mathbb{P}$.
   – Output (FORWARD, (SIGN, sid, ssid, M_i, m, bsn, \hat{p}), H_j) to S.
6. On input (SIGNPROCEED, sid, ssid) from M_i.
   – Output (FORWARD, (SIGNPROCEED, sid, ssid), M_i) to S.

Verify
7. Verify. On input (VERIFY, sid, m, bsn, \sigma, \hat{p}, RL) from some party V.
   – Set $f \leftarrow 0$ if at least one of the following conditions hold:
     • There is a $gsk' \in RL$ such that identify($m, bsn, \sigma, gsk'$) = 1.
   – If $f \neq 0$, set $f \leftarrow \text{ver}(m, bsn, \sigma, \hat{p})$.
   – Add $(m, bsn, \sigma, RL, f)$ to VerResults and output (VERIFIED, sid, f) to V.

Link
8. Link. On input (LINK, sid, m_0, \sigma_0, \hat{p}_0, m_1, \sigma_1, \hat{p}_1, bsn) from some party V with $bsn \neq \perp$.
   – Output $\bot$ to V if at least one signature tuple $(m_0, bsn, \sigma_0, \hat{p}_0)$ or $(m_1, bsn, \sigma_1, \hat{p}_1)$ is not valid, which is verified via the VERIFY interface with RL $= \emptyset$.
   – Set $f \leftarrow \text{link}(m_0, \sigma_0, m_1, \sigma_1, bsn)$.
   – Output (LINK, sid, f) to V.

Output
9. On input (OUTPUT, P, m) from S.
   – Output $(m)$ to P.

Fig. 10: Functionality $\mathcal{F}$ for Game 5

Setup
– Unchanged.
Join
– Unchanged.
Sign
– Unchanged.
Verify
– Nothing to simulate.
Link
– Nothing to simulate.
Output
– When any simulated party “P” outputs a message $m$ which is not explicitly handled by S yet, S sends (OUTPUT, P, m) to $\mathcal{F}$.

Fig. 11: Simulator for Game 5
Setup
Unchanged

Join
3. Join Request. On input \((\text{JOIN}, \text{sid}, \text{jsid}, \mathcal{M}_i)\) from host \(\mathcal{H}_j\).
   – Create a join session record \((\text{jsid}, \mathcal{M}_i, \mathcal{H}_j, \text{status})\) with \(\text{status} \leftarrow \text{request}\).
   – Output \((\text{JOINSTART}, \text{sid}, \text{jsid}, \mathcal{M}_i, \mathcal{H}_j)\) to \(\mathcal{S}\).
4. Join Request Delivery. On input \((\text{JOINSTART}, \text{sid}, \text{jsid})\) from \(\mathcal{S}\).
   – Update the session record \((\text{jsid}, \mathcal{M}_i, \mathcal{H}_j, \bot, \text{status})\) to \(\text{status} \leftarrow \text{delivered}\).
   – Abort if \(\mathcal{I}\) or \(\mathcal{M}_i\) is honest and a record \((\mathcal{M}_i, *, *, *) \in \text{Members already exists}\).
   – Output \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, \text{attrs})\) to \(\mathcal{S}\).
5. Join Proceed. On input \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, \text{gsk})\) from \(\mathcal{S}\).
   – Look up record \((\text{jsid}, \mathcal{M}_i, \mathcal{H}_j, \text{attrs}, \text{status})\) with \(\text{status} = \text{complete}\).
   – If \(\mathcal{M}_i\) and \(\mathcal{H}_j\) are honest, set \(\text{gsk} \leftarrow \bot\).
   – Add \((\mathcal{M}_i, \mathcal{H}_j, \text{gsk}, \text{attrs})\) into \text{Members} and output \((\text{JOINED}, \text{sid}, \text{jsid})\) to \(\mathcal{H}_j\).

Sign
7. On input \((\text{SIGN}, \text{sid}, \text{ssid}, \mathcal{M}_i, \text{bsn}, \hat{p})\) from host \(\mathcal{H}_j\) with \(\hat{p} \in \mathbb{P}\).
   – Output \((\text{FORWARD}, \text{SIGN}, \text{sid}, \text{ssid}, \mathcal{M}_i, \text{bsn}, \hat{p})\) to \(\mathcal{S}\).
8. On input \((\text{SIGNCOMPLETE}, \text{sid}, \text{ssid})\) from \(\mathcal{M}_i\).
   – Output \((\text{FORWARD}, \text{SIGNCOMPLETE}, \text{sid}, \text{ssid})\) to \(\mathcal{S}\).

Verify
9. Verify. On input \((\text{VERIFY}, \text{sid}, \text{m}, \text{bsn}, \sigma, \hat{p}, \mathcal{RL})\) from some party \(\mathcal{V}\).
   – Set \(f \leftarrow 0\) if at least one of the following conditions hold:
     ● There is a \(\text{gsk}' \in \mathcal{RL}\) such that \(\text{identify}(\text{m}, \text{bsn}, \sigma, \text{gsk}' ) = 1\).
     – If \(f \neq 0\), set \(f \leftarrow \text{ver}(\text{m}, \text{bsn}, \sigma, \hat{p})\).
     – Add \((\text{m}, \text{bsn}, \sigma, \text{RL}, f)\) to \(\text{VerResults}\) and output \((\text{VERIFIED}, \text{sid}, f)\) to \(\mathcal{V}\).

Link
10. Link. On input \((\text{LINK}, \text{sid}, \text{m}_0, \sigma_0, \text{bsn}_0, \text{m}_1, \sigma_1, \text{bsn})\) from some party \(\mathcal{V}\) with \(\text{bsn} \neq \bot\).
    – Output \(\bot\) to \(\mathcal{V}\) if at least one signature tuple \((\text{m}_0, \text{bsn}, \sigma_0, \text{bsn}_0)\) or \((\text{m}_1, \text{bsn}, \sigma_1, \text{bsn})\) is not valid, which is verified via the \text{VERIFY} interface with \(\mathcal{RL} = \emptyset\).
    – Set \(f \leftarrow \text{link}(\text{m}_0, \sigma_0, \text{bsn}_0, \text{bsn})\).
    – Output \((\text{LINK}, \text{sid}, f)\) to \(\mathcal{V}\).

Output
11. On input \((\text{OUTPUT}, \mathcal{P}, \text{m})\) from \(\mathcal{S}\).
    – Output \((\text{m})\) to \(\mathcal{P}\).

Fig. 12: Functionality \(\mathcal{F}\) for Game 6
Setup, Sign: Unchanged.

Join

**Honest \( \mathcal{H}, I \)**
- \( S \) receives \((\text{JOINSTART}, \text{sid}, \text{jsid}, M_i, \mathcal{H}_i)\) from \( F \).
  - \( S \) simulates the real-world protocol via giving “\( I \)” input \((\text{JOIN}, \text{sid}, \text{jsid}, M_i)\) and waits for output \((\text{JOINPROCEED}, \text{sid}, \text{jsid}, M_i, \mathcal{H}_i)\) from “\( I \)”.
  - If \( M_i \) is honest, \( S \) knows \( tsk \) as it is simulating \( M_i \). If \( M_i \) is corrupted, \( S \) runs \text{Ext} to extract \( tsk \) from proof \( \pi_t \). Since \( S \) simulates the honest host \( \mathcal{H}_i \), it knows \( hsk \). Finally, \( S \) computes \( gsk \leftarrow tsk + hsk \mod p \).
  - \( S \) sends \((\text{JOINSTART}, \text{sid}, \text{jsid})\) to \( F \).
- On input \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, \text{attrs})\) from \( F \).
  - \( S \) continues the simulation by giving “\( I \)” input \((\text{JOINPROCEED}, \text{sid}, \text{jsid}, \text{attrs})\), and waits for output \((\text{JOINED}, \text{sid}, \text{jsid})\) from “\( H_i \)”.
  - Output \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, gsk)\) to \( F \).

**Honest \( \mathcal{H}, \text{Corrupt} \ I \)**
- On input \((\text{JOINSTART}, \text{sid}, \text{jsid}, M_i, \mathcal{H}_i)\) from \( F \).
  - \( S \) simulates the real-world protocol via giving “\( H_i \)” input \((\text{JOIN}, \text{sid}, \text{jsid}, M_i)\) and waits for output \((\text{JOINPROCEED}, \text{sid}, \text{jsid})\) from “\( H_i \)”.
  - \( S \) knows which attributes \( \text{attrs} \) the corrupted issuer issued to “\( H_i \)”, as it simulates “\( H_i \)”.
  - \( S \) sends \((\text{JOINSTART}, \text{sid}, \text{jsid})\) to \( F \).
- Upon receiving \((\text{JOINPROCEED}, \text{sid}, \text{jsid}, M_i)\) from \( F \), \( S \) sends \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, \text{attrs})\) to \( F \) on behalf of \( I \).
- Upon receiving \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, \text{attrs})\) from \( F \), \( S \) sends \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, \bot)\) to \( F \).

**Honest \( M, \text{Corrupt} \ I \)**
- \( S \) notices the join as “\( I \)” outputs \((\text{JOINPROCEED}, \text{sid}, \text{jsid}, M_i)\).
  - \( S \) knows the identity of the host \( \mathcal{H}_i \) involved in the join session, as it is simulating “\( M_i \)”.
  - \( S \) takes \( tsk \) from simulating “\( M_i \)” and runs \text{Ext} to extract \( hsk \) from proof \( \pi_h \). Then \( S \) sets \( gsk \leftarrow tsk + hsk \mod p \).
  - \( S \) sends \((\text{JOIN}, \text{sid}, \text{jsid}, M_i)\) on behalf of \( \mathcal{H}_i \) to \( F \).
- \( S \) receives \((\text{JOINSTART}, \text{sid}, \text{jsid}, M_i, \mathcal{H}_i)\) from \( F \).
  - \( S \) continues the simulation of “\( M_i \)” until “\( I \)” outputs \((\text{JOINPROCEED}, \text{sid}, \text{jsid}, M_i)\).
  - \( S \) sends \((\text{JOINSTART}, \text{sid}, \text{jsid})\) to \( F \).
- Upon receiving \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, \text{attrs})\) from \( F \), \( S \) sends \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, gsk)\) to \( F \).
- Upon receiving \((\text{JOINED}, \text{sid}, \text{jsid})\) from \( F \) as host \( \mathcal{H}_i \) is corrupted, \( S \) continues the simulation via giving “\( I \)” input \((\text{JOINPROCEED}, \text{sid}, \text{jsid}, \text{attrs})\).

**Honest \( I, \text{Corrupt} \ M \)**
- \( S \) notices the join as “\( I \)” receives \((\text{SENT}, (M_i, I, \text{sid}'), \text{jsid}, (\text{tpk}, C, \pi_t, \pi_h), \mathcal{H}_i')\) from \( F_{\text{adv}} \).
  - \( S \) runs \text{Ext} to extract \( tsk \) from proof \( \pi_t \) and uses \text{Ext} to extract \( hsk \) from proof \( \pi_h \). Then, \( S \) sets \( gsk \leftarrow tsk + hsk \mod p \).
  - \( S \) does not know the exact identity of the host who launched the join session. So, \( S \) chooses an arbitrary corrupt host \( \mathcal{H}_i \) and proceeds as if it is the host who launched the join session. For corrupt platforms, the exact identity of the host does not matter.
  - \( S \) sends \((\text{JOIN}, \text{sid}, \text{jsid}, M_i)\) to \( F \) on behalf of \( \mathcal{H}_i \).
- \( S \) receives \((\text{JOINSTART}, \text{sid}, \text{jsid}, M_i, \mathcal{H}_i)\) from \( F \).
  - \( S \) continues simulating “\( I \)” until it outputs \((\text{JOINPROCEED}, \text{sid}, \text{jsid}, M_i)\).
  - \( S \) sends \((\text{JOINSTART}, \text{sid}, \text{jsid})\) to \( F \).
- Upon receiving \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, \text{attrs})\) from \( F \), \( S \) sends \((\text{JOINCOMPLETE}, \text{sid}, \text{jsid}, gsk)\) to \( F \).
- Upon receiving \((\text{JOINED}, \text{sid}, \text{jsid})\) from \( F \) as host \( \mathcal{H}_i \) is corrupted, \( S \) continues the simulation by giving “\( I \)” input \((\text{JOINPROCEED}, \text{sid}, \text{jsid}, \text{attrs})\).

**Honest \( M, \text{Corrupt} \ I \)**
- \( S \) notices this join as “\( M_i \)” receives a message \((\text{TPMCreate}, \text{sid}, \text{jsid})\) from host \( \mathcal{H}_i \).
- \( S \) simply simulates “\( M_i \)” honestly, and does not need to involve \( F \), since \( M_i \) does not receive inputs or send outputs in the join related interfaces, and \( F \) does not guarantee any security property for platforms with corrupt hosts when the issuer is corrupted.

Verify, Link: Nothing to simulate.

Output: When any simulated party “\( P \)” outputs a message \( m \) which is not explicitly handled by \( S \) yet, \( S \) sends \((\text{OUTPUT}, P, m)\) to \( F \).

Fig. 13: Simulator for Game 6
Setup
Unchanged

Join
Unchanged

Sign with $bsn \neq \perp$

1. On input $(SIGN, sid, ssid, M, bsn, \hat{p})$ from host $H_j$ with $\hat{p} \in P$
   
   - Output $(FORWARD, (SIGN, sid, ssid, M, bsn, \hat{p}), H_j)$ to $S$.

2. On input $(SIGNPROCEED, sid, ssid)$ from $M_i$
   
   - Output $(FORWARD, (SIGNPROCEED, sid, ssid), M_i)$ to $S$.

Sign with $bsn = \perp$

3. Sign Request. On input $(SIGN, sid, ssid, M, bsn, \hat{p})$ with $bsn = \perp$ and $\hat{p} \in P$ from host $H_j$
   
   - Create a sign session record $(ssid, M, H_j, m, bsn, \hat{p}, status)$ with $status \leftarrow request$.

4. Sign Proceed. On input $(SIGNPROCEED, sid, ssid)$ from $M_i$
   
   - Look up record $(ssid, M, H_j, m, bsn, \hat{p}, status)$ to $status \leftarrow delivered$.

5. Update the session record $(ssid, M, H_j, m, bsn, \hat{p}, status)$ to $status \leftarrow verified$.

6. Set $status = ERROR$.

8. Output $(SIGNCOMPLETE, sid, ssid, gsk)$ to $S$.

9. If $M_i$ and $H_j$ are honest, ignore $\sigma$ from $S$ and internally generate a signature for a fresh or established $gsk$:

   - As $bsn = \perp$, generate $\sigma \leftarrow \text{ukgen}()$, and then store $(M, bsn, gsk)$ in DomainKeys.

   - Compute a signature as $\sigma \leftarrow \text{sig}(gsk, m, bsn, \hat{p})$.

10. If $M_i$ is honest, store $(m, bsn, \sigma, M_i, \hat{p})$ in Signed.

11. Output $(SIGNATURE, sid, ssid, \sigma)$ to $H_j$.

Verify

12. Verify. On input $(VERIFY, sid, m, bsn, \sigma, \hat{p}, RL)$ from some party $V$.

   - Set $f \leftarrow 0$ if at least one of the following conditions hold:

     - There is a $gsk' \in RL$ such that identify$(m, bsn, \sigma, gsk') = 1$.

   - If $f \neq 0$, set $f \leftarrow \text{ver}(m, bsn, \sigma, \hat{p})$.

13. Add $(m, \sigma, RL, f)$ to $\text{VerResults}$ and output $(VERIFIED, sid, f)$ to $V$.

Link

14. Link. On input $(LINK, sid, m_0, \sigma_0, \hat{p}_0, m_1, \sigma_1, \hat{p}_1, bsn)$ from some party $V$ with $bsn \neq \perp$

   - Output $\perp$ to $V$ if at least one signature tuple $(m_0, bsn, \sigma_0, \hat{p}_0)$ or $(m_1, bsn, \sigma_1, \hat{p}_1)$ is not valid, which is verified via the VERIFY interface with $RL = \emptyset$.

   - Set $f \leftarrow \text{link}(m_0, \sigma_0, m_1, \sigma_1, bsn)$.

   - Output $(LINK, sid, f)$ to $V$.

Output

15. On input $(OUTPUT, P, m)$ from $S$.

   - Output $(m)$ to $P$.

Fig. 14: Functionality $F$ for Game 7
Setup
- Unchanged.

Join
- Unchanged.

Sign with $bsn \neq \bot$
- Upon receiving $(\text{FORWARD}, (\text{SIGN}, sid, ssid, M_i, m, bsn, \hat{p}), \mathcal{H}_j)$ from $\mathcal{F}$, $S$ provides “$\mathcal{H}_j$” with input $(\text{SIGN}, sid, ssid, M_i, m, bsn, \hat{p})$.
- Upon receiving $(\text{FORWARD}, (\text{SIGNPROCEED}, sid, ssid), M_i)$ from $\mathcal{F}$, $S$ provides “$M_i$” with input $(\text{SIGNPROCEED}, sid, ssid)$.

Sign with $bsn = \bot$

$Honest \ M, \mathcal{H}$
- Upon receiving $(\text{SIGNSTART}, sid, ssid, m, bsn, \hat{p}, M_i, \mathcal{H}_j)$ with $bsn = \bot$ from $\mathcal{F}$.
  - $S$ starts the simulation via giving “$\mathcal{H}_j$” input $(\text{SIGN}, sid, ssid, M_i, m, bsn, \hat{p})$.
  - When “$M_i$” outputs $(\text{SIGNPROCEED}, sid, ssid, m, bsn, \hat{p})$, $S$ sends $(\text{SIGNSTART}, sid, ssid)$ to $\mathcal{F}$.
- Upon receiving $(\text{SIGNCOMPLETE}, sid, ssid)$ from $\mathcal{F}$.
  - $S$ continues the simulation by giving “$M_i$” input $(\text{SIGNPROCEED}, sid, ssid)$.
  - When “$\mathcal{H}_j$” outputs $(\text{SIGNATURE}, sid, ssid, \sigma)$, $S$ sends $(\text{SIGNCOMPLETE}, sid, ssid, \bot)$ to $\mathcal{F}$.

$Honest \ \mathcal{H}, \ Corrupt \ M$
- Upon receiving $(\text{SIGNSTART}, sid, ssid, m, bsn, \hat{p}, M_i, \mathcal{H}_j)$ with $bsn = \bot$ from $\mathcal{F}$.
  - $S$ sends $(\text{SIGNSTART}, sid, ssid)$ to $\mathcal{F}$.
- Upon receiving $(\text{SIGNPROCEED}, sid, ssid, m, bsn, \hat{p})$ from $\mathcal{F}$ as $M_i$ is corrupted.
  - $S$ starts the simulation by giving “$\mathcal{H}_j$” input $(\text{SIGN}, sid, ssid, M_i, m, bsn, \hat{p})$.
  - When “$\mathcal{H}_j$” outputs $(\text{SIGNATURE}, sid, ssid, \sigma)$, $S$ sends $(\text{SIGNPROCEED}, sid, ssid)$ to $\mathcal{F}$ on behalf of $M_i$.
- Upon receiving $(\text{SIGNCOMPLETE}, sid, ssid)$ from $\mathcal{F}$.
  - $S$ sends $(\text{SIGNCOMPLETE}, sid, ssid, \sigma)$ to $\mathcal{F}$.

$Honest \ M, \ Corrupt \ \mathcal{H}$
- $S$ notices this sign session as “$M_i$” receives $(sid, ssid, m, bsn, \hat{p})$ from $\mathcal{H}_j$.
  - $S$ sends $(\text{SIGN}, sid, ssid, M_i, m, bsn, \hat{p})$ to $\mathcal{F}$ on behalf of $\mathcal{H}_j$.
- Upon receiving $(\text{SIGNSTART}, sid, ssid, m, bsn, \hat{p}, M_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - $S$ continues the simulation of “$M_i$” until it outputs $(\text{SIGNPROCEED}, sid, ssid, m, bsn, \hat{p})$.
  - $S$ sends $(\text{SIGNSTART}, sid, ssid)$ to $\mathcal{F}$.
- Upon receiving $(\text{SIGNCOMPLETE}, sid, ssid)$ from $\mathcal{F}$.
  - $S$ sends $(\text{SIGNCOMPLETE}, sid, ssid, \bot)$ to $\mathcal{F}$.
- Upon receiving $(\text{SIGNATURE}, sid, ssid, \bot)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupted.
  - $S$ continues the simulation via giving “$M_i$” input $(\text{SIGNPROCEED}, sid, ssid)$.

Verify
- Nothing to simulate.

Link
- Nothing to simulate.

Output
- When any simulated party “$\mathcal{P}$” outputs a message $m$ which is not explicitly handled by $S$ yet, $S$ sends $(\text{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$.

Fig. 15: Simulator for Game 7
Setup
  Unchanged

Join
  Unchanged

Sign
7. Sign Request. On input \((\text{SIGN}, \text{sid}, \text{ssid}, M_i, m, bsn, \hat{p})\) with \(bsn = \bot\) and \(\hat{p} \in \mathcal{P}\) from host \(H_j\).
   - Create a sign session record \(\langle \text{ssid}, M_i, H_j, m, bsn, \hat{p}, \text{status} \rangle\) with \(\text{status} \leftarrow \text{request}\).
   - Output \((\text{SIGNSTART}, \text{sid}, \text{ssid}, m, bsn, \hat{p}, M_i, H_j)\) to \(S\).
8. Sign Request Delivery. On input \((\text{SIGNSTART}, \text{sid}, \text{ssid})\) from \(S\).
   - Update the session record \(\langle \text{ssid}, M_i, H_j, m, bsn, \hat{p}, \text{status} \rangle\) to \(\text{status} \leftarrow \text{delivered}\).
   - Output \((\text{SIGNPROCEED}, \text{sid}, \text{ssid}, m, bsn, \hat{p})\) to \(M_i\).
9. Sign Proceed. On input \((\text{SIGNPROCEED}, \text{sid}, \text{ssid})\) from \(M_i\).
   - Look up record \(\langle \text{ssid}, M_i, H_j, m, bsn, \hat{p}, \text{status} \rangle\) with \(\text{status} = \text{delivered}\).
   - Output \((\text{SIGNCOMPLETE}, \text{sid}, \text{ssid})\) to \(S\).
10. Signature Generation. On input \((\text{SIGNCOMPLETE}, \text{sid}, \text{ssid}, \sigma)\) from \(S\).
    - If \(M_i\) and \(H_j\) are honest, ignore \(\sigma\) from \(S\) and internally generate a signature for a fresh or established \(gsk\):
      - If \(bsn \neq \bot\), retrieve \(gsk\) from \(\langle M_i, bsn, gsk \rangle \in \text{DomainKeys}\) for \(\langle M_i, bsn \rangle\). If no such \(gsk\) exists or \(bsn = \bot\), generate \(gsk \leftarrow \text{ukgen}()\) and store \(\langle M_i, bsn, gsk \rangle\) in \(\text{DomainKeys}\).
      - Compute a signature as \(\sigma \leftarrow \text{sig}(gsk, m, bsn, \hat{p})\).
    - If \(M_i\) is honest, store \(\langle m, bsn, \sigma, M_i, \hat{p} \rangle\) in \(\text{Signed}\).
    - Output \((\text{SIGNATURE}, \text{sid}, \text{ssid}, \sigma)\) to \(H_j\).

Verify
11. Verify. On input \((\text{VERIFY}, \text{sid}, m, bsn, \sigma, \hat{p}, \text{RL})\) from some party \(V\).
    - Set \(f \leftarrow 0\) if at least one of the following conditions hold:
      - There is a \(gsk' \in \text{RL}\) such that \(\text{identify}(m, bsn, \sigma, gsk') = 1\).
    - If \(f \neq 0\), set \(f \leftarrow \text{ver}(m, bsn, \sigma, \hat{p})\).
    - Add \(\langle m, bsn, \sigma, \text{RL}, f \rangle\) to \(\text{VerResults}\) and output \((\text{VERIFIED}, \text{sid}, f)\) to \(V\).

Link
12. Link. On input \((\text{LINK}, \text{sid}, m_0, \sigma_0, \hat{p}_0, m_1, \sigma_1, \hat{p}_1, bsn)\) from some party \(V\) with \(bsn \neq \bot\).
    - Output \(\bot\) to \(V\) if at least one signature tuple \((m_0, bsn, \sigma_0, \hat{p}_0)\) or \((m_1, bsn, \sigma_1, \hat{p}_1)\) is not valid, which is verified via the VERIFY interface with \(\text{RL} = \emptyset\).
    - Set \(f \leftarrow \text{link}(m_0, \sigma_0, m_1, \sigma_1, bsn)\).
    - Output \((\text{LINK}, \text{sid}, f)\) to \(V\).

Fig. 16: Functionality \(\mathcal{F}\) for Game 8
Setup
– Unchanged.

Join
– Unchanged.

Sign

Honest $M, H$
– Upon receiving $(\text{SIGNSTART}, \text{sid}, \text{ssid}, m, \text{bsn}, \hat{p}, M_i, H_j)$ from $F$.
  • $S$ starts the simulation via giving \("H_j\" input $(\text{SIGN}, \text{sid}, \text{ssid}, M_i, m, \text{bsn}, \hat{p})$.
  • When \("M_i\" outputs $(\text{SIGNPROCEED}, \text{sid}, \text{ssid}, m, \text{bsn}, \hat{p})$, $S$ sends $(\text{SIGNSTART}, \text{sid}, \text{ssid})$ to $F$.
– Upon receiving $(\text{SIGNCOMPLETE}, \text{sid}, \text{ssid})$ from $F$.
  • $S$ continues the simulation by giving \("M_i\" input $(\text{SIGNPROCEED}, \text{sid}, \text{ssid})$.
  • When \("H_j\" outputs $(\text{SIGNATURE}, \text{sid}, \text{ssid}, \sigma)$, $S$ sends $(\text{SIGNCOMPLETE}, \text{sid}, \text{ssid}, \bot)$ to $F$.

Corrupt $M$
– Upon receiving $(\text{SIGNSTART}, \text{sid}, \text{ssid}, m, \text{bsn}, \hat{p}, M_i, H_j)$ from $F$.
  • $S$ sends $(\text{SIGNSTART}, \text{sid}, \text{ssid})$ to $F$.
– Upon receiving $(\text{SIGNPROCEED}, \text{sid}, \text{ssid}, m, \text{bsn}, \hat{p})$ from $F$ as $M_i$ is corrupted.
  • $S$ starts the simulation by giving \("H_j\" input $(\text{SIGN}, \text{sid}, \text{ssid}, M_i, m, \text{bsn}, \hat{p})$.
  • When \("H_j\" outputs $(\text{SIGNATURE}, \text{sid}, \text{ssid}, \sigma)$, $S$ sends $(\text{SIGNPROCEED}, \text{sid}, \text{ssid})$ to $F$ on behalf of $M_i$.
– Upon receiving $(\text{SIGNCOMPLETE}, \text{sid}, \text{ssid})$ from $F$.
  • $S$ sends $(\text{SIGNCOMPLETE}, \text{sid}, \text{ssid}, \sigma)$ to $F$.

Honest $H$, Corrupt $M$
– $S$ notices this sign session as \("M_i\" receives $(\text{sid}, \text{ssid}, m, \text{bsn}, \hat{p})$ from $H_j$.
  • $S$ sends $(\text{SIGN}, \text{sid}, \text{ssid}, M_i, m, \text{bsn}, \hat{p})$ to $F$ on behalf of $H_j$.
– Upon receiving $(\text{SIGNSTART}, \text{sid}, \text{ssid}, m, \text{bsn}, \hat{p}, M_i, H_j)$ from $F$.
  • $S$ continues the simulation of \("M_i\" until it outputs $(\text{SIGNPROCEED}, \text{sid}, \text{ssid}, m, \text{bsn}, \hat{p})$.
  • $S$ sends $(\text{SIGNSTART}, \text{sid}, \text{ssid})$ to $F$.
– Upon receiving $(\text{SIGNCOMPLETE}, \text{sid}, \text{ssid})$ from $F$.
  • $S$ sends $(\text{SIGNCOMPLETE}, \text{sid}, \text{ssid}, \bot)$ to $F$.
– Upon receiving $(\text{SIGNATURE}, \text{sid}, \text{ssid}, \bot)$ from $F$ as $H_j$ is corrupted.
  • $S$ continues the simulation via giving \("M_i\" input $(\text{SIGNPROCEED}, \text{sid}, \text{ssid})$.

Verify
– Nothing to simulate.

Link
– Nothing to simulate.

Fig. 17: Simulator for Game 8
### Setup
Unchanged

### Join
Unchanged

### Sign
7. **Sign Request.** On input \((\text{SIGN}, \text{sid}, \text{ssid}, M, bsn, \mathcal{H}_j)\) with \(bsn = \perp\) and \(\mathcal{H}_j \in \mathcal{P}\) from host \(\mathcal{H}_j\).
   - If \(\mathcal{H}_j\) is honest and no entry \((M_i, \mathcal{H}_j, *, \text{attrs})\) with \(p(\text{attrs}) = 1\) exists in \(\text{Members}\), abort.
   - Create a sign session record \((\text{ssid}, M_i, \mathcal{H}_j, m, bsn, \mathcal{H}_j, \mathcal{P} , \mathcal{H}_j)\) with status \(\leftarrow \text{request}\).
   - Output \((\text{SIGNSTART}, \text{sid}, \text{ssid}, l(m, bsn, \mathcal{H}_j), M_i, \mathcal{H}_j)\) to \(\mathcal{S}\).
8. **Sign Request Delivery.** On input \((\text{SIGNSTART}, \text{sid}, \text{ssid})\) from \(\mathcal{S}\).
   - Update the session record \((\text{ssid}, M_i, \mathcal{H}_j, m, bsn, \mathcal{P} , \mathcal{H}_j)\) to status \(\leftarrow \text{delivered}\).
   - Output \((\text{SIGNPROCEED}, \text{sid}, \text{ssid}, m, bsn, \mathcal{H}_j)\) to \(\mathcal{M}_i\).
9. **Sign Proceed.** On input \((\text{SIGNPROCEED}, \text{sid}, \text{ssid})\) from \(\mathcal{M}_i\).
   - Look up record \((\text{ssid}, M_i, \mathcal{H}_j, m, bsn, \mathcal{P} , \mathcal{H}_j)\) with status \(\leftarrow \text{delivered}\).
   - Output \((\text{SIGNCOMPLETE}, \text{sid}, \text{ssid})\) to \(\mathcal{S}\).
10. **Signature Generation.** On input \((\text{SIGNCOMPLETE}, \text{sid}, \text{ssid}, \mathcal{P} , \mathcal{H}_j)\) from \(\mathcal{S}\).
    - If \(\mathcal{M}_i\) and \(\mathcal{H}_j\) are honest, ignore the signature \(\sigma\) from \(\mathcal{S}\) and internally generate a signature for a fresh or established \(gsk\):
      - If \(bsn \neq \perp\), retrieve \(gsk\) from \((M_i, bsn, gsk) \in \text{DomainKeys}\) for \((M_i, bsn)\). If no such \(gsk\) exists or \(bsn = \perp\), generate \(gsk \leftarrow \text{ukgen()}\) and store \((M_i, bsn, gsk)\) in \(\text{DomainKeys}\).
      - Compute a signature as \(\sigma \leftarrow \text{sig}(gsk, m, bsn, \mathcal{H}_j)\).
    - If \(\mathcal{M}_i\) is honest, store \((m, bsn, \sigma, M_i, \mathcal{H}_j)\) in \(\text{Signed}\).
    - Output \((\text{SIGNATURE}, \text{sid}, \text{ssid}, \mathcal{P} , \mathcal{H}_j)\) to \(\mathcal{H}_j\).

### Verify
11. **Verify.** On input \((\text{VERIFY}, \text{sid}, m, bsn, \sigma, \mathcal{H}_j, \mathcal{P} )\) from some party \(\mathcal{V}\).
    - Set \(f \leftarrow 0\) if at least one of the following conditions hold:
      - There is a \(gsk' \in \mathcal{P}\) such that \(\text{verify}(m, bsn, \sigma, gsk') = 1\).
    - If \(f \neq 0\), set \(f \leftarrow \text{ver}(m, bsn, \sigma, \mathcal{H}_j)\).
    - Add \((m, bsn, \sigma, \mathcal{P} , \mathcal{H}_j, f)\) to \(\text{VerResults}\) and output \((\text{VERIFIED}, \text{sid}, f)\) to \(\mathcal{V}\).

### Link
12. **Link.** On input \((\text{LINK}, \text{sid}, m_0, \sigma_0, \mathcal{H}_j, \mathcal{P} , \mathcal{H}_0, \mathcal{P} , \mathcal{H}_0)\) from some party \(\mathcal{V}\) with \(bsn \neq \perp\).
    - Output \(\perp\) to \(\mathcal{V}\) if at least one signature tuple \((m_0, bsn, \sigma_0, \mathcal{H}_0)\) or \((m_1, bsn, \sigma_1, \mathcal{H}_1)\) is not valid, which is verified via the \(\text{VERIFY}\) interface with \(\mathcal{P} = \emptyset\).
    - Set \(f \leftarrow \text{link}(m_0, \sigma_0, m_1, \sigma_1, bsn)\).
    - Output \((\text{LINK}, \text{sid}, f)\) to \(\mathcal{V}\).

---

Fig. 18: Functionality \(\mathcal{F}\) for Game 9
Setup
– Unchanged.

Join
– Unchanged.

Sign

Honest $M, \mathcal{H}$
– Upon receiving $(\text{SIGNSTART}, sid, ssid, l, M_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  • $S$ takes a dummy message $m'$, basename $bsn'$ and attribute predicate $\tilde{p}'$ such that $l(m', bsn', \tilde{p}') = l$ and $\tilde{p}'$ holds for the platform's attributes which are learned by $S$ from the join protocol.
  • $S$ starts the simulation via giving “$\mathcal{H}_j$,” input $(\text{SIGN}, sid, ssid, M_i, m', bsn', \tilde{p}')$.
  • When “$M_i$” outputs $(\text{SIGNPROCEED}, sid, ssid, m', bsn', \tilde{p}')$, $S$ sends $(\text{SIGNSTART}, sid, ssid)$ to $\mathcal{F}$.
– Upon receiving $(\text{SIGNCOMPLETE}, sid, ssid)$ from $\mathcal{F}$.
  • $S$ continues the simulation by giving “$M_i$,” input $(\text{SIGNPROCEED}, sid, ssid)$.
  • When “$\mathcal{H}_j$” outputs $(\text{SIGNATURE}, sid, ssid, \sigma)$, $S$ sends $(\text{SIGNCOMPLETE}, sid, ssid, \bot)$ to $\mathcal{F}$.

Honest $\mathcal{H}$, Corrupt $M$
– Upon receiving $(\text{SIGNSTART}, sid, ssid, l, M_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  • $S$ sends $(\text{SIGNSTART}, sid, ssid)$ to $\mathcal{F}$.
– Upon receiving $(\text{SIGNPROCEED}, sid, ssid, m, bsn, \hat{p})$ from $\mathcal{F}$ as $M_i$ is corrupted.
  • $S$ starts the simulation by giving “$\mathcal{H}_j$,” input $(\text{SIGN}, sid, ssid, M_i, m, bsn, \hat{p})$.
  • When “$\mathcal{H}_j$” outputs $(\text{SIGNATURE}, sid, ssid, \sigma)$, $S$ sends $(\text{SIGNPROCEED}, sid, ssid)$ to $\mathcal{F}$ on behalf of $M_i$.
– Upon receiving $(\text{SIGNCOMPLETE}, sid, ssid)$ from $\mathcal{F}$.
  • $S$ sends $(\text{SIGNCOMPLETE}, sid, ssid, \sigma)$ to $\mathcal{F}$.

Honest $M$, Corrupt $\mathcal{H}$
– $S$ notices this sign session as “$M_i$,” receives $(sid, ssid, m, bsn, \hat{p})$ from $\mathcal{H}_j$.
  • $S$ sends $(\text{SIGN}, sid, ssid, M_i, m, bsn, \hat{p})$ to $\mathcal{F}$ on behalf of $\mathcal{H}_j$.
– Upon receiving $(\text{SIGNSTART}, sid, ssid, l, M_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  • $S$ continues the simulation of “$M_i$” until it outputs $(\text{SIGNPROCEED}, sid, ssid, \bot)$.
  • $S$ sends $(\text{SIGNSTART}, sid, ssid)$ to $\mathcal{F}$.
– Upon receiving $(\text{SIGNCOMPLETE}, sid, ssid)$ from $\mathcal{F}$.
  • $S$ sends $(\text{SIGNCOMPLETE}, sid, ssid, \bot)$ to $\mathcal{F}$.
– Upon receiving $(\text{SIGNATURE}, sid, ssid, \bot)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupted.
  • $S$ continues the simulation via giving “$M_i$,” input $(\text{SIGNPROCEED}, sid, ssid)$.

Verify
– Nothing to simulate.

Link
– Nothing to simulate.

Fig. 19: Simulator for Game 9
Setup

Unchanged

Join

3. Join Request. On input (JOIN, sid, jsid, M_i) from host H_j.
   - Create a join session record (jsid, M_i, H_j, status) with status ← request.
   - Output (JOINSTART, sid, jsid, M_i, H_j) to S.

4. Join Request Delivery. On input (JOINSTART, sid, jsid) from S.
   - Update the session record (jsid, M_i, H_j, ∅, status) to status ← delivered.
   - Abort if I or M_i is honest and a record (M_i, *, *, *) ∈ Members already exists.
   - Output (JOINCOMPLETE, sid, jsid, M_i) to I.

5. Join Proceed. On input (JOINCOMPLETE, sid, jsid, attrs) from I with attrs ∈ A_1 × · · · × A_n.
   - Update the session record (jsid, M_i, H_j, ∅, status) to ∅ ← attrs and status ← complete.
   - Output (JOINCOMPLETE, sid, jsid, attrs) to S, where attrs ← ∅ if M_i and H_j are honest
     and attrs ← attrs otherwise.

6. Platform Key Generation. On input (JOINCOMPLETE, sid, jsid, gsk) from S.
   - Look up record (jsid, M_i, H_j, attrs, status) with status = complete.
   - If M_i and H_j are honest, set gsk ← ∅.
   - Add (M_i, H_j, gsk, attrs) into Members and output (JOINED, sid, jsid) to H_j.

Sign

7. Sign Request. On input (SIGN, sid, ssid, M_i, m, bsn, p) with bsn = ∅ and p ∈ P from host H_j.
   - If H_j is honest and no entry (M_i, H_j, *, attrs) with p(attrs) = 1 exists in Members, abort.
   - Create a sign session record (ssid, M_i, H_j, m, bsn, p, status) with status ← request.
   - Output (SIGNSTART, sid, ssid, l(m, bsn, p), M_i, H_j) to S.

8. Sign Request Delivery. On input (SIGNSTART, sid, ssid) from S.
   - Update the session record (ssid, M_i, H_j, m, bsn, p, status) to status ← delivered.
   - Output (SIGNCOMPLETE, sid, ssid, m, bsn, p) to M_i.

9. Sign Proceed. On input (SIGNCOMPLETE, sid, ssid) from M_i.
   - Look up record (ssid, M_i, H_j, m, bsn, p, status) with status = delivered.
   - Output (SIGNCOMPLETE, sid, ssid) to S.

10. Signature Generation. On input (SIGNCOMPLETE, sid, ssid, σ) from S.
    - If I is honest, check that (M_i, H_j, *, attrs) with p(attrs) = 1 exists in Members.
    - If M_i and H_j are honest, ignore σ from S and internally generate a signature for a fresh or established gsk:
      - If bsn ̸= ∅, retrieve gsk from (M_i, bsn, gsk) ∈ DomainKeys for (M_i, bsn). If no such gsk exists or bsn = ∅,
        generate gsk ← ukgen() and store (M_i, bsn, gsk) in DomainKeys.
    - Compute a signature as σ ← sig(gsk, m, bsn, p).
    - If M_i is honest, store (m, bsn, σ, M_i, p) in Signed.
    - Output (SIGNATURE, sid, ssid, σ) to H_j.

Verify

11. Verify. On input (VERIFY, sid, m, bsn, σ, p, RL) from some party V.
    - Set f ← 0 if at least one of the following conditions hold:
      - There is a gsk′ ∈ RL such that identify(m, bsn, σ, gsk′) = 1.
    - If f ̸= 0, set f ← ver(m, bsn, σ, p).
    - Add (m, bsn, σ, RL, f) to VerResults and output (VERIFIED, sid, f) to V.

Link

12. Link. On input (LINK, sid, m_0, σ_0, p_0, m_1, σ_1, p_1, bsn) from some party V with bsn ̸= ∅.
    - Output ⊥ to V if at least one signature tuple (m_0, bsn, σ_0, p_0) or (m_1, bsn, σ_1, p_1) is not valid, which is verified via
      the VERIFY interface with RL = ∅.
    - Set f ← link(m_0, σ_0, m_1, σ_1, bsn).
    - Output (LINK, sid, f) to V.

Fig. 20: Functionality F for Game 10
Setup
– Unchanged.

Join

\textit{Honest M, H, I}
– Upon receiving (\texttt{JOINSTART, sid, jsid, M_i, H_j}) from \mathcal{F}, \mathcal{S} does the following:
  \begin{itemize}
  \item \mathcal{S} simulates the real-world join protocol via giving “H_j” input (\texttt{JOIN, sid, jsid, M_i}).
  \item When “I” outputs (\texttt{JOINPROCEED, sid, jsid, M_i}), \mathcal{S} sends (\texttt{JOINSTART, sid, jsid}) to \mathcal{F}.
  \end{itemize}
– Upon receiving (\texttt{JOINCOMPLETE, sid, jsid, attrs}) from \mathcal{F}.
  \begin{itemize}
  \item \mathcal{S} does not know the attributes, as it receives attrs = \bot. Thus, \mathcal{S} picks a random \texttt{attrs' \in A_1 \times \ldots \times A_n}.
  \item \mathcal{S} continues the simulation by giving “I” input (\texttt{JOINPROCEED, sid, jsid, attrs'}).
  \item When “H_j” outputs (\texttt{JOINED, sid, jsid}), \mathcal{S} outputs (\texttt{JOINCOMPLETE, sid, jsid, gsk}) to \mathcal{F}.
  \end{itemize}

\textit{Other Cases}
– Unchanged.

Sign

\textit{Honest M, H}
– Upon receiving (\texttt{SIGNSTART, sid, ssid, l, M_i, H_j}) from \mathcal{F}.
  \begin{itemize}
  \item \mathcal{S} takes a dummy message \texttt{m', basename bsn' and attribute predicate \textbf{\texttt{\hat{p}'}} such that } l(m', bsn', \textbf{\texttt{\hat{p}'}}) = l \text{ and } \textbf{\texttt{\hat{p}'}} \text{ holds for the dummy attributes that are chosen at random by } \mathcal{S} \text{ in the join protocol}.
  \item \mathcal{S} starts the simulation via giving “H_j” input (\texttt{SIGN, sid, ssid, M_i, m', bsn', \hat{p}'}).
  \item When “M_i” outputs (\texttt{SIGNPROCEED, sid, ssid, m', bsn', \hat{p}'}), \mathcal{S} sends (\texttt{SIGNSTART, sid, ssid}) to \mathcal{F}.
  \end{itemize}
– Upon receiving (\texttt{SIGNCOMPLETE, sid, ssid}) from \mathcal{F}.
  \begin{itemize}
  \item \mathcal{S} continues the simulation by giving “M_i” input (\texttt{SIGNPROCEED, sid, ssid}).
  \item When “H_j” outputs (\texttt{SIGNATURE, sid, ssid, \sigma}), \mathcal{S} sends (\texttt{SIGNCOMPLETE, sid, ssid, \bot}) to \mathcal{F}.
  \end{itemize}

\textit{Other Cases}
– Unchanged.

Verify
– Nothing to simulate.

Link
– Nothing to simulate.

Fig. 21: Simulator for Game 10
Setup
Unchanged

Join
3. Join Request. On input (JOIN, sid, jsid, Mi) from host Hj.
   - Create a join session record ⟨jsid, Mi, Hj, status⟩ with status ← request.
   - Output (JOINSTART, sid, jsid, Mi, Hj) to S.
4. Join Request Delivery. On input (JOINSTART, sid, jsid) from S.
   - Update the session record ⟨jsid, Mi, Hj, ⊥, status⟩ to status ← delivered.
   - Abort if I or Mi is honest and a record ⟨Mi, *, *, *⟩ ∈ Members already exists.
   - Output (JOINPROCEED, sid, jsid, Mi) to I.
5. Join Proceed. On input (JOINPROCEED, sid, jsid, attrs) from I with attrs ∈ A1 × · · · × An.
   - Update the session record ⟨jsid, Mi, Hj, ⊥, status⟩ to ⊥ ← attrs and status ← complete.
   - Output (JOINCOMPLETE, sid, jsid, attrs) to S, where attrs′ ← ⊥ if Mi and Hj are honest and attrs′ ← attrs otherwise.
6. Platform Key Generation. On input (JOINCOMPLETE, sid, jsid, gsk) from S.
   - Look up record ⟨jsid, Mi, Hj, attrs, status⟩ with status = complete.
   - If Mi and Hj are honest, set gsk ← ⊥.
   - Else verify that the provided gsk is eligible via checking
     - CheckGskHonest(gsk) = 1 if Mi is honest and Hj is corrupted.
     - CheckGskCorrupt(gsk) = 1 if Mi is corrupted.
   - Add ⟨Mi, Hj, gsk, attrs⟩ into Members and output (JOINED, sid, jsid) to Hj.

Sign
7. Sign Request. On input (SIGN, sid, ssid, Mi, m, bsn, ̂p) with bsn = ⊥ and ̂p ∈ P from host Hj.
   - If Hj is honest and no entry ⟨Mi, Hj, *, attrs⟩ with ̂p(attrs) = 1 exists in Members, abort.
   - Create a sign session record ⟨ssid, Mi, Hj, m, bsn, ̂p, status⟩ with status ← request.
   - Output (SIGNSTART, sid, ssid, l(m, bsn, ̂p), Mi, Hj) to S.
8. Sign Request Delivery. On input (SIGNSTART, sid, ssid) from S.
   - Update the session record ⟨ssid, Mi, Hj, m, bsn, ̂p, status⟩ to status ← delivered.
   - Output (SIGNPROCEED, sid, ssid, m, bsn, ̂p) to Mi.
9. Sign Proceed. On input (SIGNPROCEED, sid, ssid) from Mi.
   - Look up record ⟨ssid, Mi, Hj, m, bsn, ̂p, status⟩ with status = delivered.
   - Output (SIGNCOMPLETE, sid, ssid) to S.
10. Signature Generation. On input (SIGNCOMPLETE, sid, ssid, σ) from S.
    - If I is honest, check that ⟨Mi, Hj, *, attrs⟩ with ̂p(attrs) = 1 exists in Members.
    - If Mi and Hj are honest, ignore σ from S and internally generate a signature for a fresh or established gsk:
      - If bsn = ⊥, retrieve gsk from ⟨Mi, bsn, gsk⟩ ∈ DomainKeys for ⟨Mi, bsn⟩. If no such gsk exists or bsn = ⊥, generate gsk ← ukgen(i). Check that CheckGskHonest(gsk) = 1 and store ⟨Mi, bsn, gsk⟩ in DomainKeys.
      - Compute a signature as σ ← sig(gsk, m, bsn, ̂p).
    - If Mi is honest, store ⟨m, bsn, σ, Mi, ̂p⟩ in Signed.
    - Output (SIGNATURE, sid, ssid, σ) to Hj.

Verify
Unchanged

Link
Unchanged

Fig. 22: Functionality ℋ for Game 11
Setup
– Unchanged.

Join
– Unchanged.

Sign
– Unchanged.

Verify
– Unchanged.

Link
– Unchanged.

Fig. 23: Simulators for Games 11-17

7. Sign Request. On input (SIGN, sid, ssid, Mi, m, bsn, ˆp) with bsn = ⊥ and ˆp ∈ P from host Hj.
   – If Hj is honest and no entry (Mi, Hj, *, attrs) with ˆp(attrs) = 1 exists in Members, abort.
   – Create a sign session record (ssid, Mi, Hj, m, bsn, ˆp, status) with status ← request.
   – Output (SIGNSTART, ssid, Mi, h(m, bsn, ˆp), Mj, Hj) to S.

8. Sign Request Delivery. On input (SIGNSTART, sid, ssid) from S.
   – Update the session record (ssid, Mi, Hj, m, bsn, ˆp, status) to status ← delivered.
   – Output (SIGNPROCEED, sid, ssid, m, bsn, ˆp) to Mi.

9. Sign Proceed. On input (SIGNPROCEED, sid, ssid) from Mi.
   – Look up record (ssid, Mi, Hj, m, bsn, ˆp, status) with status = delivered.
   – Output (SIGNCOMPLETE, sid, ssid) to S.

10. Signature Generation. On input (SIGNCOMPLETE, sid, ssid, σ) from S.
    – If I is honest, check that (Mi, Hj, *, attrs) with ˆp(attrs) = 1 exists in Members.
    – If Mi and Hj are honest, ignore the signature σ from S and internally generate a signature for a fresh or established gsk:
      • If bsn = ⊥, retrieve gsk from (Mi, bsn, gsk) ∈ DomainKeys for (Mi, bsn). If no such gsk exists or bsn = ⊥,
        generate gsk ← ukgen(). Check that CheckGskHonest(gsk) = 1 and store (Mi, bsn, gsk) in DomainKeys.
      • Compute a signature as σ ← sig(gsk, m, bsn, ˆp) and check ver(m, bsn, σ, ˆp) = 1.
      • Check that identify(m, bsn, σ, gsk) = 1 and check that there is no Mi’ ≠ Mi with key gsk’ registered in
        Members or DomainKeys with identify(m, bsn, σ, gsk’) = 1.
    – If Mi is honest, store (m, bsn, σ, Mi, ˆp) in Signed.
    – Output (SIGNATURE, sid, ssid, σ) to Hj.

Verify
– Unchanged

Link
– Unchanged

Fig. 24: Functionality $\mathcal{F}$ for Game 12
Setup
Unchanged

Join
Unchanged

Sign
7. Sign Request. On input (SIGN, sid, ssid, M_i, m, bsn, \(\hat{p}\)) with bsn = \(\perp\) and \(\hat{p}\) \(\in \mathbb{P}\) from host \(H_j\).
   - If \(H_j\) is honest and no entry \((M_i, H_j, *, attrs)\) with \(\hat{p}(attrs) = 1\) exists in Members, abort.
   - Create a sign session record \((ssid, M_i, H_j, m, bsn, \hat{p}, status)\) with \(status \leftarrow request\).
   - Output \((\text{SIGNSTART}, sid, ssid, l(m, bsn, \hat{p}), M_i, H_j)\) to \(S\).
8. Sign Request Delivery. On input (SIGNSTART, sid, ssid) from \(S\).
   - Update the session record \((ssid, M_i, H_j, m, bsn, \hat{p}, status)\) to \(status \leftarrow delivered\).
   - Output (SIGNPROCEED, sid, ssid, m, bsn, \(\hat{p}\)) to \(M_i\).
9. Sign Proceed. On input (SIGNPROCEED, sid, ssid) from \(M_i\).
   - Look up record \((ssid, M_i, H_j, m, bsn, \hat{p}, status)\) with \(status = delivered\).
   - Output (SIGNCOMPLETE, sid, ssid) to \(S\).
10. Signature Generation. On input (SIGNCOMPLETE, sid, ssid, \(\sigma\)) from \(S\).
    - If \(I\) is honest, check that \((M_i, H_j, *, attrs)\) with \(\hat{p}(attrs) = 1\) exists in Members.
    - If \(M_i\) and \(H_j\) are honest, ignore \(\sigma\) from \(S\) and internally generate a signature for a fresh or established gsk:
      - If bsn \(\neq \perp\), retrieve gsk from \((M_i, bsn, gsk) \in \text{DomainKeys}\) for \((M_i, bsn)\). If no such gsk exists or bsn = \(\perp\), generate gsk \(\leftarrow \text{ukgen()}\). Check that CheckGskHonest(gsk) = 1 and store \((M_i, bsn, gsk)\) in DomainKeys.
      - Compute a signature as \(\sigma \leftarrow \text{sig}(gsk, m, bsn, \hat{p})\) and check \(\text{ver}(m, bsn, \sigma, \hat{p}) = 1\).
      - Check that \(\text{identify}(m, bsn, \sigma, gsk) = 1\) and check that there is no \(M_i' \neq M_i\) with key gsk' registered in Members or DomainKeys with identify(m, bsn, \(\sigma\), gsk') = 1.
    - If \(M_i\) is honest, store \((m, bsn, \sigma, M_i, \hat{p})\) in Signed.
    - Output (SIGNATURE, sid, ssid, \(\sigma\)) to \(H_j\).

Verify
11. Verify. On input (VERIFY, sid, m, bsn, \(\sigma\), \(\hat{p}\), RL) from some party \(V\).
    - Retrieve all pairs \((M_i, gsk)\) from \((M_i, *, gsk, *) \in \text{Members}\) and \((M_i, *, gsk_i) \in \text{DomainKeys}\) such that
      - \(\text{identify}(m, bsn, \sigma, gsk) = 1\) if at least one of the following conditions hold:
        - More than one key gsk_i was found.
        - There is a gsk_i' \(\in RL\) such that \(\text{identify}(m, bsn, \sigma, gsk_i') = 1\).
      - Set \(f \leftarrow \text{ver}(m, bsn, \sigma, \hat{p})\).
    - Add \((m, bsn, \sigma, RL, f)\) to VerResults and output (VERIFIED, sid, f) to \(V\).

Link
12. Link. On input (LINK, sid, m_0, \(\sigma_0\), \(\hat{p}_0\), m_1, \(\sigma_1\), \(\hat{p}_1\), bsn) from some party \(V\) with bsn \(\neq \perp\).
    - Output \(\perp\) to \(V\) if at least one signature tuple \((m_0, bsn, \sigma_0, \hat{p}_0)\) or \((m_1, bsn, \sigma_1, \hat{p}_1)\) is not valid, which is verified via the VERIFY interface with RL = \(\emptyset\).
    - Set \(f \leftarrow \text{link}(m_0, \sigma_0, m_1, \sigma_1, bsn)\).
    - Output (LINK, sid, f) to \(V\).

Fig. 25: Functionality \(\mathcal{F}\) for Game 13
Setup
Unchanged

Join
Unchanged

Sign
7. Sign Request. On input \((\text{SIGN}, \text{sid}, \text{ssid}, \mathcal{M}_i, m, \text{bsn}, \hat{p})\) with \(\text{bsn} = \bot\) and \(\hat{p} \in \mathbb{P}\) from host \(\mathcal{H}_j\).
   - If \(\mathcal{H}_j\) is honest and no entry \((\mathcal{M}_i, \mathcal{H}_j, *, \text{attrs})\) with \(\hat{p}(\text{attrs}) = 1\) exists in \text{Members}, abort.
   - Create a sign session record \((\text{ssid}, \mathcal{M}_i, \mathcal{H}_j, m, \text{bsn}, \hat{p}, \text{status})\) with \(\text{status} \leftarrow \text{request}\).
   - Output \((\text{SIGNSTART}, \text{sid}, \text{ssid}, l(m, \text{bsn}, \hat{p}), \mathcal{M}_i, \mathcal{H}_j)\) to \(\mathcal{S}\).
8. Sign Request Delivery. On input \((\text{SIGNSTART}, \text{sid}, \text{ssid})\) from \(\mathcal{S}\).
   - Update the session record \((\text{ssid}, \mathcal{M}_i, \mathcal{H}_j, m, \text{bsn}, \hat{p}, \text{status})\) to \(\text{status} \leftarrow \text{delivered}\).
   - Output \((\text{SIGNPROCEED}, \text{sid}, \text{ ssid}, m, \text{bsn}, \hat{p})\) to \(\mathcal{M}_i\).
9. Sign Proceed. On input \((\text{SIGNPROCEED}, \text{sid}, \text{ssid})\) from \(\mathcal{M}_i\).
   - Look up record \((\text{ssid}, \mathcal{M}_i, \mathcal{H}_j, m, \text{bsn}, \hat{p}, \text{status})\) with \(\text{status} = \text{ delivered}\).
   - Output \((\text{SIGNCOMPLETE}, \text{sid}, \text{ ssid})\) to \(\mathcal{S}\).
10. Signature Generation. On input \((\text{SIGNCOMPLETE}, \text{sid}, \text{ssid}, \sigma)\) from \(\mathcal{S}\).
    - If \(\mathcal{I}\) is honest, check that \((\mathcal{M}_i, \mathcal{H}_j, *, \text{attrs})\) with \(\hat{p}(\text{attrs}) = 1\) exists in \text{Members}.
    - If \(\mathcal{M}_i\) and \(\mathcal{H}_j\) are honest, ignore \(\sigma\) from \(\mathcal{S}\) and internally generate a signature for a fresh or established \(\text{gsk}\):
      - If \(\text{bsn} \neq \bot\), retrieve \text{gsk} from \((\mathcal{M}_i, \text{bsn}, \text{gsk}) \in \text{DomainKeys}\) for \((\mathcal{M}_i, \text{bsn})\). If no such \text{gsk} exists or \(\text{bsn} = \bot\), generate \text{gsk} \leftarrow \text{ukgen}(). Check that \text{CheckGskHonest}() = 1 and store \((\mathcal{M}_i, \text{bsn}, \text{gsk})\) in DomainKeys.
      - Compute a signature as \(\sigma \leftarrow \text{sig}(\text{gsk}, m, \text{bsn}, \hat{p})\) and check \(\text{ver}(m, \text{bsn}, \sigma, \hat{p}) = 1\).
      - Check that \text{identify}(m, \text{bsn}, \sigma, \text{gsk}) = 1 and check that there is no \(\mathcal{M}_i' \neq \mathcal{M}_i\) with key \text{gsk}' registered in \text{Members} or \text{DomainKeys} with \text{identify}(m, \text{bsn}, \sigma, \text{gsk}') = 1.
    - If \(\mathcal{M}_i\) is honest, store \((m, \text{bsn}, \sigma, \mathcal{M}_i, \hat{p})\) in Signed.
    - Output \((\text{SIGNATURE}, \text{sid}, \text{ssid}, \sigma)\) to \(\mathcal{H}_j\).

Verify
11. Verify. On input \((\text{VERIFY}, \text{sid}, m, \text{bsn}, \sigma, \hat{p}, \text{RL})\) from some party \(\mathcal{V}\).
    - Retrieve all pairs \((\mathcal{M}_i, \text{gsk}_i)\) from \((\mathcal{M}_i, *, \text{gsk}_i, *) \in \text{Members}\) and \((\mathcal{M}_i, *, \text{gsk}_i) \in \text{DomainKeys}\) such that \text{identify}(m, \text{bsn}, \sigma, \text{gsk}_i) = 1. Set \(f \leftarrow 0\) if at least one of the following conditions hold:
      - More than one key \text{gsk}_i was found.
      - \(\mathcal{I}\) is honest and no pair \((\mathcal{M}_i, \text{gsk}_i)\) was found for which an entry \((\mathcal{M}_i, *, *, \text{attrs}) \in \text{Members}\) with \(\hat{p}(\text{attrs}) = 1\) exists.
      - There is a \text{gsk}' \in \text{RL} such that \text{identify}(m, \text{bsn}, \sigma, \text{gsk}') = 1.
    - If \(f \neq 0\), set \(f \leftarrow \text{ver}(m, \text{bsn}, \sigma, \hat{p})\).
    - Add \((m, \text{bsn}, \sigma, \text{RL}, f)\) to VerResults and output \((\text{VERIFIED}, \text{sid}, f)\) to \(\mathcal{V}\).

Link
12. Link. On input \((\text{LINK}, \text{sid}, m_0, \sigma_0, \hat{p}_0, m_1, \sigma_1, \hat{p}_1, \text{bsn})\) from some party \(\mathcal{V}\) with \(\text{bsn} \neq \bot\).
    - Output \(\bot\) to \(\mathcal{V}\) if at least one signature tuple \((m_0, \text{bsn}, \sigma_0, \hat{p}_0)\) or \((m_1, \text{bsn}, \sigma_1, \hat{p}_1)\) is not valid, which is verified via the VERIFY interface with \(\text{RL} = \emptyset\).
    - Set \(f \leftarrow \text{link}(m_0, \sigma_0, m_1, \sigma_1, \text{bsn})\).
    - Output \((\text{LINK}, \text{sid}, f)\) to \(\mathcal{V}\).

Fig. 26: Functionality \(\mathcal{F}\) for Game 14
Setup
Unchanged

Join
Unchanged

Sign
7. Sign Request. On input (SIGN, sid, ssid, M_i, m, bsn, p) with bsn = ∅ and p ∈ P from host H_j.
   - If H_j is honest and no entry (M_i, H_j, *, attrs) with p(attrs) = 1 exists in Members, abort.
   - Create a sign session record (ssid, M_i, H_j, m, bsn, p, status) with status ← request.
   - Output (SIGNSTART, sid, ssid, l(m, bsn, p), M_i, H_j) to S.

8. Sign Request Delivery. On input (SIGNSTART, sid, ssid) from S.
   - Update the session record (ssid, M_i, H_j, m, bsn, p, status) to status ← delivered.
   - Output (SIGNPROCEED, sid, ssid, m, bsn, p) to M_i.

9. Sign Proceed. On input (SIGNPROCEED, sid, ssid) from M_i.
   - Look up record (ssid, M_i, H_j, m, bsn, p, status) with status = delivered.
   - Output (SIGNCOMPLETE, sid, ssid) to S.

10. Signature Generation. On input (SIGNCOMPLETE, sid, ssid, σ) from S.
    - If I is honest, check that (M_i, H_j, *, attrs) with p(attrs) = 1 exists in Members.
    - If M_i and H_j are honest, ignore σ from S and internally generate a signature for a fresh or established gsk:
      * If bsn ̸= ∅, retrieve gsk from (M_i, bsn, gsk) ∈ DomainKeys for (M_i, bsn). If no such gsk exists or bsn = ∅, generate gsk ← ukgen(). Check that CheckGskHonest(gsk) = 1 and store (M_i, bsn, gsk) in DomainKeys.
      * Compute a signature as σ ← sig(gsk, m, bsn, p) and check ver(m, bsn, σ, p) = 1.
      * Check that identify(m, bsn, σ, gsk) = 1 and check that there is no M_j ≠ M_i with key gsk' registered in Members or DomainKeys with identify(m, bsn, σ, gsk') = 1.
    - If M_i is honest, store (m, bsn, σ, M_i, p) in Signed.
    - Output (SIGNATURE, sid, ssid, σ) to H_j.

Verify
11. Verify. On input (VERIFY, sid, m, bsn, σ, p, RL) from some party V.
    - Retrieve all pairs (M_i, gsk_i) from (M_i, *, gsk_i) ∈ Members and (M_i, *, gsk_i) ∈ DomainKeys such that identify(m, bsn, σ, gsk) = 1. Set f ← 0 if at least one of the following conditions hold:
      * More than one key gsk was found.
      * I is honest and no pair (M_i, gsk_i) was found for which an entry (M_i, *, attrs) ∈ Members with p(attrs) = 1 exists.
    - If there is an honest M_i, but no entry (m, bsn, *, M_i, p) ∈ Signed exists.
    - If there is a gsk' ∈ RL such that identify(m, bsn, σ, gsk') = 1.
      * Set f ← 0, set f ← ver(m, bsn, σ, p).
      * Add (m, bsn, σ, RL, f) to VerResults and output (VERIFIED, sid, f) to V.

Link
12. Link. On input (LINK, sid, m_o, σ_o, p_o, m_1, σ_1, p_1, bsn) from some party V with bsn ̸= ∅.
    - Output ⊥ to V if at least one signature tuple (m_0, bsn, σ_0, p_0) or (m_1, bsn, σ_1, p_1) is not valid, which is verified via the VERIFY interface with RL = ∅.
    - Set f ← link(m_o, σ_o, m_1, σ_1, bsn).
    - Output (LINK, sid, f) to V.

Fig. 27: Functionality $F$ for Game 15
### Setup
Unchanged

### Join
Unchanged

### Sign

7. **Sign Request.** On input $(\text{SIGN}, \text{sid}, \text{ssid}, M_i, m, \text{bsn}, \hat{p})$ with $\text{bsn} = \bot$ and $\hat{p} \in \mathcal{P}$ from host $H_j$.
   - If $H_j$ is honest and no entry $(M_i, H_j, *, \text{attrs})$ with $\hat{p}(\text{attrs}) = 1$ exists in Members, abort.
   - Create a sign session record $(\text{ssid}, M_i, H_j, m, \text{bsn}, \hat{p}, \text{status})$ with $\text{status} \leftarrow \text{request}$.
   - Output $(\text{SIGNSTART}, \text{sid}, \text{ssid}, l(m, \text{bsn}, \hat{p}), M_i, H_j)$ to $S$.

8. **Sign Request Delivery.** On input $(\text{SIGNSTART}, \text{sid}, \text{ssid})$ from $S$.
   - Update the session record $(\text{ssid}, M_i, H_j, m, \text{bsn}, \hat{p}, \text{status})$ to $\text{status} \leftarrow \text{delivered}$.
   - Output $(\text{SIGNPROCEED}, \text{sid}, \text{ssid}, m, \text{bsn}, \hat{p})$ to $M_i$.

9. **Sign Proceed.** On input $(\text{SIGNPROCEED}, \text{sid}, \text{ssid})$ from $M_i$.
   - Look up record $(\text{ssid}, M_i, H_j, m, \text{bsn}, \hat{p}, \text{status})$ with $\text{status} = \text{delivered}$.
   - Output $(\text{SIGNCOMPLETE}, \text{sid}, \text{ssid})$ to $S$.

10. **Signature Generation.** On input $(\text{SIGNCOMPLETE}, \text{sid}, \text{ssid}, \sigma)$ from $S$.
    - If $I$ is honest, check that $(M_i, H_j, *, \text{attrs})$ with $\hat{p}(\text{attrs}) = 1$ exists in Members.
    - If $M_i$ and $H_j$ are honest, ignore $\sigma$ from $S$ and internally generate a signature for a fresh or established gsk:
      - If $\text{bsn} = \bot$, retrieve gsk from $(M_i, \text{bsn}, \text{gsk}) \in \text{DomainKeys}$ for $(M_i, \text{bsn})$. If no such gsk exists or $\text{bsn} = \bot$, generate $\text{gsk} \leftarrow \text{ukgen}()$. Check that $\text{CheckGskHonest}(\text{gsk}) = 1$ and store $(M_i, \text{bsn}, \text{gsk})$ in DomainKeys.
      - Compute a signature as $\sigma \leftarrow \text{sig}(\text{gsk}, m, \text{bsn}, \hat{p})$ and check $\text{ver}(m, \text{bsn}, \sigma, \hat{p}) = 1$.
      - Check that $\text{identify}(m, \text{bsn}, \sigma, \text{gsk}) = 1$ and check that there is no $M'_i \neq M_i$ with key $\text{gsk}'$ registered in Members or DomainKeys with $\text{identify}(m, \text{bsn}, \sigma, \text{gsk}') = 1$.
    - If $M_i$ is honest, store $(m, \text{bsn}, \sigma, M_i, \hat{p})$ in Signed.
    - Output $(\text{SIGNATURE}, \text{sid}, \text{ssid}, \sigma)$ to $H_j$.

### Verify

11. **Verify.** On input $(\text{VERIFY}, \text{sid}, m, \text{bsn}, \sigma, \hat{p}, \text{RL})$ from some party $V$.
    - Retrieve all pairs $(M_i, \text{gsk})$ from $(M_i, *, \text{gsk}, *, *) \in \text{Members}$ and $(M_i, *, *, \text{gsk}) \in \text{DomainKeys}$ such that $\text{identify}(m, \text{bsn}, \sigma, \text{gsk}) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
      - More than one key $\text{gsk}_i$ was found.
      - $I$ is honest and no pair $(M_i, \text{gsk}_i)$ was found for which an entry $(M_i, *, *, \text{attrs}) \in \text{Members}$ with $\hat{p}(\text{attrs}) = 1$ exists.
      - There is an honest $M_i$ but no entry $(m, \text{bsn}, *, M_i, \hat{p}) \in \text{Signed}$ exists.
      - There is a $\text{gsk}' \in \text{RL}$ such that $\text{identify}(m, \text{bsn}, \sigma, \text{gsk}') = 1$ and no pair $(M_i, \text{gsk}_i)$ for an honest $M_i$ was found.
    - If $f \neq 0$, set $f \leftarrow \text{ver}(m, \text{bsn}, \sigma, \hat{p})$.
    - Add $(m, \text{bsn}, \sigma, \text{RL})$ to VerResults and output $(\text{VERIFIED}, \text{sid}, f)$ to $V$.

### Link

12. **Link.** On input $(\text{LINK}, \text{sid}, m_0, \sigma_0, \hat{p}_0, m_1, \sigma_1, \hat{p}_1, \text{bsn})$ from some party $V$ with $\text{bsn} \neq \bot$.
    - Output $\bot$ to $V$ if at least one signature tuple $(m_0, \text{bsn}, \sigma_0, \hat{p}_0)$ or $(m_1, \text{bsn}, \sigma_1, \hat{p}_1)$ is not valid, which is verified via the VERIFY interface with $\text{RL} = \emptyset$.
    - Set $f \leftarrow \text{link}(m_0, \sigma_0, m_1, \sigma_1, \text{bsn})$.
    - Output $(\text{LINK}, \text{sid}, f)$ to $V$.

---

Fig. 28: Functionality $\mathcal{F}$ for Game 16
Fig. 29: Functionality $\mathcal{F}$ for Game 17