Universally Composable Direct Anonymous Attestation

Conference Paper · March 2016

DOI: 10.1007/978-3-662-49387-8_10

Jan Camenisch
Dfinity

Manu Drijvers
IBM

Anja Lehmann
IBM

Universally Composable Direct Anonymous Attestation*

Jan Camenisch[1], Manu Drijvers[1,2], and Anja Lehmann[1]

[1] IBM Research – Zurich, Säumerstrasse 4, 8803 Rüschlikon, Switzerland
{jca,mdr,anj}@zurich.ibm.com
[2] Department of Computer Science, ETH Zürich, 8092 Zürich, Switzerland

Abstract. Direct Anonymous Attestation (DAA) is one of the most complex cryptographic algorithms that has been deployed in practice. In spite of this and the long body of work on the subject, there is still no fully satisfactory security definition for DAA. This was already acknowledged by Bernard et al. (IJIC'13) who showed that in existing models even insecure protocols can be proven secure. Bernard et al. therefore proposed an extensive set of security games which, however, aim only at a simplified setting termed pre-DAA. In pre-DAA the host platform that runs a TPM is assumed to be trusted. Consequently, their notion does not guarantee any security if a TPM is embedded in a potentially corrupt host which is a significant restriction. In this paper, we give a comprehensive security definition for full DAA in the form of an ideal functionality in the Universal Composability model. Our definition considers the host and TPM to be separate entities that can be in different corruption states. None of the existing DAA schemes satisfy our strong security notion. We therefore propose a realization that is based on a DAA scheme supported by the TPM 2.0 standard and prove it secure in our model.

1 Introduction

Direct Anonymous Attestation (DAA) allows a small chip, the Trusted Platform Module (TPM), that is embedded in a host computer to create attestations about the state of the host system. Such attestations, which can be seen as signatures on the current state under the TPM's secret key, convince a remote verifier that the system it is communicating with is running on top of certified hardware and is using the correct software. A crucial feature of DAA is that it performs such attestations in a privacy-friendly manner. That is, the user of the host system can choose to create attestations anonymously ensuring that her transactions are unlinkable and do not leak any information about the particular TPM being used. User-controlled linkability is also allowed and is steered by a basename `bsn`: attestations under a fresh or empty basename can not be linked whereas repeated use of a basename makes the corresponding transactions linkable.

* An extended abstract of this work was published at PKC2016. This is the full version.

DAA is one of the most complex cryptographic protocols deployed in practice. The Trusted Computing Group (TCG), the industry standardization group that designed the TPM, standardized the first DAA protocol in the TPM 1.2 specification in 2004 [23] and included support for multiple DAA schemes in the TPM 2.0 specification in 2014 [24]. Over 500 million computers with TPM chips have been sold[3], making DAA one of the largest deployments of such a complex cryptographic scheme. This sparked a strong interest in the research community in the security and efficiency of DAA schemes [3, 5–7, 14–18].

Direct Anonymous Attestation has recently also gained the attention of the FIDO alliance which aims at basing online authentication on strong cryptography rather than passwords. The FIDO approach is to choose a fresh key pair for every user account, to provide the public key to the service provider, and to authenticate users via the corresponding secret key. Adding DAA to this approach allows one to prove that the secret key is properly stored on and protected by a trusted platform.

Existing Security Definitions. Interestingly, in spite of the large scale deployment and the long body of work on the subject, DAA still lacks a sound and comprehensive security definition. There exist a number of security definitions in the literature. Unfortunately all of them have rather severe shortcomings such as allowing for obviously broken schemes to be proven secure. This was recently discussed by Bernard et al. [3] who provide an analysis of existing security notions and also propose a new security definition. In a nutshell, the existing definitions that capture the desired security properties in form of an ideal functionality either miss to treat signatures as concrete objects that can be output or stored by the verifier [5] or are unrealizable [14, 17]. The difficulty in defining a proper ideal functionality for the complex DAA setting might not be all that surprising considering the numerous (failed) attempts in modeling the much simpler standard signature scheme in the universal composability framework [1, 13].

Another line of work therefore aimed at capturing the DAA requirements in the form of game-based security properties [3, 7, 15] as a more intuitive way of modeling. However, the first attempts [7, 15] have missed to cover some of the expected security properties and also have made unconventional choices when defining unforgeability (the latter resulting in schemes being considered secure that use a *constant* value as signatures).

Realizing that the previous definitions were not sufficient, Bernard et al. [3] provided an extensive set of property-based security games. The authors consider only a simplified setting which they call pre-DAA. The simplification is that the host and the TPM are considered as single entity (the platform), thus they are both either corrupt or honest. For properties such as anonymity and non-frameability this is sufficient as they protect against a corrupt issuer and assume both the TPM and the host to be honest. Unforgeability of a TPM attestation, however, should rely only on the TPM being honest but allow the host to be corrupt. This cannot be captured in their model. In fact, shifting the

[3] http://www.trustedcomputinggroup.org/solutions/authentication

load of the computational work to the host without affecting security in case the host is corrupted is one of the main challenges when designing a DAA scheme. Therefore, a DAA security definition should allow one to formally analyze the setting of an honest TPM and a corrupt host.

This is also acknowledged by Bernard et al. [3] who, after proposing a pre-DAA secure protocol, argue how to obtain a protocol achieving full DAA security. Unfortunately, due to the absence of a full DAA security model, this argumentation is done only informally. In this paper we show that their argumentation is actually somewhat flawed: the given proof for unforgeability of the given pre-DAA proof can not be lifted (under the same assumptions) to the full DAA setting. This highlights the fact that an "almost matching" security model together with an informal argument of how to achieve the actually desired security does not provide sound guarantees beyond what is formally proved.

Thus still no satisfying security model for DAA exists to date. This lack of a sound security definition is not only a theoretic problem but has resulted in insecure schemes being deployed in practice. A DAA scheme that allows anyone to forge attestations (as it does not exclude the "trivial" TPM credential $(1, 1, 1, 1)$) has even been standardized in ISO/IEC 20008-2 [18, 20].

Our Contributions. We tackle the challenge of formally defining Direct Anonymous Attestation and provide an ideal functionality for DAA in the Universal Composability (UC) framework [12]. Our functionality models hosts and TPMs as individual parties who can be in different corruption states and comprises all expected security properties such as unforgeability, anonymity, and non-frameability. The model also includes verifier-local revocation where a verifier, when checking the validity of a signature, can specify corrupted TPMs from which he no longer accepts signatures.

We choose to define a new model rather than addressing the weaknesses of one of the existing models. The latest DAA security model by Bernard et al. [3] seem to be the best starting point. However, as their model covers pre-DAA only, changing all their definitions to full DAA would require changes to almost every aspect of them. Furthermore, given the complexity of DAA, we believe that the simulation-based approach is more natural as one has a lower risk of overlooking security properties. A functionality provides a full description of security and no oracles have to be defined as the adversary simply gets full control over corrupt parties. Furthermore, the UC framework comes with strong composability guarantees that allow for protocols to be analyzed individually and preserve that security when being composed with other protocols.

None of the existing DAA constructions [3, 6, 7, 16, 18] satisfy our notion of security. Therefore, we also propose a modified version of recent DAA schemes [3, 18] that are built from pairing-based Camenisch-Lysyanskaya signatures [9] and zero-knowledge proofs. We then rigorously prove that our scheme realizes our new functionality. By the universal composition theorem, this proves that our scheme can be composed in arbitrary ways without losing security.

Organization. The rest of this paper is structured as follows. We start with a detailed discussion of existing DAA security definitions in Section 2, with a focus on the latest one by Bernard et al. [3]. Section 3 then presents our new definition in the form of an ideal functionality in the UC framework. Section 4 introduces the building blocks required for our DAA scheme, which is presented in Section 5. The latter section also contains a discussion why the existing DAA schemes could not be proven secure in our model. The proof that the new DAA scheme fulfills our definition of security is sketched in Section 6 (the complete proof is given in Appendix B).

2 Issues in Existing Security Definitions

In this section we briefly discuss why current security definitions do not properly capture the security properties one would expect from a DAA scheme. Some of the arguments were already pointed out by Bernard et al. [3], who provide a thorough analysis of the existing DAA security definitions and also propose a new set of definitions. For the sake of completeness, we summarize and extend their findings and also give an assessment of the latest definition by Bernard et al.

Before discussing the various security definitions and their limitation, we informally describe how DAA works and what are the desired security properties. In a DAA scheme, we have four main entities: a number of trusted platform modules (TPM), a number of hosts, an issuer, and a number of verifiers. A TPM and a host together form a platform which performs the *join protocol* with the issuer who decides if the platform is allowed to become a member. Once being a member, the TPM and host together can *sign* messages with respect to basenames $\texttt{bsn}$. If a platform signs with $\texttt{bsn} = \bot$ or a fresh basename, the signature must be anonymous and unlinkable to previous signatures. That is, any verifier can check that the signature stems from a legitimate platform via a deterministic *verify* algorithm, but the signature does not leak any information about the identity of the signer. Only when the platform signs repeatedly with the same basename $\texttt{bsn} \neq \bot$, it will be clear that the resulting signatures were created by the same platform, which can be publicly tested via a (deterministic) *link* algorithm.

One requires the typical completeness properties for signatures created by honest parties:

Completeness: When an honest platform successfully creates a signature on a message m w.r.t. a basename $\texttt{bsn}$, an honest verifier will accept the signature.

Correctness of Link: When an honest platform successfully creates two signatures, σ_1 and σ_2, w.r.t. the same basename $\texttt{bsn} \neq \bot$, an honest verifier running a link algorithm on σ_1 and σ_2 will output 1. To an honest verifier, it also does not matter in which order two signatures are supplied when testing linkability between the two signatures.

The more difficult part is to define the security properties that a DAA scheme should provide in the presence of malicious parties. These properties can be informally described as follows:

Unforgeability-1: When the issuer and all TPMs are honest, no adversary can create a signature on a message m w.r.t. basename `bsn` when no platform signed m w.r.t. `bsn`.

Unforgeability-2: When the issuer is honest, an adversary can only sign in the name of corrupt TPMs. More precisely, if n TPMs are corrupt, the adversary can at most create n unlinkable signatures for the same basename $\texttt{bsn} \neq \bot$.

Anonymity: An adversary that is given two signatures, w.r.t. two different basenames or $\texttt{bsn} = \bot$, cannot distinguish whether both signatures were created by one honest platform, or whether two different honest platforms created the signatures.

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

2.1 Simulation-Based Security Definitions

A simulation-based security definition defines an ideal functionality, which can be seen as a central trusted party that receives inputs from all parties and provides outputs to them. Roughly, a protocol is called secure if its behavior is indistinguishable from the functionality.

The Brickell, Camenisch, and Chen security definition [5]. DAA was first introduced by Brickell, Camenisch, and Chen [5] along with a simulation-based security definition. The functionality has a single procedure encompassing both signature generation and verification, meaning that a signature is generated for a specific verifier and will immediately be verified by that verifier. As the signature is never output to the verifier, he only learns that a message was correctly signed, but can neither forward signatures or verify them again. Clearly this limits the scenarios in which DAA can be applied.

Furthermore, linkability of signatures with the same basename was not defined explicitly in the security definition. In the instantiation it is handled by attaching pseudonyms to signatures, and when two signatures have the same pseudonym, they must have been created by the same platform.

The Chen, Morissey, and Smart security definitions [14, 17]. An extension to the security definition by Brickell et al. was later proposed by Chen, Morissey, and Smart [17]. It aims at providing linkability as an explicit feature in the functionality. To this end, the functionality is extended with a link interface that takes as input two signatures and determines whether they link. However, as discussed before, the sign and verify interfaces are interactive and thus signatures are never sent as output to parties, so it is not possible to provide them as input either. This was realized by the authors who later proposed a new simulation-based security definition [14] that now separates the generation of signatures from their verification by outputting signatures. Unfortunately, the functionality models the signature generation in a too simplistic way: signatures are simply random values, even when the TPM is corrupt. Furthermore, the verify interface refuses all requests when the issuer is corrupt. Clearly, both these behaviours are not realizable by any protocol.

2.2 Property-Based Security Definitions

Given the difficulties in properly defining ideal functionalities, there is also a line of work that captures DAA features via property-based definitions. Such definitions capture every security property in a separate security game.

The Brickell, Chen, and Li security definition [7]. The first property-based security definition is by Brickell, Chen, and Li [7]. They define security games for anonymity, and "user-controlled traceability". The latter aims to capture our unforgeability-1 and unforgeability-2 requirements. Unfortunately, this definition has several major shortcomings that were already discussed in detail by Bernard et al. [3].

The first problem is that the game for unforgeability-1 considers insecure schemes to be secure. The adversary in the unforgeability-1 game has oracle access to the honest parties from whom he can request signatures on messages and basenames of his choice. The adversary then wins if he can come up with a valid signature that is not a previous oracle response. This last requirement allows trivially insecure schemes to win the security game: assume a DAA scheme that outputs the hash of the TPM's secret key gsk as signature, i.e., the signature is independent of the message. Clearly, this should be an insecure scheme as the adversary, after having seen one signature can provide valid signatures on arbitrary messages of his choice. However, this scheme is secure according to the unforgeability-1 game, as there reused signatures are not considered a forgery.

Another issue is that the game for unforgeability-2 is not well defined. The goal of the adversary is to supply a signature σ, a message m, a basename $\mathtt{bsn} \neq \bot$, and a signer's identity ID. The adversary wins if another signature "associated with the same ID" exists, but the signatures do not link. Firstly, there is no check on the validity of the supplied signature, which makes winning trivial for the adversary. Secondly, "another signature associated with the same ID" is not precisely defined, but we assume it to mean that the signature was the result of a signing query with that ID. However, then the adversary is limited to tamper with at most one of the signatures, whereas the second one is enforced to be honestly generated and unmodified. Thirdly, there is no check on the relation between the signature and the supplied ID. We expect that the intended behavior is that the supplied signature uses the key of ID, but there is no way to enforce this. Now an adversary can simply make a signing query with $(m, \mathtt{bsn}, ID_1)$, thus obtaining σ, and win the game with $(\sigma, m, \mathtt{bsn}, ID_2)$.

The definition further lacks a security game that captures the non-frameability requirement. This means a scheme with a link algorithm that always outputs 1 can be proven secure. Chen [15] extends the definition to add non-frameability, but this extension inherits all the aforementioned problems from [7].

The Bernard et al. security definition [3]. Realizing that the previous security definitions are not sufficient, Bernard et al. [3] provide an extensive set of property-based security definitions covering all expected security requirements.

The main improvement is the way signatures are identified. An identify algorithm is introduced that takes a signature and a TPM key, and outputs whether the key was used to create the signature, which is possible as signatures are uniquely identifiable if the secret key is known. In all their game definitions, the keys of honest TPMs are known, allowing the challenger to identify which key was used to create the signature, solving the problems related to the imprecisely defined ID in the Brickell, Chen, and Li definition.

However, the security games make a simplifying assumption, namely that the platform, consisting of a host and a TPM, is considered as *one* party. This approach, termed "pre-DAA", suffices for anonymity and non-frameability, as there both the TPM and host have to be honest. However, for the unforgeability requirements it is crucial that the host does *not* have to be trusted. In fact, distributing the computational work between the TPM and the host, such that the load on the TPM is as small as possible and, at the same time, not requiring the host to be honest, is the main challenge in designing a DAA scheme. Therefore, a DAA security definition must be able to formally analyze this setting of an honest TPM working with a corrupt host.

The importance of such full DAA security is also acknowledged by Bernard et al. [3]. After formally proving a proposed scheme secure in the pre-DAA setting, the authors bring the scheme to the full DAA setting where the TPM and host are considered as separate parties. To obtain full DAA security, the host randomizes the issuer's credential on the TPM's public key. Bernard et al. then argue that this has no impact on the proven pre-DAA security guarantees as the host does not perform any action involving the TPM secret key. While this seems intuitively correct, it gives no guarantees whether the security properties are *provably* preserved in the full DAA setting. Indeed, the proof of unforgeability of the pre-DAA scheme, which is proven under the DL assumption, does not hold in the full DAA setting as a corrupt host could notice the simulation used in the security proof. More precisely, in the Bernard et al. scheme, the host sends values (b, d) to the TPM which are the re-randomized part of the issued credential and are supposed to have the form $b^{gsk} = d$ with gsk being the TPM's secret key. The TPM then provides a signature proof of knowledge (SPK) of gsk to the host. The pre-DAA proof relies on the DL assumption and places the unknown discrete logarithm of the challenge DL instance as the TPM key gsk. In the pre-DAA setting, the TPM then simulates the proof of knowledge of gsk for any input (b, d). This, however, is no longer possible in the full DAA setting. If the host is corrupt, he can send arbitrary values (b, d) with $b^{gsk} \neq d$ to the TPM. The TPM must only respond with a SPK if (b, d) are properly set, but relying only on the DL assumption does not allow the TPM to perform this check. Thus, the unforgeability can no longer be proven under the DL assumption. Note that the scheme could still be proven secure using the stronger static DH assumption, but the point is that a proof of pre-DAA security and a seemingly convincing but informal argument to transfer the scheme to the full DAA setting does not guarantee security in the full DAA setting.

Another peculiarity of the Bernard et al. definition is that it makes some rather strong yet somewhat hidden assumptions on the adversary's behavior. For instance, in the traceability game showing unforgeability of the credentials, the adversary must not only output the claimed forgery but also the secret keys of all TPMs. For a DAA protocol this implicitly assumes that the TPM secret key can be extracted from every signature. Similarly, in games such as non-frameability or anonymity that capture security against a corrupt issuer, the issuer's key is generated honestly within the game, instead of being chosen by the adversary. For any realization this assumes either a trusted setup setting or an extractable proof of correctness of the issuer's secret key.

In the scheme proposed by Bernard et al. [3], none of these implicit assumptions hold though: the generation of the issuer key is not extractable or assumed to be trusted, and the TPM's secret key cannot be extracted from every signature, as the rewinding for this would require exponential time. Note that these assumptions are indeed necessary to guarantee security for the proposed scheme. If the non-frameability game would allow the issuer to choose its own key, it could choose $y = 0$ and win the game. Ideally, a security definition should not impose such assumptions or protocol details. If such assumptions are necessary though, then they should be made explicit to avoid pitfalls in the protocol design.

3 A New Security Definition for DAA

In this section we present our security definition for DAA, which is defined as an ideal functionality $\mathcal{F}^{l}_{\mathsf{daa}}$ in the UC framework [12]. In UC, an environment $\mathcal{E}$ passes inputs to and receives outputs from the protocol parties. The network is controlled by an adversary $\mathcal{A}$ that may communicate freely with $\mathcal{E}$. In the ideal world, the parties forward their inputs to the ideal functionality $\mathcal{F}$, which then (internally) performs the defined task and creates the party's outputs that they forward to $\mathcal{E}$.

Roughly, a protocol Π is said to securely realize a functionality $\mathcal{F}$ if the real world in which the protocol is used is as secure as the ideal world where the ideal functionality is used, meaning for every adversary performing an attack in the real world, there is an ideal world adversary (often called simulator) $\mathcal{S}$ that performs the same attack in the ideal world. More precisely, a protocol Π is secure if for every adversary $\mathcal{A}$ and evironment $\mathcal{E}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{E}$ cannot distinguish interacting with the real world with Π and $\mathcal{A}$ from interacting with the ideal world with $\mathcal{F}$ and $\mathcal{S}$.

3.1 Ideal Functionality $\mathcal{F}^l_{\mathsf{daa}}$

We now formally define our ideal functionality $\mathcal{F}^l_{\mathsf{daa}}$. We assume static corruptions, i.e., the adversary decides upfront which parties are corrupt and makes this information known to the functionality. The UC framework allows us to focus our analysis on a single scheme instance with a globally unique session identifier sid. Here we use session identifiers of the form $sid = (\mathcal{I}, sid')$ for some issuer $\mathcal{I}$ and a unique string sid'. To allow several sub-sessions for the join and sign related interfaces we use unique sub-session identifiers $jsid$ and $ssid$. Our ideal functionality $\mathcal{F}^l_{\mathsf{daa}}$ is further parametrized by a leakage function $l : \{0,1\}^* \to \{0,1\}^*$, that models the information leakage that occurs in the communication between a host $\mathcal{H}_i$ and TPM $\mathcal{M}_j$.

We first briefly describe the main interfaces, then present the full functionality $\mathcal{F}^l_{\mathsf{daa}}$ and finally discuss in depth why $\mathcal{F}^l_{\mathsf{daa}}$ implements the desired security properties.

Setup. The SETUP interface on input $sid = (\mathcal{I}, sid')$ initiates a new DAA session for the issuer $\mathcal{I}$ and expects the adversary to provide a number of algorithms $(\mathsf{ukgen}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify})$ that will be used inside the functionality. These algorithms are used as follows:

- $gsk \xleftarrow{\$} \mathsf{ukgen}()$ will be used to generate keys gsk for honest TPMs.
- $\sigma \xleftarrow{\$} \mathsf{sig}(gsk, m, bsn)$ will also be used for honest TPMs. On input a key gsk, a message m, and a basename $\mathtt{bsn}$, it outputs a signature σ.
- $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$ will be used in the verify interface. On input a signature σ, a message m, and a basename $\mathtt{bsn}$, it outputs $f = 1$ if the signature is valid, and $f = 0$ otherwise.
- $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$ will be used in the link interface. It takes two tuples (σ, m), (σ', m') and a basename $\mathtt{bsn}$ as input and outputs $f = 1$ to indicate that both signature are generated by the same TPM and $f = 0$ otherwise.
- $f \leftarrow \mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk)$ outputs $f = 1$ if σ is a signature on m with respect to $\mathtt{bsn}$ under key gsk, and $f = 0$ otherwise. We will use $\mathsf{identify}$ in several places to ensure consistency, e.g., whenever a new key gsk is generated or provided by the adversary.

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

Join. When the setup is completed, a host $\mathcal{H}_j$ can request to join with a TPM $\mathcal{M}_i$ using the JOIN interface. Only if the issuer gives his approval through the $\mathsf{JOINPROCEED}$ interface, the join will complete and $\mathcal{F}^l_{\mathsf{daa}}$ stores $\langle \mathcal{M}_i, \mathcal{H}_j, gsk \rangle$ in an internal list $\texttt{Members}$. If the host or TPM are corrupt, gsk has to be provided by the adversary. If both are honest, $\mathcal{F}^l_{\mathsf{daa}}$ stores $gsk \leftarrow \bot$.

On the first glance, it might seem a bit surprising that we let the adversary also provide gsk when the host is corrupt but the TPM is honest. However we use gsk inside the functionality only to reflect the anonymity properties according to the set of corrupted parties. Only if the entire platform if honest, one can guarantee anonymity and then we enforce $gsk \leftarrow \bot$. Note that the value of gsk has in particular no impact on the unforgeability guarantees that are always enforced by $\mathcal{F}^l_{\mathsf{daa}}$ if $\mathcal{M}_i$ is honest.

Sign. Once a platform has joined, the host $\mathcal{H}_j$ can call the SIGN interface to request a DAA signature from a TPM $\mathcal{M}_i$ on a message m with respect to a basename $\texttt{bsn}$. If the issuer is honest, only platforms $\langle \mathcal{M}_i, \mathcal{H}_j, gsk \rangle \in \texttt{Members}$ can sign. The TPM is notified and has to give its explicit approval through the $\mathsf{SIGNPROCEED}$ interface. If the host or TPM is corrupt, the signature σ has to be input by the adversary. When both are honest, the signature is generated via the sig algorithm. For this, $\mathcal{F}^l_{\mathsf{daa}}$ first chooses a fresh key gsk whenever an honest platform (honest host and honest TPM) wishes to sign under a *new* basename, which naturally enforces unlinkability and anonymity of those signatures. Every newly generated key is stored as $\langle \mathcal{M}_i, \texttt{bsn}, gsk \rangle$ in a list $\texttt{DomainKeys}$ and will be re-used whenever the honest platform wants to sign under the same $\texttt{bsn}$ again. For honest TPMs, the generated or adversarial provided signature is also stored as $\langle \sigma, m, \texttt{bsn}, \mathcal{M}_i \rangle$ in a list $\texttt{Signed}$.

Verify. The verify interface VERIFY allows any party $\mathcal{V}$ to check whether σ is a valid signature on m with respect to $\texttt{bsn}$. The functionality will use its internal records to determine whether σ is a proper signature. Here we also use the helper algorithm $\mathsf{identify}$ to determine which of the gsk values stored by $\mathcal{F}^l_{\mathsf{daa}}$ belongs to that signature. If the key belongs to an honest TPM, then an entry $\langle *, m, \texttt{bsn}, \mathcal{M}_i \rangle \in \texttt{Signed}$ must exist. For signatures of corrupt TPMs, $\mathcal{F}^l_{\mathsf{daa}}$ checks that a valid signature would not violate any of the expected properties, e.g., whether the signature links to another signature by an honest TPM.

The interface also provides verifier-local revocation, as it accepts a revocations list $\texttt{RL}$ as additional input which is a list of gsk values from which the verifier does not accept signatures anymore. To ensure that this does not harm the anonymity of honest TPMs, $\mathcal{F}^l_{\mathsf{daa}}$ ignores all honest gsk values for the revocation check.

If the $\mathcal{F}^l_{\mathsf{daa}}$ did find some reason why the signature should *not* be valid, it sets the output to $f \leftarrow 0$. Otherwise, it determines the verification result f using the ver algorithm. Finally, the functionality keeps track of this result by adding $\langle \sigma, m, \texttt{bsn}, \texttt{RL}, f \rangle$ to a list $\texttt{VerResults}$.

Link. Any party $\mathcal{V}$ can use the LINK interface to learn whether two signatures (σ, σ'), on messages (m, m') respectively, generated with the same basename $\texttt{bsn}$ originate from the same TPM or not. Similarly as for verification, $\mathcal{F}^l_{\mathsf{daa}}$ then first uses its internal records and helper functions to determine if there is any evidence for linkage or non-linkage. If such evidence is found, then the output bit

f is set accordingly to 0 or 1. When the functionality has no evidence that the signatures must or must not belong together, it determines the linking result via the link algorithm.

The full definition of $\mathcal{F}^l_{\mathsf{daa}}$ is given in Figures 1 and 2. To save on repeating and non-essential notation, we use the following conventions in our definition:

- All requests other than the SETUP are ignored until one setup phase is completed. For such requests, $\mathcal{F}$ outputs $\perp$ to the caller immediately.
- Whenever the functionality performs a check that fails, it outputs $\perp$ directly to the caller of the respective interface.
- We require the link algorithm to be symmetric: $\mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn}) \leftrightarrow \mathsf{link}(\sigma', m', \sigma, m, \mathtt{bsn})$. To guarantee this, whenever we write that $\mathcal{F}$ runs $\mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$, it runs $\mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$ and $\mathsf{link}(\sigma', m', \sigma, m, \mathtt{bsn})$. If the results are equal, it continues as normal with the result, and otherwise $\mathcal{F}$ outputs $\perp$ to the adversary.
- When $\mathcal{F}$ runs one of the algorithms sig, ver, $\mathsf{identify}$, link, and ukgen, it does so without maintaining state. This means all user keys have the same distribution, signatures are equally distributed for the same input, and ver, $\mathsf{identify}$, and link invocations only depend on the current input, not on previous inputs.

We will further use two "macros" to determine if a gsk is consistent with the functionality's records or not. This is checked at several places in our functionality and also depends on whether the gsk belongs to an honest or corrupt TPM. The first macro $\mathsf{CheckGskHonest}$ is used when the functionality stores a new TPM key gsk that belongs to an honest TPM, and checks that none of the existing valid signatures are identified as belonging to this TPM key. The second macro $\mathsf{CheckGskCorrupt}$ is used when storing a new gsk that belongs to a corrupt TPM, and checks that the new gsk does not break the identifiability of signatures, i.e., it checks that there is no other known TPM key gsk', unequal to gsk, such that both keys are identified as the owner of a signature. Both functions output a bit b where $b = 1$ indicates that the new gsk is consistent with the stored information, whereas $b = 0$ signals an invalid key. The two macros are defined as follows.

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

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

Setup

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

Join

3. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$
   – Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status \leftarrow request$.
   – Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.
4. `Join Request Delivery.` On input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{S}$
   – Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ to $status \leftarrow delivered$.
   – Abort if $\mathcal{I}$ or $\mathcal{M}_i$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in$ `Members` already exists **(ii)**.
   – Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
5. `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$
   – Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ to $status \leftarrow complete$.
   – Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{S}$.
6. `Platform Key Generation.` On input $(\mathsf{JOINCOMPLETE}, sid, jsid, gsk)$ from $\mathcal{S}$.
   – Look up record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = complete$.
   – If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, set $gsk \leftarrow \bot$.
   – Else, verify that the provided gsk is eligible by checking
   - $\mathsf{CheckGskHonest}(gsk) = 1$ **(iii)** if $\mathcal{H}_j$ is corrupt and $\mathcal{M}_i$ is honest, or
   - $\mathsf{CheckGskCorrupt}(gsk) = 1$ **(iv)** if $\mathcal{M}_i$ is corrupt.

   – Insert $\langle \mathcal{M}_i, \mathcal{H}_j, gsk\rangle$ into `Members` and output $(\mathsf{JOINED}, sid, jsid)$ to $\mathcal{H}_j$.

Fig. 1. The Setup and Join related interfaces of $\mathcal{F}^l_{\mathsf{daa}}$. *(The roman numbers are labels for the different checks made within the functionality and will be used as references in the detailed analysis in Section 3.2)*

3.2 Detailed Analysis of $\mathcal{F}^l_{\mathsf{daa}}$

We now argue that our functionality enforces the desired unforgeability, anonymity, and non-frameability properties we informally introduced in Section 2.

In terms of completeness and correctness, we further have to add to three more properties: consistency of verify, consistency of link, and symmetry of link. These properties are trivially achieved for property-based definitions, where one simply requires the algorithms to be deterministic, and the link algorithm to be symmetric. In a simulation-based definition, however, the behavior of a functionality may depend on its state, which is why we explicitly show that we achieve these properties.

We start with the security related properties unforgeability, anonymity and non-frameability, and then discuss the correctness and consistency properties.

Unforgeability. We consider two unforgeability properties, depending on all TPMs being honest or some of them being corrupt. The issuer is of course always honest when aiming at unforgeability. Firstly, if all TPMs are honest, an adversary cannot create a signature on a message m with respect to basename

Sign

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

Verify

11. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
    - Retrieve all pairs $(gsk_i, \mathcal{M}_i)$ from $\langle \mathcal{M}_i, *, gsk_i \rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i, *, gsk_i \rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
      - More than one key gsk_i was found **(ix)**.
      - $\mathcal{I}$ is honest and no pair $(gsk_i, \mathcal{M}_i)$ was found **(x)**.
      - There is an honest $\mathcal{M}_i$ but no entry $\langle *, m, \mathtt{bsn}, \mathcal{M}_i \rangle \in \mathtt{Signed}$ exists **(xi)**.
      - There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$ and no pair $(gsk_i, \mathcal{M}_i)$ for an honest $\mathcal{M}_i$ was found **(xii)**.
    - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$ **(xiii)**.
    - Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f \rangle$ to $\mathtt{VerResults}$, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

12. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \bot$.
    - Output $\bot$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$) **(xiv)**.
    - For each gsk_i in $\mathtt{Members}$ and $\mathtt{DomainKeys}$ compute $b_i \leftarrow \mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk_i)$ and $b'_i \leftarrow \mathsf{identify}(\sigma', m', \mathtt{bsn}, gsk_i)$ and do the following:
      - Set $f \leftarrow 0$ if $b_i \neq b'_i$ for some i **(xv)**.
      - Set $f \leftarrow 1$ if $b_i = b'_i = 1$ for some i **(xvi)**.
    - If f is not defined yet, set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.
    - Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 2. The Sign, Verify, and Link related interfaces of $\mathcal{F}^{l}_{\mathsf{daa}}$

$\mathtt{bsn}$ when no honest TPM signed m with respect to $\mathtt{bsn}$. By Check **(x)**, the signature must trace to some TPMs gsk. As we assumed all TPMs to be honest, Check **(xi)** will reject any signature on messages not signed by that TPM.

Secondly, when some TPMs are corrupt, an adversary cannot forge signatures with more distinct 'identities' than there are corrupt TPMs. More precisely, when the adversary corrupted n TPMs, he cannot create more than n unlinkable signatures for the same $\mathtt{bsn} \neq \perp$, and when no honest TPM signed under $\mathtt{bsn}$ too. We show that for any $n + 1$ signatures $\{\sigma_i, m_i, \mathtt{bsn}\}_{0 \geq i \geq n}$, we cannot have that all signatures verify, m_i was not signed with respect to $\mathtt{bsn}$ by an honest TPMs, and every pair of signatures is unlinkable.

If all signatures verify, by Check **(x)**, each of the $n+1$ signatures must trace to exactly one pair $(\mathcal{M}_i, gsk_i)$. Given the fact that no honest TPM signed m_i with respect to $\mathtt{bsn}$, by Check **(xi)**, we must have that every TPM in the list of tracing $(\mathcal{M}_i, gsk_i)$ pairs is corrupt. Furthermore, we know that all $(\mathcal{M}_i, gsk_i)$ come from $\mathtt{Members}$, as only honest TPMs occur in $\mathtt{DomainKeys}$. Since the issuer is honest, Check **(ii)** enforces that every TPM can join at most once, i.e., there can be at most n pairs $(\mathcal{M}_i, gsk_i)$ of corrupt TPMs in $\mathtt{Members}$. Thus, the traced list of $(\mathcal{M}_i, gsk_i)$ pairs must contain at least one duplicate entry. By Check **(xvi)**, the two signatures that trace to the same gsk must link, showing that the adversary cannot forge more than n unlinkable signatures with a single $\mathtt{bsn} \neq \perp$.

Anonymity. Anonymity of signatures created by an honest TPM $\mathcal{M}_i$ and host $\mathcal{H}_j$ is guaranteed by $\mathcal{F}^l_{\mathsf{daa}}$ due to the random choice of gsk for every signature. More precisely, if the platform is honest, our functionality does not store any unique gsk for the pair $(\mathcal{M}_i, \mathcal{H}_j)$ in $\mathtt{Members}$, but leaves the key unassigned. Whenever a new signature is requested for an unused basename $\mathtt{bsn}$, $\mathcal{F}^l_{\mathsf{daa}}$ first draws a fresh key $gsk \leftarrow \mathsf{ukgen}$ under which it then creates the signature using the sig algorithm. The combination of basename and key is stored as $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle$ in a list $\mathtt{DomainKeys}$, and gsk is re-used whenever $\mathcal{M}_i$ wishes to sign under the same $\mathtt{bsn} \neq \perp$ again.

That is, two signatures with different basenames or with basename $\mathtt{bsn} = \perp$ are distributed in exactly the same way for all honest platforms, independent of whether the signatures are created for the same platform or for two distinct platforms.

Verifier-local revocation is enabled via the revocation list attribute $\mathtt{RL}$ in the VERIFY interface and allows to "block" signatures of exposed gsk's. This revocation feature should not be exploitable to trace honest users, though, as that would clearly break anonymity. To this end, $\mathcal{F}^l_{\mathsf{daa}}$ ignores $gsk \in \mathtt{RL}$ in the revocation test when the key belongs to an honest TPM (Check **(xii)**).

Note that the anonymity property dictated the use of the sig algorithm in $\mathcal{F}^l_{\mathsf{daa}}$. We only use the algorithm if the platform is honest though, whereas for corrupt platforms the simulator is allowed to provide the signature (which then could depend on the identity of the signer). This immediately reflects that anonymity is only guaranteed if both the TPM and host are honest.

Non-frameability. An honest platform $(\mathcal{M}_i, \mathcal{H}_j)$ cannot be framed, meaning that no one can create signatures on messages that the platform never signed but that link to signatures the platform did create. Note that this definition also crucially relies on both $\mathcal{M}_i$ and $\mathcal{H}_j$ being honest. Intuitively, one might expect that only the TPM $\mathcal{M}_i$ is required to be honest, and the host could be corrupt. However, that would be unachievable. We can never control the signatures that a corrupt $\mathcal{H}_j$ outputs. In particular, the host could additionally run a corrupt TPM that joined as well, and create signatures using the corrupt TPM's key instead of using $\mathcal{M}_i$'s contribution. The resulting signature can not be protected from framing, as it uses a corrupt TPM's key. Thus, for a meaningful non-frameability guarantee, the host has to be honest too. The issuer can of course be corrupt.

We now show that when an honest platform $(\mathcal{M}_i, \mathcal{H}_j)$ created a signature σ on message m and under basename $\mathtt{bsn}$, then no other signature σ' on some m' links to σ when $(\mathcal{M}_i, \mathcal{H}_j)$ never signed m' with respect to $\mathtt{bsn}$. The first requirement in LINK is that both signatures must be valid (Check **(xiv)**). By completeness (discussed below) we know that $\sigma, m, \mathtt{bsn}$ generated by the honest platform is valid, and that it traces to some key gsk. If the second signature $\sigma', m', \mathtt{bsn}$ is valid too, we know by the Check **(xi)** in the VERIFY interface that the signature cannot trace to the same gsk, because $(\mathcal{M}_i, \mathcal{H}_j)$ has never signed $m', \mathtt{bsn}'$. Finally, by Check **(xv)** that ensures that the output of $\mathsf{identify}$ must be consistent for *all* used keys, the output of LINK is set to $f \leftarrow 0$.

Completeness. The functionality guarantees completeness, i.e., when an honest platform successfully creates a signature, this signature will be accepted by honest verifiers. More precisely, when honest TPM $\mathcal{M}_i$ with honest host $\mathcal{H}_j$ signs m with respect to $\mathtt{bsn}$ resulting in a signature σ, a verifier will accept $(\sigma, m, \mathtt{bsn})$. To show this, we argue that the four checks the functionality makes (Check **(ix)**, Check **(x)**, Check **(xi)**, and Check **(xii)**) do not set f to 0, and that ver will accept the signature.

Check **(ix)** will not trigger, as by Check **(viii)** there was no honest TPM other than $\mathcal{M}_i$ with a key matching this signature yet and, by Check **(iv)**, Check **(iv)**, and Check **(v)**, gsk values matching σ cannot be added to $\mathtt{Members}$ and $\mathtt{DomainKeys}$.

Check **(x)** will not trigger as we have an entry $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle \in \mathtt{DomainKeys}$, and by Check **(vii)** we know this one matches σ.

In Check **(xi)**, $\mathcal{F}^l_{\mathsf{daa}}$ finds all honest TPMs that have a key matching this signature, and checks whether they signed m with respect to $\mathtt{bsn}$. By Check **(viii)**, at the time of signing there were no other TPMs with keys matching this signature and, by Check **(iii)** and Check **(v)**, no honest TPM can get a key matching this signature. The only honest TPM with a matching key is $\mathcal{M}_i$, but as he honestly signed m with respect to $\mathtt{bsn}$, we have an entry $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i \rangle \in \mathtt{Signed}$ ensuring that the check does not trigger.

The revocation test Check **(xii)** does not trigger as by Check **(vii)** we know that honest TPM $\mathcal{M}_i$ has a key matching this signature.

As all previous checks did not apply, $\mathcal{F}^l_{\mathsf{daa}}$ sets the verification outcome using the ver algorithm, we now show that ver will accept the signature. $\mathcal{F}^l_{\mathsf{daa}}$ checked that ver accepts the signature in Check **(vi)**, and by Check **(i)** and the fact that $\mathcal{F}^{l}_{\mathsf{daa}}$ does not maintain state for the algorithms, the verification algorithm output only depends on its input, so ver outputs 1 and $\mathcal{F}^{l}_{\mathsf{daa}}$ accepts the signature.

Correctness of Link. If an honest platform signs multiple times with the same basename, the resulting signatures will link. Let platform ($\mathcal{M}_i$, $\mathcal{H}_j$) sign messages m_1 and m_2 with basename $\mathtt{bsn} \neq \bot$, resulting in signatures σ_1 and σ_2 respectively. By completeness, both signatures verify, so Check **(xiv)** does not trigger. By Check **(vii)**, both signatures identify to some gsk, which results in Check **(xvi)** setting the signatures as linked.

Consistency of Verify. This property ensures that calling the VERIFY interface with the same input multiple times gives the same result every time. To prevent the functionality from becoming unnecessarily complex, we only enforce consistency for valid signatures. That is, whenever a signature was accepted, it will remain valid, whereas an invalid signature could become valid at a later time.

Suppose a signature σ on message m with basename $\mathtt{bsn}$ was verified successfully with revocation list $\mathtt{RL}$. We now show that in any future verification with the same $\mathtt{RL}$ will lead to the same result. To show this, we argue that the four checks the functionality makes (Check **(ix)**, Check **(x)**, Check **(xi)**, and Check **(xii)**) do not set f to 0, and that ver will accept the signature.

Check **(ix)** makes sure that at most one key gsk matches the signature σ, meaning that for at most one gsk we have $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk) = 1$. This check does not cause rejection of the signature, as the signature previously verified, and by Check **(ix)** we have that at most one gsk matched the signature at that time. After that, the signature was placed in $\mathtt{VerResults}$, which means Check **(iii)**, Check **(iv)**, and Check **(v)** prevent adding gsk values that match σ, so the number of matching gsk values has not changed and Check **(ix)** still passes.

Check **(x)** does not apply. If $\mathcal{I}$ is corrupt, the check trivially does not trigger. If $\mathcal{I}$ is honest, from the previous verification we have that there was precisely one key matching, and as argued for the previous check, no matching gsk values can be added, so we must still have precisely one matching gsk.

To show that Check **(xi)** does not apply, we must show that for every honest TPM that has a key matching this signature, that TPM has signed m with respect to $\mathtt{bsn}$. The check previously passed, so we know that at that point for any matching $\mathcal{M}_i$ there is a satisfying entry in $\mathtt{Signed}$. No new TPMs matching this signature can be found, as Check **(iii)** and Check **(v)** prevent honest TPMs from registering a key that matches an existing signature.

Check **(xii)**, the revocation check, did not reject σ in the previous verification. By the fact that $\mathsf{identify}$ is deterministic Check **(i)** and executed without maintaining state, it will not do so now.

As the four checks $\mathcal{F}^{l}_{\mathsf{daa}}$ makes did not apply, $\mathcal{F}^{l}_{\mathsf{daa}}$ uses the verification algorithm ver. Since the signature was previously accepted, by Check **(xiii)** ver must have accepted the signature. By the fact that ver is deterministic (Check **(i)**) and executed without maintaining state, it will also accept now.

Consistency of Link. We also want to ensure that calling the LINK interface with the same input multiple times gives the same result every time. Here we guarantee consistency for both outputs $f \in \{0, 1\}$ i.e., if LINK outputs f for some input $(\sigma, m, \sigma', m', \mathtt{bsn})$, the result will always be f.

Suppose we have signatures σ and σ' on messages m and m' respectively, both with respect to basename $\mathtt{bsn}$, that have been linked with output $f \in \{0, 1\}$ before. We now show that the same result f will be given in future queries, by showing that Check **(xiv)** will not cause an output of $\perp$, and by showing that Check **(xv)**, Check **(xvi)**, and the link algorithm are consistent.

$\mathcal{F}^{l}_{\mathsf{daa}}$ will not output $\perp$, as by the previous output $f \neq \perp$ we know that the verification of both signatures must have passed. As VERIFY is consistent for valid signatures, this test in Check **(xiv)** will pass again.

Check **(xv)** and Check **(xvi)** are consistent. They depend on the gsk values in $\mathtt{Members}$ and $\mathtt{DomainKeys}$ that match the signatures and are retrieved via the deterministic $\mathsf{identify}$ algorithm. The matching gsk values cannot have changed as Check **(iii)**, Check **(iv)**, and Check **(v)** prevent conflicting gsk values to be added to these lists. The link algorithm used to in the final step is deterministic by Check **(i)**. Thus, LINK will consistently generate the same output bit f.

Symmetry of Link. The link interface is symmetric, i.e., it does not matter whether one gives input $(\mathsf{LINK}, \sigma, m, \sigma', m', \mathtt{bsn})$ or $(\mathsf{LINK}, \sigma', m', \sigma, m, \mathtt{bsn})$. Both signatures are verified, the order in which this happens does not change the outcome. Next $\mathcal{F}^{l}_{\mathsf{daa}}$ finds matching keys for the signatures, and as $\mathsf{identify}$ is executed without state, it does not matter whether it first tries to match σ or σ'. The next checks are based on the equality of the b_i and b'_i values, which clearly is symmetric. Finally $\mathcal{F}^{l}_{\mathsf{daa}}$ uses the link algorithm, which is enforced to be symmetric as $\mathcal{F}^{l}_{\mathsf{daa}}$ will abort as soon as it detects link not being symmetric.

4 Building Blocks

In this section we introduce the building blocks for our construction. Apart from standard building blocks such as pairing-based CL-signatures [9] and zero-knowledge proofs, we also provide a new functionality $\mathcal{F}_{\mathsf{auth}*}$ that captures the semi-authenticated channel that is present in the DAA setting.

4.1 Bilinear Maps

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

4.2 Camenisch-Lysyanskaya Signature

We now recall the pairing-based Camenisch-Lysyanskaya (CL) signature scheme [9] that allows for efficient proofs of signature possession and is the basis for the DAA scheme we extend. The scheme uses a bilinear group $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ that is available to all algorithms.

Key generation. The key generation algorithm chooses $x \xleftarrow{\$} \mathbb{Z}_q$ and $y \xleftarrow{\$} \mathbb{Z}_q$, and sets $sk \leftarrow (x, y)$, $pk \leftarrow (X, Y)$, where $X \leftarrow g_2^x$ and $Y \leftarrow g_2^y$.

Signature. On input a message m and secret key $sk = (x, y)$, choose a random $a \xleftarrow{\$} \mathbb{G}_1$, and output the signature $\sigma \leftarrow (a, a^y, a^{x+mxy})$.

Verification. On input a public key $pk = (X, Y)$, message m, and purported signature $\sigma = (a, b, c)$, output 1 if the following verification equations hold, and 0 otherwise: $a \neq 1_{\mathbb{G}_1}, e(a, Y) = e(b, g_2)$ and $e(a \cdot b^m, X) = e(c, g_2)$.

This signature scheme is existentially unforgeable against a chosen-message attack (EUF-CMA) under the LRSW assumption [21], which is proven in [9]. Certain schemes [3, 18], including ours, add a fourth element $d = b^m$ to the signature, which allows more efficient proofs of knowledge of a message signed by a signature. This extended CL signature is as secure as the original CL signature: Any adversary that can create a standard CL forgery (a, b, c) on message m can forge an extended CL signature by adding $d = b^m$. Any adversary that can create an extended CL forgery (a, b, c, d) on m can forge a standard CL signature, by adding $d = b^m$ to the signing oracle outputs, and omitting d from the final forgery.

4.3 Proof Protocols

When referring to the zero-knowledge proofs of knowledge of discrete logarithms and statements about them, we will follow the notation introduced by Camenisch and Stadler [11] and formally defined by Camenisch, Kiayias, and Yung [8].

For instance, $PK\{(a, b, c) : y = g^a h^b \wedge \tilde{y} = \tilde{g}^a \tilde{h}^c\}$ denotes a "*zero-knowledge Proof of Knowledge of integers a, b and c such that $y = g^a h^b$ and $\tilde{y} = \tilde{g}^a \tilde{h}^c$ holds,*" where $y, g, h, \tilde{y}, \tilde{g}$ and $\tilde{h}$ are elements of some groups $\mathbb{G} = \langle g \rangle = \langle h \rangle$ and $\tilde{\mathbb{G}} = \langle \tilde{g} \rangle = \langle \tilde{h} \rangle$. Given a protocol in this notation, it is straightforward to derive an actual protocol implementing the proof [8]. Indeed, the computational complexities of the proof protocol can be easily derived from this notation: for each term $y = g^a h^b$, the prover and the verifier have to perform an equivalent computation, and to transmit one group element and one response value for each exponent.

SPK denotes a signature proof of knowledge, that is a non-interactive transformation of a proof with the Fiat-Shamir heuristic [19] in the random oracle model [2]. From these non-interactive proofs, the witness can be extracted by rewinding the prover and programming the random oracle. Alternatively, these proofs can be extended to be online-extractable, by verifiably encrypting the witness to a public key defined in the common reference string (CRS). Now a simulator controlling the CRS can extract the witness without rewinding by decrypting the ciphertext. A practical instantiation is given by Camenisch and Shoup [10] using Paillier encryption, secure under the DCR assumption [22].

4.4 Semi-Authenticated Channels via $\mathcal{F}_{\mathsf{auth}*}$

In the join protocol of DAA, it is crucial that the TPM and issuer authenticate to each other, such that only authentic TPMs can create signatures. This is not an ordinary authenticated channel, as all communication is channeled via the host, that can read the messages, block the communication, or append messages. There exist several sub-protocols and setup settings in the DAA context that provide this type of special authenticated channels, of which an overview is given by Bernard et al. [3]. These constructions require the TPM to have a key pair, the endorsement key, of which the public part is known to the issuer. In practice, the TPM manufacturer certifies the public key using traditional PKI, allowing an issuer to verify that this public key indeed belongs to a certain TPM. If the endorsement key is a key for a signature scheme, the TPM can send an authenticated message to the issuer by signing the message. If a public key encryption key is used, this can be used to exchange a MAC key to authenticate later messages.

We design a functionality $\mathcal{F}_{\mathsf{auth}*}$ modeling the desired channel, which allows us to rather use the abstract functionality in the protocol design instead of a concrete sub-protocol. Then, any protocol that securely realizes $\mathcal{F}_{\mathsf{auth}*}$ can be used for the initial authentication.

The functionality must capture the fact that the sender S sends a message containing an authenticated and an unauthenticated part to the receiver R, while giving some forwarder F (this role will be played by the host) the power to block the message or replace the unauthenticated part, and giving the adversary the power to replace the forwarder's message and block the communication. We capture these requirements in $\mathcal{F}_{\mathsf{auth}*}$, defined in Figure 3.

1. On input $(\mathsf{SEND}, sid, ssid, m_1, m_2, F)$ from S. Check that $sid = (S, R, sid')$ for some R an output $(\mathsf{REPLACE}1, sid, ssid, m_1, m_2, F)$ to $\mathcal{S}$.
2. On input $(\mathsf{REPLACE}1, sid, ssid, m_2')$ from $\mathcal{S}$, output $(\mathsf{APPEND}, sid, ssid, m_1, m_2')$ to F.
3. On input $(\mathsf{APPEND}, sid, ssid, m_2'')$ from F, output $(\mathsf{REPLACE}2, sid, ssid, m_1, m_2'')$ to $\mathcal{S}$.
4. On input $(\mathsf{REPLACE}2, sid, ssid, m_2''')$ from $\mathcal{S}$, output $(\mathsf{SENT}, sid, ssid, m_1, m_2''')$ to R.

Fig. 3. The special authenticated communication functionality $\mathcal{F}_{\mathsf{auth}*}$

Clearly we can realize this functionality using the endorsement key and a signature scheme or public key encryption scheme.

5 Construction

In this section, we present our DAA scheme that securely implements $\mathcal{F}_{\mathsf{daa}}^{l}$. While our scheme is similar to previous constructions [3, 6, 7, 16, 18], it required several modifications in order to fulfill all of our desired security guarantees. We give a detailed discussion of the changes with respect to previous versions in Section 5.2.

The high-level idea of our DAA scheme is as follows. In the join protocol, a platform, consisting of a TPM $\mathcal{M}_i$ and host $\mathcal{H}_j$, receives a credential (a, b, c, d) from the issuer $\mathcal{I}$ which is a Camenisch-Lysyanskaya signature [9] on some TPM chosen secret gsk. After joining, the platform can sign any message m w.r.t. some basename $\texttt{bsn}$. To this end, the host first randomizes the credential (a, b, c, d) to $(a' = a^r, b' = b^r, c' = c^r, d' = d^r)$ for a random r and then lets the TPM $\mathcal{M}_i$ create a signature proof of knowledge (SPK) on m showing that $b'^{gsk} = d'$. To obtain user-controlled linkability for basenames $\texttt{bsn} \neq \perp$, pseudonyms are attached to the signature. Pseudonyms are similar to BLS signatures [4] on the basename and have the form $\texttt{nym} = H_1(\texttt{bsn})^{gsk}$ for some hash function H_1. Whenever a basename $\texttt{bsn} \neq \perp$ is used, the SPK generated by the TPM also proves that the pseudonym is well-formed.

5.1 Our DAA protocol Π_{daa}

We now present our DAA scheme in detail, and also give a simplified overview of the join and sign protocols in Figures 4 and 5 respectively.

We assume that a common reference string functionality $\mathcal{F}_{\mathsf{crs}}^{D}$ and a certificate authority functionality $\mathcal{F}_{\mathsf{ca}}$ are available to all parties. The later allows the issuer to register his public key, and $\mathcal{F}_{\mathsf{crs}}^{D}$ is used to provide all entities with the system parameters comprising a security parameter τ, a bilinear group $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order q with generators g_1, g_2 and bilinear map e, generated via $\mathcal{G}(1^\tau)$. We further use a random oracle $H_1 : \{0, 1\}^* \to \mathbb{G}_1$.

For the communication between the TPM and issuer (via the host) in the join protocol, we use our semi-authenticated channel $\mathcal{F}_{\mathsf{auth}*}$ introduced in Section 4.4. For all communication between a host and TPM we assume the secure message transmission functionality $\mathcal{F}_{\mathsf{smt}}^{l}$ (enabling authenticated and encrypted communication). In practice, $\mathcal{F}_{\mathsf{smt}}^{l}$ is naturally guaranteed by the physical proximity of the host and TPM forming the platform, i.e., if both are honest an adversary can neither alter nor read their internal communication. To make the protocol more readable, we simply say that host $\mathcal{H}_i$ sends a message to, or receives a message from TPM $\mathcal{M}_j$, instead of explicitly calling $\mathcal{F}_{\mathsf{smt}}^{l}$ with sub-session IDs etc. For definitions of the standard functionalities $\mathcal{F}_{\mathsf{crs}}^{D}, \mathcal{F}_{\mathsf{ca}}$ and $\mathcal{F}_{\mathsf{smt}}^{l}$ we refer to Appendix A.

In the description of the protocol, we assume that parties call $\mathcal{F}_{\mathsf{crs}}^{D}$ and $\mathcal{F}_{\mathsf{ca}}$ to retrieve the necessary key material whenever they use a public key of another party. In case of the issuer public key, every protocol participant first verifies the correctness of the key by checking the proof π and that $Y \neq 1_{\mathbb{G}_2}$ (with π, Y being parts of the public key as defined below). Further, if any of the checks in the protocol fails, the protocol ends with a failure message $\perp$. The protocol also outputs $\perp$ whenever a party receives an input or message it does not expect (e.g., protocol messages arriving in the wrong order.)

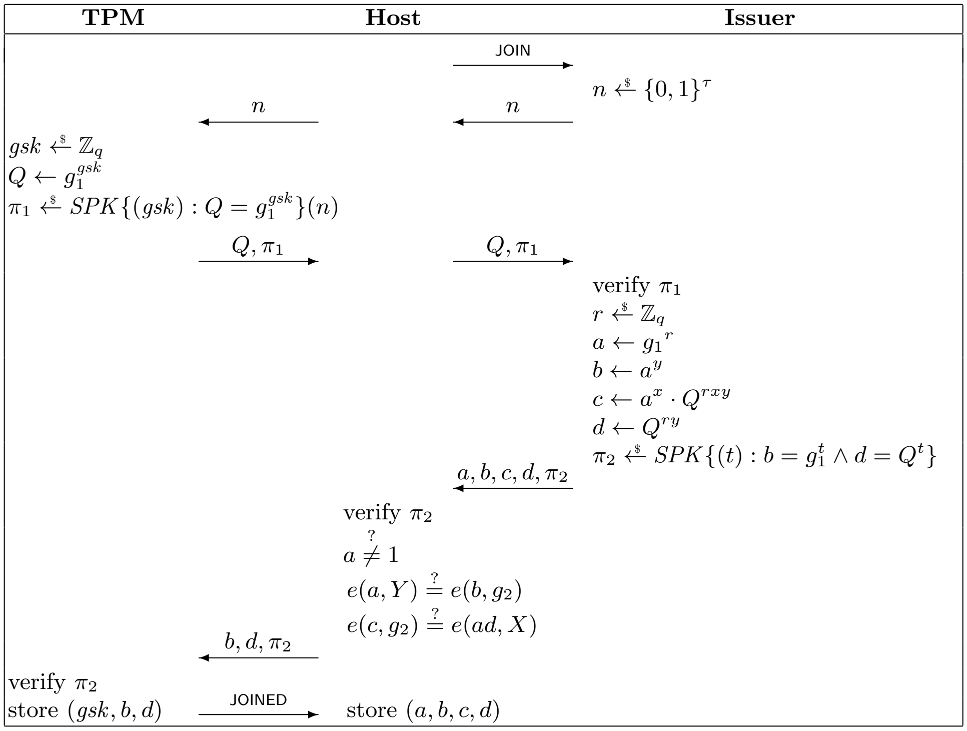
TPM
Host
Issuer
JOIN
$n \xleftarrow{\$} \{0,1\}^\tau$
n
n
$gsk \xleftarrow{\$} \mathbb{Z}_q$
$Q \leftarrow g_1^{gsk}$
$\pi_1 \xleftarrow{\$} SPK\{(gsk) : Q = g_1^{gsk}\}(n)$
Q, π_1
Q, π_1
verify π_1
$r \xleftarrow{\$} \mathbb{Z}_q$
$a \leftarrow {g_1}^r$
$b \leftarrow a^y$
$c \leftarrow a^x \cdot Q^{rxy}$
$d \leftarrow Q^{ry}$
$\pi_2 \xleftarrow{\$} SPK\{(t) : b = g_1^t \wedge d = Q^t\}$
a, b, c, d, π_2
verify π_2
$a \overset{?}{\neq} 1$
$e(a, Y) \overset{?}{=} e(b, g_2)$
$e(c, g_2) \overset{?}{=} e(ad, X)$
b, d, π_2
verify π_2
store (gsk, b, d)
JOINED
store (a, b, c, d)

Fig. 4. Overview of the join protocol

Setup. In the setup phase, the issuer $\mathcal{I}$ creates a key pair of the CL-signature scheme and registers the public key with $\mathcal{F}_{\mathsf{ca}}$.

1. $\mathcal{I}$ upon input (SETUP, sid) generates his key pair:

- Check that $sid = (\mathcal{I}, sid')$ for some sid'.
- Choose $x, y \xleftarrow{\$} \mathbb{Z}_q$, and set $X \leftarrow g_2^x, Y \leftarrow g_2^y$. Initiate $\mathcal{L}_{\mathsf{JOINED}} \leftarrow \emptyset$.
- Prove that the key is well-formed in $\pi \xleftarrow{\$} SPK\{(x, y) : X = g_2^x \wedge Y = g_2^y\}$.
- Register the public key (X, Y, π) at $\mathcal{F}_{\mathsf{ca}}$, and store the secret key as (x, y).
- Output $(\mathsf{SETUPDONE}, sid)$.

Join. The join protocol runs between the issuer $\mathcal{I}$ and a platform, consisting of a TPM $\mathcal{M}_i$ and a host $\mathcal{H}_j$. The platform authenticates to the issuer and, if the issuer allows, obtains a credential that subsequently enables the platform to create signatures. To distinguish several join sessions that might run in parallel, we use a unique sub-session identifier $jsid$ that is given as input to all parties.

1. $\mathcal{H}_j$ upon input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ parses $sid = (\mathcal{I}, sid')$ and sends the message $(\mathsf{JOIN}, sid, jsid)$ to $\mathcal{I}$.
2. $\mathcal{I}$ upon receiving $(\mathsf{JOIN}, sid, jsid)$ from a party $\mathcal{H}_j$ chooses a fresh nonce $n \xleftarrow{\$} \{0,1\}^\tau$ and sends $(sid, jsid, n)$ back to $\mathcal{H}_j$.

3. $\mathcal{H}_j$ upon receiving $(sid, jsid, n)$ from $\mathcal{I}$, sends $(sid, jsid, n)$ to $\mathcal{M}_i$.
4. $\mathcal{M}_i$ upon receiving $(sid, jsid, n)$ from $\mathcal{H}_j$, generates its secret key:
- Check that no *completed* key record exists.
- Choose $gsk \xleftarrow{\$} \mathbb{Z}_q$ and store the key as $(sid, jsid, \mathcal{H}_j, gsk, \perp)$.
- Set $Q \leftarrow g_1^{gsk}$ and compute $\pi_1 \xleftarrow{\$} SPK\{(gsk) : Q = g_1^{gsk}\}(n)$.
- Send (Q, π_1) via the host to $\mathcal{I}$ using $\mathcal{F}_{\mathsf{auth}*}$, i.e., invoke $\mathcal{F}_{\mathsf{auth}*}$ on input $(\mathsf{SEND}, (\mathcal{M}_i, \mathcal{I}, sid), jsid, (Q, \pi_1), \mathcal{H}_j)$.
5. $\mathcal{H}_j$ upon receiving $(\mathsf{APPEND}, (\mathcal{M}_i, \mathcal{I}, sid), jsid, Q, \pi_1)$ from $\mathcal{F}_{\mathsf{auth}*}$, forwards the message to $\mathcal{I}$ by sending $(\mathsf{APPEND}, (\mathcal{M}_i, \mathcal{I}, sid), jsid, \mathcal{H}_j)$ to $\mathcal{F}_{\mathsf{auth}*}$. It also keeps state as $(sid, jsid, Q)$.
6. $\mathcal{I}$ upon receiving $(\mathsf{SENT}, (\mathcal{M}_i, \mathcal{I}, sid), jsid, (Q, \pi_1), \mathcal{H}_j)$ from $\mathcal{F}_{\mathsf{auth}*}$ verifies π_1 and checks that $\mathcal{M}_i \notin \mathcal{L}_{\mathsf{JOINED}}$. It stores $(sid, jsid, Q, \mathcal{M}_i, \mathcal{H}_j)$ and outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$.

The join session is completed when the issuer receives an explicit input telling him to proceed with join session $jsid$.

1. $\mathcal{I}$ upon input $(\mathsf{JOINPROCEED}, sid, jsid)$ generates the CL credential:
- Retrieve the record $(sid, jsid, Q, \mathcal{M}_i, \mathcal{H}_j)$ and add $\mathcal{M}_i$ to $\mathcal{L}_{\mathsf{JOINED}}$.
- Choose $r \xleftarrow{\$} \mathbb{Z}_q$ and compute $a \leftarrow g_1^r$, $b \leftarrow a^y$, $c \leftarrow a^x \cdot Q^{rxy}$, $d \leftarrow Q^{ry}$.
- Prove correctness of the signature in $\pi_2 \xleftarrow{\$} SPK\{(t) : b = g_1^t \wedge d = Q^t\}$.
- Send the credential (a, b, c, d) to the host $\mathcal{H}_j$ by giving $\mathcal{F}_{\mathsf{auth}*}$ input $(\mathsf{SEND}, (\mathcal{I}, \mathcal{M}_i, sid), jsid, (b, d, \pi_2), (a, c), \mathcal{H}_j)$.
2. $\mathcal{H}_j$ upon receiving $(\mathsf{APPEND}, (\mathcal{I}, \mathcal{M}_i, sid), jsid, (b, d, \pi_2), (a, c))$ from $\mathcal{F}_{\mathsf{auth}*}$ verifies the credential (a, b, c, d) and forwards (b, d, π_2) to $\mathcal{M}_i$:
- Retrieve $(sid, jsid, Q)$ and verify π_2 w.r.t. Q.
- Verify the credential as $a \neq 1$, $e(a, Y) = e(b, g_2)$, and $e(c, g_2) = e(a \cdot d, X)$.
- Send $(\mathsf{APPEND}, (\mathcal{I}, \mathcal{M}_i, sid), jsid, \perp)$ to $\mathcal{F}_{\mathsf{auth}*}$.
3. $\mathcal{M}_i$ upon receiving $(\mathsf{SENT}, (\mathcal{I}, \mathcal{M}_i, sid), jsid, (b, d, \pi_2), \perp)$ from $\mathcal{F}_{\mathsf{auth}*}$, completes the join:
- Retrieve the record $(sid, jsid, \mathcal{H}_j, gsk, \perp)$ and verify π_2 with respect to $Q \leftarrow g_1^{gsk}$.
- Complete the record to $(sid, jsid, \mathcal{H}_j, gsk, (b, d))$ and send $(sid, jsid, \mathsf{JOINED})$ to $\mathcal{H}_j$.
4. $\mathcal{H}_j$ upon receiving $(sid, jsid, \mathsf{JOINED})$ from $\mathcal{M}_i$ stores $(sid, \mathcal{M}_i, (a, b, c, d))$ and outputs $(\mathsf{JOINED}, sid, jsid)$.

Sign. The sign protocol runs between a TPM $\mathcal{M}_i$ and a host $\mathcal{H}_j$. After joining, together they can sign a message m with respect to a basename $\mathtt{bsn}$. Again, we use a unique sub-session identifier $ssid$ to allow for multiple sign sessions.

1. $\mathcal{H}_j$ upon input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ re-randomizes the CL-credential:
- Retrieve the join record $(sid, \mathcal{M}_i, (a, b, c, d))$.
- Choose $r \xleftarrow{\$} \mathbb{Z}_q$ and set $(a', b', c', d') \leftarrow (a^r, b^r, c^r, d^r)$.

TPM gsk, b, d		**Host** $(a, b, c, d), m, \mathtt{bsn}$
		$r \xleftarrow{\$} \mathbb{Z}_q$ $a' \leftarrow a^r, b' \leftarrow b^r, c' \leftarrow c^r, d' \leftarrow d^r$
	$\xleftarrow{m, \mathtt{bsn}, r}$	
$b' \leftarrow b^r, d' \leftarrow d^r$ if $\mathtt{bsn} \neq \perp$: $\mathtt{nym} \leftarrow H_1(\mathtt{bsn})^{gsk}$ $\pi \xleftarrow{\$} SPK\{(gsk) : d' = b'^{gsk} \wedge \mathtt{nym} = H_1(\mathtt{bsn})^{gsk}\}(\mathtt{bsn}, m)$		
	$\xrightarrow{\pi, \mathtt{nym}}$	
		output $\sigma \leftarrow (a', b', c', d', \mathtt{nym}, \pi)$

Fig. 5. Overview of the sign protocol

- Send $(sid, ssid, m, \mathtt{bsn}, r)$ to $\mathcal{M}_i$ and store $(sid, ssid, (a', b', c', d'))$

2. $\mathcal{M}_i$ upon receiving $(sid, ssid, m, \mathtt{bsn}, r)$ from $\mathcal{H}_j$ asks for permission to proceed.

- Check that a complete join record $(sid, \mathcal{H}_j, gsk, (b, d))$ exists.
- Store $(sid, ssid, m, \mathtt{bsn}, r)$ and output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$.

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

1. $\mathcal{M}_i$ upon input $(\mathsf{SIGNPROCEED}, sid, ssid)$ computes the SPK and $\mathtt{nym}$:

- Retrieve records $(sid, *, \mathcal{H}_j, gsk, (b, d))$ and $(sid, ssid, m, \mathtt{bsn}, r)$.
- Compute $b' \leftarrow b^r, d' \leftarrow d^r$.
- If $\mathtt{bsn} = \perp$, set $\mathtt{nym} = \perp$ and compute $\pi \xleftarrow{\$} SPK\{(gsk) : d' = b'^{gsk}\}(m, \mathtt{bsn})$.
- If $\mathtt{bsn} \neq \perp$, set $\mathtt{nym} = H_1(\mathtt{bsn})^{gsk}$ and compute the SPK on (m,$\mathtt{bsn}$) as $\pi \xleftarrow{\$} SPK\{(gsk) : \mathtt{nym} = H_1(\mathtt{bsn})^{gsk} \wedge d' = b'^{gsk}\}(m, \mathtt{bsn})$.
- Send $(sid, ssid, \pi, \mathtt{nym})$ to $\mathcal{H}_j$.

2. $\mathcal{H}_j$ upon receiving $(sid, ssid, \pi, \mathtt{nym})$ from $\mathcal{H}_j$, retrieves $(sid, ssid, (a', b', c', d'))$ and outputs $(\mathsf{SIGNATURE}, sid, ssid, (a', b', c', d', \pi, \mathtt{nym}))$.

Verify. The verify algorithm allows anyone to check whether a signature σ on message m with respect to basename $\mathtt{bsn}$ is valid, i.e., stems from a certified TPM. To test whether the signature originates from a TPM that did get corrupted, the verifier can pass a revocation list $\mathtt{RL}$ to the algorithm. This list contains the keys of corrupted TPMs he no longer wishes to accept signatures from.

1. $\mathcal{V}$ upon input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ verifies the signature:

- Parse σ as $(a, b, c, d, \pi, \mathtt{nym})$.
- Verify π with respect to $(m, \mathtt{bsn})$ and $\mathtt{nym}$ (if $\mathtt{bsn} \neq \perp$).
- Check that $a \neq 1$, $e(a, Y) = e(b, g_2)$, and $e(c, g_2) = e(a \cdot d, X)$.
- For every $gsk_i \in \mathtt{RL}$, check that $b^{gsk_i} \neq d$.
- If all tests pass, set $f \leftarrow 1$, otherwise $f \leftarrow 0$.
- Output $(\mathsf{VERIFIED}, sid, f)$.

Link. With the link algorithm, anyone can test whether two signatures (σ, m), (σ', m') that were generated for the same basename $\texttt{bsn} \neq \bot$, stem from the same TPM.

1. $\mathcal{V}$ upon input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \texttt{bsn})$ verifies the signatures and compares the pseudonyms contained in σ, σ':

- Check that $\texttt{bsn} \neq \bot$ and that both signatures σ, σ' are valid.
- Output $\bot$ if any check failed.
- Parse the signatures as $(a, b, c, d, \pi, \texttt{nym}) \leftarrow \sigma$, $(a', b', c', d', \pi', \texttt{nym}') \leftarrow \sigma'$.
- If $\texttt{nym} = \texttt{nym}'$, set $f \leftarrow 1$, otherwise $f \leftarrow 0$.
- Output (LINK, sid, f).

5.2 Differences with Previous Schemes

Our proposed protocol is very similar to previous CL-based DAA constructions [3, 6, 7, 16, 18] which, however, do not immediately satisfy our security notion. For each part of the protocol we now discuss the weaknesses of these previous schemes and the way our solution overcomes them.

Setup. In our scheme, the issuer is required to prove knowledge of his secret key (x, y) in the proof π and to make π part of his public key. This proof allows the simulator in the security proof to extract the issuers secret key. Such an extraction capability is crucial for the security proof and was missing in the existing schemes, even though some of the corresponding security models implicitly assumed the extractability of the issuers key (as discussed in Section 2.2).

Join. In the join protocol, we reintroduced a proof π_1 of gsk by the TPM, that was present in many previous works but omitted in the scheme by Bernard et al. [3]. Additionally, our scheme contains the proof π_2 by the issuer, which was introduced by Bernard et al.

Many previous schemes [6, 7, 16, 18] let the TPM prove knowledge of the discrete log of $Q = g^{gsk}$ in the join protocol. Bernard et al. removed this proof by reducing the forgery of a join credential to the security of a blind signature scheme, and using a unforgeability notion that requires the adversary to output all secret keys. This assumes that all these secrets are extractable which, if extraction by rewinding is used, would require exponential time. We realize efficient extraction by adding the SPK π_1 in which the TPM proves of knowledge of gsk to the join protocol and allowing only a logarithmic number of simultaneous join sessions.

The second proof $\pi_2 \overset{\$}{\leftarrow} SPK\{(t) : b = g_1^t \wedge d = Q^t\}$ in our join protocol was introduced by Bernard et al. and lets the issuer prove correctness of his credential computations, which none of the previous works did. We also use this proof as it allows to simulate a TPM without knowing the secret key gsk. This is required in our reduction where we use the unknown discrete logarithm of a DL or DDH instance as the key of a TPM.

Sign. We change the communication between the TPM and host to prevent the TPM from leaking information about its secret key gsk to the host, and we only use pseudonyms when required.

Chen, Page, and Smart [18] let the host send a randomized b value of the credential to the TPM, which responded with $d = b^{gsk}$. This gives information to the host that cannot be simulated without knowing gsk, which prevents a proof of unforgeability under the DL assumption, and requires the stronger static DH assumption. The scheme by Bernard et al. [3] has a similar problem: The host sends (b, d) to the TPM, and the TPM responds with a proof proving that $b^{gsk} = d$. Now the TPM should only output a valid proof for valid inputs, i.e, when $b^{gsk} = d$. A simulator mimicking a TPM in the security proof, however, cannot decide this when reducing to the DL problem, a stronger assumption is required to prove unforgeability in their scheme.

We apply the fix by Xi et al. [25], in which the host sends the randomness r used to randomize the credential. This does not give the host any new information on gsk, which is why we can prove unforgeability under the DL assumption.

Some schemes [6, 7, 16, 18] always attached a pseudonym to signatures to support revocation, even when the basename $\mathtt{bsn}$ was equal to $\perp$. However, we can perform the revocation check on the credential: $b^{gsk} \overset{?}{=} d$, so the pseudonym can be omitted when $\mathtt{bsn} = \perp$ for a more efficient scheme.

Verify. We add a check $a \neq 1_{\mathbb{G}_1}$ to the verification algorithm, which many of the previous schemes [6, 7, 16, 18] are lacking. Without this check, schemes tolerate a trivial issuer credential $(1_{\mathbb{G}_1}, 1_{\mathbb{G}_1}, 1_{\mathbb{G}_1}, 1_{\mathbb{G}_1})$ that allows anyone to create valid DAA signatures, which clearly breaks unforgeability. Note that [18] has been ISO standardized [20] with this flaw.

The verification algorithm also checks $b \neq 1_{\mathbb{G}_1}$, which is not present in any of the previous schemes. A credential with $b = 1_{\mathbb{G}_1}$ leads to $d = 1_{\mathbb{G}_1}$, and lets any gsk match the credential, which is undesirable as we no longer have a unique matching gsk. An adversarial issuer can create such credentials by choosing its secret key $y = 0$. This case is "excluded" by the non-frameability property of Bernard et al. [3] which assumes that even a corrupt issuer creates his keys honestly, so $y = 0$ will occur with negligible probability only. We avoid such an assumption and simply add the check $b \neq 1_{\mathbb{G}_1}$.

6 Security Proof Sketch

Theorem 1. *The protocol Π_{daa} presented in Section 5 securely realizes $\mathcal{F}^{l}_{\mathsf{daa}}$ in the $(\mathcal{F}_{\mathsf{auth*}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}^{l}_{\mathsf{smt}}, \mathcal{F}^{D}_{\mathsf{crs}})$-hybrid model using random oracles and static corruptions, if the DL and DDH assumptions hold, the CL signature [9] is unforgeable, and the proofs-of-knowledge are online extractable.*

As CL signatures are unforgeable under the LRSW assumption [21], and we can instantiate the SPKs to be online extractable under the DCR assumption [22], we obtain the following corollary:

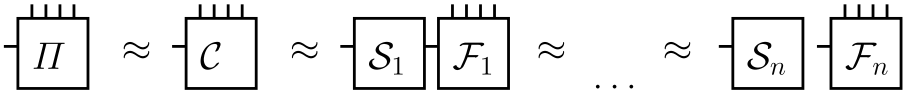

Fig. 6. Visualization of the proof strategy

Corollary 1. *The protocol Π_{daa} presented in Section 5 instantiated with online extractable proofs securely realizes $\mathcal{F}^l_{\mathsf{daa}}$ in the $(\mathcal{F}_{\mathsf{auth}*}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}^l_{\mathsf{smt}}, \mathcal{F}^D_{\mathsf{crs}})$-hybrid model using random oracles and static corruptions under the DL, DDH, LRSW, and DCR assumptions.*

Instead of relying on *online extractable* SPKs one could also use extraction by rewinding, which would yield a more efficient scheme. However, one needs to take special care that the rewinding does not require exponential time in the security proof. The only SPK we constantly have to extract from in our security proof is π_1 used in the join protocol. Thus, we can avoid the exponential blow-up by letting the issuer limit the number of simultaneous join sessions to be logarithmic in the security parameter. Since we keep the way in which the simulator extracts witnesses abstract in the proof of Theorem 1, the very same simulator proves the scheme with extraction by rewinding secure. Note though, that the UC framework does not allow rewinding at all, i.e., this only proves the instantiation using extraction by rewinding secure in a stand-alone fashion, but one cannot claim composability guarantees.

To show that no environment $\mathcal{E}$ can distinguish the real world, in which it is working with Π_{daa} and adversary $\mathcal{A}$, from the ideal world, in which it uses $\mathcal{F}^l_{\mathsf{daa}}$ with simulator $\mathcal{S}$, we use a sequence of games. We start with the real world protocol execution. In the next game we construct one entity $\mathcal{C}$ that runs the real world protocol for all honest parties. Then we split $\mathcal{C}$ into two pieces, a functionality $\mathcal{F}$ and a simulator $\mathcal{S}$, where $\mathcal{F}$ receives all inputs from honest parties and sends the outputs to honest parties. We start with a useless functionality, and gradually change $\mathcal{F}$ and update $\mathcal{S}$ accordingly, to end up with the full $\mathcal{F}^l_{\mathsf{daa}}$ and a satisfying simulator. This strategy is depicted in Figure 6.

Due to space constraints, we present the complete security proof including all intermediate functionalities and simulators in Appendix B. Here an overview of the game hops is given, along with an explanation how we can show indistinguishability between the games.

Game 1: This is the real world protocol.

Game 2: The challenger $\mathcal{C}$ now receives all inputs and simulates the real world protocol for honest parties. Since $\mathcal{C}$ gets all inputs, it can simply run the real world protocol. It also simulates all hybrid functionalities, but does so honestly, so $\mathcal{E}$ does not see any difference. By construction, this is equivalent to the previous game.

Game 3: We now split $\mathcal{C}$ into a "dummy functionality" $\mathcal{F}$ and simulator $\mathcal{S}$. $\mathcal{F}$ receives all inputs, and simply forwards them to $\mathcal{S}$. $\mathcal{S}$ simulates the real world protocol and sends the outputs it generates to $\mathcal{F}$, who then outputs them to $\mathcal{E}$. This game only restructures the previous game.

Game 4: In this game we let our intermediate $\mathcal{F}$ handle the setup related interfaces using the procedure specified in $\mathcal{F}^l_{\mathsf{daa}}$. Consequently, $\mathcal{F}$ expects to receive the algorithms $(\mathsf{ukgen}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify})$ from the simulator. For $\mathsf{ukgen}, \mathsf{ver}, \mathsf{link}$ and $\mathsf{identify}$, $\mathcal{S}$ can simply provide the algorithms from the real-world protocol, where it omits the revocation check from ver. The sig algorithm, though, must contain the issuer's private key, so $\mathcal{S}$ has to be able to get that value.

When $\mathcal{I}$ is honest, $\mathcal{S}$ will receive a message from $\mathcal{F}$ asking for the algorithms, which informs $\mathcal{S}$ what is happening and allows him to start simulating the issuer. Since $\mathcal{S}$ is running the issuer, it knows its secret key and sets the sig algorithm accordingly.

When $\mathcal{I}$ is corrupt, $\mathcal{S}$ starts the simulation when the issuer registers his key with $\mathcal{F}_{\mathsf{ca}}$ that is controlled by the simulator. Since the public key includes a proof of knowledge of the issuer's secret key, $\mathcal{S}$ can extract the secret key from there and define the sig algorithm accordingly. By the simulation soundness of the SPK, this game hop is indistinguishable for the adversary.

Game 5: $\mathcal{F}$ now handles the verify and link queries using the provided algorithms ver and link from the previous game, rather than forwarding the queries to $\mathcal{S}$. We do not let $\mathcal{F}$ perform the additional checks (Check **(ix)** - Check **(xvi)**) done by $\mathcal{F}^l_{\mathsf{daa}}$, though, but add these only later. For Check **(xii)**, $\mathcal{F}$ rejects a signature when a matching $gsk' \in \mathtt{RL}$ is found, but does not exclude honest TPMs from this check yet.

Because verify and link do not involve network traffic, the simulator does not have to simulate traffic either, we must only make sure the outputs do not change. $\mathcal{F}$ executes the algorithms that $\mathcal{S}$ supplied, and $\mathcal{S}$ supplied them in such a way that they are equivalent to the real world algorithms, so the outcome will clearly be equivalent.

Game 6: In this step we change $\mathcal{F}$ to also handle the join-related interfaces, meaning it will receive the inputs and generate the outputs. We let $\mathcal{F}$ run the same procedure as $\mathcal{F}^l_{\mathsf{daa}}$, but again omit the additional checks (Check **(ii)**-Check **(iv)**).

We have to ensure that $\mathcal{F}$ outputs the same values as the real world did. As the join interfaces do not output crypto values, but only messages like start and complete, we simply have to guarantee that whenever the real world protocol would reach a certain output, the functionality also allows that output, and vice versa.

For the direction from the real world to the functionality this is clearly given, since $\mathcal{F}$ does not perform additional checks and thus will always proceed for any input it receives from $\mathcal{S}$. For all outputs triggered by $\mathcal{F}$, the simulator has to give explicit approval which allows $\mathcal{S}$ to block any output by $\mathcal{F}$ if the real world protocol would not proceed at a certain point.

If the host and/or TPM are honest, the simulator knows the identities $\mathcal{M}_i, \mathcal{H}_j$ and correctly uses them towards $\mathcal{F}$ as well as in the simulation. If both, the

TPM and host are corrupt but the issuer is honest, then $\mathcal{S}$ cannot determine the identity of the host, as the host does not authenticate towards the issuer in the real world. This does not impact the real world simulation of the issuer, but the simulator has to choose an arbitrary corrupt host $\mathcal{H}_j$ now when invoking $\mathcal{F}$ with the JOIN call. This will only result in a different host being stored in the $\mathtt{Members}$ list in $\mathcal{F}$, but $\mathcal{F}$ never uses this identity when the corresponding TPM is corrupt.

In the final join interface $\mathsf{JOINCOMPLETE}$, the simulator has to provide the secret key gsk of the TPM. When the TPM is honest, $\mathcal{S}$ knows the key anyway and if the TPM is corrupt, $\mathcal{S}$ extracts the key from the proof π_1. Note that $\mathcal{F}$ sets $gsk \leftarrow \bot$ when both the TPM and host are honest. However, this has no impact yet, as the signatures are still created by the simulator and the verify and link interfaces of $\mathcal{F}$ do not run the additional checks that make use of the internally stored records and keys. Overall, this game hop is indistinguishable by the simulation soundness of the SPK π_1.

Game 7, 8, 9, 10: Over the next four game hops we transform $\mathcal{F}$ such that it internally handles the signing queries instead of merely forwarding them to $\mathcal{S}$. Again, $\mathcal{F}$ uses the sign interfaces from $\mathcal{F}^l_{\mathsf{daa}}$, with the difference that it does not perform the Check **(v)**-Check **(viii)** which we only add in a later game. As argued in the previous game, the procedures are defined such that $\mathcal{S}$ has to approve every output by $\mathcal{F}$, which allows it to block any output that would not happen in the real world protocol.

When the TPM or the host is corrupt, $\mathcal{S}$ has to provide the signature value, which it takes from the real world simulation, and thus perfectly mimics the real world output. When both the TPM and the host are honest, $\mathcal{F}$ creates the signatures internally in an unlinkable way: It chooses a new gsk per basename and TPM, or per signature when $\mathtt{bsn} = \bot$ and then runs the sig algorithm for that fresh key. $\mathcal{F}$ keeps the internally chosen keys $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle$ in a list $\mathtt{DomainKeys}$ to ensure consistency if a TPM wishes to reuse the basename.

This change is indistinguishable under the DDH assumption: Suppose an environment can distinguish a signature by an honest party with the gsk it joined with from a signature by the same party but with a different gsk. Then we can use that environment to break a DDH instance α, β, γ by simulating the join and the first signature using the unknown $log_{g_1}(\alpha)$ as gsk, and for the second signature we use the unknown $log_\beta(\gamma)$ as gsk. If the environment notices a difference, we know that $log_{g_1}(\alpha) \neq log_\beta(\gamma)$, solving the DDH problem.

In the reduction we have to be able to simulate the TPM without knowing gsk, but merely based on $\alpha = g_1^{gsk}$. A TPM uses gsk to set $Q \leftarrow g_1^{gsk}$ in the join protocol, to do proofs π_1 in joining and π in signing, and to compute pseudonyms. In simulation, we set $Q \leftarrow \alpha$ and we simulate all proofs π_1 and π. For pseudonyms, the power over the random oracle is used: $\mathcal{S}$ chooses $H_1(\mathtt{bsn}) = g_1^r$ for $r \xleftarrow{\$} \mathbb{Z}_q$, and sets $\mathtt{nym} \leftarrow \alpha^r = H_1(\mathtt{bsn})^{gsk}$ without knowing gsk. Note that the proof π_2 the issuer makes in the join protocol is crucial for our simulation as well, since the TPM does not have to use gsk to check $b^{gsk} \overset{?}{=} d$, it can simply verify π_2.

Game 11: In this game we let $\mathcal{F}$ additionally check the validity of every new *gsk* that is generated or received in the join and sign interface.

If the TPM is corrupt, $\mathcal{F}$ checks that $\mathsf{CheckGskCorrupt}(gsk) = 1$ for the *gsk* that the simulator extracted from π_1 (Check **(iv)**). This check prevents the adversary from choosing different keys $gsk \neq gsk'$ that both fit to the same signature. In our protocol there exists only a single *gsk* for every valid signature where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk) = 1$, and thus this check will never fail.

For keys of honest TPMs, $\mathcal{F}$ verifies that $\mathsf{CheckGskHonest}(gsk) = 1$ whenever it receives or generates a new *gsk* (Check **(iii)** and Check **(v)**). With these checks we avoid the registration of keys for which matching signatures already exist. Since keys for honest TPMs are chosen uniformly at random from an exponentially large group and every signature has exactly one matching key, the chance that a signature under that key already exists is negligible.

Game 12: We now add the checks to $\mathcal{F}$ that $\mathcal{F}_{\mathsf{daa}}^{l}$ runs in the sign interfaces when internally generating signatures for honest platforms. After creating a signature, $\mathcal{F}$ checks whether the signature verifies and matches the right key (Check **(vi)** and Check **(vii)**). As $\mathcal{S}$ supplied proper algorithms and the signature scheme is complete, these checks will obviously always succeed.

$\mathcal{F}$ also checks with the help of its internal key records `Members` and `DomainKeys` that no one else already has a key which would match this newly generated signature (Check **(viii)**). If this fails, we can solve the DL problem: We simulate a TPM using the unknown discrete logarithm of the DL instance as *gsk* like in the DDH reduction before. If a matching *gsk* is found, then we have a solution to the DL problem.

Game 13, 14, 15, 16: In these four game hops, we let $\mathcal{F}$ perform the four additional checks that are done by $\mathcal{F}_{\mathsf{daa}}^{l}$ in the verification interface and show that this does not change the verification outcome.

The first check prevents multiple *gsk* values matching one signature, but as $\mathsf{identify}$ considers the discrete log relation between b and d from the credential, and $b \neq 1$, there exists only one $gsk \in \mathbb{Z}_q$ such that $b^{gsk} = d$ (Check **(ix)**).

If the issuer is honest, the second check prevents signing with join credentials that were not issued by the issuer (Check **(x)**). We can reduce this to the unforgeability of the CL signature. The signing oracle is now used to create credentials, and when a credential is verified that was not signed by the issuer, it must be a forgery.

$\mathcal{F}$ prevents signatures that use the key and credential of an honest TPM, but are for messages that this TPM never signed (Check **(xi)**). We can reduce the occurrence of such a signature to the DL problem. Again we simulate a TPM using the unknown discrete logarithm of the problem instance. When a signature is verified for a message that the TPM never signed, we know that the proof π is not simulated, so we can extract *gsk* from it, breaking the DL assumption.

The last check prevents the revocation of honest TPMs (Check **(xii)**), which we can reduce to the DL problem as well. We simulate the TPM using the DL instance, and if a matching key is placed on the revocation list, this must be the discrete logarithm of the problem instance.

Game 17: We now let $\mathcal{F}$ perform all the additional checks $\mathcal{F}^l_{\mathsf{daa}}$ makes for link queries. If it notices a key that matches one signature but not the other, $\mathcal{F}$ states the signatures are not linked. If it notices one key that matches both signatures, it outputs that the signatures are linked. The output of $\mathcal{F}$ based on these checks is still consistent with the output which the link algorithm would give: If there is a gsk that matches one signature but not the other, by soundness of π we have that the pseudonyms are not based on the same gsk. As $H_1(\mathtt{bsn})$ generates $\mathbb{G}_1$ with overwhelming probability, the pseudonyms differ and link would output 0. If there is a gsk that matches both signatures, by soundness of π we have that the pseudonyms are based on the same gsk and must be equal, resulting in link outputting 1.

Now $\mathcal{F}$ is equal to $\mathcal{F}^l_{\mathsf{daa}}$, concluding our proof sketch.

Acknowledgements. This work was supported by the European Commission through the Seventh Framework Programme, under grant agreements #321310 for the PERCY grant and #318424 for the project FutureID.

References

1. Backes, M., Hofheinz, D.: How to break and repair a universally composable signature functionality. Information Security 2004.
2. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. ACM CCS 1993.
3. Bernhard, D., Fuchsbauer, G., Ghadafi, E., Smart, N., Warinschi, B.: Anonymous attestation with user-controlled linkability. International Journal of Information Security 12(3), (2013)
4. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil pairing. ASIACRYPT 2001.
5. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. ACM CCS 2004.
6. Brickell, E., Chen, L., Li, J.: A new direct anonymous attestation scheme from bilinear maps. Trusted Computing - Challenges and Applications 2008.
7. Brickell, E., Chen, L., Li, J.: Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. International Journal of Information Security 8(5), (2009)
8. Camenisch, J., Kiayias, A., Yung, M.: On the portability of generalized schnorr proofs. EUROCRYPT 2009.
9. Camenisch, J., Lysyanskaya, A.: Signature schemes and anonymous credentials from bilinear maps. CRYPTO 2004.
10. Camenisch, J., Shoup, V.: Practical verifiable encryption and decryption of discrete logarithms. CRYPTO 2003.
11. Camenisch, J., Stadler, M.: Efficient group signature schemes for large groups (extended abstract). CRYPTO 1997.
12. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. ePrint Archive Report 2000/067.

13. Canetti, R.: Universally composable signatures, certification and authentication. ePrint Archive, Report 2003/239.
14. Chen, L., Morrissey, P., Smart, N.: DAA: Fixing the pairing based protocols. ePrint Archive, Report 2009/198.
15. Chen, L.: A DAA scheme requiring less tpm resources. Information Security and Cryptology 2010.
16. Chen, L., Morrissey, P., Smart, N.P.: Pairings in trusted computing (invited talk). PAIRING 2008.
17. Chen, L., Morrissey, P., Smart, N.: On proofs of security for DAA schemes. Provable Security 2008.
18. Chen, L., Page, D., Smart, N.: On the design and implementation of an efficient DAA scheme. Smart Card Research and Advanced Application 2010.
19. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. CRYPTO 1986.
20. International Organization for Standardization: ISO/IEC 20008-2: Information technology - Security techniques - Anonymous digital signatures - Part 2: Mechanisms using a group public key (2013)
21. Lysyanskaya, A., Rivest, R.L., Sahai, A., Wolf, S.: Pseudonym systems. SAC 1999.
22. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. EUROCRYPT 1999.
23. Trusted Computing Group: TPM main specification version 1.2 (2004)
24. Trusted Computing Group: Trusted platform module library specification, family "2.0" (2014)
25. Xi, L., Yang, K., Zhang, Z., Feng, D.: DAA-related APIs in TPM 2.0 revisited. Trust and Trustworthy Computing 2014.

A Auxiliary Ideal Functionalities

In this section, we formally define the ideal functionalities we use as subroutines in our protocol.

A.1 Certification Authority

We use the ideal certification authority functionality $\mathcal{F}_{\mathsf{ca}}$ as defined in [13].

1. Upon receiving the first message $(\mathsf{Register}, sid, v)$ from party P, send $(\mathsf{Registered}, sid, v)$ to the adversary; upon receiving ok from the adversary, and if $sid = P$ and this is the first request from P, then record the pair (P, v).
2. Upon receiving a message $(\mathsf{Retrieve}, sid)$ from party P', send $(\mathsf{Retrieve}, sid, P')$ to the adversary, and wait for an ok from the adversary. Then, if there is a recorded pair (sid, v) output $(\mathsf{Retrieve}, \mathsf{sid}, \mathsf{v})$ to P'. Else output $(\mathsf{Retrieve}, sid, \perp)$ to P'.

Fig. 7. Ideal certification authority functionality $\mathcal{F}_{\mathsf{ca}}$ from [13]

A.2 Secure Message Transmission

We further use the ideal secure message transmission functionality as defined in the 2013 version of [12]. This functionality is parametrized by a leakage function $l : \{0,1\}^* \to \{0,1\}^*$.

For our security proof, we require the leakage function l to fulfill the following property: $l(b) = l(b') \to l(a, b) = l(a, b')$ for all a, b, b'. This is a natural requirement, as most secure channels will at most leak the length of the plaintext, for which this property holds.

1. Upon receiving input $(\mathsf{Send}, S, R, sid, m)$ from S, send $(\mathsf{Sent}, S, R, sid, l(m))$ to the adversary, generate a private delayed output $(\mathsf{Sent}, S, sid, m)$ to R and halt.
2. Upon receiving $(\mathsf{Corrupt}, sid, P)$ from the adversary, where $P \in \{S, R\}$, disclose m to the adversary. Next, if the adversary provides a value m', and $P = S$, and no output has been yet written to R, then output $(\mathsf{Sent}, S, sid, m')$ to R and halt.

Fig. 8. Ideal secure message transmission functionality $\mathcal{F}^{l}_{\mathsf{smt}}$ from [12]

A.3 Common Reference String

For the crs functionality we use the 2005 version of [12]. This functionality is parametrized by a distribution D, from which the crs is sampled.

1. When receiving input (CRS, sid) from party P, first verify that $sid = (\mathcal{P}, sid')$ where $\mathcal{P}$ is a set of identities, and that $P \in \mathcal{P}$; else ignore the input. Next, if there is no value r recorded then choose and record $r \xleftarrow{\$} D$. Finally, send a public delayed output (CRS, sid, r) to P.

Fig. 9. Ideal crs functionality $\mathcal{F}_{\mathsf{crs}}^{D}$ from [12]

B Proof of Theorem 1

Proof. We have to prove that our scheme realizes $\mathcal{F}^{l}_{\mathsf{daa}}$, which means proving that for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every environment $\mathcal{E}$ we have $\mathrm{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} \approx \mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{E}}$. To prove this a sequence of games is used. First we define all intermediate functionalities and simulators, and then we prove that they are all indistinguishable from each other.

B.1 Functionalities and Simulators

Setup

1. On input (SETUP, sid) from $\mathcal{I}$.
– Output $(\mathsf{FORWARD}, (\mathsf{SETUP}, sid), \mathcal{I})$ to $\mathcal{S}$.

Join

1. On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
– Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
2. On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$
– Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid), \mathcal{I})$ to $\mathcal{S}$.

Sign

1. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.
– Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn}), \mathcal{H}_j)$ to $\mathcal{S}$.
2. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
– Output $\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

1. On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from $\mathcal{V}$,
– Output $(\mathsf{FORWARD}, (\mathsf{VERIFY}, sid, vsid, m, \mathtt{bsn}, \sigma, \mathtt{RL}), \mathcal{V})$ to $\mathcal{S}$.

Link

1. On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from $\mathcal{V}$.
– Output $(\mathsf{FORWARD}, (\mathsf{LINK}, sid, lsid, \sigma, m, \sigma', m', \mathtt{bsn}), \mathcal{V})$ to $\mathcal{S}$.

Output

1. On input $(\mathsf{OUTPUT}, \mathcal{P}, m)$ from $\mathcal{S}$.
– Output (m) to $\mathcal{P}$.

Fig. 10. $\mathcal{F}$ for Game 3

When any simulated party "$\mathcal{P}$" outputs a message m, $\mathcal{S}$ sends $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$.

KeyGen

– Upon receiving $(\mathsf{FORWARD}, (\mathsf{SETUP}, sid), \mathcal{I})$ from $\mathcal{F}$.

• Give "$\mathcal{I}$" input (SETUP, sid).

Join

– Upon receiving input $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j)$ from $\mathcal{F}$.

• Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$.

– Upon receiving input $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid), \mathcal{I})$ from $\mathcal{F}$.

• Give "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, sid, jsid)$.

Sign

– Upon receiving $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn}), \mathcal{H}_j)$ from $\mathcal{F}$.

• Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$.

– Upon receiving $\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ from $\mathcal{F}$.

• Give "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

Verify

– Upon receiving $(\mathsf{FORWARD}, (\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL}), \mathcal{V})$ from $\mathcal{F}$.

• Give "$\mathcal{V}$" input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$.

Link

– Upon receiving $(\mathsf{FORWARD}, (\mathsf{LINK}, sid, lsid, \sigma, m, \sigma', m', \mathtt{bsn}), \mathcal{V})$ from $\mathcal{F}$.

• Give "$\mathcal{V}$" input $(\mathsf{LINK}, sid, lsid, \sigma, m, \sigma', m', \mathtt{bsn})$.

Fig. 11. Simulator Game 3

Setup

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

Join

1. On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
– Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
2. On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$
– Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid), \mathcal{I})$ to $\mathcal{S}$.

Sign

1. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.
– Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn}), \mathcal{H}_j)$ to $\mathcal{S}$.
2. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
– Output $\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

1. On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from $\mathcal{V}$,
– Output $(\mathsf{FORWARD}, (\mathsf{VERIFY}, sid, vsid, m, \mathtt{bsn}, \sigma, \mathtt{RL}), \mathcal{V})$ to $\mathcal{S}$.

Link

1. On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from $\mathcal{V}$.
– Output $(\mathsf{FORWARD}, (\mathsf{LINK}, sid, lsid, \sigma, m, \sigma', m', \mathtt{bsn}), \mathcal{V})$ to $\mathcal{S}$.

Output

1. On input $(\mathsf{OUTPUT}, \mathcal{P}, m)$ from $\mathcal{S}$.
– Output (m) to $\mathcal{P}$.

Fig. 12. $\mathcal{F}$ for GAME 4

When any simulated party "$\mathcal{P}$" outputs a message m that is not explicitly handled by $\mathcal{S}$ yet, $\mathcal{S}$ sends $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$.

KeyGen

Honest $\mathcal{I}$

– On input (SETUP, sid) from $\mathcal{F}$.
- Try to parse sid as $\mathcal{I}, sid'$, output $\perp$ to $\mathcal{I}$ if that fails.
- Give "$\mathcal{I}$" input (SETUP, sid).
- When "$\mathcal{I}$" outputs $(\mathsf{SETUPDONE}, sid)$, $\mathcal{S}$ takes its private key x, y.
- Define $\mathsf{sig}(gsk, m, \mathtt{bsn})$ as follows: compute $(a, b, c, d) \leftarrow CL.Sig(x, y; gsk)$, if $\mathtt{bsn} \neq \perp$, $\mathtt{nym} = H_1(\mathtt{bsn})^{gsk}$ and $\pi \leftarrow SPK\{(gsk) : d = b^{gsk} \wedge \mathtt{nym} = H_1(\mathtt{bsn})^{gsk}\}(m, \mathtt{bsn})$, if $\mathtt{bsn} = \perp$, $\mathtt{nym} = \perp$ and $\pi \leftarrow SPK\{(gsk) : d = b^{gsk}\}(m, \mathtt{bsn})$, and output $(a, b, c, d, \pi, \mathtt{nym})$.
- Define $\mathsf{ver}(\sigma, m, \mathtt{bsn})$ as follows: parse σ as $(a, b, c, d, \pi, \mathtt{nym})$ and check whether π is valid on $(m, \mathtt{bsn})$, $a \neq 1_{\mathbb{G}_1}$, $b \neq 1_{\mathbb{G}_1}$, $e(a, Y) = e(b, g_2)$, $e(c, g_2) = e(a \cdot d, X)$. If so output 1, otherwise 0.
- Define $\mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$ as follows: parse the signatures as $(a, b, c, d, \pi, \mathtt{nym}) \leftarrow \sigma$, $(a', b', c', d', \pi', \mathtt{nym}') \leftarrow \sigma'$. If $\mathtt{nym} = \mathtt{nym}'$, output 1, otherwise 0.
- Define $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk)$ as follows: parse σ as $(a, b, c, d, \pi, \mathtt{nym})$ and check $gsk \in \mathbb{Z}_q$, $\mathsf{ver}(\sigma, m, \mathtt{bsn}) = 1$ and $d = b^{gsk}$. If so, output 1, otherwise 0.
- Define ukgen as follows: take $gsk \in_R \mathbb{Z}_q$ and output gsk.
- $\mathcal{S}$ sends $(\mathsf{KEYS}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.

Corrupt $\mathcal{I}$

– $\mathcal{S}$ notices this setup as it notices $\mathcal{I}$ registering a public key with "$\mathcal{F}_{\mathsf{ca}}$" with $sid = (\mathcal{I}, sid')$.
- If the registered key is of the form X, Y, π and π is valid, $\mathcal{S}$ extracts x, y from π.
- $\mathcal{S}$ defines the algorithms $\mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen}$ as before, but now depending on the extracted key.
- $\mathcal{S}$ sends (SETUP, sid) to $\mathcal{F}$ on behalf of $\mathcal{I}$.

– On input (KEYGEN, sid) from $\mathcal{F}$.
- $\mathcal{S}$ sends $(\mathsf{KEYS}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.

– On input $(\mathsf{SETUPDONE}, sid)$ from $\mathcal{F}$
- $\mathcal{S}$ continues simulating "$\mathcal{I}$".

Join

Unchanged.

Sign

Unchanged.

Verify

Unchanged.

Link

Unchanged.

Fig. 13. Simulator Game 4

Setup

1. `Issuer Setup.` On input (SETUP, sid) from issuer $\mathcal{I}$

– Verify that $sid = (\mathcal{I}, sid')$ and output (SETUP, sid) to $\mathcal{S}$.

2. `Set Algorithms.` On input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{S}$

– Check that ver, link and $\mathsf{identify}$ are deterministic.

– Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

Join

1. On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.

– Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.

2. On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$

– Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid), \mathcal{I})$ to $\mathcal{S}$.

Sign

1. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.

– Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn}), \mathcal{H}_j)$ to $\mathcal{S}$.

2. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.

– Output $\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.

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

• There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.

– If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.

– Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f \rangle$ to `VerResults`, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \bot$.

– Output $\bot$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).

– Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.

– Output (LINK, sid, f) to $\mathcal{V}$.

Output

1. On input $(\mathsf{OUTPUT}, \mathcal{P}, m)$ from $\mathcal{S}$.

– Output (m) to $\mathcal{P}$.

Fig. 14. $\mathcal{F}$ for Game 5

When any simulated party "$\mathcal{P}$" outputs a message m that is not explicitly handled by $\mathcal{S}$ yet, $\mathcal{S}$ sends $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$.

KeyGen

Unchanged.

Join

Unchanged.

Sign

Unchanged.

Verify

Nothing to simulate.

Link

Nothing to simulate.

Fig. 15. Simulator Game 5

Setup

1. `Issuer Setup.` On input (SETUP, sid) from issuer $\mathcal{I}$
- Verify that $sid = (\mathcal{I}, sid')$ and output (SETUP, sid) to $\mathcal{S}$.

2. `Set Algorithms.` On input $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{S}$
- Check that ver, link and $\mathsf{identify}$ are deterministic.
- Store $(sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, sid)$ to $\mathcal{I}$.

Join

1. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$
- Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status \leftarrow request$.
- Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.

2. `Join Request Delivery.` On input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{S}$
- Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ to $status \leftarrow delivered$.
- Abort if $\mathcal{I}$ or $\mathcal{M}_i$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
- Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.

3. `Complete Join.` On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$
- Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ to $status \leftarrow complete$.
- Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{S}$.

4. `Key Generation.` On input $(\mathsf{JOINCOMPLETE}, sid, jsid, gsk)$ from $\mathcal{S}$.
- Look up record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = complete$.
- If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, set $gsk \leftarrow \perp$.
- Insert $\langle \mathcal{M}_i, \mathcal{H}_j, gsk\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid)$ to $\mathcal{H}_j$.

Sign

1. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.
- Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn}), \mathcal{H}_j)$ to $\mathcal{S}$.

2. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
- Output $\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
- Set $f \leftarrow 0$ if at least one of the following conditions hold:
- • There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.
- If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.
- Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f\rangle$ to $\mathtt{VerResults}$, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \perp$.
- Output $\perp$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
- Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.
- Output (LINK, sid, f) to $\mathcal{V}$.

Output

1. On input $(\mathsf{OUTPUT}, \mathcal{P}, m)$ from $\mathcal{S}$.
- Output (m) to $\mathcal{P}$.

Fig. 16. $\mathcal{F}$ for GAME 6

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

KeyGen

Unchanged.

Join

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

– $\mathcal{S}$ receives $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
- It simulates the real world protocol by giving “$\mathcal{H}_j$” input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ and waits for output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ from “$\mathcal{I}$”.
- If $\mathcal{M}_i$ is corrupt, $\mathcal{S}$ extracts gsk from proof π_1 stores it. If $\mathcal{M}_i$ is honest, it already knows gsk as it is simulating $\mathcal{M}_i$.
- $\mathcal{S}$ sends $(\mathsf{JOINSTART}, sid, jsid)$ to $\mathcal{F}$.

– On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
- $\mathcal{S}$ continues the simulation by giving “$\mathcal{I}$” input $(\mathsf{JOINPROCEED}, sid, jsid)$, and waits for output $(\mathsf{JOINED}, sid, jsid)$ from “$\mathcal{H}_j$”.
- Output $(\mathsf{JOINCOMPLETE}, sid, jsid, gsk)$ to $\mathcal{F}$.

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

– On input $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
- $\mathcal{S}$ simulates the real world protocol by giving “$\mathcal{H}_j$” input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ and waits for output $(\mathsf{JOINED}, sid, jsid)$ from “$\mathcal{H}_j$”.
- $\mathcal{S}$ sends $(\mathsf{JOINSTART}, sid, jsid)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{F}$.
- $\mathcal{S}$ sends $(\mathsf{JOINPROCEED}, sid, jsid)$ to $\mathcal{F}$ on behalf of $\mathcal{I}$

– Upon receiving $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
- Send $(\mathsf{JOINCOMPLETE}, sid, jsid, \perp)$ to $\mathcal{F}$.

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

– $\mathcal{S}$ notices this join as “$\mathcal{M}_i$” receives a nonce n from $\mathcal{H}_j$
- $\mathcal{S}$ makes a join query on $\mathcal{H}_j$'s behalf by sending $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
- $\mathcal{S}$ continues the simulation of “$\mathcal{M}_i$” until “$\mathcal{I}$” outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$.
- $\mathcal{S}$ sends $(\mathsf{JOINSTART}, sid, jsid)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
- $\mathcal{S}$ sends $(\mathsf{JOINCOMPLETE}, sid, jsid, gsk)$ to $\mathcal{F}$, where gsk is taken from simulating “$\mathcal{M}_i$”.

– Upon receiving $(\mathsf{JOINED}, sid, jsid)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupt.
- $\mathcal{S}$ gives “$\mathcal{I}$” input $(\mathsf{JOINPROCEED}, sid, jsid)$.

Simulator continues on next page.

Fig. 17. First part of Simulator Game 6

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

– $\mathcal{S}$ notices this join as "$\mathcal{I}$" receives $(\mathsf{SENT}, sid', (Q, \pi_1), \mathcal{H}'_j)$ from $\mathcal{F}_{\mathsf{auth}*}$.

- Parse sid' as $(\mathcal{M}_i, \mathcal{I}, sid)$. $\mathcal{S}$ extracts gsk from π_1.
- Note that $\mathcal{S}$ does not know the identity of the host that initiated this join, so it chooses some corrupt $\mathcal{H}_j$ and proceeds as if this is the host that initiated the join protocol. Even though this probably is not the correct host, it will only put a different host in `Members`, and the identities of hosts in this list are only used while creating signatures for platforms with an honest TPM or host, so for a fully corrupt platform it does not matter.
- $\mathcal{S}$ makes a join query with $\mathcal{M}_i$ by sending $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{F}$ on behalf of $\mathcal{H}_j$.

– Upon receiving $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.

- $\mathcal{S}$ continues simulating "$\mathcal{I}$" until it outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$.
- $\mathcal{S}$ sends $(\mathsf{JOINSTART}, sid, jsid)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.

- $\mathcal{S}$ sends $(\mathsf{JOINCOMPLETE}, sid, jsid, gsk)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{JOINED}, sid, jsid)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupt.

- $\mathcal{S}$ continues the simulation by giving "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, sid, jsid)$.

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

– $\mathcal{S}$ notices this join as "$\mathcal{M}_i$" receives a message n from $\mathcal{H}_j$

– $\mathcal{S}$ simply simulates $\mathcal{M}_i$ honestly, there is no need to involve $\mathcal{F}$ as $\mathcal{M}_i$ does not receive inputs or send outputs in the join procedure.

Sign
Unchanged.
Verify
Nothing to simulate.
Link
Nothing to simulate.

Fig. 18. Second part of Simulator GAME 6

Setup

Unchanged.

Join

Unchanged.

Sign with $\texttt{bsn} \neq \bot$

1. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \texttt{bsn})$ with $\texttt{bsn} \neq \bot$ from host $\mathcal{H}_j$.
– Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \texttt{bsn}), \mathcal{H}_j)$.
2. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
– Output $\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Sign with $\texttt{bsn} = \bot$

1. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \texttt{bsn})$ with $\texttt{bsn} = \bot$ from host $\mathcal{H}_j$.
– Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \texttt{bsn}, status\rangle$ with $status \leftarrow request$.
– Output $(\mathsf{SIGNSTART}, sid, ssid, m, \texttt{bsn}, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.
2. `Sign Request Delivery.` On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.
– Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \texttt{bsn}, status\rangle$ to $status \leftarrow delivered$.
– Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \texttt{bsn})$ to $\mathcal{M}_i$.
3. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
– Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \texttt{bsn}, status\rangle$ with $status = delivered$.
– Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.
4. `Signature Generation.` On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.
– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:
• If $\texttt{bsn} \neq \bot$, retrieve gsk from $\langle \mathcal{M}_i, \texttt{bsn}, gsk\rangle \in \texttt{DomainKeys}$ for $(\mathcal{M}_i, \texttt{bsn})$. If no such gsk exists or $\texttt{bsn} = \bot$, set $gsk \leftarrow \mathsf{ukgen}()$ and store $\langle \mathcal{M}_i, \texttt{bsn}, gsk\rangle$ in $\texttt{DomainKeys}$.
• Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \texttt{bsn})$.
– If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \texttt{bsn}, \mathcal{M}_i\rangle$ in $\texttt{Signed}$.
– Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \texttt{bsn}, \sigma, \texttt{RL})$ from some party $\mathcal{V}$.
– Set $f \leftarrow 0$ if at least one of the following conditions hold:
• There is a $gsk' \in \texttt{RL}$ where $\mathsf{identify}(\sigma, m, \texttt{bsn}, gsk') = 1$.
– If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \texttt{bsn})$.
– Add $\langle \sigma, m, \texttt{bsn}, \texttt{RL}, f\rangle$ to $\texttt{VerResults}$, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \texttt{bsn})$ from some party $\mathcal{V}$ with $\texttt{bsn} \neq \bot$.
– Output $\bot$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \texttt{bsn})$ or $(\sigma', m', \texttt{bsn})$ is not valid (verified via the `verify` interface with $\texttt{RL} = \emptyset$).
– Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \texttt{bsn})$.
– Output (LINK, sid, f) to $\mathcal{V}$.

Output

1. On input $(\mathsf{OUTPUT}, \mathcal{P}, m)$ from $\mathcal{S}$.
– Output (m) to $\mathcal{P}$.

Fig. 19. $\mathcal{F}$ for GAME 7

When any simulated party "$\mathcal{P}$" outputs a message m that is not explicitly handled by $\mathcal{S}$ yet, $\mathcal{S}$ sends $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$.

KeyGen

Unchanged.

Join

Unchanged.

Sign with $\mathtt{bsn} \neq \perp$

– Upon receiving $(\mathsf{FORWARD}, (\mathsf{SIGN}, \mathit{sid}, \mathit{ssid}, \mathcal{M}_i, m, \mathtt{bsn}), \mathcal{H}_j)$ from $\mathcal{F}$.

• Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, \mathit{sid}, \mathit{ssid}, \mathcal{M}_i, m, \mathtt{bsn})$.

– Upon receiving $\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid}), \mathcal{M}_i)$ from $\mathcal{F}$.

• Give "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid})$.

Sign with $\mathtt{bsn} = \perp$

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

– Upon receiving $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid}, m, \mathtt{bsn}, \mathcal{M}_i, \mathcal{H}_j)$ with $\mathtt{bsn} = \perp$ from $\mathcal{F}$.

• $\mathcal{S}$ starts the simulation by giving "$\mathcal{H}_j$" input $(\mathsf{SIGN}, \mathit{sid}, \mathit{ssid}, \mathcal{M}_i, m', \mathtt{bsn}')$.

• When "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid}, m', \mathtt{bsn}')$, send $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid})$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid})$ from $\mathcal{F}$.

• Continue the simulation by giving "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid})$.

• When "$\mathcal{M}_i$" outputs $(\mathsf{SIGNATURE}, \mathit{sid}, \mathit{ssid}, \sigma)$, send $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid}, \perp)$ to $\mathcal{F}$.

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

– Upon receiving $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid}, m, \mathtt{bsn}, \mathcal{M}_i, \mathcal{H}_j)$ with $\mathtt{bsn} = \perp$ from $\mathcal{F}$.

• Send $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid})$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid}, m, \mathtt{bsn})$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.

• Starts the simulation by giving "$\mathcal{H}_j$" input $(\mathsf{SIGN}, \mathit{sid}, \mathit{ssid}, \mathcal{M}_i, m, \mathtt{bsn})$.

• When "$\mathcal{H}_j$" outputs $(\mathsf{SIGNATURE}, \mathit{sid}, \mathit{ssid}, \sigma)$, sends $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid})$ to $\mathcal{F}$ on behalf of $\mathcal{M}_i$.

– Upon receiving $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid})$ from $\mathcal{F}$.

• Send $\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid}, \sigma)$ to $\mathcal{F}$.

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

– $\mathcal{S}$ notices this sign as "$\mathcal{M}_i$" receives $m, \mathtt{bsn}, r$ with $\mathtt{bsn} = \perp$ from $\mathcal{H}_j$.

• Make a sign query on $\mathcal{H}_j$'s behalf by sending $(\mathsf{SIGN}, \mathit{sid}, \mathit{ssid}, \mathcal{M}_i, m, \mathtt{bsn})$.

– Upon receiving $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid}, m, \mathtt{bsn}, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.

• Continue the simulation of "$\mathcal{M}_i$" until it outputs $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid}, m, \mathtt{bsn})$.

• Sends $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid})$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid})$ from $\mathcal{F}$.

• Send $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid}, \perp)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{SIGNATURE}, \mathit{sid}, \mathit{ssid}, \sigma)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupt.

• Continue the simulation by giving "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid})$.

Verify

Nothing to simulate.

Link

Nothing to simulate.

Fig. 20. Simulator GAME 7

Setup
Unchanged.
Join
Unchanged.
Sign

1. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.
– Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status\rangle$ with $status \leftarrow request$.
– Output $(\mathsf{SIGNSTART}, sid, ssid, m, \mathtt{bsn}, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.
2. `Sign Request Delivery.` On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.
– Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status\rangle$ to $status \leftarrow delivered$.
– Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ to $\mathcal{M}_i$.
3. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
– Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status\rangle$ with $status = delivered$.
– Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.
4. `Signature Generation.` On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.
– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:
• If $\mathtt{bsn} \neq \bot$, retrieve gsk from $\langle \mathcal{M}_i, \mathtt{bsn}, gsk\rangle \in \mathtt{DomainKeys}$ for $(\mathcal{M}_i, \mathtt{bsn})$. If no such gsk exists or $\mathtt{bsn} = \bot$, set $gsk \leftarrow \mathsf{ukgen}()$ and store $\langle \mathcal{M}_i, \mathtt{bsn}, gsk\rangle$ in $\mathtt{DomainKeys}$.
• Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \mathtt{bsn})$.
– If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i\rangle$ in $\mathtt{Signed}$.
– Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
– Set $f \leftarrow 0$ if at least one of the following conditions hold:
• There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.
– If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.
– Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f\rangle$ to $\mathtt{VerResults}$, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \bot$.
– Output $\bot$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
– Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.
– Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 21. $\mathcal{F}$ for Game 8

KeyGen

Unchanged.

Join

Unchanged.

Sign

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

– Upon receiving $(\mathsf{SIGNSTART}, sid, ssid, m, \mathtt{bsn}, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.

• $\mathcal{S}$ starts the simulation by giving "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$.

• When "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$, send $(\mathsf{SIGNSTART}, sid, ssid)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ from $\mathcal{F}$.

• Continue the simulation by giving "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

• When "$\mathcal{M}_i$" outputs $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$, send $(\mathsf{SIGNCOMPLETE}, sid, ssid, \bot)$ to $\mathcal{F}$.

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

– Upon receiving $(\mathsf{SIGNSTART}, sid, ssid, m, \mathtt{bsn}, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.

• Send $(\mathsf{SIGNSTART}, sid, ssid)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.

• Starts the simulation by giving "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$.

• When "$\mathcal{H}_j$" outputs $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$, sends $(\mathsf{SIGNPROCEED}, sid, ssid)$ to $\mathcal{F}$ on behalf of $\mathcal{M}_i$.

– Upon receiving $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ from $\mathcal{F}$.

• Send $\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ to $\mathcal{F}$.

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

– $\mathcal{S}$ notices this sign as "$\mathcal{M}_i$" receives $m, \mathtt{bsn}, r$ from $\mathcal{H}_j$.

• Make a sign query on $\mathcal{H}_j$'s behalf by sending $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$.

– Upon receiving $(\mathsf{SIGNSTART}, sid, ssid, m, \mathtt{bsn}, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.

• Continue the simulation of "$\mathcal{M}_i$" until it outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$.

• Sends $(\mathsf{SIGNSTART}, sid, ssid)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ from $\mathcal{F}$.

• Send $(\mathsf{SIGNCOMPLETE}, sid, ssid, \bot)$ to $\mathcal{F}$.

– Upon receiving $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupt.

• Continue the simulation by giving "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

Verify

Nothing to simulate.

Link

Nothing to simulate.

Fig. 22. Simulator GAME 8

Setup

Unchanged.

Join

Unchanged.

Sign

1. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.
- Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status\rangle$ with $status \leftarrow request$.
- Output $(\mathsf{SIGNSTART}, sid, ssid, l(m, \mathtt{bsn}), \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.
2. `Sign Request Delivery.` On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.
- Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status\rangle$ to $status \leftarrow delivered$.
- Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ to $\mathcal{M}_i$.
3. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
- Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status\rangle$ with $status = delivered$.
- Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.
4. `Signature Generation.` On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.
- If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:
- • If $\mathtt{bsn} \neq \bot$, retrieve gsk from $\langle \mathcal{M}_i, \mathtt{bsn}, gsk\rangle \in \mathtt{DomainKeys}$ for $(\mathcal{M}_i, \mathtt{bsn})$. If no such gsk exists or $\mathtt{bsn} = \bot$, set $gsk \leftarrow \mathsf{ukgen}()$ and store $\langle \mathcal{M}_i, \mathtt{bsn}, gsk\rangle$ in $\mathtt{DomainKeys}$.
- • Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \mathtt{bsn})$.
- If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i\rangle$ in $\mathtt{Signed}$.
- Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
- Set $f \leftarrow 0$ if at least one of the following conditions hold:
- • There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.
- If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.
- Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f\rangle$ to $\mathtt{VerResults}$, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \bot$.
- Output $\bot$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
- Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.
- Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 23. $\mathcal{F}$ for Game 9

KeyGen
Unchanged.
Join
Unchanged.
Sign
Honest $\mathcal{M}$, $\mathcal{H}$:

- Upon receiving $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid}, l, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Take a dummy $m', \mathtt{bsn}'$ such that $l(m', \mathtt{bsn}') = l$.
  - $\mathcal{S}$ starts the simulation by giving "$\mathcal{H}_j$" input $(\mathsf{SIGN}, \mathit{sid}, \mathit{ssid}, \mathcal{M}_i, m', \mathtt{bsn}')$.
  - When "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid}, m', \mathtt{bsn}')$, send $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid})$ to $\mathcal{F}$.
- Upon receiving $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid})$ from $\mathcal{F}$.
  - Continue the simulation by giving "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid})$.
  - When "$\mathcal{M}_i$" outputs $(\mathsf{SIGNATURE}, \mathit{sid}, \mathit{ssid}, \sigma)$, send $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid}, \bot)$ to $\mathcal{F}$.

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

- Upon receiving $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid}, l, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Send $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid})$ to $\mathcal{F}$.
- Upon receiving $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid}, m, \mathtt{bsn})$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Starts the simulation by giving "$\mathcal{H}_j$" input $(\mathsf{SIGN}, \mathit{sid}, \mathit{ssid}, \mathcal{M}_i, m, \mathtt{bsn})$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{SIGNATURE}, \mathit{sid}, \mathit{ssid}, \sigma)$, sends $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid})$ to $\mathcal{F}$ on behalf of $\mathcal{M}_i$.
- Upon receiving $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid})$ from $\mathcal{F}$.
  - Send $\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid}, \sigma)$ to $\mathcal{F}$.

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

- $\mathcal{S}$ notices this sign as "$\mathcal{M}_i$" receives $m, \mathtt{bsn}, r$ from $\mathcal{H}_j$.
  - Make a sign query on $\mathcal{H}_j$'s behalf by sending $(\mathsf{SIGN}, \mathit{sid}, \mathit{ssid}, \mathcal{M}_i, m, \mathtt{bsn})$.
- Upon receiving $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid}, l, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Continue the simulation of "$\mathcal{M}_i$" until it outputs $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid}, m, \mathtt{bsn})$.
  - Sends $(\mathsf{SIGNSTART}, \mathit{sid}, \mathit{ssid})$ to $\mathcal{F}$.
- Upon receiving $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid})$ from $\mathcal{F}$.
  - Send $(\mathsf{SIGNCOMPLETE}, \mathit{sid}, \mathit{ssid}, \bot)$ to $\mathcal{F}$.
- Upon receiving $(\mathsf{SIGNATURE}, \mathit{sid}, \mathit{ssid}, \sigma)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupt.
  - Continue the simulation by giving "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, \mathit{sid}, \mathit{ssid})$.

Verify
Nothing to simulate.
Link
Nothing to simulate.

Fig. 24. Simulator Game 9

Setup

Unchanged.

Join

Unchanged.

Sign

1. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.

– If $\mathcal{I}$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`, abort.

– Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status \leftarrow request$.

– Output $(\mathsf{SIGNSTART}, sid, ssid, l(m, \mathtt{bsn}), \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.

2. `Sign Request Delivery.` On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.

– Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ to $status \leftarrow delivered$.

– Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ to $\mathcal{M}_i$.

3. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.

– Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status = delivered$.

– Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.

4. `Signature Generation.` On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.

– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:

• If $\mathtt{bsn} \neq \bot$, retrieve gsk from $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle \in$ `DomainKeys` for $(\mathcal{M}_i, \mathtt{bsn})$. If no such gsk exists or $\mathtt{bsn} = \bot$, set $gsk \leftarrow \mathsf{ukgen}()$ and store $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle$ in `DomainKeys`.

• Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \mathtt{bsn})$.

– If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i \rangle$ in `Signed`.

– Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.

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

• There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.

– If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.

– Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f \rangle$ to `VerResults`, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \bot$.

– Output $\bot$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).

– Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.

– Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 25. $\mathcal{F}$ for Game 10

KeyGen
Unchanged.
Join
Unchanged.
Sign
Unchanged
Verify
Unchanged.
Link
Unchanged.

Fig. 26. Simulator GAME 10

Setup

Unchanged.

Join

1. Join Request. On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$
– Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status \leftarrow request$.
– Output $(\mathsf{JOINSTART}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.
2. Join Request Delivery. On input $(\mathsf{JOINSTART}, sid, jsid)$ from $\mathcal{S}$
– Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ to $status \leftarrow delivered$.
– Abort if $\mathcal{I}$ or $\mathcal{M}_i$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in \mathtt{Members}$ already exists.
– Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
3. Complete Join. On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$
– Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ to $status \leftarrow complete$.
– Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{S}$.
4. Key Generation. On input $(\mathsf{JOINCOMPLETE}, sid, jsid, gsk)$ from $\mathcal{S}$.
– Look up record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = complete$.
– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, set $gsk \leftarrow \bot$.
– Else, verify that the provided gsk is eligible by checking
• $\mathsf{CheckGskHonest}(gsk) = 1$ if $\mathcal{H}_j$ is corrupt and $\mathcal{M}_i$ is honest, or
• $\mathsf{CheckGskCorrupt}(gsk) = 1$ if $\mathcal{M}_i$ is corrupt.
– Insert $\langle \mathcal{M}_i, \mathcal{H}_j, gsk\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid)$ to $\mathcal{H}_j$.

Sign

1. Sign Request. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.
– If $\mathcal{I}$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in $\mathtt{Members}$, abort.
– Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status\rangle$ with $status \leftarrow request$.
– Output $(\mathsf{SIGNSTART}, sid, ssid, l(m, \mathtt{bsn}), \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.
2. Sign Request Delivery. On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.
– Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status\rangle$ to $status \leftarrow delivered$.
– Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ to $\mathcal{M}_i$.
3. Sign Proceed. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
– Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status\rangle$ with $status = delivered$.
– Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.
4. Signature Generation. On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.
– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:
• If $\mathtt{bsn} \neq \bot$, retrieve gsk from $\langle \mathcal{M}_i, \mathtt{bsn}, gsk\rangle \in \mathtt{DomainKeys}$ for $(\mathcal{M}_i, \mathtt{bsn})$. If no such gsk exists or $\mathtt{bsn} = \bot$, set $gsk \leftarrow \mathsf{ukgen}()$. Check $\mathsf{CheckGskHonest}(gsk) = 1$ and store $\langle \mathcal{M}_i, \mathtt{bsn}, gsk\rangle$ in $\mathtt{DomainKeys}$.
• Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \mathtt{bsn})$.
– If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i\rangle$ in $\mathtt{Signed}$.
– Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

Unchanged.

Link

Unchanged.

Fig. 27. $\mathcal{F}$ for Game 11

KeyGen
Unchanged.
Join
Unchanged.
Sign
Unchanged
Verify
Unchanged.
Link
Unchanged.

Fig. 28. Simulator GAME 11

Setup

Unchanged.

Join

Unchanged.

Sign

1. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.
– If $\mathcal{I}$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`, abort.
– Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status \leftarrow request$.
– Output $(\mathsf{SIGNSTART}, sid, ssid, l(m, \mathtt{bsn}), \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.
2. `Sign Request Delivery.` On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.
– Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ to $status \leftarrow delivered$.
– Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ to $\mathcal{M}_i$.
3. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
– Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status = delivered$.
– Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.
4. `Signature Generation.` On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.
– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:
• If $\mathtt{bsn} \neq \bot$, retrieve gsk from $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle \in$ `DomainKeys` for $(\mathcal{M}_i, \mathtt{bsn})$. If no such gsk exists or $\mathtt{bsn} = \bot$, set $gsk \leftarrow \mathsf{ukgen}()$. Check $\mathsf{CheckGskHonest}(gsk) = 1$ and store $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle$ in `DomainKeys`.
• Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \mathtt{bsn})$ and check $\mathsf{ver}(\sigma, m, \mathtt{bsn}) = 1$.
• Check $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk) = 1$ and check that there is no $\mathcal{M}_i' \neq \mathcal{M}_i$ with key gsk' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.
– If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i \rangle$ in `Signed`.
– Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
– Set $f \leftarrow 0$ if at least one of the following conditions hold:
• There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.
– If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.
– Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f \rangle$ to `VerResults`, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \bot$.
– Output $\bot$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
– Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.
– Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 29. $\mathcal{F}$ for GAME 12

KeyGen
Unchanged.
Join
Unchanged.
Sign
Unchanged
Verify
Unchanged.
Link
Unchanged.

Fig. 30. Simulator GAME 12

Setup

Unchanged.

Join

Unchanged.

Sign

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

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
   - Retrieve all pairs $(gsk_i, \mathcal{M}_i)$ from $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ `Members` and $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one key gsk_i was found.
     - There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.
   - Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f \rangle$ to `VerResults`, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \bot$.
   - Output $\bot$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.
   - Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 31. $\mathcal{F}$ for Game 13

KeyGen
Unchanged.
Join
Unchanged.
Sign
Unchanged
Verify
Unchanged.
Link
Unchanged.

Fig. 32. Simulator Game 13

Setup

Unchanged.

Join

Unchanged.

Sign

1. Sign Request. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.

– If $\mathcal{I}$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in Members, abort.

– Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status \leftarrow request$.

– Output $(\mathsf{SIGNSTART}, sid, ssid, l(m, \mathtt{bsn}), \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.

2. Sign Request Delivery. On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.

– Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ to $status \leftarrow delivered$.

– Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ to $\mathcal{M}_i$.

3. Sign Proceed. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.

– Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status = delivered$.

– Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.

4. Signature Generation. On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.

– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:

• If $\mathtt{bsn} \neq \perp$, retrieve gsk from $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle \in$ DomainKeys for $(\mathcal{M}_i, \mathtt{bsn})$. If no such gsk exists or $\mathtt{bsn} = \perp$, set $gsk \leftarrow \mathsf{ukgen}()$. Check $\mathsf{CheckGskHonest}(gsk) = 1$ and store $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle$ in DomainKeys.

• Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \mathtt{bsn})$ and check $\mathsf{ver}(\sigma, m, \mathtt{bsn}) = 1$.

• Check $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk) = 1$ and check that there is no $\mathcal{M}_i' \neq \mathcal{M}_i$ with key gsk' registered in Members or DomainKeys with $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.

– If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i \rangle$ in Signed.

– Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

1. Verify. On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.

– Retrieve all pairs $(gsk_i, \mathcal{M}_i)$ from $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ Members and $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ DomainKeys where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:

• More than one key gsk_i was found.

• $\mathcal{I}$ is honest and no pair $(gsk_i, \mathcal{M}_i)$ was found.

• There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.

– If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.

– Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f \rangle$ to VerResults, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. Link. On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \perp$.

– Output $\perp$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the verify interface with $\mathtt{RL} = \emptyset$).

– Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.

– Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 33. $\mathcal{F}$ for Game 14

KeyGen
Unchanged.
Join
Unchanged.
Sign
Unchanged
Verify
Unchanged.
Link
Unchanged.

Fig. 34. Simulator GAME 14

Setup

Unchanged.

Join

Unchanged.

Sign

1. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.
– If $\mathcal{I}$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`, abort.
– Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status \leftarrow request$.
– Output $(\mathsf{SIGNSTART}, sid, ssid, l(m, \mathtt{bsn}), \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.
2. `Sign Request Delivery.` On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.
– Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ to $status \leftarrow delivered$.
– Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ to $\mathcal{M}_i$.
3. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
– Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status = delivered$.
– Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.
4. `Signature Generation.` On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.
– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:
• If $\mathtt{bsn} \neq \perp$, retrieve gsk from $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle \in$ `DomainKeys` for $(\mathcal{M}_i, \mathtt{bsn})$. If no such gsk exists or $\mathtt{bsn} = \perp$, set $gsk \leftarrow \mathsf{ukgen}()$. Check $\mathsf{CheckGskHonest}(gsk) = 1$ and store $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle$ in `DomainKeys`.
• Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \mathtt{bsn})$ and check $\mathsf{ver}(\sigma, m, \mathtt{bsn}) = 1$.
• Check $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk) = 1$ and check that there is no $\mathcal{M}_i' \neq \mathcal{M}_i$ with key gsk' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.
– If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i \rangle$ in `Signed`.
– Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
– Retrieve all pairs $(gsk_i, \mathcal{M}_i)$ from $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ `Members` and $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
• More than one key gsk_i was found.
• $\mathcal{I}$ is honest and no pair $(gsk_i, \mathcal{M}_i)$ was found.
• There is an honest $\mathcal{M}_i$ but no entry $\langle *, m, \mathtt{bsn}, \mathcal{M}_i \rangle \in$ `Signed` exists.
• There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.
– If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.
– Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f \rangle$ to `VerResults`, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \perp$.
– Output $\perp$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
– Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.
– Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 35. $\mathcal{F}$ for Game 15

KeyGen
Unchanged.
Join
Unchanged.
Sign
Unchanged
Verify
Unchanged.
Link
Unchanged.

Fig. 36. Simulator GAME 15

Setup

Unchanged.

Join

Unchanged.

Sign

1. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.
– If $\mathcal{I}$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`, abort.
– Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status \leftarrow request$.
– Output $(\mathsf{SIGNSTART}, sid, ssid, l(m, \mathtt{bsn}), \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.
2. `Sign Request Delivery.` On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.
– Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ to $status \leftarrow delivered$.
– Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ to $\mathcal{M}_i$.
3. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
– Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status = delivered$.
– Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.
4. `Signature Generation.` On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.
– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:
- If $\mathtt{bsn} \neq \perp$, retrieve gsk from $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle \in$ `DomainKeys` for $(\mathcal{M}_i, \mathtt{bsn})$. If no such gsk exists or $\mathtt{bsn} = \perp$, set $gsk \leftarrow \mathsf{ukgen}()$. Check $\mathsf{CheckGskHonest}(gsk) = 1$ and store $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle$ in `DomainKeys`.
- Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \mathtt{bsn})$ and check $\mathsf{ver}(\sigma, m, \mathtt{bsn}) = 1$.
- Check $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk) = 1$ and check that there is no $\mathcal{M}_i' \neq \mathcal{M}_i$ with key gsk' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.

– If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i \rangle$ in `Signed`.
– Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
– Retrieve all pairs $(gsk_i, \mathcal{M}_i)$ from $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ `Members` and $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
- More than one key gsk_i was found.
- $\mathcal{I}$ is honest and no pair $(gsk_i, \mathcal{M}_i)$ was found.
- There is an honest $\mathcal{M}_i$ but no entry $\langle *, m, \mathtt{bsn}, \mathcal{M}_i \rangle \in$ `Signed` exists.
- There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$ and no pair $(gsk_i, \mathcal{M}_i)$ for an honest $\mathcal{M}_i$ was found.

– If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.
– Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f \rangle$ to `VerResults`, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \perp$.
– Output $\perp$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
– Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.
– Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 37. $\mathcal{F}$ for Game 16

KeyGen
Unchanged.
Join
Unchanged.
Sign
Unchanged
Verify
Unchanged.
Link
Unchanged.

Fig. 38. Simulator Game 16

Setup

Unchanged.

Join

Unchanged.

Sign

1. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, \mathtt{bsn})$ from host $\mathcal{H}_j$.

– If $\mathcal{I}$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`, abort.

– Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status \leftarrow request$.

– Output $(\mathsf{SIGNSTART}, sid, ssid, l(m, \mathtt{bsn}), \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{S}$.

2. `Sign Request Delivery.` On input $(\mathsf{SIGNSTART}, sid, ssid)$ from $\mathcal{S}$.

– Update the session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ to $status \leftarrow delivered$.

– Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, \mathtt{bsn})$ to $\mathcal{M}_i$.

3. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.

– Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, \mathtt{bsn}, status \rangle$ with $status = delivered$.

– Output $(\mathsf{SIGNCOMPLETE}, sid, ssid)$ to $\mathcal{S}$.

4. `Signature Generation.` On input $(\mathsf{SIGNCOMPLETE}, sid, ssid, \sigma)$ from $\mathcal{S}$.

– If $\mathcal{M}_i$ and $\mathcal{H}_j$ are honest, ignore the adversary's signature and internally generate the signature for a fresh or established gsk:

• If $\mathtt{bsn} \neq \perp$, retrieve gsk from $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle \in$ `DomainKeys` for $(\mathcal{M}_i, \mathtt{bsn})$. If no such gsk exists or $\mathtt{bsn} = \perp$, set $gsk \leftarrow \mathsf{ukgen}()$. Check $\mathsf{CheckGskHonest}(gsk) = 1$ and store $\langle \mathcal{M}_i, \mathtt{bsn}, gsk \rangle$ in `DomainKeys`.

• Compute signature as $\sigma \leftarrow \mathsf{sig}(gsk, m, \mathtt{bsn})$ and check $\mathsf{ver}(\sigma, m, \mathtt{bsn}) = 1$.

• Check $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk) = 1$ and check that there is no $\mathcal{M}_i' \neq \mathcal{M}_i$ with key gsk' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$.

– If $\mathcal{M}_i$ is honest, store $\langle \sigma, m, \mathtt{bsn}, \mathcal{M}_i \rangle$ in `Signed`.

– Output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

1. `Verify.` On input $(\mathsf{VERIFY}, sid, m, \mathtt{bsn}, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.

– Retrieve all pairs $(gsk_i, \mathcal{M}_i)$ from $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ `Members` and $\langle \mathcal{M}_i, *, gsk_i \rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:

• More than one key gsk_i was found.

• $\mathcal{I}$ is honest and no pair $(gsk_i, \mathcal{M}_i)$ was found.

• There is an honest $\mathcal{M}_i$ but no entry $\langle *, m, \mathtt{bsn}, \mathcal{M}_i \rangle \in$ `Signed` exists.

• There is a $gsk' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk') = 1$ and no pair $(gsk_i, \mathcal{M}_i)$ for an honest $\mathcal{M}_i$ was found.

– If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, \mathtt{bsn})$.

– Add $\langle \sigma, m, \mathtt{bsn}, \mathtt{RL}, f \rangle$ to `VerResults`, output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

1. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', \mathtt{bsn})$ from some party $\mathcal{V}$ with $\mathtt{bsn} \neq \perp$.

– Output $\perp$ to $\mathcal{V}$ if at least one signature tuple $(\sigma, m, \mathtt{bsn})$ or $(\sigma', m', \mathtt{bsn})$ is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).

– For each gsk_i in `Members` and `DomainKeys` compute $b_i \leftarrow \mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk_i)$ and $b_i' \leftarrow \mathsf{identify}(\sigma', m', \mathtt{bsn}, gsk_i)$ and do the following:

• Set $f \leftarrow 0$ if $b_i \neq b_i'$ for some i.

• Set $f \leftarrow 1$ if $b_i = b_i' = 1$ for some i.

– If f is not defined yet, Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', \mathtt{bsn})$.

– Output (LINK, sid, f) to $\mathcal{V}$.

Fig. 39. $\mathcal{F}$ for Game 17

KeyGen
Unchanged.
Join
Unchanged.
Sign
Unchanged
Verify
Unchanged.
Link
Unchanged.

Fig. 40. Simulator Game 17

B.2 Indistinguishability of games

Game 1: This is the real world protocol.

Game 2: $\mathcal{C}$ receives all inputs for honest parties and simulates the real world protocol for honest parties. By construction, this is equivalent to the real world.

Game 3: We now split $\mathcal{C}$ into two pieces, $\mathcal{F}$ and $\mathcal{S}$. We let $\mathcal{F}$ evolve to become $\mathcal{F}_{\mathsf{daa}}$ and $\mathcal{S}$ to become the simulator. $\mathcal{F}$ behaves as an ideal functionality, so the messages it sends are authenticated and immediate, so $\mathcal{A}$ will not notice them. $\mathcal{F}$ receives all the inputs, who forwards them to $\mathcal{S}$. $\mathcal{S}$ will simulate the real world protocol for all honest parties, and sends the outputs to $\mathcal{F}$, who forwards them to $\mathcal{E}$. Outputs generated by parties simulated by $\mathcal{S}$ are not sent anywhere, only $\mathcal{S}$ notices them. $\mathcal{S}$ sends an equivalent output to $\mathcal{F}$ using an OUTPUT message, such that $\mathcal{F}$ will generate the same output.

This game is simply Game 2 but structured differently, so Game 3 = Game 2.

Game 4: We now change the behavior of $\mathcal{F}$ in the setup interface, and store algorithms in $\mathcal{F}$. Note that $\mathcal{F}$ now checks the structure of *sid* for honest issuer $\mathcal{I}$, and aborts when it is not of the expected form. This abort will not change the view of $\mathcal{E}$, as $\mathcal{I}$ performs the same check upon receiving this input.

The simulated real world does not change, as $\mathcal{I}$ gives "$\mathcal{I}$" the correct input. For corrupt $\mathcal{I}$, $\mathcal{S}$ also extracts the secret key and calls the setup interface on $\mathcal{I}$'s behalf, but clearly this does not change $\mathcal{E}$'s view, so Game 4 = Game 3.

Game 5: $\mathcal{F}$ now handles verification and link queries instead of forwarding them to $\mathcal{S}$. There are no protocol messages, so we only have to make sure the output is equal.

The verification algorithm $\mathcal{F}$ uses is almost equal to the real world protocol. The only difference is the that the ver algorithm that $\mathcal{F}$ uses does not contain the revocation check. $\mathcal{F}$ performs this check separately, so the outcomes are equal.

The linking protocol outputs $\perp$ when it is called with invalid signatures. $\mathcal{F}$ does the same, it verifies the signatures and outputs $\perp$ when one of the signatures is not valid. The protocol then checks the pseudonyms for equality, which is exactly what $\mathcal{F}$ does, showing that the outputs will be equal. $\mathcal{F}$ requires link to be symmetrical and outputs $\perp$ to $\mathcal{E}$ when it notices that it is not. This algorithm is symmetrical, so this abort will not happen and we have Game 5 = Game 4.

Game 6: The join interface of $\mathcal{F}$ is now changed. It stores which members joined, and if $\mathcal{I}$ is honest, stores the *gsk* key with which corrupt TPMs joined. $\mathcal{S}$ extracts this from proof π in the simulated real world.

If extraction by rewinding is used, a fresh nonce must be included in the proof, guaranteeing that this proof is fresh and we can extract from it. Note that we can rewind here as the honest $\mathcal{I}$ only allows a logarithmic number of simultaneous join sessions, so there is no risk of requiring exponential time to rewind every join session.

$\mathcal{S}$ always has enough information to simulate the real world protocol, except when only the issuer is honest. It then does not know which host initiated this join, so it cannot make a join query with $\mathcal{F}$ on that host's behalf. However, it is sufficient to take any corrupt host, as this will only result in a different identity in $\texttt{Members}$, and in $\mathcal{F}_{\mathsf{daa}}$ this identity only matters for honest hosts.

Note that we need the TPM to be assured that b, d, π_2 was sent by the issuer. If not, the host could send these values without involving the honest issuer, resulting in $\mathcal{E}$ not seeing the JOINPROCEED output in the real world. We have to make a query with $\mathcal{F}$ such that the platform whereas in the ideal world, $\mathcal{E}$ would receive this output when $\mathcal{I}$ is honest. Since the TPM knows that b, d, π_2 was sent by $\mathcal{I}$, we are assured that $\mathcal{I}$ was involved in this join and $\mathcal{E}$ sees the JOINPROCEED output.

We must argue that $\mathcal{F}$ does not prevent an execution that was previously allowed. $\mathcal{F}$ only has one check that may cause it to abort, if $\mathcal{M}$ already registered and $\mathcal{I}$ is honest. Since in the protocol, $\mathcal{I}$ checks this before outputting JOINPROCEED, $\mathcal{F}$ will never abort.

As $\mathcal{S}$ can simulate the real world protocol and keep everything in sync with $\mathcal{F}$, the view of $\mathcal{E}$ does not change, GAME 6 = GAME 5.

Game 7: When signing with $\texttt{bsn} = \bot$, $\mathcal{F}$ now creates anonymous signatures for honest platforms using the algorithms defined in setup. Clearly the process for creating signatures with $\texttt{bsn} \neq \bot$ has not changed. To show that $\mathcal{E}$ cannot notice that signatures with $\texttt{bsn} = \bot$ are now made in a different way by $\mathcal{F}$, we make this change gradually.

In GAME 7.$k.k'$, $\mathcal{F}$ forwards all signing inputs with $\mathcal{M}_i, i < k$ to $\mathcal{S}$, and $\mathcal{S}$ creates signatures as before. For signing inputs with $\mathcal{M}_k$, the first k' inputs are handled by $\mathcal{F}$, and later inputs will be forwarded to $\mathcal{S}$. We now have GAME 7.0.0 = GAME 6. When increasing k', in polynomially many steps GAME 7.k,k' will be equal to GAME 7.$k+1$.0 Repeating this process will make k large enough to include all TPMs, so for some k, k', we have GAME 7 = GAME 7.$k.k'$. Therefore, to show that GAME 7 = GAME 6, it suffices to show that increasing k' by one is indistinguishable. Note that in this reduction we allow $\mathcal{S}$ and $\mathcal{F}$ to share information, as in this reduction the separation of $\mathcal{S}$ and $\mathcal{F}$ is irrelevant.

We now show that anyone distinguishing GAME 7.$k.k'$ from GAME 7.$k.k'+1$ can solve DDH. We modify $\mathcal{S}$ working with $\mathcal{F}$ parametrized by k, k' such that if it receives a DDH tuple, it is equivalent to GAME 7.$k.k'$, and otherwise equivalent to GAME 7.$k.k'+1$.

$\mathcal{S}$ receives a DDH instance $\tilde{g}, \alpha, \beta, \gamma \in \mathbb{G}_1$ and must answer whether $log_{\tilde{g}}(\alpha) \cdot log_{\tilde{g}}(\beta) = log_{\tilde{g}}(\gamma)$. $\mathcal{S}$ simulates $\mathcal{M}_k$ using the unknown discrete logarithm of α as its gsk by setting $Q \leftarrow \alpha$, simulating proof π_1 in join. Note that the issuers proof π_2 helps the simulation of a TPM without knowing its gsk, as we don't need gsk to check $b^{gsk} = d$.

The first k' signatures for $\mathcal{M}_k$ are fully anonymous, generated by $\mathcal{F}$.

In the $k'+1$-th signature for $\mathcal{M}_k$, we modify $\mathcal{F}$ to output a signature using the DDH instance: it computes credential $a \leftarrow \beta^{1/y}, b \leftarrow \beta, c \leftarrow a^x \cdot \gamma^x, d \leftarrow \gamma$, using x, y that $\mathcal{S}$ learned during the setup protocol. It simulates the proof π and outputs a, b, c, d, π.

Signing queries with $\mathcal{M}_k$ after the $k' + 1$-th one are handled by $\mathcal{S}$. When $\mathcal{S}$ must create signatures for $\mathcal{M}_k$, it executes the protocol honestly except that it simulates π.

Note that now the first k' signing queries are based on different gsk values for every signature, the $k' + 1$-th query is also based on a fresh gsk if the DDH instance is not a DDH tuple or is based on the gsk from join if it is a DDH tuple, and all later signatures are based on the gsk from join. Now any distinguisher between Game 7.k.k' and Game 7.k.$k' + 1$ can solve DDH.

Game 8: When signing with $\texttt{bsn} \neq \bot$, $\mathcal{F}$ now creates pseudonymous signatures for honest platforms using the algorithms defined in setup. To show that $\mathcal{E}$ cannot notice that signatures with $\texttt{bsn} \neq \bot$ are now made in a different way by $\mathcal{F}$, we make this change gradually.

In Game 8.k.k', $\mathcal{F}$ forwards all signing inputs with $\mathcal{M}_i, i < k$ to $\mathcal{S}$, and $\mathcal{S}$ creates signatures as before. For signing inputs with $\mathcal{M}_k$, signing queries with the first k' basenames that are hashed are handled by $\mathcal{F}$, and later inputs will be forwarded to $\mathcal{S}$. We now have Game 8.0.0 = Game 7. When increasing k', in polynomially many steps Game 8.k,k' will be equal to Game 8.$k + 1$.0, as there can only be polynomially many basenames hashed. Repeating this process will make k large enough to include all TPMs, so for some k, k', we have Game 8 = Game 8.k.k'. Therefore, to show that Game 8 = Game 7, it suffices to show that increasing k' by one is indistinguishable.

We now show that anyone distinguishing Game 8.k.k' from Game 8.k.$k'+1$ can solve DDH. We modify $\mathcal{S}$ working with $\mathcal{F}$ parametrized by k, k' such that if it receives a DDH tuple, it is equivalent to Game 8.k.k', and otherwise equivalent to Game 8.k.$k' + 1$.

$\mathcal{S}$ receives a DDH instance $\tilde{g}, \alpha, \beta, \gamma \in \mathbb{G}_1$ and must answer whether $log_{\tilde{g}}(\alpha) \cdot log_{\tilde{g}}(\beta) = log_{\tilde{g}}(\gamma)$. $\mathcal{S}$ answers H_1 queries with g_1^r for some $r \in_R \mathbb{Z}_q$, maintaining consistency, except the k'-th query, in which it returns β^r for $r \in_R \mathbb{Z}_q$. $\mathcal{S}$ simulates $\mathcal{M}_k$ using the unknown discrete logarithm α as its gsk by setting $Q \leftarrow \alpha$, simulating proof π_1 in join.

Signatures with the first k' basenames are handled by $\mathcal{F}$.

When creating signatures with the $k' + 1$-th basename for $\mathcal{M}_k$, we modify $\mathcal{F}$ to output a signature using the DDH instance: it computes credential $a \leftarrow \beta^{1/y}, b \leftarrow \beta, c \leftarrow a^x \cdot \gamma^x, d \leftarrow \gamma$, randomizes the credential by raising the four values to a random exponent, sets $\texttt{nym} \leftarrow \delta^r$ where r is taken from computing the random oracle output for the $k' + 1$-th basename. Finally it simulates π and outputs $a, b, c, d, \texttt{nym}, \pi$.

Signing queries with $\mathcal{M}_k$ and later basenames, $\mathcal{S}$ creates signatures using the unknown gsk that it used in the join protocol. It honestly executes the real world protocol, except that it simulates the proof π and the way it computes the pseudonym: It computes $\texttt{nym} \leftarrow \alpha^r$, where r is taken from answering the random oracle query on the basename.

Note that now signatures with the first k' basenames are based on different gsk values for basename, signatures with the $k'+1$-th basename are also based on a fresh gsk if the DDH instance is not a DDH tuple or is based on the gsk from join if it is a DDH tuple, and signatures for all later basenames are based on the gsk from join. Now any distinguisher between GAME 8.k.k' and GAME 8.k.$k'+1$ can solve DDH.

As all outputs are generated by $\mathcal{F}$ now, $\mathcal{S}$ no longer forwards outputs to $\mathcal{F}$ so we can remove the forward output interface of $\mathcal{F}$.

Game 9: $\mathcal{F}$ now no longer informs $\mathcal{S}$ of which message and basename are being signed. When the TPM and Host are both honest, $\mathcal{S}$ does not learn $m, \texttt{bsn}$ to simulate the real world, only leakage $l(m, \texttt{bsn})$. It now chooses $m', \texttt{bsn}'$ such that $l(m, \texttt{bsn}) = l(m', \texttt{bsn}')$ and uses these values to simulate the real world protocol. The host will send $m', \texttt{bsn}', r$ over a secure channel, leaking $l(m', \texttt{bsn}', r)$ to $\mathcal{A}$, but since $l(m, \texttt{bsn}) = l(m', \texttt{bsn}')$, we have $l(m, \texttt{bsn}, r) = l(m', \texttt{bsn}', r)$. The leakage that $\mathcal{A}$ sees is therefore consistent with the input.

Game 10: $\mathcal{F}$ now only allows platforms that joined to sign when $\mathcal{I}$ is honest. This check will not change the view of $\mathcal{E}$, using $\mathcal{S}$ from GAME 9. Before signing with some $\mathcal{M}_i$ in the real world, an honest host will check whether it joined with $\mathcal{M}_i$ and abort otherwise, so for honest hosts there is no difference. An honest TPM $\mathcal{M}_i$ only signs when it has joined with that host, and when an honest $\mathcal{M}_i$ performs the join protocol with a corrupt $\mathcal{H}_j$ and honest $\mathcal{I}$, the simulator will make a join query with $\mathcal{F}$, ensuring that $\mathcal{M}_i$ and $\mathcal{H}_j$ are in $\texttt{Members}$. Since $\mathcal{F}$ still allows any signing that could take place in the real world, GAME 10 $\approx$ GAME 9.

Game 11: When storing a new gsk, $\mathcal{F}$ checks $\mathsf{CheckGskCorrupt}(gsk) = 1$ or $\mathsf{CheckGskHonest}(gsk) = 1$. We now show that these checks will never fail.

Note that we only consider valid signatures from $\texttt{VerResults}$, and $\texttt{Signed}$ only contains valid signatures (added for honest TPM and host) and $\perp$ (added for honest TPM with corrupt host). As $\mathsf{identify}(\perp) = 0$, we only have to consider valid signatures.

Any signature that passes verification has $b \neq 1$, and since $\mathbb{G}_1$ is a prime order group, there is only one $gsk \in \mathbb{Z}_q$ that has $b^{gsk} = d$. From this property, it follows that $\mathsf{CheckGskCorrupt}$ can never fail.

The gsk values that enter $\mathsf{CheckGskHonest}$ are taken uniformly at random from $\mathbb{Z}_q$, which has exponential size, meaning that the probability that one of the existing signatures contains that gsk is negligible. Therefore we have GAME 11 $\approx$ GAME 10.

Game 12: $\mathcal{F}$ now performs some checks on honestly generated signatures. First, it checks that these signatures verify. This check will pass with overwhelming probability: sig creates a valid proof and ensures a, b, c, d have the correct structure such that the pairing tests will pass. Verification can only fail when a or b are $1_{\mathbb{G}_1}$, which happens with negligible probability.

Second, it makes sure $\mathsf{identify}(\sigma, m, \texttt{bsn}, gsk) = 1$. $\mathcal{F}$ running sig sets b, d such that $b^{gsk} = d$, so $\mathsf{identify}(\sigma, m, \texttt{bsn}, gsk) = 1$.

Third, it checks that no honest user is already using gsk. We reduce this check happening with non-negligible probability to solving the DL problem. $\mathcal{F}$ receives an instance $h \in \mathbb{G}_1$ of the DL problem and must answer $log_{g_1}(h)$. Only polynomial many gsk values are created in signing requests, $\mathcal{F}$ chooses one of those at random. Instead of setting $gsk \leftarrow \mathtt{ukgen}$, $\mathcal{F}$ creates a credential on h, determines the $\mathtt{nym}$ using the power $\mathcal{S}$ has over random oracle, and simulates the π. When $\mathcal{F}$ would reuse this key (this happens when $\mathtt{bsn} \neq \bot$), it repeats the same process. When a key matching any of these signatures is found in `Members` or `DomainKeys`, this must be the discrete log of h, as there is only one gsk matching a signature.

As every check passes with overwhelming probability, we have Game 12 $\approx$ Game 11.

Game 13: $\mathcal{F}$ now performs an additional check during verification, it checks whether it finds multiple gsk values matching this signature, and if so, it rejects the signature. We now show that this check does not change the verification outcome, as any signature that would previously pass will still pass.

If the signature would previously pass verification, we have $\mathsf{ver}(\sigma, m, \mathtt{bsn}) = 1$, meaning $b \neq 1_{\mathbb{G}_1}$. As $\mathbb{G}_1$ is a prime order group, there exists only one $gsk \in \mathbb{Z}_q$ with $b^{gsk} = d$. Therefore Game 13 = Game 12.

Game 14: When $\mathcal{I}$ is honest, $\mathcal{F}$ now only accepts credentials on gsk values that $\mathcal{I}$ issued. Under the existential unforgeability of the CL signature, this check changes the verification outcome only with negligible probability.

$\mathcal{C}$ receives a CL public key, which it registers with a simulated proof. When $\mathcal{I}$ must create a credential in the join protocol, it takes the extracted gsk and sends it to the signing oracle and receives a, b, c. It computes $d = b^{gsk}$ and continues as before. $\mathcal{F}$ now uses the signing oracle to create credentials when signing for honest platforms. Note that all the gsk values that $\mathcal{I}$ signs are stored in `Members` or `DomainKeys`.

When a verifier sees a signature $\sigma = (a, b, c, d, \pi, \mathtt{nym})$ such that $d \neq b^{gsk}$ for all gsk values in `Members` and `DomainKeys`, it extracts gsk' from π. By soundness of the proof, a, b, c is a valid CL signature on gsk', allowing $\mathcal{C}$ to win the unforgeability game.

As the CL signature is unforgeable under the LRSW assumption, Game 14 = Game 13.

Game 15: $\mathcal{F}$ now prevents forging signatures using an honest TPM's gsk. We make this change gradually, and in Game 15.i, we do this check for the first $\mathcal{M}_i$ TPMs. We show that any environment able to distinguish Game 15.$i-1$ and Game 15.i can break the DL assumption.

$\mathcal{S}$ receives an DL instance h, and simulates $\mathcal{M}_i$ as follows: it uses $Q \leftarrow h$ in the join protocol to register the unknown discrete logarithm as its gsk value, along with a simulated proof π_1.

It answers H_1 queries by taking $r \in \mathbb{Z}_q$ and returning g_1^r, while maintaining consistency.

Signing queries for $\mathcal{M}_i$ are answered by simulating π, and if $\mathtt{bsn} \neq \perp$ setting $\mathtt{nym} \leftarrow h^r$, where r is taken from the H_1 query on $\mathtt{bsn}$.

In verification, $\mathcal{F}$ now skips the check that at least one pair $(\mathcal{M}_i, gsk_i)$ must be found, as it cannot do this check for $\mathcal{M}_i$.

Only polynomially many verification queries can be made. $\mathcal{F}$ picks one verification query at random, and if it is a valid signature, it extracts gsk from π. With non-negligible probability, this is the discrete logarithm of h.

Game 16: $\mathcal{F}$ now prevents honest TPMs from being revoked. Any environment that can put a gsk on the revocation list that matches an honest TPMs signature can break the DL problem. We show this in two steps: first $\mathcal{F}$ prevents this for pairs $(\mathcal{M}_i, gsk)$ from $\mathtt{Members}$, and after that also for pairs $(\mathcal{M}_i, gsk)$ from $\mathtt{DomainKeys}$.

If this check aborts for a pair found in $\mathtt{Members}$, $\mathcal{E}$ can solve the DL problem. $\mathcal{S}$ receives an instance $h \in \mathbb{G}_1$ of the DL problem and must answer $log_{g_1}(h)$. $\mathcal{S}$ chooses an honest TPM at random and, as described in previous games, simulates this TPM using the unknown discrete logarithm of h as its secret key. When a gsk matching one of this TPMs signatures is found in the revocation list this must be the discrete log of h, as there is only one gsk matching a signature.

If this check aborts for a pair found in $\mathtt{DomainKeys}$, $\mathcal{E}$ can solve the DL problem. $\mathcal{F}$ receives an instance $h \in \mathbb{G}_1$ of the DL problem and must answer $log_{g_1}(h)$. Only polynomial many gsk values are created in signing requests, $\mathcal{F}$ chooses one of those at random. Instead of setting $gsk \leftarrow \mathsf{ukgen}$, $\mathcal{F}$ creates a credential on h, determines the $\mathtt{nym}$ using its power over the random oracle, and simulates the π. When $\mathcal{F}$ would reuse this key (this happens when $\mathtt{bsn} \neq \perp$), it repeats the same process. When a key matching any of these signatures is found in the revocation list this must be the discrete log of h, as there is only one gsk matching a signature.

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

As the signatures already verified, we have $b \neq 1_{\mathbb{G}_1}$, and since $\mathbb{G}_1$ is prime order there is one unique $gsk \in \mathbb{Z}_q$ with $\mathsf{identify}(\sigma, m, \mathtt{bsn}, gsk)$. If one gsk matches one of the signatures but not the other, then by soundness of the proof, $\mathtt{nym} \neq \mathtt{nym}'$ and link would also output 0. If both signatures match some gsk, then by soundness of the proof, we have $\mathtt{nym} = \mathtt{nym}'$ and link would also output 1. Therefore we have Game 17 = Game 16.

The functionality in Game 17 is equal to $\mathcal{F}^{l}_{\mathsf{daa}}$, completing our sequence of games.