Anonymous Attestation with Subverted TPMs [⋆]

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 Zurich, 8092 Zürich, Switzerland

Abstract. Various sources have revealed that cryptographic standards and components have been subverted to undermine the security of users, reigniting research on means to achieve security in presence of such subverted components. In this paper we consider direct anonymous attestation (DAA) in this respect. This standardized protocol allows a computer with the help of an embedded TPM chip to remotely attest that it is in a healthy state. Guaranteeing that different attestations by the same computer cannot be linked was an explicit and important design goal of the standard in order to protect the privacy of the user of the computer. Surprisingly, none of the standardized or otherwise proposed DAA protocols achieves privacy when the TPM is subverted, but they all rely on the honesty of the TPM. As the TPM is a piece of hardware, it is hardly possible to tell whether or not a given TPM follows the specified protocol. In this paper we study this setting and provide a new protocol that achieves privacy also in presence of subverted TPMs.

1 Introduction

Direct anonymous attestation (DAA) is a cryptographic protocol for a platform consisting of a host and a TPM chip (Trusted Platform Module). The TPM serves as a trust anchor of the platform and anonymously attests either to the host's current state or some other message chosen by the host. Thus, DAA can be used to convince a communication partner that the platform has not been compromised, i.e., modified by malware. The main design goal of DAA is that such attestations are anonymous, i.e., while a verifier can check that the signature stems from a legitimate platform, it does not learn the identity of the platform, or even recognize that multiple attestations stem from the same platform.

DAA was introduced by Brickell, Camenisch, and Chen [BCC04] for the Trusted Computing Group and was standardized in the TPM 1.2 specification in 2004 [Tru04]. Their paper inspired a large body of work on DAA schemes [BCL08, CMS08, CF08, BCL09, Che09, CPS10, BL10, BFG$^+$13b, CDL16b, CDL16a], including more efficient schemes using bilinear pairings as well as different security definitions and proofs. One result of these works is the recent TPM 2.0 specification [Tru14, Int15] that includes support for multiple pairing-based DAA schemes, two of which are standardized by ISO [Int13]. Over 500 million TPMs

[⋆] This work has been supported by the ERC under Grant PERCY #321310.

have been sold, [3]making DAA probably the most complex cryptographic scheme that is widely implemented. Recently, the protocol has gotten renewed attention for authentication: An extension of DAA called EPID is used in Intel SGX [CD16], the most recent development in the area of trusted computing. Further, the FIDO alliance, an industry consortium designing standards for strong user authentication, is in the process of standardizing a specification using DAA to attest that authentication keys are securely stored [CDE$^+$].

The first version of the TPM specification and attestation protocol had received strong criticism from privacy groups and data protection authorities as it imposed linkability and full identification of all attestations. As a consequence, guaranteeing the privacy of the platform, i.e., ensuring that an attestation does not carry any identifier, became an important design criteria for such hardware-based attestation. Indeed, various privacy groups and data protection authorities had been consulted in the design process of DAA.

Trusting Hardware for Privacy? Surprisingly, despite the strong concerns of having to trust a piece of hardware when TPMs and hardware-based attestation were introduced, the problem of privacy-preserving attestation in the presence of fraudulent hardware has not been fully solved yet. The issue is that the original DAA protocol as well as all other DAA protocols crucially rely on the honesty of the entire platform, i.e., host and TPM, for guaranteeing privacy. Clearly, assuming that the host is honest is unavoidable for privacy, as it communicates directly with the outside world and can output any identifying information it wants. However, further requiring that the TPM behaves fully honest and aims to preserve the host's privacy is an unnecessarily strong assumption and contradicts the initial design goal of not having to trust the TPM.

Even worse, it is impossible to verify this strong assumption as the TPM is a chip that comes with pre-installed software, to which the user only has black-box access. While black-box access might allow one to partly verify the TPM's functional correctness, it is impossible to validate its *privacy* guarantees. A compromised TPM manufacturer can ship TPMs that provide seemingly correct outputs, but that are formed in a way that allows dedicated entities (knowing some trapdoor) to trace the user, for instance by encoding an identifier in a nonce that is hashed as part of the attestation signature. It could further encode its secret key in attestations, allowing a fraudulent manufacturer to *frame* an honest host by signing a statement on behalf of the platform. We stress that such attacks are possible on all current DAA schemes, meaning that, by compromising a TPM manufacturer, all TPMs it produces can be used as mass surveillance devices. The revelations of subverted cryptographic standards [PLS13, BBG13] and tampered hardware [Gre14] indicate that such attack scenarios are very realistic.

In contrast to the TPM, the host software can be verified by the user, e.g., being compiled from open source, and will likely run on hardware that is not under the control of the TPM manufacturer. Thus, while the honesty of the host

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

is vital for the platform's privacy and there are means to verify or enforce such honesty, requiring the TPM to be honest is neither necessary nor verifiable.

1.1 Our Contribution

In this paper we address this problem of anonymous attestation without having to trust a piece of hardware, a problem which has been open for more than a decade. We further exhibit a new DAA protocol that provides privacy even if the TPM is subverted. More precisely, our contributions are twofold: we first show how to model subverted parties within the Universal Composability (UC) model and then propose a protocol that is secure against subverted TPMs.

Modeling Subversion Attacks in UC. We modify the UC-functionality of DAA recently proposed by Camenisch, Drijvers, and Lehmann [CDL16b] to model the preserved privacy guarantees in the case where the TPM is corrupt and the host remains honest. Modeling corruption in the sense of subverted parties is not straightforward: if the TPM was simply controlled by the adversary, then, using the standard UC corruption model, only very limited privacy can be achieved. The TPM has to see and approve every message it signs but, when corrupted, all these messages are given to the adversary as well. In fact, the adversary will learn which particular TPM is asked to sign which message. That is, the adversary can later recognize a certain TPM attestation via its message, even if the signatures are anonymous.

Modeling corruption of TPMs like this gives the adversary much more power than in reality: even if a TPM is subverted and runs malicious algorithms, it is still embedded into a host who controls all communication with the outside world. Thus, the adversary cannot communicate directly with the TPM, but only via the (honest) host. To model such subversions more accurately, we introduce *isolated* corruptions in UC. When a TPM is corrupted like this, we allow the ideal-world adversary (simulator) to specify a piece of code that the isolated, yet subverted TPM will run. Other than that, the adversary has no control over the isolated corrupted party, i.e., it cannot directly interact with the isolated TPM and cannot see its state. Thus, the adversary will also not automatically learn anymore which TPM signed which message.

A New DAA Protocol with Optimal Privacy. We further discuss why the existing DAA protocols do not offer privacy when the TPM is corrupt and propose a new DAA protocol which we prove to achieve our strong security definition. In contrast to most existing schemes, we construct our protocol from generic building blocks which yields a more modular design. A core building block are *split signatures* which allow two entities – in our case the TPM and host – each holding a secret key share to jointly generate signatures. Using such split keys and signatures is a crucial difference compared with all existing schemes, where only the TPM contributed to the attestation key which inherently limits the possible privacy guarantees. We also redesign the overall protocol such that the main part of the attestation, namely proving knowledge of a membership credential on the attestation key, can be done by the host instead of the TPM.

By shifting more responsibility and computations to the host, we do not only increase privacy, but also achieve stronger notions of non-frameability and unforgeability than all previous DAA schemes. Interestingly, this design change also improves the efficiency of the TPM, which is usually the bottleneck in a DAA scheme. In fact, we propose a pairing-based instantiation of our generic protocol which, compared to prior DAA schemes, has the most efficient TPM signing operation. This comes for the price of higher computational costs for the host and verifier. However, we estimate signing and verification times of around 20ms, which is sufficiently fast for most practical applications.

1.2 Related Work

The idea of combining a piece of tamper-resistant hardware with a user-controlled device was first suggested by Chaum [Cha92] and applied to the context of e-cash by Chaum and Pedersen [CP93], which got later refined by Cramer and Pedersen [CP94] and Brands [Bra94]. A user-controlled wallet is required to work with a piece of hardware, the observer, to be able to withdraw and spend e-cash. The wallet ensures the user's privacy while the observer prevents a user from double-spending his e-cash. Later, Brands in 2000 [Bra00] considered the more general case of user-bound credentials where the user's secret key is protected by a smart card. Brands proposes to let the user's host add randomness to the smart card contribution as a protection against subliminal channels. All these works use a blind signature scheme to issue credentials to the observers and hence such credentials can only be used a single time.

Young and Yung further study the protection against subverted cryptographic algorithms with their work on kleptography [YY97a, YY97b] in the late 1990s. Recently, caused by the revelations of subverted cryptographic standards [PLS13, BBG13] and tampered hardware [Gre14] as a form of mass-surveillance, this problem has again gained substantial attention.

Subversion-Resilient Cryptography. Bellare et al. [BPR14] provided a formalization of algorithm-substitution attacks and considered the challenge of securely encrypting a message with an encryption algorithm that might be compromised. Here, the corruption is limited to attacks where the subverted party's behavior is indistinguishable from that of a correct implementation, which models the goal of the adversary to remain undetected. This notion of algorithm-substitution attacks was later applied to signature schemes, with the goal of preserving unforgeability in the presence of a subverted signing algorithm [AMV15].

However, these works on subversion-resilient cryptography crucially rely on honestly generated keys and aim to prevent key or information leakage when the algorithms using these keys get compromised.

Recently, Russell et al. [RTYZ16a, RTYZ16b] extended this line of work by studying how security can be preserved when *all* algorithms, including the key generation can be subverted. The authors also propose immunization strategies for a number of primitives such as one-way permutations and signature schemes. The approach of replacing a correct implementation with an indistinguishable yet corrupt one is similar to the approach in our work, and like Russell et al. we allow the subversion of all algorithms, and aim for security (or rather privacy) when the TPM behaves maliciously already when generating the keys.

The DAA protocol studied in this work is more challenging to protect against subversion attacks though, as the signatures produced by the TPM must not only be unforgeable and free of a subliminal channel which could leak the signing key, but also be anonymous and unlinkable, i.e., signatures must not leak any information about the signer even when the key is generated by the adversary. Clearly, allowing the TPM to run subverted keys requires another trusted entity on the user's side in order to hope for any privacy-protecting operations. The DAA setting naturally satisfies this requirement as it considers a platform to consist of two individual entities: the TPM and the host, where all of TPM's communication with the outside world is run via the host.

Reverse Firewalls. This two-party setting is similar to the concept of reverse firewalls recently introduced by Mironov and Stephens-Davidowitz [MS15]. A reverse firewall sits in between a user's machine and the outside world and guarantees security of a joint cryptographic operation even if the user's machine has been compromised. Moreover, the firewall-enhanced scheme should maintain the original functionality and security, meaning the part run on the user's computer must be fully functional and secure on its own without the firewall. Thus, the presence of a reverse firewall can enhance security if the machine is corrupt but is not the source of security itself. This concept has been proven very powerful and manages to circumvent the negative results of resilience against subversion-attacks [DMSD16, CMY$^+$16].

The DAA setting we consider in this paper is not as symmetric as a reverse firewall though. While both parties contribute to the unforgeability of attestations, the privacy properties are only achievable if the host is honest. In fact, there is no privacy towards the host, as the host is fully aware of the identity of the embedded TPM. The requirement of privacy-protecting and unlinkable attestation only applies to the final output produced by the host.

Divertible Protocols & Local Adversaries. A long series of related work explores divertible and mediated protocols [BD95, OO90, BBS98, AsV08], where a special party called the mediator controls the communication and removes hidden information in messages by rerandomizing them. The host in our protocol resembles the mediator, as it adds randomness to every contribution to the signature from the TPM. However, in our case the host is a normal protocol participant, whereas the mediator's sole purpose is to control the communication.

Alwen et al. [AKMZ12] and Canetti and Vald [CV12] consider local adversaries to model isolated corruptions in the context of multi-party protocols. These works thoroughly formalize the setting of multi-party computations where several parties can be corrupted, but are controlled by different and non-colluding adversaries. In contrast, the focus of this work is to limit the communication channel that the adversary has to the corrupted party itself. We leverage the flexibility of the UC model to define such isolated corruptions.

Generic MPC. Multi-party computation (MPC) was introduced by Yao [Yao82] and allows a set of parties to securely compute any function on private inputs. Although MPC between the host and TPM could solve our problem, a negative result by Katz and Ostrovsky [KO04] shows that this would require at least five rounds of communication, whereas our tailored solution is much more efficient. Further, none of the existing MPC models considers the type of subverted corruptions that is crucial to our work, i.e., one first would have to extend the existing models and schemes to capture such isolated TPM corruption. This holds in particular for the works that model tamper-proof hardware [Kat07, HPV16], as therein the hardware is assumed to be "perfect" and unsubvertable.

TPM2.0 Interfaces & Subliminal Channels. Camenisch et al. [CCD$^+$17] recently studied the DAA-related interfaces that are provided by hardware modules following the current TPM2.0 specification, and propose a revision to obtain better security and privacy guarantees from such hardware. The current APIs do not allow to prove the unforgeability of the TPM's parts in the DAA protocols, and provide a static Diffie-Hellman oracle. Fixes to these problems have been proposed, but they create new issues: they enable a fraudulent TPM to encode information into an attestation signature, which could be used to break anonymity or to leak the secret key. This creates a subliminal channel already on the hardware level, which would annihilate any privacy guarantees against malicious TPMs that are achieved on the protocol level. Camenisch et al. address this problem and present a revised set of interfaces that allow for provable security and do not introduce a subliminal channel. Further, two new DAA protocols are presented that can be build from these revised APIs and guarantee privacy even when the hardware is subverted, which is termed *strong* privacy and builds upon our isolated corruption model. In contrast to our work, the protocols in [CCD$^+$17] do not provide privacy against malicious TPMs in the standard corruption model, and the privacy guarantees in the isolated model are slightly weaker than in our optimal privacy definition. We give a brief comparison of strong and optimal privacy in Section 2.3 and refer to [CCD$^+$17] for a detailed discussion. The protocols proposed in [CCD$^+$17] are realizable with only minor modifications to the TPM specification, though, whereas our protocol with optimal privacy would require more significant changes.

2 A Security Model for DAA with Optimal Privacy

This section presents our security definition for anonymous attestation with optimal privacy. First, we informally describe how DAA works and what the desired security and (optimal) privacy properties are. Then we present our formal definition in Section 2.1, and describe how it improves upon existing work in Section 2.2. Finally, in Section 2.3, we elaborate on the inherent limitations the

UC framework imposes on privacy in the presence of fully corrupted parties and introduce the concept of *isolated corruptions*, which allow one to overcome this limitations yet capture the power of subverted TPMs.

High-Level Functional and Security Properties. In a DAA scheme, we have four kinds of entities: a number of TPMs, 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 bsn, where the basename steers the platform's anonymity. If a platform signs with a fresh basename, the signature must be anonymous and unlinkable to any 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. However, signatures the platform makes with the *same* basename can be linked to each other via a (deterministic) *link* algorithm.

For security, one requires **unforgeability**: when the issuer is honest, the adversary can only sign in the name of corrupt platforms. More precisely, if n platforms are corrupt, the adversary can forge at most n unlinkable signatures for one basename. By corrupt platform we mean that both the host and TPM are corrupt, and thus a platform is called honest if at least one of the TPM or host is honest. This is in fact stronger than the unforgeability notion covered in all previous definitions which only rely on the honesty of the TPM.

Non-frameability captures the property that no adversary can create signatures on a message m w.r.t. basename bsn that links to a signature created by a platform with an honest host, when this platform never signed m w.r.t. bsn.

Finally, we require **anonymity** for attestations. An adversary that is given two signatures, w.r.t. two *different* basenames cannot determine whether both signatures stem from the same platform. All previous works considered anonymity only for fully honest platforms, i.e., consisting of an honest TPM and honest host, whereas our goal is to guarantee anonymity even if the TPM is corrupt. Note that anonymity can only hold if the host is honest, though, as it has full control over its output and can, e.g., always choose to append its identity to a signature. Thus, the best one can hope for is preserved anonymity when the TPM is corrupt but the host is honest, which is the setting that this work addresses.

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

Informally, a protocol Π securely realizes an ideal functionality $\mathcal{F}$ if the real world is as secure as the ideal world. For every adversary $\mathcal{A}$ attacking the real

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

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

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`.
   - Generate the signature for a fresh or established key: *(strong privacy)*
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor τ' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify & Link

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

Fig. 1. Our ideal functionality $\mathcal{F}_{\mathsf{pdaa}}$ for DAA with optimal privacy.

world, there exists an ideal world attacker or simulator $\mathcal{S}$ that performs an equivalent attack on the ideal world. As $\mathcal{F}$ performs the task at hand in an ideal fashion, i.e., $\mathcal{F}$ is secure by construction, there are no meaningful attacks on the ideal world, so there are no meaningful attacks on the real world. More precisely, Π securely realizes $\mathcal{F}$ if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that no environment $\mathcal{E}$ can distinguish the real world (with Π and $\mathcal{A}$) from the ideal world (with $\mathcal{F}$ and $\mathcal{S}$).

2.1 Ideal Functionality $\mathcal{F}_{\mathsf{pdaa}}$

We now formally define our ideal DAA-with-optimal-privacy functionality $\mathcal{F}_{\mathsf{pdaa}}$, which is based on $\mathcal{F}_{\mathsf{daa}}^{l}$ by Camenisch et al. [CDL16b]. The crucial difference between the two functionalities is the resilience against corrupt TPMs: $\mathcal{F}_{\mathsf{daa}}^{l}$ guarantees anonymity, non-frameability and unforgeability only when both the TPM and the host are honest. Our modified version $\mathcal{F}_{\mathsf{pdaa}}$ guarantees all properties as long as the host is honest, i.e., even when the TPM is corrupt. We explain these differences in detail in Section 2.2. We start by describing the interfaces and guaranteed security properties in an informal manner, and present the detailed definition of $\mathcal{F}_{\mathsf{pdaa}}$ in Figure 1.

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

Note that the ver and link algorithms assist the functionality only for signatures that are not generated by $\mathcal{F}_{\mathsf{pdaa}}$ itself. For signatures generated by the functionality, $\mathcal{F}_{\mathsf{pdaa}}$ will enforce correct verification and linkage using its internal records. While 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 $\mathsf{link}(\sigma, m, \sigma', m', bsn) \leftrightarrow \mathsf{link}(\sigma', m', \sigma, m, bsn)$.

Join. A host $\mathcal{H}_j$ can request to join with a TPM $\mathcal{M}_i$ using the JOIN interface. If both the TPM and the issuer approve the join request, the functionality stores an internal membership record for $\mathcal{M}_i, \mathcal{H}_j$ in $\mathtt{Members}$ indicating that from now on that platform is allowed to create attestations.

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

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

Sign. After joining, a host $\mathcal{H}_j$ can request a signature on a message m with respect to basename bsn using the SIGN interface. The signature will only be created when the TPM $\mathcal{M}_i$ explicitly agrees to signing m w.r.t. bsn and a join record for $\mathcal{M}_i, \mathcal{H}_j$ in `Members` exists (if the issuer is honest).

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

Our functionality enforces this privacy property whenever the host is honest. Note, however, that $\mathcal{F}_{\mathsf{pdaa}}$ does not behave differently when the host is corrupt, as in this case its output does not matter due to way corruptions are handled in UC. That is, $\mathcal{F}_{\mathsf{pdaa}}$ always outputs anonymous signatures to the host, but if the host is corrupt, the signature is given to the adversary, who can choose to discard it and output anything else instead.

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

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

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

Finally, for ensuring unforgeability, the signed message, basename, and platform are stored in `Signed` which will be used when verifying signatures.

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

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

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

The full definition of $\mathcal{F}_{\mathsf{pdaa}}$ is given in Figure 1. Note that when $\mathcal{F}_{\mathsf{pdaa}}$ 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.

2.2 Comparison with $\mathcal{F}_{\mathsf{daa}}^{l}$

Our functionality $\mathcal{F}_{\mathsf{pdaa}}$ is a strengthened version of $\mathcal{F}_{\mathsf{daa}}^{l}$ [CDL16b], as it requires fewer trust assumptions on the TPM for anonymity, non-frameability and unforgeability. It also includes a syntactical change which allows for more efficient constructions, as we discuss at the end of this section.

Optimal Privacy. The most important difference is that $\mathcal{F}_{\mathsf{daa}}^{l}$ guarantees anonymity only when both the TPM and the host are honest, whereas our modified version $\mathcal{F}_{\mathsf{pdaa}}$ guarantees anonymity as long as the host is honest, i.e., even when the TPM is corrupt. As discussed, the honesty of the host is strictly necessary, as privacy is impossible to guarantee otherwise.

In the ideal functionality $\mathcal{F}_{\mathsf{daa}}^{l}$ proposed by Camenisch et al. [CDL16b] the signatures are created in the $\mathsf{SIGNPROCEED}$ step in two different ways, depending on whether the TPM is honest or not. For the case of a corrupt TPM, the

Corruption Setting	$\mathcal{F}^l_{\mathsf{daa}}$	$\mathcal{F}_{\mathsf{pdaa+}}$	$\mathcal{F}_{\mathsf{pdaa}}$	
Honest host, honest TPM	+	+	+	
Honest host, isolated corrupt TPM	-	(+)	+	*optimal privacy*
Honest host, fully corrupt TPM	-	-	(+)	*conditional privacy*
Corrupt host	-	-	-	*impossible*

Table 1. Overview of privacy guarantees by $\mathcal{F}^l_{\mathsf{daa}}$ [CDL16b], $\mathcal{F}_{\mathsf{pdaa+}}$ [CCD⁺17] and $\mathcal{F}_{\mathsf{pdaa}}$ (this work).

signature is provided by the adversary, which reflects that the adversary can recognize and link the signatures and $\mathcal{F}^l_{\mathsf{daa}}$ does not guarantee any privacy. If the TPM (and the host) is honest, $\mathcal{F}^l_{\mathsf{daa}}$ creates anonymous signatures inside the functionality using the signing algorithm sig and ukgen. As signatures are generated with fresh keys for every new basename, the functionality enforces the desired unlinkability and anonymity.

In our functionality $\mathcal{F}_{\mathsf{pdaa}}$, we also apply that approach of internally and anonymously creating signatures to the case where the TPM is corrupt, instead of relying on a signature input by the adversary. Thus, $\mathcal{F}_{\mathsf{pdaa}}$ guarantees the same strong privacy for both settings of a corrupt and honest TPM. In fact, for the sake of simplicity we let $\mathcal{F}_{\mathsf{pdaa}}$ even generate the signatures for corrupt hosts within the functionality now (whereas $\mathcal{F}^l_{\mathsf{daa}}$ used adversarially provided ones). However, as $\mathcal{F}_{\mathsf{pdaa}}$ outputs that signature to the host $\mathcal{H}_i$, who will be the adversary if $\mathcal{H}_i$ is corrupt, the behaviour of $\mathcal{F}_{\mathsf{pdaa}}$ with respect to privacy does not matter in that case: the adversary can simply ignore the output. We present a summary of the privacy properties guaranteed by $\mathcal{F}^l_{\mathsf{daa}}$ and $\mathcal{F}_{\mathsf{pdaa}}$ in Table 1.

Another difference between both functionalities is that in $\mathcal{F}_{\mathsf{pdaa}}$ we assume a direct communication channel between the host and TPM, which is necessary to achieve the desired privacy properties (see Section 2.3). Note that in the real-world, such a direct channel is naturally enforced 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, or even notice that communication is happening. Consequently, our functionality gets a bit simpler compared to $\mathcal{F}^l_{\mathsf{daa}}$ as we omit in JOIN and SIGN all dedicated interfaces and outputs that informed the simulator about communication between $\mathcal{H}_j$ and $\mathcal{M}_i$ and waited for a proceed input by the simulator to complete their communication.

Stronger Non-Frameability and Unforgeability. While the focus of this work is strengthening the privacy properties in the presence of a subverted TPM, we also lift the trust assumption for non-frameability and unforgeability. Whereas $\mathcal{F}^l_{\mathsf{daa}}$ and all other prior security models [BCC04, BCL09] guarantee non-frameability only if the entire platform is honest, our modified definition $\mathcal{F}_{\mathsf{pdaa}}$ enforces that property as long as the host is honest. Our stronger version of non-frameability is enforced by modifying the $\mathsf{JOINPROCEED}$ interface such that it allows the adversary to provide a tracing trapdoor τ (which steers the non-frameability checks by $\mathcal{F}_{\mathsf{pdaa}}$) only when the host is corrupt, as it sets $\tau \leftarrow \perp$ whenever the host is honest. This replaces the original condition of discarding the adversarial τ when both, the host and TPM are honest. Note that similar to anonymity, requiring an honest host is strictly necessary for non-frameability too, as we can never control the signatures that a corrupt host outputs. In particular, a corrupt host with an honest TPM could additionally run a corrupt TPM and "frame itself" by outputting signatures from the corrupt TPM.

In terms of unforgeability, all previous definitions including $\mathcal{F}_{\mathsf{daa}}^{l}$ solely rely on the honesty of the TPM (and issuer of course). In $\mathcal{F}_{\mathsf{pdaa}}$ we provide a stronger version and guarantee that attestations cannot be forged unless the entire platform is corrupted, i.e., here we ensure unforgeability if at least one of two entities, TPM or host, is honest. This change is reflected in our functionality $\mathcal{F}_{\mathsf{pdaa}}$ as follows: In the SIGNPROCEED interface we store the host identity as part of the signature record $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j \rangle \in \mathtt{Signed}$ when signatures are created. Further, the VERIFY interface now requires the existence of such record whenever the signature to be verified belongs to an honest host or honest TPM. In $\mathcal{F}_{\mathsf{daa}}^{l}$ only $\langle \sigma, m, bsn, \mathcal{M}_i \rangle$ was stored and required when the TPM was honest. For unforgeability, relaxing the condition on the honesty of the TPM is not as crucial as for privacy and non-frameability. Thus, if only the standard unforgeability notion is sufficient, one can easily derive a functionality with optimal privacy but standard unforgeability by reverting the changes we just described.

Dedicated Tracing Key. Our functionality also includes some syntactical changes. $\mathcal{F}_{\mathsf{daa}}^{l}$ uses keys gsk for two purposes: to create signatures for honest platforms (via sig), and to trace signatures (via identify) when enforcing non-frameability and unforgeability. A key gsk can be provided by the adversary when a JOIN request is completed for a corrupt host, or is generated internally via ukgen whenever an anonymous signature is created. In $\mathcal{F}_{\mathsf{pdaa}}$ we split this into two dedicated values: gsk which is used to sign, and τ to trace signatures. Consequently, the identify algorithm now takes τ instead of gsk as input. The adversary has to provide τ in the JOIN interface, as its input is only used to ensure that a corrupt host cannot impersonate or frame another honest platform. The internally created keys are used for both, signing and tracing, and hence we modify ukgen to output a tuple (gsk, τ) instead of gsk only.

The idea behind that change is to allow for more efficient schemes, as the tracing key τ is usually a value that needs to be extracted by the simulator in the security proof. In the scheme we propose, it is sufficient that τ is the public key of the platform whereas gsk is its secret key. Using only a single gsk would have required the join protocol to include an extractable encryption of the platform's secret key, which would not only be less efficient but also a questionable protocol design. Clearly, our approach is more general than in $\mathcal{F}_{\mathsf{daa}}^{l}$, one can simply set $\tau = gsk$ to derive the same definition as $\mathcal{F}_{\mathsf{daa}}^{l}$.

2.3 Modeling Subverted Parties in the UC Framework

As just discussed, our functionality $\mathcal{F}_{\mathsf{pdaa}}$ guarantees that signatures created with an honest host are unlinkable and do not leak any information about the signing platform, even if the TPM is corrupt. However, the adversary still learns the message and basename when the TPM is corrupt, due to the way UC models corruptions. We discuss how this standard corruption model inherently limits the achievable privacy level, and then present our approach of isolated corruptions which allow one to overcome this limitation yet capture the power of subverted TPMs. While we discuss the modeling of isolated corruptions in the context of our DAA functionality, we consider the general concept to be of independent interest as it is applicable to any other scenario where such subversion attacks can occur.

Conditional Privacy under Full TPM Corruption. According to the UC corruption model, the adversary gains full control over a corrupted party, i.e., it receives all inputs to that party and can choose its responses. For the case of a corrupt TPM this means that the adversary sees the message m and basename bsn whenever the honest host wants to create a signature. In fact, the adversary will learn which particular TPM $\mathcal{M}_i$ is asked to sign m w.r.t. bsn. Thus, even though the signature σ on m w.r.t. bsn is then created by $\mathcal{F}_{\mathsf{pdaa}}$ and does not leak any information about the identity of the signing platform, the adversary might still be able to recognize the platform's identity via the signed values. That is, if a message m or basename bsn is unique, i.e., only a single (and corrupt) TPM has ever signed m w.r.t. bsn, then, when later seeing a signature on m w.r.t. bsn, the adversary can derive which platform had created the signature.

A tempting idea for better privacy would be to change the functionality such that the TPM does not receive the message and basename when asked to approve an attestation via the SIGNPROCEED message. As a result, this information will not be passed to the adversary if the TPM is corrupt. However, that would completely undermine the purpose of the TPM that is supposed to serve as a trust anchor: verifiers accept a DAA attestation because they know a trusted TPM has approved them. Therefore, it is essential that the TPM sees and acknowledges the messages it signs.

Thus, in the presence of a fully corrupt TPM, the amount of privacy that can be achieved depends which messages and basenames are being signed – the more unique they are, the less privacy $\mathcal{F}_{\mathsf{pdaa}}$ guarantees.

Optimal Privacy under *Isolated* TPM Corruption. The aforementioned leakage of all messages and basenames that are signed by a corrupt TPM is enforced by the UC corruption model. Modeling corruption of TPMs like this gives the adversary much more power than in reality: even if a TPM is subverted and runs malicious algorithms, it is still embedded into a host who controls all communication with the outside world. Thus, the adversary cannot communicate directly with the TPM, but only via the (honest) host.

To model such subversions more accurately and study the privacy achievable in the presence of subverted TPMs, we define a relaxed level of corruption that we call *isolated corruption*. When the adversary corrupts a TPM in this manner, it can specify code for the TPM but cannot directly communicate with the TPM.

We formally define such isolated corruptions via the body-shell paradigm used to model UC corruptions [Can00]. Recall that the body of a party defines

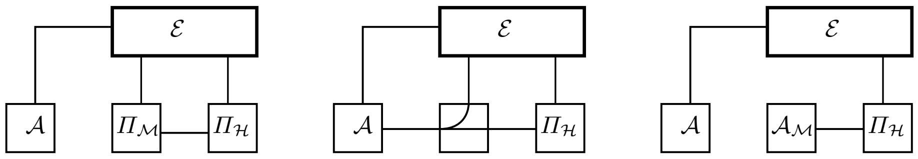

Fig. 2. Modeling of corruption in the real world. Left: an honest TPM applies the protocol $\Pi_{\mathcal{M}}$, and communicates with the host running $\Pi_{\mathcal{H}}$. Middle: a corrupt TPM sends any input the adversary instructs it to, and forwards any messages received to the adversary. Right: an isolated corrupt TPM is controlled by an isolated adversary $\mathcal{A}_{\mathcal{M}}$, who can communicate with the host, but not with any other entities.

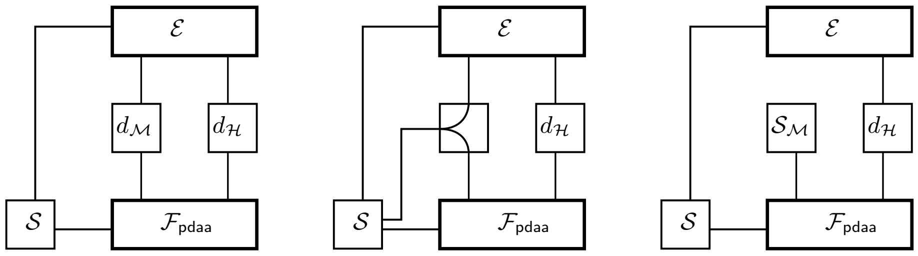

Fig. 3. Modeling of corruption in the ideal world. Left: an honest TPM is a dummy party $d_{\mathcal{M}}$ that forwards inputs and outputs between the environment $\mathcal{E}$ and the functionality $\mathcal{F}_{\mathsf{pdaa}}$. Middle: a corrupt TPM sends any input the adversary instructs it to, and forwards any subroutine output to the adversary. Right: an isolated corrupt TPM is controlled by an isolated simulator $\mathcal{S}_{\mathcal{M}}$, who may send inputs and receive outputs from $\mathcal{F}_{\mathsf{pdaa}}$, but not communicate with any other entities.

its behavior, whereas the shell models the communication with that party. Thus, for our isolated corruptions, the adversary gets control over the body but not the shell. Interestingly, this is exactly the inverse of honest-but-curious corruptions in UC, where the adversary controls the shell and thus sees all inputs and outputs, but cannot change the body, i.e., the parties behavior remains honest.

In our case, an adversary performing an isolated corruption can provide a body, which models the tampered algorithms that an isolated corrupt TPM may use. The shell remains honest though and handles inputs, and subroutine outputs, and only forwards the ones that are allowed to the body. In the real world, the shell would only allow communication with the host in which the TPM is embedded. In the ideal world, the shell allows inputs to and outputs from the functionality, and blocks anything else.

Figure 2 and Figure 3 depict the different levels of corruption in the real world and ideal world, respectively. In the ideal word, an isolated corruption of a TPM replaces the dummy TPM that forwards inputs and outputs between the environment and the ideal functionality with an *isolated simulator* comprising of the adversarial body and honest shell.

When designing a UC functionality, then all communication between a host and the "embedded" party that can get corrupted in such isolated manner must be modeled as direct channel (see e.g., the SIGN related interfaces in $\mathcal{F}_{\mathsf{pdaa}}$). Otherwise the simulator/adversary will be aware of the communication between both parties and can delay or block messages, which would contradict the concept of an isolated corruption where the adversary has no direct channel to the embedded party. Note that the perfect channel of course only holds if the host entity is honest, if it is corrupt (in the standard sense), the adversary can see and control all communication via the host anyway.

With such isolated adversaries we specify much stronger privacy. The adversary no longer automatically learns which isolated corrupt TPM signed which combination of messages and basenames, and the signatures created by $\mathcal{F}_{\mathsf{pdaa}}$ are guaranteed to be unlinkable. Of course the message m and basename bsn must not leak information about the identity of the platform. In certain applications, the platform would sign data generated or partially controlled by other functions contained in a TPM. This is out of scope of the attestation scheme, but the higher level scheme using $\mathcal{F}_{\mathsf{pdaa}}$ should ensure that this does not happen, by, e.g., letting the host randomize or sanitize the message.

Comparison with Strong Privacy ($\mathcal{F}_{\mathsf{pdaa+}}$). Recently, Camenisch et al. [CCD+17] proposed a variant $\mathcal{F}_{\mathsf{pdaa+}}$ of our functionality that, when considering only isolated TPM corruptions, provides an intermediate level of anonymity, termed *strong privacy* (the + in $\mathcal{F}_{\mathsf{pdaa+}}$ refers to the addition of attributes and signature-based revocation). In $\mathcal{F}_{\mathsf{pdaa+}}$ all signatures are generated internally by the functionally, just as in optimal privacy. The difference is that in strong privacy these signatures are revealed to the TPM which can then base its behavior on the signature value. Thus, while the actual signature shown to the TPM is still guaranteed to be anonymous, the TPM can influence the final distribution of the signatures by blocking certain values. In the isolated corruption model, where the corrupt TPM cannot communicate the learned signatures to the adversary, $\mathcal{F}_{\mathsf{pdaa+}}$ provides an interesting relaxation of optimal privacy which allows for significantly simpler constructions as shown in [CCD+17].

3 Insufficiency of Existing DAA Schemes

Our functionality $\mathcal{F}_{\mathsf{pdaa}}$ requires all signatures on a message m with a fresh basename bsn to have the same distribution, even when the TPM is corrupt. None of the existing DAA schemes can be used to realize $\mathcal{F}_{\mathsf{pdaa}}$ when the TPM is corrupted (either fully or isolated). The reason is inherent to the common protocol design that underlies all DAA schemes so far, i.e., there is no simple patch that would allow upgrading the existing solutions to achieve optimal privacy.

In a nutshell, in all existing DAA schemes, the TPM chooses a secret key gsk for which it blindly receives a membership credential of a trusted issuer. To create a signature on message m with basename bsn, the platform creates a signature proof of knowledge signing message m and proving knowledge of gsk and the membership credential.

In the original RSA-based DAA scheme [BCC04], and the more recent qSDH-based schemes [CF08, BL11, BL10, CDL16a], the proof of knowledge of the membership credential is created jointly by the TPM and host. After jointly computing the commitment values of the proof, the host computes the hash over these values and sends the hash c to the TPM. To prevent leaking information about its key, the TPM must ensure that the challenge is a hash of fresh values. In all the aforementioned schemes this is done by letting the TPM choose a fresh nonce n and computing the final hash as $c' \leftarrow \mathsf{H}(n, c)$. An adversarial TPM can embed information in n instead of taking it uniformly at random, clearly altering the distribution of the proof and thus violating the desired privacy guarantees.

At a first glance, deriving the hash for the proof in a more robust manner might seem a viable solution to prevent such leakage. For instance, setting the nonce as $n \leftarrow n_t \oplus n_h$, with n_t being the TPM's and n_h the host's contribution, and letting the TPM commit to n_t before receiving n_h. While this indeed removes the leakage via the nonce, it still reveals the hash value $c' \leftarrow \mathsf{H}(n, c)$ to the TPM with the hash becoming part of the completed signature. Thus, the TPM can base its behavior on the hash value and, e.g., only sign messages for hashes that start with a 0-bit. When considering only isolated corruptions for the TPM, the impact of such leakage is limited though as argued by Camenisch et al. [CCD+17] and formalized in their notion of strong privacy. In fact, Camenisch et al. show that by using such jointly generated nonces, and also letting the host contribute to the platform's secret key, the existing DAA schemes can be modified to achieve strong privacy in the isolated corruption model. However, it clearly does not result in signatures that are equally distributed as required by our functionality, and thus the approach is not sufficient to obtain *optimal* privacy.

The same argument applies to the existing LRSW-based DAA schemes [CPS10, BFG+13b, CDL16b], where the proof of a membership credential is done solely by the TPM, and thus can leak information via the Fiat-Shamir hash output again. The general problem is that the signature proofs of knowledge are not randomizable. If the TPM would create a randomizable proof of knowledge, e.g., a Groth-Sahai proof [GS08], the host could randomize the proof to remove any hidden information, but this would yield a highly inefficient signing protocol for the TPM.

4 Building Blocks

In this section we introduce the building blocks for our DAA scheme. In addition to standard components such as additively homomorphic encryption and zero-knowledge proofs, we introduce two non-standard types of signature schemes. One signature scheme we require is for the issuer to blindly sign the public key of the TPM and host. The second signature scheme is needed for the TPM and host to jointly create signed attestations, which we term *split signatures*.

The approach of constructing a DAA scheme from modular building blocks rather than basing it on a concrete instantiation was also used by Bernhard et al. [BFG+13b, BFG13a]. As they considered a simplified setting, called pre-DAA, where the host and platform have a joint corruption state, and we aim for much stronger privacy, their "linkable indistinguishable tag" is not sufficient for our construction. We replace this with our split signatures.

As our protocol requires "compatible" building blocks, i.e., the different schemes have to work in the same group, we assume the availability of public system parameters $spar \xleftarrow{\$} \mathsf{SParGen}(\tau)$ generated for security parameter τ. We give *spar* as dedicated input to the individual key generation algorithms instead of the security parameter τ. For the sake of simplicity, we omit the system parameters as dedicated input to all other algorithms and assume that they are given as implicit input.

4.1 Proof Protocols

Let $NIZK\{(w) : s(w)\}(ctxt)$ denote a generic non-interactive zero-knowledge proof that is bound to a certain context *ctxt* and proves knowledge of a witness w such that statement $s(w)$ is true. Sometimes we need witnesses to be online-extractable, which we denote by underlining them: $NIZK\{(\underline{w_1}, w_2) : s(w_1, w_2)\}$ allows for online extraction of w_1.

All the *NIZK* we give have efficient concrete instantiations for the instantiations we propose for our other building blocks. We will follow the notation introduced by Camenisch and Stadler [CS97] and formally defined by Camenisch, Kiayias, and Yung [CKY09] for these protocols. For instance, $PK\{(a) : y = g^a\}$ denotes a "*zero-knowledge Proof of Knowledge of integer a such that* $y = g^a$ *holds.*" $SPK\{\ldots\}(m)$ denotes a signature proof of knowledge on m, that is a non-interactive transformation of a proof with the Fiat-Shamir heuristic [FS87].

4.2 Homomorphic Encryption Schemes

We require an encryption scheme $(\mathsf{EncKGen}, \mathsf{Enc}, \mathsf{Dec})$ that is semantically secure and that has a cyclic group $\mathbb{G} = \langle g \rangle$ of order q as message space. It consists of a key generation algorithm $(epk, esk) \xleftarrow{\$} \mathsf{EncKGen}(spar)$, where *spar* defines the group $\mathbb{G}$, an encryption algorithm $C \xleftarrow{\$} \mathsf{Enc}(epk, m)$, with $m \in \mathbb{G}$, and a decryption algorithm $m \leftarrow \mathsf{Dec}(esk, C)$.

We further require that the encryption scheme has an appropriate *homomorphic property*, namely that there is an efficient operation $\odot$ on ciphertexts such that, if $C_1 \in \mathsf{Enc}(epk, m_1)$ and $C_2 \in \mathsf{Enc}(epk, m_2)$, then $C_1 \odot C_2 \in \mathsf{Enc}(epk, m_1 \cdot m_2)$. We will also use exponents to denote the repeated application of $\odot$, e.g., C^2 to denote $C \odot C$.

ElGamal Encryption. We use the ElGamal encryption scheme [ElG86], which is homomorphic and chosen plaintext secure. The semantic security is sufficient for our construction, as the parties always prove to each other that they formed the ciphertexts correctly. Let *spar* define a group $\mathbb{G} = \langle g \rangle$ of order q such that the DDH problem is hard w.r.t. τ, i.e., q is a τ-bit prime.

$\mathsf{EncKGen}(spar)$: Pick $x \xleftarrow{\$} \mathbb{Z}_q$, compute $y \leftarrow g^x$, and output $esk \leftarrow x$, $epk \leftarrow y$.

$\mathsf{Enc}(epk, m):$ To encrypt a message $m \in \mathbb{G}$ under $epk = y$, pick $r \xleftarrow{\$} \mathbb{Z}_q$ and output the ciphertext $(C_1, C_2) \leftarrow (y^r, g^r m)$.

$\mathsf{Dec}(esk, C):$ On input the secret key $esk = x$ and a ciphertext $C = (C_1, C_2) \in \mathbb{G}^2$, output $m' \leftarrow C_2 \cdot C_1^{-1/x}$.

4.3 Signature Schemes for Encrypted Messages

We need a signature scheme that supports the signing of encrypted messages and must allow for (efficient) proofs proving that an encrypted value is correctly signed and proving knowledge of a signature that signs an encrypted value. Dual-mode signatures [CL15] satisfy these properties, as therein signatures on plaintext as well as on encrypted messages can be obtained. As we do not require signatures on plaintexts, though, we can use a simplified version.

A signature scheme for encrypted messages consists of the algorithms $(\mathsf{SigKGen}, \mathsf{EncSign}, \mathsf{DecSign}, \mathsf{Vf})$ and also uses an encryption scheme $(\mathsf{EncKGen}, \mathsf{Enc}, \mathsf{Dec})$ that is compatible with the message space of the signature scheme. In particular, the algorithms working with encrypted messages or signatures also get the keys $(epk, esk) \xleftarrow{\$} \mathsf{EncKGen}(spar)$ of the encryption scheme as input.

$\mathsf{SigKGen}(spar):$ On input the system parameters, this algorithm outputs a public verification key spk and secret signing key ssk.

$\mathsf{EncSign}(ssk, epk, C):$ On input signing key ssk, a public encryption key epk, and ciphertext $C = \mathsf{Enc}(epk, m)$, outputs an "encrypted" signature $\overline{\sigma}$ of C.

$\mathsf{DecSign}(esk, spk, \overline{\sigma}):$ On input an "encrypted" signature $\overline{\sigma}$, secret decryption key esk and public verification key spk, outputs a standard signature σ.

$\mathsf{Vf}(spk, \sigma, m):$ On input a public verification key spk, signature σ and message m, outputs 1 if the signature is valid and 0 otherwise.

For correctness, we require that any message encrypted with honestly generated keys that is honestly signed decrypts to a valid signature. More precisely, for any message m, we require

$$\begin{aligned}\mathsf{Pr}\Big[\mathsf{Vf}(spk, \sigma, m) = 1 \ \mid\ & spar \leftarrow \mathsf{SParGen}(\tau), (spk, ssk) \xleftarrow{\$} \mathsf{SigKGen}(spar), \\ & (epk, esk) \leftarrow \mathsf{EncKGen}(spar), C \leftarrow \mathsf{Enc}(spar, epk, m), \\ & \bar{\sigma} \leftarrow \mathsf{EncSign}(ssk, epk, c), \sigma \leftarrow \mathsf{DecSign}(esk, spk, \bar{\sigma})\Big].\end{aligned}$$

We use the unforgeability definition of [CL15], but omit the oracle for signatures on plaintext messages. Note that the oracle $\mathcal{O}^{\mathsf{EncSign}}$ will only sign correctly computed ciphertexts, which is modeled by providing the message and public encryption key as input and let the oracle encrypt the message itself before signing it. When using the scheme, this can easily be enforced by asking the signature requester for a proof of correct ciphertext computation, and, indeed, in our construction such a proof is needed for other reasons as well.

```
Experiment $\mathsf{Exp}^{\mathsf{ESIG\text{-}forge}}_{\mathcal{A},\mathsf{ESIG},\mathsf{Enc}}(\mathbb{G},\tau)$:
  $spar \leftarrow \mathsf{SParGen}(\tau)$
  $(spk, ssk) \xleftarrow{\$} \mathsf{SigKGen}(spar)$
  $\mathbf{L} \leftarrow \emptyset$
  $(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{EncSign}(ssk,\cdot,\cdot)}}(spar, spk)$
      where $\mathcal{O}^{\mathsf{EncSign}}$ on input $(epk_i, m_i)$:
          add $m_i$ to the list of queried messages $\mathbf{L} \leftarrow \mathbf{L} \cup m_i$
          compute $C_i \xleftarrow{\$} \mathsf{Enc}(epk_i, m_i)$
          return $\overline{\sigma} \xleftarrow{\$} \mathsf{EncSign}(ssk, epk_i, C_i)$
  return 1 if $\mathsf{Vf}(spk, \sigma^*, m^*) = 1$ and $m^* \notin \mathbf{L}$
```

Fig. 4. Unforgeability experiment for signatures on encrypted messages.

Definition 1. (Unforgeability of Signatures for Encrypted Messages). *We say a signature scheme for encrypted messages is* unforgeable *if for any efficient algorithm $\mathcal{A}$ the probability that the experiment given in Figure 9 returns 1 is negligible (as a function of τ).*

AGOT+ Signature Scheme. To instantiate the building block of signatures for encrypted messages we will use the AGOT+ scheme of [CL15], which was shown to be a secure instantiation of a dual-mode signature, hence is also secure in our simplified setting. Again, as we do not require signatures on plaintext messages we omit the standard signing algorithm. The AGOT+ scheme is based on the structure-preserving signature scheme by Abe et al. [AGOT14], which is proven to be unforgeable in the generic group model.

The AGOT+ scheme assumes the availability of system parameters $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, x)$, where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are groups of prime order q generated by g_1, g_2, and $e(g_1, g_2)$ respectively, e is a non-degenerate bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, and x is an additional random group element in $\mathbb{G}_1$.

$\mathsf{SigKGen}(spar)$: Draw $v \xleftarrow{\$} \mathbb{Z}_q$, compute $y \leftarrow g_2^v$, and return $spk = y$, $ssk = v$.

$\mathsf{EncSign}(ssk, epk, M)$: On input a proper encryption $M = \mathsf{Enc}(epk, m)$ of a message $m \in \mathbb{G}_1$ under epk, and secret key $ssk = v$, choose a random $u \xleftarrow{\$} \mathbb{Z}_q^*$, and output the (partially) encrypted signature $\overline{\sigma} = (r, S, T, w)$:

$$r \leftarrow g_2^u,\ \ S \leftarrow (M^v \odot \mathsf{Enc}(epk, x))^{1/u},\ \ T \leftarrow (S^v \odot \mathsf{Enc}(epk, g_1))^{1/u},\ \ w \leftarrow g_1^{1/u}.$$

$\mathsf{DecSign}(esk, spk, \overline{\sigma})$: Parse $\overline{\sigma} = (r, S, T, w)$, compute $s \leftarrow \mathsf{Dec}(esk, S)$, $t \leftarrow \mathsf{Dec}(esk, T)$ and output $\sigma = (r, s, t, w)$.

$\mathsf{Vf}(spk, \sigma, m)$: Parse $\sigma = (r, s, t, w')$ and $spk = y$ and output 1 iff $m, s, t \in \mathbb{G}_1$, $r \in \mathbb{G}_2$, $e(s, r) = e(m, y) \cdot e(x, g_2)$, and $e(t, r) = e(s, y) \cdot e(g_1, g_2)$.

Note that for notational simplicity, we consider w part of the signature, i.e., $\sigma = (r, s, t, w)$, altough signature verification will ignore w. As pointed out by Abe et al., a signature $\sigma = (r, s, t)$ can be randomized using the randomization token w to obtain a signature $\sigma' = (r', s', t')$ by picking a random $u' \xleftarrow{\$} \mathbb{Z}_q^*$ and computing $r' \leftarrow r^{u'}, \quad s' \leftarrow s^{1/u'}, \quad t' \leftarrow (tw^{(u'-1)})^{1/u'^2}$.

For our construction, we also require the host to prove that it knows an encrypted signature on an encrypted message. In Section 6 we describe how such a proof can be done.

4.4 Split Signatures

The second signature scheme we require must allow two different parties, each holding a share of the secret key, to jointly create signatures. Our DAA protocol performs the joined public key generation and the signing operation in a strict sequential order. That is, the first party creates his part of the key, and the second party receiving the 'pre-public key' generates a second key share and completes the joined public key. Similarly, to sign a message the first signer creates a 'pre-signature' and the second signer completes the signature. We model the new signature scheme for that particular sequential setting rather than aiming for a more generic building block in the spirit of threshold or multi-signatures, as the existence of a strict two-party order allows for substantially more efficient constructions.

We term this new building block *split signatures* partially following the notation by Bellare and Sandhu [BS01] who formalized different two-party settings for RSA-based signatures where the signing key is split between a client and server. Therein, the case "MSC" where the first signature contribution is produced by an external server and then completed by the client comes closest to out setting.

Formally, we define a split signature scheme as a tuple of the algorithms $\mathsf{SSIG} = (\mathsf{PreKeyGen}, \mathsf{CompleteKeyGen}, \mathsf{VerKey}, \mathsf{PreSign}, \mathsf{CompleteSign}, \mathsf{Vf})$:

$\mathsf{PreKeyGen}(spar)$: On input the system parameters, this algorithm outputs the pre-public key ppk and the first share of the secret signing key ssk_1.

$\mathsf{CompleteKeyGen}(ppk)$: On input the pre-public key, this algorithm outputs a public verification key spk and the second secret signing key ssk_2.

$\mathsf{VerKey}(ppk, spk, ssk_2)$: On input the pre-public key ppk, the full public key spk, and a secret key share ssk_2, this algorithm outputs 1 iff the pre-public key combined with secret key part ssk_2 leads to full public key spk.

$\mathsf{PreSign}(ssk_1, m)$: On input a secret signing key share ssk_1, and message m, this algorithm outputs a pre-signature σ'.

$\mathsf{CompleteSign}(ppk, ssk_2, m, \sigma')$: On input the pre-public key ppk, the second signing key share ssk_2, message m, and pre-signature σ', this algorithm outputs the completed signature σ.

$\mathsf{Vf}(spk, \sigma, m)$: On input the public key spk, signature σ, and message m, this algorithm outputs a bit b indicating whether the signature is valid or not.

We require a number of security properties from our split signatures. The first one is unforgeability which must hold if at least one of the two signers is honest. This is captured in two security experiments: type-1 unforgeability allows the

```
Experiment $\mathsf{Exp}^{\mathsf{Unforgeability\text{-}1}}_{\mathcal{A}}(\tau)$:
  $spar \xleftarrow{\$} \mathsf{SParGen}(1^\tau)$
  $(ppk, state) \leftarrow \mathcal{A}(spar)$
  $(spk, ssk_2) \leftarrow \mathsf{CompleteKeyGen}(ppk)$
  $\mathbf{L} \leftarrow \emptyset$
  $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{CompleteSign}(ppk, ssk_2, \cdot, \cdot)}}(state, spk)$
    where $\mathcal{O}^{\mathsf{CompleteSign}}$ on input $(m_i, \sigma'_i)$:
      set $\mathbf{L} \leftarrow \mathbf{L} \cup m_i$
      return $\sigma_i \leftarrow \mathsf{CompleteSign}(ppk, ssk_2, m_i, \sigma'_i)$
  return 1 if $\mathsf{Vf}(spk, \sigma^*, m^*) = 1$ and $m^* \notin \mathbf{L}$
```

```
Experiment $\mathsf{Exp}^{\mathsf{Unforgeability\text{-}2}}_{\mathcal{A}}(\tau)$:
  $spar \xleftarrow{\$} \mathsf{SParGen}(1^\tau)$
  $(ppk, ssk_1) \leftarrow \mathsf{PreKeyGen}(spar)$
  $\mathbf{L} \leftarrow \emptyset$
  $(m^*, \sigma^*, spk, ssk_2) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{PreSign}(ssk_1, \cdot)}}(spar, ppk)$
    where $\mathcal{O}^{\mathsf{PreSign}}$ on input $m_i$:
      set $\mathbf{L} \leftarrow \mathbf{L} \cup m_i$
      return $\sigma'_i \leftarrow \mathsf{PreSign}(ssk_1, m_i)$
  return 1 if $\mathsf{Vf}(spk, \sigma^*, m^*) = 1$, and $m^* \notin \mathbf{L}$
    and $\mathsf{VerKey}(ppk, spk, ssk_2) = 1$
```

Fig. 5. Unforgeability-1 (1st signer is corrupt) and unforgeability-2 (2nd signer is corrupt) experiments.

first signer to be corrupt, and type-2 unforgeability considers a corrupt second signer. Our definitions are similar to the ones by Bellare and Sandhu, with the difference that we do not assume a trusted dealer creating *both* secret key shares. Instead, we let the adversary output the key share of the party he controls. For type-2 unforgeability we must ensure, though, that the adversary indeed integrates the honestly generated pre-key *ppk* when producing the completed public key *spk*, which we verify via VerKey. Formally, unforgeability for split signatures is defined as follows.

Definition 2. (Type-1/2 Unforgeability of SSIG). *A split signature scheme is* type-1/2 unforgeable *if for any efficient algorithm $\mathcal{A}$ the probability that the experiments given in Figure 5 return 1 is negligible (as a function of τ).*

Further, we need a property that we call *key-hiding*, which ensures that signatures do not leak any information about the public key for which they are generated. This is needed in the DAA scheme to get unlinkability even in the presence of a corrupt TPM that contributes to the signatures and knows part of the secret key, yet should not be able to recognize “his” signatures afterwards. Our key-hiding notion is somewhat similar in spirit to key-privacy for encryption schemes as defined by Bellare et al. [BBDP01], which requires that a ciphertext should not leak anything about the public key under which it is encrypted.

Formally, this is captured by giving the adversary a challenge signature for a chosen message either under the real or a random public key. Clearly, the property can only hold as long as the real public key *spk* is not known to the adversary, as otherwise he can simply verify the challenge signature. As we want the property to hold even when the first party is corrupt, the adversary can choose the first part of the secret key and also contribute to the challenge signature. The adversary is also given oracle access to $\mathcal{O}^{\mathsf{CompleteSign}}$ again, but is not allowed to query the message used in the challenge query, as he could win trivially otherwise (by the requirement of signature-uniqueness defined below and the determinism of CompleteSign). The formal experiment for our key-hiding property is given below. The oracle $\mathcal{O}^{\mathsf{CompleteSign}}$ is defined analogously as in type-1 unforgeability.

Definition 3. (Key-hiding property of SSIG). *We say a split signature scheme is* key-hiding *if for any efficient algorithm $\mathcal{A}$ the probability that the experiment given in Figure 6 returns 1 is negligible (as a function of τ).*

```
$\textbf{Experiment}\ \mathsf{Exp}_{\mathcal{A}}^{\mathsf{Key\text{-}Hiding}}(\tau)$:
  $spar \xleftarrow{\$} \mathsf{SParGen}(1^\tau)$
  $(ppk, state) \xleftarrow{\$} \mathcal{A}(spar)$
  $(spk, ssk_2) \xleftarrow{\$} \mathsf{CompleteKeyGen}(ppk)$
  $\mathbf{L} \leftarrow \emptyset$
  $(m, \sigma', state') \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{CompleteSign}(ppk, ssk_2, \cdot, \cdot)}}(state)$
  $b \xleftarrow{\$} \{0,1\}$
  if $b = 0$ (signature under spk):
      $\sigma \leftarrow \mathsf{CompleteSign}(ppk, ssk_2, m, \sigma')$
  if $b = 1$ (signature under random key):
      $(ppk^*, ssk_1^*) \xleftarrow{\$} \mathsf{PreKeyGen}(spar)$
      $(spk^*, ssk_2^*) \xleftarrow{\$} \mathsf{CompleteKeyGen}(ppk^*)$
      $\sigma' \xleftarrow{\$} \mathsf{PreSign}(ssk_1^*, m)$
      $\sigma \leftarrow \mathsf{CompleteSign}(ppk^*, ssk_2^*, m, \sigma')$
  $b' \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{CompleteSign}(ppk, ssk_2, \cdot, \cdot)}}(state', \sigma)$
  return 1 if $b = b'$, $m \notin \mathbf{L}$, and $\mathsf{Vf}(spk, \sigma, m) = 1$
```

Fig. 6. Key-hiding experiment for split signatures.

For correctness, we require that honestly created signatures always pass verification:

$$\mathsf{Pr}\Big[\mathsf{Vf}(spar, spk, \sigma, m) = 1 \ \mid \ spar \leftarrow \mathsf{SParGen}(\tau), \\ (ppk, spk_1) \xleftarrow{\$} \mathsf{PreKeyGen}(spar), (spk, ssk_2) \xleftarrow{\$} \mathsf{CompleteKeyGen}(ppk), \\ \sigma' \xleftarrow{\$} \mathsf{PreSign}(ssk_1, m), \sigma' \leftarrow \mathsf{CompleteSign}(ppk, ssk_2, m, \sigma')\Big]$$

We also require two uniqueness properties for our split signatures. The first is *key-uniqueness*, which states that every signature is only valid under one public key.

Definition 4. (Key-uniqueness of split signatures). *We say a split signature scheme has* key-uniqueness *if for any efficient algorithm $\mathcal{A}$ the probability that the experiment given in Figure 7 returns 1 is negligible (as a function of τ).*

```
$\textbf{Experiment}\ \mathsf{Exp}_{\mathcal{A}}^{\mathsf{Key\text{-}Uniqueness}}(\tau)$:
  $spar \xleftarrow{\$} \mathsf{SParGen}(1^\tau)$
  $(\sigma, spk_0, spk_1, m) \xleftarrow{\$} \mathcal{A}(spar)$
  return 1 if $spk_0 \neq spk_1$, $\mathsf{Vf}(spar, spk_0, \sigma, m) = 1$, and $\mathsf{Vf}(spar, spk_1, \sigma, m) = 1$
```

Fig. 7. Key-uniqueness experiment for split signatures.

The second uniqueness property required is *signature-uniqueness*, which guarantees that one can compute only a single valid signature on a certain message under a certain public key.

Definition 5. (Signature-uniqueness of split signatures). *We say a split signature scheme has* signature uniqueness *if for any efficient algorithm $\mathcal{A}$ the probability that the experiment given in Figure 8 returns 1 is negligible (as a function of τ).*

Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{Signature\text{-}Uniqueness}}(\tau)$:
$spar \xleftarrow{\$} \mathsf{SParGen}(1^\tau)$
$(\sigma_0, \sigma_1, spk, m) \xleftarrow{\$} \mathcal{A}(spar)$
return 1 if $\sigma_0 \neq \sigma_1$, $\mathsf{Vf}(spar, spk, \sigma_0, m) = 1$, and $\mathsf{Vf}(spar, spk, \sigma_1, m) = 1$

Fig. 8. Signature-uniqueness experiment for split signatures.

Instantiation of split signatures (split-BLS). To instantiate split signatures, we use a modified BLS signature [BLS04]. Let H be a hash function $\{0,1\} \rightarrow \mathbb{G}_1^*$ and the public system parameters be the description of a bilinear map, i.e., $spar = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e, q)$.

$\mathsf{PreKeyGen}(spar):$ Take $ssk_1 \xleftarrow{\$} \mathbb{Z}_q^*$, set $ppk \leftarrow g_2^{ssk_1}$, and output (ppk, ssk_1).
$\mathsf{CompleteKeyGen}(spar, ppk):$ Check $ppk \in \mathbb{G}_2$ and $ppk \neq 1_{\mathbb{G}_2}$. Take $ssk_2 \xleftarrow{\$} \mathbb{Z}_q^*$ and compute $spk \leftarrow ppk^{ssk_2}$. Output (spk, ssk_2).
$\mathsf{VerKey}(spar, ppk, spk, ssk_2):$ Output 1 iff $ppk \neq 1_{\mathbb{G}_2}$ and $spk = ppk^{ssk_2}$.
$\mathsf{PreSign}(spar, ssk_1, m):$ Output $\sigma' \leftarrow \mathsf{H}(m)^{ssk_1}$.
$\mathsf{CompleteSign}(spar, ppk, ssk_2, m, \sigma'):$ If $e(\sigma', g_2) = e(\mathsf{H}(m), ppk)$, output $\sigma \leftarrow \sigma'^{ssk_2}$, otherwise $\perp$.
$\mathsf{Vf}(spar, spk, \sigma, m):$ Output 1 iff $\sigma \neq 1_{\mathbb{G}_1}$ and $e(\sigma, g_2) = e(\mathsf{H}(m), spk)$.

The proof of the following theorem is given in Appendix A.

Theorem 1. *The split-BLS signature scheme is a secure split signature scheme, satisfying correctness, unforgeability-1, unforgeability-2, key-hiding, key-uniqueness, and signature-uniqueness, under the computational co-Diffie-Hellman assumption and the DDH assumption in $\mathbb{G}_1$, in the random oracle model.*

5 Construction

This section describes our DAA protocol achieving optimal privacy. On a very high level, the protocol follows the core idea of existing DAA protocols: The platform, consisting of the TPM and a host, first generates a secret key *gsk* that gets blindly certified by a trusted issuer. Subsequently, the platform can use the key *gsk* to sign attestations and basenames and then prove that it has a valid credential on the signing key, certifying the trusted origin of the attestation.

This high-level procedure is the main similarity to existing schemes though, as we significantly change the role of the host to satisfy our notion of optimal privacy. First, we no longer rely on a single secret key *gsk* that is fully controlled by the TPM. Instead, both the TPM and host generate secret shares, *tsk* and *hsk* respectively, that lead to a joint public key *gpk*. For privacy reasons, we cannot reveal this public key to the issuer in the join protocol, as any exposure of the joint public key would allow to trace any subsequent signed attestations of the platform. Thus, we let the issuer sign only an encryption of the public key, using the signature scheme for encrypted messages. When creating this membership credential *cred* the issuer is assured that the blindly signed key is formed correctly and the credential is strictly bound to that unknown key.

After having completed the JOIN protocol, the host and TPM can together sign a message m with respect to a basename bsn. Both parties use their individual key shares and create a split signature on the message and basename (denoted as *tag*), which shows that the platform intended to sign this message and basename, and a split signature on only the basename (denoted as *nym*), which is used as a pseudonym. Recall that attestations from one platform with the same basename should be linkable. By the uniqueness of split signatures, *nym* will be constant for one platform and basename and allow for such linkability. Because split signatures are key-hiding, we can reveal *tag* and *nym* while preserving the unlinkability of signatures with different basenames.

When signing, the host proves knowledge of a credential that signs gpk. Note that the host can create the full proof of knowledge because the membership credential signs a joint public key. In existing DAA schemes, the membership credential signs a TPM secret, and therefore the TPM must always be involved to prove knowledge of the credential, which prevents optimal privacy as we argued in Section 3.

5.1 Our DAA Protocol with Optimal Privacy Π_{pdaa}

We now present our generic DAA protocol with optimal privacy Π_{pdaa} in detail. Let $\mathsf{SSIG} = (\mathsf{PreKeyGen}, \mathsf{CompleteKeyGen}, \mathsf{VerKey}, \mathsf{PreSign}, \mathsf{CompleteSign}, \mathsf{Vf})$ denote a secure split signature scheme, as defined in Section 4.4, and let $\mathsf{ESIG} = (\mathsf{SigKGen}, \mathsf{EncSign}, \mathsf{DecSign}, \mathsf{Vf})$ denote a secure signature scheme for encrypted messages, as defined in Section 4.3. In addition, we use a CPA secure encryption scheme $\mathsf{ENC} = (\mathsf{EncKGen}, \mathsf{Enc}, \mathsf{Dec})$. We require all these algorithms to be compatible, meaning they work with the same system parameters.

We further assume that functionalities $(\mathcal{F}_{\mathsf{crs}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{auth}*})$ are available to all parties. The certificate authority functionality $\mathcal{F}_{\mathsf{ca}}$ allows the issuer to register his public key, and we assume that parties call $\mathcal{F}_{\mathsf{ca}}$ to retrieve the public key whenever needed. As the issuer key (ipk, π_{ipk}) also contains a proof of well-formedness, we also assume that each party retrieving the key will verify π_{ipk}.

The common reference string functionality $\mathcal{F}_{\mathsf{crs}}$ provides all parties with the system parameters *spar* generated via $\mathsf{SParGen}(1^{\tau})$. All the algorithms of the building blocks take *spar* as an input, which we omit – except for the key generation algorithms – for ease of presentation.

For the communication between the TPM and issuer (via the host) in the join protocol, we use the semi-authenticated channel $\mathcal{F}_{\mathsf{auth}*}$ introduced by Camenisch et al. [CDL16b]. This functionality abstracts the different options on how to realize the authenticated channel between the TPM and issuer that is established via an unauthenticated host. We assume the host and TPM can communicate directly, meaning that they have an authenticated and perfectly secure channel. This models the physical proximity of the host and TPM forming the platform: if the host is honest an adversary can neither alter nor read their internal communication, or even notice that communication is happening.

To make the protocol more readable, we omit the explicit calls to the sub-functionalities with sub-session IDs and simply say e.g., issuer $\mathcal{I}$ registers its public key with $\mathcal{F}_{\mathsf{ca}}$. For definitions of the standard functionalities $\mathcal{F}_{\mathsf{crs}}$ and $\mathcal{F}_{\mathsf{ca}}$ we refer to [Can00, Can04]. All used functionalities are also presented in Appendix C.

`1. Issuer Setup.` In the setup phase, the issuer $\mathcal{I}$ creates a key pair of the signature scheme for encrypted messages and registers the public key with $\mathcal{F}_{\mathsf{ca}}$.

(a) $\mathcal{I}$ upon input (SETUP, sid) generates his key pair:
  - Check that $sid = (\mathcal{I}, sid')$ for some sid'.
  - Get $(ipk, isk) \xleftarrow{\$} \mathsf{ESIG.SigKGen}(spar)$ and prove knowledge of the secret key via $\pi_{ipk} \leftarrow \mathsf{NIZK}\{(\underline{isk}) : (ipk, isk) \in \mathsf{ESIG.SigKGen}(spar)\}(sid)$.
  - Initiate $\mathcal{L}_{\mathsf{JOINED}} \leftarrow \emptyset$.
  - Register the public key (ipk, π_{ipk}) at $\mathcal{F}_{\mathsf{ca}}$, and store $(isk, \mathcal{L}_{\mathsf{JOINED}})$.
  - Output $(\mathsf{SETUPDONE}, sid)$.

Join Protocol. 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 the platform to join, obtains a credential *cred* that subsequently enables the platform to create signatures. The credential is a signature on the encrypted joint public key *gpk* to which the host and TPM each hold a secret key share. To show the issuer that a TPM has contributed to the joint key, the TPM reveals an authenticated version of his (public) key contribution to the issuer and the host proves that it correctly incorporated that share in *gpk*. A unique sub-session identifier *jsid* distinguishes several join sessions that might run in parallel.

`2. Join Request.` The join request is initiated by the host.

(a) Host $\mathcal{H}_j$, on input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ parses $sid = (\mathcal{I}, sid')$ and sends $(sid, jsid)$ to $\mathcal{M}_i$.[4]

(b) TPM $\mathcal{M}_i$, upon receiving $(sid, jsid)$ from a party $\mathcal{H}_j$, outputs $(\mathsf{JOIN}, sid, jsid)$.

`3.` $\boldsymbol{\mathcal{M}}$`-Join Proceed.` The join session proceeds when the TPM receives an explicit input telling him to proceed with the join session *jsid*.

(a) TPM $\mathcal{M}_i$, on input $(\mathsf{JOIN}, sid, jsid)$ creates a key share for the split signature and sends it authenticated to the issuer (via the host):
  - Run $(tpk, tsk) \xleftarrow{\$} \mathsf{SSIG.PreKeyGen}(spar)$.
  - Send tpk over $\mathcal{F}_{\mathsf{auth*}}$ to $\mathcal{I}$ via $\mathcal{H}_j$, and store the key $(sid, \mathcal{H}_j, tsk)$.

[4] Recall that we use direct communication between a TPM and host, i.e., this message is authenticated and unnoticed by the adversary.

(b) When $\mathcal{H}_j$ notices $\mathcal{M}_i$ sending tpk over $\mathcal{F}_{\mathsf{auth}*}$ to the issuer, it generates its key share for the split signature and appends an encryption of the jointly produced gpk to the message sent towards the issuer.
   – Complete the split signature key as $(gpk, hsk) \xleftarrow{\$} \mathsf{SSIG.CompleteKeyGen}(tpk)$.
   – Create an ephemeral encryption key pair $(epk, esk) \xleftarrow{\$} \mathsf{EncKGen}(spar)$.
   – Encrypt gpk under epk as $C \xleftarrow{\$} \mathsf{Enc}(epk, gpk)$.
   – Prove that C is an encryption of a public key gpk that is correctly derived from the TPM public key share tpk:

$$\pi_{\mathsf{JOIN},\mathcal{H}} \leftarrow \mathsf{NIZK}\{(\underline{gpk}, hsk) : C \in \mathsf{Enc}(epk, gpk) \;\wedge \\ \mathsf{SSIG.VerKey}(tpk, gpk, hsk) = 1\}(sid, jsid).$$

   – Append $(\mathcal{H}_j, epk, C, \pi_{\mathsf{JOIN},\mathcal{H}})$ to the message $\mathcal{M}_i$ is sending to $\mathcal{I}$ over $\mathcal{F}_{\mathsf{auth}*}$ and store $(sid, jsid, \mathcal{M}_i, esk, hsk, gpk)$.

(c) $\mathcal{I}$, upon receiving tpk authenticated by $\mathcal{M}_i$ and $(\mathcal{H}_j, epk, C, \pi_{\mathsf{JOIN},\mathcal{H}})$ in the unauthenticated part, verifies that the request is legitimate:
   – Verify $\pi_{\mathsf{JOIN},\mathcal{H}}$ w.r.t. the authenticated tpk and check that $\mathcal{M}_i \notin \mathcal{L}_{\mathsf{JOINED}}$.
   – Store $(sid, jsid, \mathcal{H}_j, \mathcal{M}_i, epk, C)$ and output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$.

4. `𝓘-Join Proceed.` The join session is completed when the issuer receives an explicit input telling him to proceed with join session $jsid$.

(a) $\mathcal{I}$ upon input $(\mathsf{JOINPROCEED}, sid, jsid)$ signs the encrypted public key C using the signature scheme for encrypted messages:
   – Retrieve $(sid, jsid, \mathcal{H}_j, \mathcal{M}_i, epk, C)$ and set $\mathcal{L}_{\mathsf{JOINED}} \leftarrow \mathcal{L}_{\mathsf{JOINED}} \cup \mathcal{M}_i$.
   – Sign C as $cred' \xleftarrow{\$} \mathsf{ESIG.EncSign}(isk, epk, C)$ and prove that it did so correctly. (This proof is required to allow verification in the security proof: ENC is only CPA-secure and thus we cannot decrypt $cred'$.)

$$\pi_{\mathsf{JOIN},\mathcal{I}} \leftarrow \mathsf{NIZK}\{isk : cred' \in \mathsf{ESIG.EncSign}(isk, epk, C) \;\wedge \\ (ipk, isk) \in \mathsf{ESIG.SigKGen}(spar)\}(sid, jsid).$$

   – Send $(sid, jsid, cred', \pi_{\mathsf{JOIN},\mathcal{I}})$ to $\mathcal{H}_j$ (via the network).

(b) Host $\mathcal{H}_j$, upon receiving $(sid, jsid, cred', \pi_{\mathsf{JOIN},\mathcal{I}})$ decrypts and stores the membership credential:
   – Retrieve the session record $(sid, jsid, \mathcal{M}_i, esk, hsk, gpk)$.
   – Verify proof $\pi_{\mathsf{JOIN},\mathcal{I}}$ w.r.t. $ipk, cred', C$ and decrypt the credential as $cred \leftarrow \mathsf{ESIG.DecSign}(esk, cred')$.
   – Store the completed key record $(sid, hsk, tpk, gpk, cred, \mathcal{M}_i)$ and output $(\mathsf{JOINED}, sid, jsid)$.

Sign Protocol. 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 w.r.t. a basename bsn using the split signature. Sub-session identifier $ssid$ distinguishes multiple sign sessions.

`5. Sign Request.` The signature request is initiated by the host.

(a) $\mathcal{H}_j$ upon input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ prepares the signature process:
  - Check that it joined with $\mathcal{M}_i$ (i.e., a completed key record for $\mathcal{M}_i$ exists).
  - Create signature record $(sid, ssid, \mathcal{M}_i, m, bsn)$.
  - Send $(sid, ssid, m, bsn)$ to $\mathcal{M}_i$.

(b) $\mathcal{M}_i$, upon receiving $(sid, ssid, m, bsn)$ from $\mathcal{H}_j$, stores $(sid, ssid, \mathcal{H}_j, m, bsn)$ and outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$.

`6. Sign Proceed.` The signature is completed when $\mathcal{M}_i$ gets permission to proceed for *ssid*.

(a) $\mathcal{M}_i$ on input $(\mathsf{SIGNPROCEED}, sid, ssid)$ creates the first part of the split signature on m w.r.t. *bsn*:
  - Retrieve the signature request $(sid, ssid, \mathcal{H}_j, m, bsn)$ and key $(sid, \mathcal{H}_j, tsk)$.
  - Set $tag' \xleftarrow{\$} \mathsf{SSIG.PreSign}(tsk, (0, m, bsn))$ and $nym' \xleftarrow{\$} \mathsf{SSIG.PreSign}(tsk, (1, bsn))$.
  - Send $(sid, ssid, tag', nym')$ to $\mathcal{H}_j$.

(b) $\mathcal{H}_j$ upon receiving $(sid, ssid, tag', nym')$ from $\mathcal{M}_i$ completes the signature:
  - Retrieve the signature request $(sid, ssid, \mathcal{M}_i, m, bsn)$ and key $(sid, hsk, tpk, gpk, cred, \mathcal{M}_i)$.
  - Compute $tag \leftarrow \mathsf{SSIG.CompleteSign}(hsk, tpk, (0, m, bsn), tag')$.
  - Compute $nym \leftarrow \mathsf{SSIG.CompleteSign}(hsk, tpk, (1, bsn), nym')$.
  - Prove that *tag* and *nym* are valid split signatures under public key *gpk* and that it owns a valid issuer credential *cred* on *gpk*, without revealing *gpk* or *cred*.

$$\pi_{\mathsf{SIGN}} \leftarrow \mathsf{NIZK}\{(gpk, cred) : \mathsf{ESIG.Vf}(ipk, cred, gpk) = 1 \ \wedge$$
$$\mathsf{SSIG.Vf}(gpk, tag, (0, m, bsn)) = 1 \ \wedge \ \mathsf{SSIG.Vf}(gpk, nym, (1, bsn)) = 1\}$$

  - Set $\sigma \leftarrow (tag, nym, \pi_{\mathsf{SIGN}})$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$.

Verify & Link. Any party can use the following verify and link algorithms to determine the validity of a signature and whether two signatures for the same basename were created by the same platform.

`7. Verify.` The verify algorithm allows one to check whether a signature σ on message m w.r.t. basename *bsn* and private key revocation list $\mathtt{RL}$ is valid.

(a) $\mathcal{V}$ upon input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, \mathtt{RL})$ verifies the signature:
  - Parse σ as $(tag, nym, \pi_{\mathsf{SIGN}})$.
  - Verify π_{SIGN} with respect to m, *bsn*, *tag*, and *nym*.
  - For every $gpk_i \in \mathtt{RL}$, check that $\mathsf{SSIG.Vf}(gpk_i, nym, (1, bsn)) \neq 1$.
  - If all tests pass, set $f \leftarrow 1$, otherwise $f \leftarrow 0$.
  - Output $(\mathsf{VERIFIED}, sid, f)$.

8. `Link`. The link algorithm allows one to check whether two signatures σ and σ', on messages m and m' respectively, that were generated for the same basename bsn were created by the same platform.

(a) $\mathcal{V}$ upon input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', bsn)$ verifies the signatures and compares the pseudonyms contained in σ, σ':
   - Check that both signatures σ and σ' are valid with respect to (m, bsn) and (m', bsn) respectively, using the `Verify` algorithm with $\mathtt{RL} \leftarrow \emptyset$. Output $\perp$ if they are not both valid.
   - Parse the signatures as $(tag, nym, \pi_{\mathsf{SIGN}})$ and $(tag', nym', \pi'_{\mathsf{SIGN}})$.
   - If $nym = nym'$, set $f \leftarrow 1$, otherwise $f \leftarrow 0$.
   - Output (LINK, sid, f).

5.2 Security

We now prove that that our generic protocol is a secure DAA scheme with optimal privacy under isolated TPM corruptions (and also achieves conditional privacy under full TPM corruption) as defined in Section 2.

Theorem 2. *Our protocol Π_{pdaa} described in Section 5, securely realizes $\mathcal{F}_{\mathsf{pdaa}}$ defined in Section 2, in the $(\mathcal{F}_{\mathsf{auth}*}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{crs}})$-hybrid model, provided that*
- SSIG *is a secure split signature scheme (as defined in Section 4.4),*
- ESIG *is a secure signature scheme for encrypted messages,*
- ENC *is a CPA-secure encryption scheme, and*
- NIZK *is a zero-knowledge, simulation-sound and online-extractable (for the underlined values) proof system.*

To prove Theorem 2, we have to show that there exists a simulator $\mathcal{S}$ as a function of $\mathcal{A}$ such that no environment can distinguish Π_{pdaa} and $\mathcal{A}$ from $\mathcal{F}_{\mathsf{pdaa}}$ and $\mathcal{S}$. We let the adversary perform both isolated corruptions and full corruptions on TPMs, showing that this proof both gives optimal privacy with respect to adversaries that only perform isolated corruptions on TPMs, and conditional privacy otherwise. The full proof is given in Appendix D, we present a proof sketch below.

Proof Sketch

Setup. For the setup, the simulator has to provide the functionality the required algorithms $(\mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$, where $\mathsf{sig}, \mathsf{ver}, \mathsf{link}$, and ukgen simply reflect the corresponding real-world algorithms. Thereby the signing algorithm also includes the issuer's secret key. When the issuer is corrupt, $\mathcal{S}$ can learn the issuer secret key by extracting from the proof π_{ipk}. When the issuer is honest, it is simulated by $\mathcal{S}$ in the real-world and thus $\mathcal{S}$ knows the secret key.

The algorithm $\mathsf{identify}(\sigma, m, bsn, \tau)$ that is used by $\mathcal{F}_{\mathsf{pdaa}}$ to internally ensure consistency and non-frameability is defined as follows: parse σ as $(tag, nym, \pi_{\mathsf{SIGN}})$ and output $\mathsf{SSIG.Vf}(\tau, nym, (1, bsn))$. Recall that τ is a tracing trapdoor that is either provided by the simulator (when the host is corrupt) or generated internally by $\mathcal{F}_{\mathsf{pdaa}}$ whenever a new gpk is generated.

Join. The join-related interfaces of $\mathcal{F}_{\mathsf{pdaa}}$ notify $\mathcal{S}$ about any triggered join request by a platform consisting of host $\mathcal{H}_j$ and TPM $\mathcal{M}_i$ such that $\mathcal{S}$ can simulate the real-world protocol accordingly. If the host is corrupt, the simulator also has to provide the functionality with the tracing trapdoor τ. For our scheme the joint key *gpk* of the split signature serves that purpose. For privacy reasons the key is never revealed, but the host proves knowledge and correctness of the key in $\pi_{\mathsf{JOIN},\mathcal{H}}$. Thus, if the host is corrupt, the simulator extracts *gpk* from this proof and gives it $\mathcal{F}_{\mathsf{pdaa}}$.

Sign. For platforms with an honest host, $\mathcal{F}_{\mathsf{pdaa}}$ creates anonymous signatures using the sig algorithm $\mathcal{S}$ defined in the setup phase. Thereby, $\mathcal{F}_{\mathsf{pdaa}}$ enforces unlinkability by generating and using fresh platform keys via ukgen whenever a platform requests a signature for a new basename. For signature requests where a platform repeatedly uses the same basename, $\mathcal{F}_{\mathsf{pdaa}}$ re-uses the corresponding key accordingly. We now briefly argue that no environment can notice this difference. Recall that signatures consist of signatures *tag* and *nym*, and a proof π_{SIGN}, with the latter proving knowledge of the platform's key *gpk* and credential *cred*, such that *tag* and *nym* are valid under *gpk* which is in turn certified by *cred*. Thus, for every new basename, the credential *cred* is now based on different keys *gpk*. However, as we never reveal these values but only prove knowledge of them in π_{SIGN}, this change is indistinguishable to the environment.

The signature *tag* and pseudonym *nym*, that are split signatures on the message and basename, are revealed in plain though. For repeated attestations under the same basename, $\mathcal{F}_{\mathsf{pdaa}}$ consistently re-uses the same key, whereas the use of a fresh basename will now lead to the disclosure of split signatures under different keys. The key-hiding property of split signatures guarantees that this change is unnoticeable, even when the TPM is corrupt and controls part of the key. Note that the key-hiding property requires that the adversary does not know the joint public key *gpk*, which we satisfy as *gpk* is never revealed in our scheme; the host only proves knowledge of the key in $\pi_{\mathsf{JOIN},\mathcal{H}}$ and π_{SIGN}.

Verify. For the verification of DAA signatures $\mathcal{F}_{\mathsf{pdaa}}$ uses the provided ver algorithm but also performs additional checks that enforce the desired non-frameability and unforgeability properties. We show that these additional checks will fail with negligible probability only, and therefore do not noticeably change the verification outcome.

First, $\mathcal{F}_{\mathsf{pdaa}}$ uses the identify algorithm and the tracing trapdoors τ_i to check that there is only a unique signer that matches to the signature that is to be verified. Recall that we instantiated the identify algorithm with the verification algorithm of the split signature scheme SSIG and $\tau = gpk$ are the (hidden) joint platform keys. By the key-uniqueness property of SSIG the check will fail with negligible probability only.

Second, $\mathcal{F}_{\mathsf{pdaa}}$ rejects the signature when no matching tracing trapdoor was found and the issuer is honest. For platforms with an honest hosts, theses trapdoors are created internally by the functionality whenever a signature is generated, and $\mathcal{F}_{\mathsf{pdaa}}$ immediately checks that the signature matches to the trapdoor (via the $\mathsf{identify}$ algorithm). For platforms where the host is corrupt, our simulator $\mathcal{S}$ ensures that a tracing trapdoor is stored in $\mathcal{F}_{\mathsf{pdaa}}$ as soon as the platform has joined (and received a credential). If a signature does not match any of the existing tracing trapdoors, it must be under a $gpk = \tau$ that was neither created by $\mathcal{F}_{\mathsf{pdaa}}$ nor signed by the honest issuer in the real-world. The proof π_{SIGN} that is part of every signature σ proves knowledge of a valid issuer credential on gpk. Thus, by the unforgeability of the signature scheme for encrypted messages ESIG, such invalid signatures can occur only with negligible probability.

Third, if $\mathcal{F}_{\mathsf{pdaa}}$ recognizes a signature on message m w.r.t. basename bsn that matches the tracing trapdoor of a platform with an honest TPM or honest host, but that platform has never signed m w.r.t. bsn, it rejects the signature. This can be reduced to unforgeability-1 (if the host is honest) or unforgeability-2 (if the TPM is honest) of the split signature scheme SSIG.

The fourth check that $\mathcal{F}_{\mathsf{pdaa}}$ makes corresponds to the revocation check in the real-world verify algorithm, i.e., it does not impose any additional check.

Link. Similar as for verification, $\mathcal{F}_{\mathsf{pdaa}}$ is not relying solely on the provided link algorithm but performs some extra checks when testing for the linkage between two signatures σ and σ'. It again uses $\mathsf{identify}$ and the internally stored tracing trapdoor to derive the final linking output. If there is one tracing trapdoor matching one signature but not the other, it outputs that they are not linked. If there is one tracing trapdoor matching both signatures, it enforces the output that they are linked. Only if no matching tracing trapdoor is found, $\mathcal{F}_{\mathsf{pdaa}}$ derives the output via link algorithm.

We now show that the two checks and decisions imposed by $\mathcal{F}_{\mathsf{pdaa}}$ are consistent with the real-world linking algorithm. In the real world, signatures $\sigma = (tag, nym, \pi_{\mathsf{SIGN}})$ and $\sigma' = (tag', nym', \pi'_{\mathsf{SIGN}})$ w.r.t basename bsn are linked iff $nym = nym'$. Tracing trapdoors are instantiated by the split signature scheme public keys gpk, and $\mathsf{identify}$ verifies nym under the key gpk. If one key matches one signature but not the other, then by the fact that the verification algorithm of the split signatures is deterministic, we must have $nym \neq nym'$, showing that the real world algorithm also outputs unlinked. If one key matches both signatures, we have $nym = nym'$ by the signature-uniqueness of split signatures, so the real-world algorithm also outputs linked. $\square$

6 Concrete Instantiation and Efficiency

In this section we describe on a high level how to efficiently instantiate the generic building blocks to instantiate our generic DAA scheme presented in Section 5.

The split signature scheme is instantiated with the split-BLS signatures (as described in Section 4.4), the signatures for encrypted messages with the AGOT+ signature scheme (as described in Section 4.3) and the encryption scheme with ElGamal, both working in $\mathbb{G}_2$. All the zero-knowledge proofs are instantiated with non-interactive Schnorr-type proofs about discrete logarithms, and witnesses that have to be online extractable are encrypted using ElGamal for group elements and Camenisch-Shoup encryption [CS03] for exponents. Note that the latter is only used by the issuer to prove that its key is correctly formed, i.e., every participant will only work with Camenisch-Shoup ciphertexts once.

The shared system parameters *spar* then consist of a security parameter τ, a bilinear group $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order q with generators g_1 and g_2 and bilinear map e. Further, the system parameters contain an additional random group element $x \xleftarrow{\$} \mathbb{G}_2$ for the AGOT+ signature and an ElGamal encryption key $epk_{\mathsf{crs}} \xleftarrow{\$} \mathbb{G}_2$. This crs-key allows for efficient online extractability in the security proof, as the simulator will be privy of the corresponding secret key. Finally, let $\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1^*$ be a hash function, that we model as a random oracle in the security proof.

Setup. The issuer registers the AGOT+ key $ipk = g_1^{isk}$ along with a proof π_{ipk} that ipk is well-formed. For universal composition, we need isk to be online-extractable, which can be achieved by verifiable encryption. To this end, we let the crs additionally contain a public key $(\mathsf{n}, \mathsf{y}, \mathsf{g}, \mathsf{h})$ for the CPA version of the Camenisch-Shoup encryption scheme and an additional element g to make the verifiable encryption work [CS03]. We thus instantiate the proof

$$\pi_{ipk} \leftarrow \mathsf{NIZK}\{(\underline{isk}) : (ipk, isk) \in \mathsf{ESIG.SigKGen}(spar)\}(sid)$$

as follows:

$$\pi_{ipk} \leftarrow \mathsf{SPK}\{(isk, r) : ipk = g_1^{isk} \ \wedge\ \hat{\mathsf{g}}^r \mathsf{g}^{isk} \bmod \mathsf{n} \ \wedge \\ \mathsf{g}^r \bmod \mathsf{n} \ \wedge\ \mathsf{y}^r \mathsf{h}^{isk} \bmod \mathsf{n} \ \wedge\ isk \in [-\mathsf{n}/4, \mathsf{n}/4]\}(sid)$$

Join. Using the split-BLS signature, the TPM has a secret key $tsk \in \mathbb{Z}_q^*$ and public key $tpk = g_2^{tsk}$, the host has secret key $hsk \in \mathbb{Z}_q^*$, and together they have created the public key $gpk = g_2^{tsk \cdot hsk}$.

We now show how to instantiate the proof $\pi_{\mathsf{JOIN},\mathcal{H}}$ where the host proves that C is an encryption of a correctly derived gpk. Recall that the issuer receives the M_i's public key contribution tpk authenticated from the TPM.

$$\pi_{\mathsf{JOIN},\mathcal{H}} \leftarrow \mathsf{NIZK}\{(\underline{gpk}, hsk) : C \in \mathsf{Enc}(epk, gpk) \ \wedge \\ \mathsf{SSIG.VerKey}(tpk, gpk, hsk) = 1\}(sid, jsid).$$

The joint public key gpk is encrypted under an ephemeral key epk using ElGamal with crs trapdoor epk_{crs}. We set $\rho \xleftarrow{\$} \mathbb{Z}_q$, $C_1 \leftarrow epk_{\mathsf{crs}}^{\rho}$, $C_2 \leftarrow epk^{\rho}$, $C_3 \leftarrow g_2^{\rho} \cdot gpk$ and prove:

$$\pi'_{\mathsf{JOIN},\mathcal{H}} \leftarrow \mathsf{SPK}\{(hsk, \rho) : C_1 = epk_{\mathsf{crs}}^{\rho} \ \wedge\ C_2 = epk^{\rho} \ \wedge\ C_3 = g_2^{\rho} \cdot tpk^{hsk}\}(sid, jsid).$$

The host sets $\pi_{\mathsf{JOIN},\mathcal{H}} \leftarrow (C_1, C_2, C_3, \pi'_{\mathsf{JOIN},\mathcal{H}})$ as the final proof. Note that gpk is online-extractable as it is encrypted under epk_{crs}. The issuer checks $tpk \neq 1_{\mathbb{G}_2}$ and verifies $\pi'_{\mathsf{JOIN},\mathcal{H}}$.

Next, the issuer places an AGOT+ signature on gpk. Since $gpk \in \mathbb{G}_2$, the decrypted credential has the form (r, s, t, w) which is an element of $\mathbb{G}_1 \times \mathbb{G}_2^3$. The issuer computes the credential on ciphertext (C_1, C_2, C_3) as follows: Choose a random $u, \rho_1, \rho_2 \xleftarrow{\$} \mathbb{Z}_q^*$, and compute the (partially) encrypted signature $\bar{\sigma} = (r, (S_1, S_2, S_3), (T_1, T_2, T_3), w)$:

$$r \leftarrow g_2^u, \qquad S_1 \leftarrow C_2^{v/u} epk^{\rho_1}, \qquad S_2 \leftarrow (C_3^v x)^{1/u} g_2^{\rho_1},$$
$$T_1 \leftarrow S_2^{v/u} epk^{\rho_2}, \qquad T_2 \leftarrow (S_2^v g_2)^{1/u} g_2^{\rho_2}, \qquad w \leftarrow g_2^{1/u}.$$

Then, with $\pi_{\mathsf{JOIN},\mathcal{I}}$ it proves that it signed the ciphertext correctly:

$$\pi_{\mathsf{JOIN},\mathcal{I}} \leftarrow \mathsf{NIZK}\{isk : cred' \in \mathsf{ESIG.EncSign}(isk, epk, C) \ \wedge \\ (ipk, isk) \in \mathsf{ESIG.SigKGen}(spar)\}(sid, jsid).$$

To instantiate this, we let the issuer create $\pi'_{\mathsf{JOIN},\mathcal{I}}$ as follows, using witness $u' = \frac{1}{u}$ and $isk' = \frac{isk}{u}$:

$$\pi'_{\mathsf{JOIN},\mathcal{I}} \leftarrow \mathsf{SPK}\{(u', isk', \rho_1, \rho_2) : g_2 = r^{u'} \ \wedge \ S_1 = C_2^{isk'} epk^{\rho_1} \ \wedge \\ S_2 = C_3^{isk'} x^{u'} g_2^{\rho_1} \ \wedge \ T_1 = S_1^{isk'} epk^{\rho_2} \ \wedge \ T_2 = S_2^{isk'} g_2^{u'} g_2^{\rho_2} \ \wedge \ w = g_2^{u'} \ \wedge \\ 1 = ipk^{-isk'} g_1^{u'}\}(sid, jsid).$$

The issuer outputs $\pi_{\mathsf{JOIN},\mathcal{I}} = (r, S_1, S_2, T_1, T_2, w, \pi'_{\mathsf{JOIN},\mathcal{I}})$.

Sign. In our concrete instantiation, signatures on messages and basenames are split-BLS signatures, i.e., the TPM and host jointly compute BLS signatures $tag \leftarrow \mathsf{H}(0, m, bsn)^{tsk \cdot hsk}$ and $nym \leftarrow \mathsf{H}(1, bsn)^{tsk \cdot hsk}$. Recall that we cannot reveal the joint public key gpk or the credential $cred$. Instead the host provides the proof π_{SIGN} that tag and nym are valid split signatures under public key gpk and that it owns a valid issuer credential $cred$ on gpk, without disclosing gpk and $cred$:

$$\pi_{\mathsf{SIGN}} \leftarrow \mathsf{NIZK}\{(gpk, cred) : \mathsf{ESIG.Vf}(ipk, cred, gpk) = 1 \ \wedge \\ \mathsf{SSIG.Vf}(gpk, tag, (0, m, bsn)) = 1 \ \wedge \ \mathsf{SSIG.Vf}(gpk, nym, (1, bsn)) = 1\}$$

This proof can be realized as follows: First, the host randomizes the AGOT+ credential (r, s, t, w) to (r', s', t', w) using the randomization token w. Note that this randomization allows the host to release r' (instead of encrypting it) without becoming linkable. The host then proves knowledge of the rest of the credential and gpk, such that the credential is valid under the issuer public key and signs gpk, that tag is a valid split-BLS signature on $(0, m, bsn)$ under gpk, and that nym is a valid split-BLS signature on $(1, bsn)$ under gpk. It computes the following proof:

$$\pi'_{\mathsf{SIGN}} \leftarrow \mathsf{SPK}\{(gpk, s', t') : \\ e(g_1, x) = e(r', s')e(V^{-1}, gpk) \quad \wedge \quad e(g_1, g_2) = e(r', t')e(V^{-1}, gpk) \quad \wedge \\ e(tag, g_2) = e(\mathsf{H}(0, m, bsn), gpk) \quad \wedge \quad e(nym, g_2) = e(\mathsf{H}(1, bsn), gpk)\}$$

The host finally sets $\pi_{\mathsf{SIGN}} \leftarrow (r', \pi'_{\mathsf{SIGN}})$. A verifier receiving $(tag, nym, \pi_{\mathsf{SIGN}})$ verifies π'_{SIGN} and checks $nym \neq 1_{\mathbb{G}_1}$ and $tag \neq 1_{\mathbb{G}_1}$.

6.1 Security

When using the concrete instantiations as presented above we can derive the following corollary from Theorem 2 and the required security assumptions of the deployed building blocks. We have opted for a highly efficient instantiation of our scheme, which comes for the price of stronger assumptions such as the generic group (for AGOT+ signatures) and random oracle model (for split-BLS signatures and Fiat-Shamir NIZKs). We would like to stress that our generic scheme based on abstract building blocks, presented in Section 5, does not require either of the models, and one can use less efficient instantiations to avoid these assumptions.

Corollary 1. *Our protocol Π_{pdaa} described in Section 5 and instantiated as described above, securely realizes $\mathcal{F}_{\mathsf{pdaa}}$ in the $(\mathcal{F}_{\mathsf{auth*}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{crs}})$-hybrid model under the following assumptions:*

Primitive	*Instantiation*	*Assumption*
SSIG	*split-BLS*	*co-DHP* [CHKM10], DDH in $\mathbb{G}_1$, RO model*
ESIG	*AGOT+*	*generic group model (security of AGOT)*
ENC	*ElGamal*	*DDH in $\mathbb{G}_2$*
NIZK	*Fiat-Shamir, ElGamal, Camenisch-Shoup*	*DDH in $\mathbb{G}_2$, DCR [Pai99], RO model*

6.2 Efficiency

We now give an overview of the efficiency of our protocol when instantiated as described above. Our analysis focuses on signing and verification, which will be used the most and thus have the biggest impact on the performance of the scheme.

We now discuss the efficiency of our protocol when instantiated as described above. Our analysis focuses on the signing protocol and verification, which will be used the most and thus have the biggest impact on the performance of the scheme.

TPM. Given the increased "responsibility" of the host, our protocol is actually very lightweight on the TPM's side. When signing, the TPM only performs two exponentiations in $\mathbb{G}_1$. In fact, according to the efficiency overview by Camenisch et al. [CDL16a], our scheme has the most efficient signing operation for the TPM to date. Since the TPM is typically orders of magnitude slower than the host, minimizing the TPM's workload is key to achieve an efficient scheme.

Host. The host performs more tasks than in previous DAA schemes, but remains efficient. The host runs $\mathsf{split-BLS.CompleteSign}$ twice, which costs 4 pairings and 2 exponentiations in $\mathbb{G}_1$. Next, it constructs π_{SIGN}. This involves randomizing the AGOT credential, which costs 1 exponentiation in $\mathbb{G}_1$ and 3 in $\mathbb{G}_2$. It then constructs π'_{SIGN}, which costs 3 exponentiations in $\mathbb{G}_2$ and 6 pairings. This results in total signing cost of $3\mathbb{G}_1, 6\mathbb{G}_2, 10P$ for a host.

Verifier. The verification checks the validity of $(tag, nym, \pi_{\mathsf{SIGN}})$, which consists of checking π'_{SIGN}. Computing the left-hand sides of the euqations in π'_{SIGN} costs two pairings, as $e(g_1, g_2)$ and $e(g_1, x)$ can be precomputed. Verifying the rest of the proof costs 6 pairings and 4 exponentiations in $\mathbb{G}_T$. The revocation check with a revocation list of n elements costs $n + 1$ pairings.

Estimated Performance. We measured the speed of the Apache Milagro Cryptographic Library (AMCL)[5] and found that exponentiations in $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ require 0.6ms, 1.0ms, and 1.4ms respectively. A pairing costs 1.6ms. Using these numbers, we estimate a signing time of 23.8ms for the host, and a verification time of 18.4ms, showing that also for the host our protocol is efficient enough to be used in practice. Table 2 gives an overview of the efficiency of our concrete instantiation.

	$\mathcal{M}$ Sign	$\mathcal{H}$ Sign	Verify
Operations	$2\mathbb{G}_1$	$3\mathbb{G}_1$, $6\mathbb{G}_2$, $10P$	$4\mathbb{G}_T$, $8P$
Est. Time		23.8ms	18.4ms

Table 2. Efficiency of our concrete DAA scheme.

References

[AGOT14] Masayuki Abe, Jens Groth, Miyako Ohkubo, and Mehdi Tibouchi. Unified, minimal and selectively randomizable structure-preserving signatures. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 688–712. Springer, Heidelberg, February 2014.

[AKMZ12] Joël Alwen, Jonathan Katz, Ueli Maurer, and Vassilis Zikas. Collusion-preserving computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 124–143. Springer, Heidelberg, August 2012.

[AMV15] Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. Subversion-resilient signature schemes. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 364–375. ACM Press, October 2015.

[AsV08] Joël Alwen, abhi shelat, and Ivan Visconti. Collusion-free protocols in the mediated model. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 497–514. Springer, Heidelberg, August 2008.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 566–582. Springer, Heidelberg, December 2001.

[BBG13] James Ball, Julian Borger, and Glenn Greenwald. Revealed: how US and UK spy agencies defeat internet privacy and security. Guardian Weekly, September 2013.

[5] See `https://github.com/miracl/amcl`. We used the C-version of the library, configured to use the BN254 curve. The program `benchtest_pair.c` has been used to retrieve the timings, executed on an Intel i5-4300U CPU.

[BBS98] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 127–144. Springer, Heidelberg, May / June 1998.

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

[BCL08] Ernie Brickell, Liqun Chen, and Jiangtao Li. A new direct anonymous attestation scheme from bilinear maps. In Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch, editors, *Trusted Computing - Challenges and Applications, First International Conference on Trusted Computing and Trust in Information Technologies, Trust 2008, Villach, Austria, March 11-12, 2008, Proceedings*, volume 4968 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 2008.

[BCL09] Ernie Brickell, Liqun Chen, and Jiangtao Li. Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. *Int. J. Inf. Sec.*, 8(5):315–330, 2009.

[BD95] Mike Burmester and Yvo Desmedt. A secure and efficient conference key distribution system (extended abstract). In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 275–286. Springer, Heidelberg, May 1995.

[BFG13a] David Bernhard, Georg Fuchsbauer, and Essam Ghadafi. Efficient signatures of knowledge and DAA in the standard model. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 13*, volume 7954 of *LNCS*, pages 518–533. Springer, Heidelberg, June 2013.

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

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

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

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[BPR14] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 1–19. Springer, Heidelberg, August 2014.

[Bra94] Stefan Brands. Untraceable off-line cash in wallets with observers (extended abstract). In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 302–318. Springer, Heidelberg, August 1994.

[Bra00] Stefan A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy.* MIT Press, Cambridge, MA, USA, 2000.

[BS01] Mihir Bellare and Ravi Sandhu. The security of practical two-party RSA signature schemes. Cryptology ePrint Archive, Report 2001/060, 2001. http://eprint.iacr.org/2001/060.

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

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

[CCD$^+$17] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One TPM to bind them all: Fixing TPM 2.0 for provably secure anonymous attestation. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 901–920. IEEE Computer Society, 2017.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. `http://eprint.iacr.org/2016/086`.

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

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

[CDL16b] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 234–264. Springer, Heidelberg, March 2016.

[CF08] Xiaofeng Chen and Dengguo Feng. Direct anonymous attestation for next generation TPM. *JCP*, 3(12):43–50, 2008.

[Cha92] David Chaum. Achieving electronic privacy. *Scientific american*, 267(2):96–101, 1992.

[Che09] Liqun Chen. A DAA scheme requiring less TPM resources. In Feng Bao, Moti Yung, Dongdai Lin, and Jiwu Jing, editors, *Information Security and Cryptology - 5th International Conference, Inscrypt 2009, Beijing, China, December 12-15, 2009. Revised Selected Papers*, volume 6151 of *Lecture Notes in Computer Science*, pages 350–365. Springer, 2009.

[CHKM10] Sanjit Chatterjee, Darrel Hankerson, Edward Knapp, and Alfred Menezes. Comparing two pairing-based aggregate signature schemes. *Des. Codes Cryptography*, 55(2-3):141–167, 2010.

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

[CL15] Jan Camenisch and Anja Lehmann. (Un)linkable pseudonyms for governmental databases. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 1467–1479. ACM Press, October 2015.

[CMS08] Liqun Chen, Paul Morrissey, and Nigel P. Smart. Pairings in trusted computing (invited talk). In Steven D. Galbraith and Kenneth G. Paterson, editors, *PAIRING 2008*, volume 5209 of *LNCS*, pages 1–17. Springer, Heidelberg, September 2008.

[CMY$^+$16] Rongmao Chen, Yi Mu, Guomin Yang, Willy Susilo, Fuchun Guo, and Mingwu Zhang. Cryptographic reverse firewall via malleable smooth projective hash functions. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 844–876. Springer, Heidelberg, December 2016.

[CP93] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, August 1993.

[CP94] Ronald Cramer and Torben P. Pedersen. Improved privacy in wallets with observers (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 329–343. Springer, Heidelberg, May 1994.

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

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

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

[CV12] Ran Canetti and Margarita Vald. Universally composable security with local adversaries. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 281–301. Springer, Heidelberg, September 2012.

[DMSD16] Yevgeniy Dodis, Ilya Mironov, and Noah Stephens-Davidowitz. Message transmission with reverse firewalls—secure communication on corrupted machines. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 341–372. Springer, Heidelberg, August 2016.

[ElG86] Taher ElGamal. On computing logarithms over finite fields. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 396–402. Springer, Heidelberg, August 1986.

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

[Gre14] Glenn Greenwald. No place to hide: Edward snowden, the nsa, and the u.s. surveillance state. Metropolitan Books, May 2014.

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 415–432. Springer, Heidelberg, April 2008.

[HPV16] Carmit Hazay, Antigoni Polychroniadou, and Muthuramakrishnan Venkitasubramaniam. Composable security in the tamper-proof hardware model under minimal complexity. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 367–399. Springer, Heidelberg, October / November 2016.

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

[Int15] International Organization for Standardization. ISO/IEC 11889: Information technology - Trusted platform module library, 2015.

[Kat07] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 115–128. Springer, Heidelberg, May 2007.

[KO04] Jonathan Katz and Rafail Ostrovsky. Round-optimal secure two-party computation. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 335–354. Springer, Heidelberg, August 2004.

[MS15] Ilya Mironov and Noah Stephens-Davidowitz. Cryptographic reverse firewalls. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 657–686. Springer, Heidelberg, April 2015.

[OO90] Tatsuaki Okamoto and Kazuo Ohta. Divertible zero knowledge interactive proofs and commutative random self-reducibility. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 134–148. Springer, Heidelberg, April 1990.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.

[PLS13] Nicole Perlroth, Jeff Larson, and Scott Shane. N.S.A. able to foil basic safeguards of privacy on web. The New York Times, September 2013.

[RTYZ16a] Alexander Russell, Qiang Tang, Moti Yung, and Hong-Sheng Zhou. Cliptography: Clipping the power of kleptographic attacks. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 34–64. Springer, Heidelberg, December 2016.

[RTYZ16b] Alexander Russell, Qiang Tang, Moti Yung, and Hong-Sheng Zhou. Destroying steganography via amalgamation: Kleptographically CPA secure public key encryption. Cryptology ePrint Archive, Report 2016/530, 2016. `http://eprint.iacr.org/2016/530`.

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

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

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[YY97a] Adam Young and Moti Yung. Kleptography: Using cryptography against cryptography. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 62–74. Springer, Heidelberg, May 1997.

[YY97b] Adam Young and Moti Yung. The prevalence of kleptographic attacks on discrete-log based cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 264–276. Springer, Heidelberg, August 1997.

A Proof of Theorem 1

We now prove that the split-BLS construction presented in Section 4.4 indeed satisfies our notion of secure split signatures.

Correctness. From running $\mathsf{PreKeyGen}$, we get $ssk_1 \xleftarrow{\$} \mathbb{Z}_q^*$ and $ppk \leftarrow g_2^{ssk_1}$. $\mathsf{CompleteKeyGen}$ will check that $ppk \neq 1_{\mathbb{G}_2}$, which holds as ssk_1 is taken from $\mathbb{Z}_q^*$. It then takes $ssk_2 \xleftarrow{\$} \mathbb{Z}_q^*$ and $spk \leftarrow ppk^{ssk_2}$.

When signing, $\mathsf{PreSign}$ sets $\sigma' \leftarrow \mathsf{H}(m)^{ssk_1}$. $\mathsf{CompleteSign}$ checks $e(\sigma', g_2) \overset{?}{=} e(\mathsf{H}(m), ppk)$ which holds for this σ', and computes $\sigma \leftarrow \sigma'^{ssk_2}$.

Verification checks $e(\sigma, g_2) = e(\mathsf{H}(m), spk)$, which holds as $\sigma = \mathsf{H}(m)^{ssk_1 \cdot ssk_2}$ and $spk = g_2^{ssk_1 \cdot ssk_2}$. Since both ssk_1 and ssk_2 are taken from $\mathbb{Z}_q^*$, they are both unequal to 0 and $ssk_1 \cdot ssk_2 \neq 0$. As H maps to $\mathbb{G}_1^*$, this means $\sigma \neq 1_{\mathbb{G}}$.

Unforgeability-1. We reduce breaking unforgeability-1 to breaking the co-Diffie-Hellman problem.[6] Our reduction takes as input g_1^α, g_2^α, $h \in \mathbb{G}_1$, and must compute h^α. If $\alpha = 0$, the reduction fails. $\mathcal{A}$ gives the pre-key ppk upon input the system parameters. For some unknown ssk_1, $ppk = g_2^{ssk_1}$. This reduction simulates the (unknown) second key $ssk_2 = \alpha/ssk_1$. We therefore set $spk \leftarrow g_2^\alpha = ppk^{ssk_2}$. Random oracle queries are answered with g_1^r for $r \xleftarrow{\$} \mathbb{Z}_q$, while maintaining consistency, except for a random query $\bar{m}$, where it returns h. When $\mathcal{A}$ makes a $\mathsf{CompleteSign}$ query on a message $m \neq \bar{m}$ and pre-signature σ', first check $e(\sigma', g_2) \overset{?}{=} e(\mathsf{H}(m), ppk)$, and return $\perp$ if this does not hold. Otherwise, return $\sigma \leftarrow \mathsf{H}(m)^\alpha = (g_1^\alpha)^r$, where the reduction knows r from simulating the random oracle.

When $\mathcal{A}$ outputs forgery (m^*, σ^*). With non-negligible probability, $m^* = \bar{m}$, and we have $e(\sigma^*, g_2) = e(\mathsf{H}(m), spk) = e(h, g_2^\alpha)$, so σ^* solves the computational co-DH problem.

Unforgeability-2. We reduce breaking unforgeability-2 to breaking the co-Diffie-Hellman problem.

Our reduction takes as input g_1^α, g_2^α, $h \in \mathbb{G}_1$, and must compute h^α. If $g_1^\alpha = 1_{\mathbb{G}_1}$ or $h = 1_{\mathbb{G}_1}$ the reduction fails. Give $\mathcal{A}$ input $ppk = g_2^\alpha$. When $\mathcal{A}$ makes random oracle queries, answer them with g_1^r with $r \xleftarrow{\$} \mathbb{Z}_q^*$, while maintaining consistency, except for a random query $\bar{m}$, where it returns h. When $\mathcal{A}$ makes a $\mathsf{PreSign}$ query on m, find r such that $\mathsf{H}(m) = g_1^r$ from simulating the random oracle and output signature $\sigma \leftarrow (g_1^\alpha)^r$. If $\mathcal{A}$ makes a query with $m = \bar{m}$, the reduction fails.

When $\mathcal{A}$ outputs $(m^*, \sigma^*, spk, ssk_2)$ with $\mathsf{VerKey}(spar, ppk, spk, ssk_2) = 1$, $\mathsf{Vf}(spar, spk, \sigma^*, m^*) = 1$, and m was not queried, with non-negligible probability we have $m^* = \bar{m}$ and therefore $e(\sigma^*, g_2) = e(h, spk)$. Since $spk = g_2^{\alpha\, ssk_2}$, we have

[6] As the original BLS signatures were presented in a type II pairing setting, we prove unforgeability-1 and unforgeability-2 using a sightly different version of the computational co-Diffie-Hellman assumption suitable for type III pairings. The assumption we use is called co-DHP* and is formalized by Chatterjee et al. [CHKM10].

$e(\sigma^*, g_2) = e(h, g_2^{\alpha\, ssk_2})$. As $\sigma^* \neq 1_{\mathbb{G}_1}$, we have $ssk_2 \neq 0$ and $e(\sigma^{*1/ssk_2}, g_2) = e(h, g_2^{\alpha})$, so $\sigma^{*1/ssk_2} = h^{\alpha}$ solves the co-CDH problem.

Key-Hiding. Any adversary that has non-negligible probability of winning the key hiding game breaks the DDH assumption in $\mathbb{G}_1$.

The reduction receives input $g_1^{\alpha}, g_1^{\beta}, g_1^{\gamma}$. If $g_1^{\alpha} = 1_{\mathbb{G}_1}$, $g_1^{\beta} = 1_{\mathbb{G}_1}$, or $\gamma = 1_{\mathbb{G}_1}$, the reduction fails. It receives $ppk \in \mathbb{G}_2$ from $\mathcal{A}$, after handing it the system parameters. When $\mathcal{A}$ makes random oracle queries, answer them with g_1^{r} with $r \xleftarrow{\$} \mathbb{Z}_q^*$, while maintaining consistency, except for a random query $\bar{m}$, where it returns g_1^{β}. When $\mathcal{A}$ makes a $\mathsf{CompleteSign}$ query on a message $m \neq \bar{m}$ and pre-signature σ', first check $e(\sigma', g_2) \overset{?}{=} e(\mathsf{H}(m), ppk)$, and return $\perp$ if this does not hold. Otherwise, return $\mathsf{H}(m)^{\alpha} = (g_1^{\alpha})^{r}$, where the reduction knows r from simulating the random oracle. If $\mathcal{A}$ makes a signing query on $\bar{m}$ with a valid pre-signature, the reduction fails.

When $\mathcal{A}$ outputs (m, σ'), and $m \neq \bar{m}$, the reduction fails. If $m = \bar{m}$, give the adversary $\sigma \leftarrow g_1^{\gamma}$. Continue answering the oracle queries as before.

Now, note that if $\gamma = \alpha \cdot \beta$, we have simulated the game with $b = 0$, and if not, we simulated $b = 1$, so any adversary guessing b with non-negligible probability can break DDH.

Key-Uniqueness. As we work in prime order groups, every element has a unique discrete logarithm in $\mathbb{Z}_q$. If a signature on message m verifies under keys $spk_0 \neq spk_1$, we have $e(\sigma, g_2) = e(\mathsf{H}(m), spk_b)$ for $b \in \{0, 1\}$. Let $\mathsf{H}(m)$ be g_1^{r} for some $r \in \mathbb{Z}_q^*$, and let $spk_b = g_2^{x_b}$ for some $x_b \in \mathbb{Z}_q$, and as $spk_0 \neq spk_1$, $x_0 \neq x_1$. This gives $e(\sigma, g_2) = e(g_1, g_2)^{r \cdot x_b}$. Let s be the discrete log of σ, this means $s = r \cdot x_0$ and $s = r \cdot x_1$, which contradicts $r \in \mathbb{Z}_q^*$.

Signature-Uniqueness. If two signatures $\sigma_0 \neq \sigma_1$ on message m both verify under key spk, we have $e(\sigma_b, g_2) = e(\mathsf{H}(m), spk)$ for $b \in \{0, 1\}$. Let $\mathsf{H}(m)$ be g_1^{r}, $\sigma_b = g_1^{s_b}$, and $spk = g_2^{x}$, for some $r \in \mathbb{Z}_q^*$ and $s_0, s_1, x \in \mathbb{Z}_q^3$. As $\sigma_b \neq 1_{\mathbb{G}_1}$ (by the verification check), we have $s_0 \neq 0$ and $s_1 \neq 0$. This gives $s_b = r \cdot x$, which contradicts $s_0 \neq s_1$.

For correctness, we require that any message encrypted with honestly generated keys that is honestly signed decrypts to a valid signature. More precisely, for any message m, we require

$$\Pr\Big[\mathsf{Vf}(spk, \sigma, m) = 1 \;\mid\; spar \leftarrow \mathsf{SParGen}(\tau), (spk, ssk) \xleftarrow{\$} \mathsf{SigKGen}(spar), \\ (epk, esk) \leftarrow \mathsf{EncKGen}(spar), C \leftarrow \mathsf{Enc}(spar, epk, m), \\ \bar{\sigma} \leftarrow \mathsf{EncSign}(ssk, epk, c), \sigma \leftarrow \mathsf{DecSign}(esk, spk, \bar{\sigma})\Big].$$

We use the unforgeability definition of [CL15], but omit the oracle for signatures on plaintext messages. Note that the oracle $\mathcal{O}^{\mathsf{EncSign}}$ will only sign correctly computed ciphertexts, which is modeled by providing the message and public encryption key as input and let the oracle encrypt the message itself before signing it. When using the scheme, this can easily be enforced by asking the signature requester for a proof of correct ciphertext computation, and, indeed, in our construction such a proof is needed for other reasons as well.

```
**Experiment** $\mathsf{Exp}^{\mathsf{ESIG\text{-}forge}}_{\mathcal{A},\mathsf{ESIG},\mathsf{Enc}}(\mathbb{G},\tau)$:
 $spar \leftarrow \mathsf{SParGen}(\tau)$
 $(spk, ssk) \xleftarrow{\$} \mathsf{SigKGen}(spar)$
 $\mathbf{L} \leftarrow \emptyset$
 $(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{EncSign}(ssk,\cdot,\cdot)}}(spar, spk)$
     where $\mathcal{O}^{\mathsf{EncSign}}$ on input $(epk_i, m_i)$:
         add $m_i$ to the list of queried messages $\mathbf{L} \leftarrow \mathbf{L} \cup m_i$
         compute $C_i \xleftarrow{\$} \mathsf{Enc}(epk_i, m_i)$
         return $\overline{\sigma} \xleftarrow{\$} \mathsf{EncSign}(ssk, epk_i, C_i)$
 return 1 if $\mathsf{Vf}(spk, \sigma^*, m^*) = 1$ and $m^* \notin \mathbf{L}$
```

Fig. 9. Unforgeability experiment for signatures on encrypted messages.

Definition 6. (Unforgeability of Signatures for Encrypted Messages). *We say a signature scheme for encrypted messages is* unforgeable *if for any efficient algorithm $\mathcal{A}$ the probability that the experiment given in Figure 9 returns 1 is negligible (as a function of τ).*

B Macros CheckTtdHonest and CheckTtdCorrupt

Here we define the two "macros" CheckTtdHonest and CheckTtdCorrupt that our ideal functionality $\mathcal{F}_{\mathsf{pdaa}}$ uses to to determine if a tracing trapdoor τ is consistent with the functionality's records or not. This is checked at several places in our functionality and also depends on whether the τ belongs to a platform with honest or corrupt host. The first macro CheckTtdHonest is used when the functionality stores a new key and tracing trapdoor τ that belongs to a platform with honest TPM, and checks that none of the existing valid signatures are identified as belonging to this TPM key. The second macro CheckTtdCorrupt is used when storing a new tracing trapdoor τ that belongs to a corrupt TPM, and checks that the new τ does not break the identifiability of signatures, i.e., it checks that there is no other known tracing trapdoor $\tau' \neq \tau$, such that both keys are identified as the owner of a signature. Both macros output a bit b indicating whether the new τ is consistent with the stored information or not. Note that these macros are slightly modified from the original $\mathcal{F}^{l}_{\mathsf{daa}}$ by Camenisch et al. [CDL16b]. In their functionality, the signing value *gsk* is also used as the tracing trapdoor. Our functionality is more generic, as the signing key and tracing trapdoor can have different values.

The two macros CheckTtdHonest and CheckTtdCorrupt are defined as follows:

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

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

C Auxiliary Ideal Functionalities

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

C.1 Special Authenticated Communication between TPM and Issuer

We use the authenticated channel from the TPM to the issuer via the host functionalty $\mathcal{F}_{\mathsf{auth}*}$ as defined in [CDL16b].

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 $(\mathsf{REPLACE1}, sid, ssid, m_1, m_2, F)$ to $\mathcal{S}$.
2. On input $(\mathsf{REPLACE1}, sid, ssid, m_2')$ 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 $(\mathsf{REPLACE2}, sid, ssid, m_1, m_2'')$ to $\mathcal{S}$.
4. On input $(\mathsf{REPLACE2}, sid, ssid, m_2''')$ from $\mathcal{S}$, output $(\mathsf{SENT}, sid, ssid, m_1, m_2''')$ to R.

Fig. 10. The special authenticated communication functionality $\mathcal{F}_{\mathsf{auth}*}$.

C.2 Certification Authority

We use the ideal certification authority functionality $\mathcal{F}_{\mathsf{ca}}$ as defined in [Can04], extended to allow one party to register multiple keys, i.e., we check $sid = (P, sid')$ for some sid' instead of checking $sid = P$.

1. Upon receiving the first message $(\mathsf{Register}, sid, v)$ from P, send $(\mathsf{Registered}, sid, v)$ to the adversary; upon receiving ok from the adversary, and if $sid = (P, sid')$ 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 $(\mathsf{Retrieve}, sid, \mathsf{v})$ to P'. Else output $(\mathsf{Retrieve}, sid, \perp)$ to P'.

Fig. 11. Ideal certification authority functionality $\mathcal{F}_{\mathsf{ca}}$, extended from the one by Canetti [Can04].

C.3 Common Reference String

For the crs functionality we use the 2005 version of UC [Can00]. 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 $r \overset{\$}{\leftarrow} D$. Finally, send a public delayed output (CRS, sid, r) to P.

Fig. 12. Ideal CRS functionality $\mathcal{F}_{\mathsf{crs}}^{D}$ by Canetti [Can00].

D Proof of Theorem 2 (Security of our DAA Scheme)

We now prove Theorem 2. We have to prove that our scheme realizes $\mathcal{F}_{\mathsf{pdaa}}$, which means proving that for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every environment $\mathcal{E}$ we have $\mathrm{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} \approx \mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{E}}$.

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

D.1 Functionalities and Simulators

Setup

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

Join

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, sid, jsid)$ from TPM $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid), \mathcal{M}_i)$ to $\mathcal{S}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid), \mathcal{I})$ to $\mathcal{S}$.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn), \mathcal{H}_j)$ to $\mathcal{S}$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

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

Link

8. `Link.` On input $(\mathsf{LINK}, sid, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{LINK}, sid, \sigma, m, \sigma', m', bsn), \mathcal{V})$ to $\mathcal{S}$.

Fig. 13. $\mathcal{F}$ for Game 3

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

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

Fig. 14. Simulator for GAME 3

Setup

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

2. `Join Request.` On input ($\mathsf{JOIN}, sid, jsid, \mathcal{M}_i$) from host $\mathcal{H}_j$.
   - Output ($\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j$) to $\mathcal{S}$.
3. $\mathcal{M}$ `Join Proceed.` On input ($\mathsf{JOIN}, sid, jsid$) from TPM $\mathcal{M}_i$.
   - Output ($\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid), \mathcal{M}_i$) to $\mathcal{S}$.
4. $\mathcal{I}$ `Join Proceed.` On input ($\mathsf{JOINPROCEED}, sid, jsid$) from $\mathcal{I}$.
   - Output ($\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid), \mathcal{I}$) to $\mathcal{S}$.

Sign

5. `Sign Request.` On input ($\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn$) from $\mathcal{H}_j$.
   - Output ($\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn), \mathcal{H}_j$) to $\mathcal{S}$.
6. `Sign Proceed.` On input ($\mathsf{SIGNPROCEED}, sid, ssid$) from $\mathcal{M}_i$.
   - Output ($\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i$) to $\mathcal{S}$.

Verify

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

Link

8. `Link.` On input ($\mathsf{LINK}, sid, \sigma, m, \sigma', m', bsn$) from a party $\mathcal{V}$.
   - Output ($\mathsf{FORWARD}, (\mathsf{LINK}, sid, \sigma, m, \sigma', m', bsn), \mathcal{V}$) to $\mathcal{S}$.

Fig. 15. $\mathcal{F}$ for Game 4

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

Setup

Honest $\mathcal{I}$

- On input (SETUP, sid) from $\mathcal{F}$.
  - Parse sid as $(\mathcal{I}, sid')$ and give "$\mathcal{I}$" input (SETUP, sid).
  - When "$\mathcal{I}$" outputs $(\mathsf{SETUPDONE}, sid)$, $\mathcal{S}$ takes its secret key isk and defines the following algorithms.
    - Define $\mathsf{sig}(((tsk, hsk), gpk), m, bsn)$ as follows: First, create a credential by taking encryption key $(epk, esk) \leftarrow \mathsf{EncKGen}()$. Encrypt the credential with $C \leftarrow \mathsf{Enc}(epk, gpk)$, and sign the ciphertext with $cred' \leftarrow \mathsf{EncSign}(isk, epk, C)$., and decrypt credential $cred \leftarrow \mathsf{DecSign}(esk, cred')$. Next, the algorithm performs the real world signing algorithm (performing both the tasks from the host and the TPM).
    - Define $\mathsf{ver}(\sigma, m, bsn)$ as the real world verification algorithm, except that the private-key revocation check is ommitted.
    - Define $\mathsf{link}(\sigma, m, \sigma', m', bsn)$ as the real world linking algorithm.
    - Define $\mathsf{identify}(\sigma, m, bsn, \tau)$ as follows: parse σ as $(tag, nym, \pi_{\mathsf{SIGN}})$ and check $\mathsf{SSIG.Vf}(\tau, nym, (1, bsn))$. If so, output 1, otherwise 0.
    - Define ukgen as follows: Let $(tpk, tsk) \leftarrow \mathsf{SSIG.PreKeyGen}()$, $(gpk, hsk) \leftarrow \mathsf{SSIG.CompleteKeyGen}(tpk)$, and output $((tsk, hsk), gpk)$.

    $\mathcal{S}$ sends $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.

Corrupt $\mathcal{I}$

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

Forwarded Input

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

Fig. 16. Simulator for Game 4

Setup

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

2. `Join Request.` On input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, sid, jsid)$ from TPM $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, sid, jsid), \mathcal{M}_i)$ to $\mathcal{S}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, sid, jsid), \mathcal{I})$ to $\mathcal{S}$.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn), \mathcal{H}_j)$ to $\mathcal{S}$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

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

Link

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

Fig. 17. $\mathcal{F}$ for Game 5

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

Setup

Honest $\mathcal{I}$

– On input (SETUP, sid) from $\mathcal{F}$.
  - Parse sid as $(\mathcal{I}, sid')$ and give "$\mathcal{I}$" input (SETUP, sid).
  - When "$\mathcal{I}$" outputs $(\mathsf{SETUPDONE}, sid)$, $\mathcal{S}$ takes its secret key isk and defines the following algorithms.
    * Define $\mathsf{sig}(((tsk, hsk), gpk), m, bsn)$ as follows: First, create a credential by taking encryption key $(epk, esk) \leftarrow \mathsf{EncKGen}()$. Encrypt the credential with $C \leftarrow \mathsf{Enc}(epk, gpk)$, and sign the ciphertext with $cred' \leftarrow \mathsf{EncSign}(isk, epk, C)$., and decrypt credential $cred \leftarrow \mathsf{DecSign}(esk, cred')$. Next, the algorithm performs the real world signing algorithm (performing both the tasks from the host and the TPM).
    * Define $\mathsf{ver}(\sigma, m, bsn)$ as the real world verification algorithm, except that the private-key revocation check is ommitted.
    * Define $\mathsf{link}(\sigma, m, \sigma', m', bsn)$ as the real world linking algorithm.
    * Define $\mathsf{identify}(\sigma, m, bsn, \tau)$ as follows: parse σ as $(tag, nym, \pi_{\mathsf{SIGN}})$ and check $\mathsf{SSIG.Vf}(\tau, nym, (1, bsn))$. If so, output 1, otherwise 0.
    * Define ukgen as follows: Let $(tpk, tsk) \leftarrow \mathsf{SSIG.PreKeyGen}()$, $(gpk, hsk) \leftarrow \mathsf{SSIG.CompleteKeyGen}(tpk)$, and output $((tsk, hsk), gpk)$.

    $\mathcal{S}$ sends $(\mathsf{ALG}, sid, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.

Corrupt $\mathcal{I}$

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

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

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

Forwarded Input

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

Fig. 18. Simulator for GAME 5

Setup

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

2. `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 $(\mathsf{JOIN}, sid, jsid, \mathcal{H}_j)$ to $\mathcal{M}_i$.
3. $\mathcal{M}$ `Join Proceed`. On input $(\mathsf{JOIN}, sid, jsid)$ from TPM $\mathcal{M}_i$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = request$ to $delivered$.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{A}$.
   - Abort if $\mathcal{I}$ or $\mathcal{M}_i$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in$ `Members` already exists.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
4. $\mathcal{I}$ `Join Proceed`. On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau\rangle$ into `Members` and output $(\mathsf{JOINED}, sid, jsid)$ to $\mathcal{H}_j$.

Sign

5. `Sign Request`. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn), \mathcal{H}_j)$ to $\mathcal{S}$.
6. `Sign Proceed`. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, sid, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

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

Link

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

Fig. 19. $\mathcal{F}$ for GAME 6

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

Isolated Corrupt TPM

When a TPM $\mathcal{M}_i$ becomes isolated corrupted in the simulated real world, $\mathcal{S}$ defines a local simulator $\mathcal{S}_{\mathcal{M}_i}$ that simulates an honest host with the isolated corrupt $\mathcal{M}_i$. Note that $\mathcal{M}_i$ only talks to one host, who's identity is fixed upon receiving the first message. $\mathcal{S}_{\mathcal{M}_i}$ is defined as follows.

- When $\mathcal{S}_{\mathcal{M}_i}$ receives $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{H}_j)$ as $\mathcal{M}_i$ is isolated corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, sid, jsid)$, send $(\mathsf{JOINPROCEED}, sid, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Setup

Unchanged.

Join

<u>Honest $\mathcal{M}$, $\mathcal{H}$, $\mathcal{I}$</u>

- On input $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{M}_i$" outputs $(\mathsf{JOIN}, sid, jsid, \mathcal{H}_j)$, give "$\mathcal{M}_i$" input $(\mathsf{JOIN}, sid, jsid)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$, output $(\mathsf{JOINPROCEED}, sid, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$.
  - Give "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, sid, jsid)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, sid, jsid)$, output $(\mathsf{JOINCOMPLETE}, sid, jsid, \perp)$ to $\mathcal{F}$.

<u>Honest $\mathcal{H}$, $\mathcal{I}$, Corrupt $\mathcal{M}$</u>

- When $\mathcal{S}$ receives $(\mathsf{JOIN}, sid, jsid)$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$, send $(\mathsf{JOIN}, sid, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINPROCEED}, sid, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$.
  - Give "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, sid, jsid)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, sid, jsid)$, output $(\mathsf{JOINCOMPLETE}, sid, jsid, \perp)$ to $\mathcal{F}$.

<u>Honest $\mathcal{M}$, $\mathcal{H}$, Corrupt $\mathcal{I}$</u>

- On input $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{M}_i$" outputs $(\mathsf{JOIN}, sid, jsid, \mathcal{H}_j)$, give "$\mathcal{M}_i$" input $(\mathsf{JOIN}, sid, jsid)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, sid, jsid)$, output $(\mathsf{JOINPROCEED}, sid, jsid)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ from $\mathcal{F}$ as $\mathcal{I}$ is corrupt.
  - Send $(\mathsf{JOINPROCEED}, sid, jsid)$ on $\mathcal{I}$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$.
  - output $(\mathsf{JOINCOMPLETE}, sid, jsid, \perp)$ to $\mathcal{F}$.

Fig. 20. First part of Simulator for Game 6

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

- $\mathcal{S}$ notices this join as "$\mathcal{M}_i$" outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{H}_j)$.
  - Send $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Continue simulating "$\mathcal{M}_i$" by giving it input $(\mathsf{JOINPROCEED}, sid, jsid)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$, extract gpk from $\pi_{\mathsf{JOIN},\mathcal{H}}$ and output $(\mathsf{JOINPROCEED}, sid, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
  - output $(\mathsf{JOINCOMPLETE}, sid, jsid, gpk)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINED}, sid, jsid)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupt.
  - Continue simulating "$\mathcal{I}$" by giving it input $(\mathsf{JOINPROCEED}, sid, jsid)$.

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

- When $\mathcal{S}$ receives $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ as $\mathcal{M}_i$ is corrupt.
  - Send $(\mathsf{JOIN}, sid, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, sid, jsid)$, output $(\mathsf{JOINPROCEED}, sid, jsid)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ as $\mathcal{I}$ is corrupt.
  - Send $(\mathsf{JOINPROCEED}, sid, jsid)$ on $\mathcal{I}$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINCOMPLETE}, sid, jsid, \perp)$ to $\mathcal{F}$.

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

- $\mathcal{S}$ notices this join as "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$.
  - Extract gpk from $\pi_{\mathsf{JOIN},\mathcal{H}}$ and output $(\mathsf{JOINPROCEED}, sid, jsid)$ to $\mathcal{F}$.
  - Pick some corrupt identity $\mathcal{H}_j$, and send $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{H}_j)$ as $\mathcal{M}_i$ is corrupt.
  - Send $(\mathsf{JOINPROCEED}, sid, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINPROCEED}, sid, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid, gpk)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINED}, sid, jsid)$ as $\mathcal{H}_j$ is corrupt.
  - Give "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, sid, jsid)$.

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

- $\mathcal{S}$ notices this join as "$\mathcal{M}_i$" outputs $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{H}_j)$.
  - Send $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINPROCEED}, sid, jsid)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i)$ as $\mathcal{I}$ is corrupt.
  - Send $(\mathsf{JOINPROCEED}, sid, jsid)$ on $\mathcal{I}$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, sid, jsid)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINCOMPLETE}, sid, jsid, \perp)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINED}, sid, jsid)$ as $\mathcal{H}_j$ is corrupt.
  - Give "$\mathcal{M}_i$" input $(\mathsf{JOINPROCEED}, sid, jsid)$.

Fig. 21. Second part of Simulator for GAME 6

Honest $\mathcal{H}$, $\mathcal{I}$, Isolated corrupt $\mathcal{M}$

- On input ($\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j$).
  - Give "$\mathcal{H}_j$" input ($\mathsf{JOIN}, sid, jsid, \mathcal{M}_i$).
  - When "$\mathcal{M}_i$" outputs ($\mathsf{JOINPROCEED}, sid, jsid, \mathcal{H}_j$), Give "$\mathcal{M}_i$" input ($\mathsf{JOINPROCEED}, sid, jsid$).
  - When "$\mathcal{I}$" outputs ($\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i$), output ($\mathsf{JOINPROCEED}, sid, jsid$) to $\mathcal{F}$.
- On input ($\mathsf{JOINCOMPLETE}, sid, jsid$).
  - Give "$\mathcal{I}$" input ($\mathsf{JOINPROCEED}, sid, jsid$).
  - When "$\mathcal{H}_j$" outputs ($\mathsf{JOINED}, sid, jsid$), output ($\mathsf{JOINCOMPLETE}, sid, jsid, \perp$) to $\mathcal{F}$.

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

- On input ($\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j$).
  - Give "$\mathcal{H}_j$" input ($\mathsf{JOIN}, sid, jsid, \mathcal{M}_i$).
  - When "$\mathcal{M}_i$" outputs ($\mathsf{JOINPROCEED}, sid, jsid, \mathcal{H}_j$), Give "$\mathcal{M}_i$" input ($\mathsf{JOINPROCEED}, sid, jsid$).
  - When "$\mathcal{H}_j$" outputs ($\mathsf{JOINED}, sid, jsid$), output ($\mathsf{JOINPROCEED}, sid, jsid$) to $\mathcal{F}$.
- When $\mathcal{S}$ receives ($\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i$) as $\mathcal{I}$ is corrupt.
  - Send ($\mathsf{JOINPROCEED}, sid, jsid$) on $\mathcal{I}$'s behalf to $\mathcal{F}$.
- On input ($\mathsf{JOINCOMPLETE}, sid, jsid$) from $\mathcal{F}$.
  - output ($\mathsf{JOINCOMPLETE}, sid, jsid, \perp$) to $\mathcal{F}$.

Forwarded Input

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

Fig. 22. Third part of Simulator for GAME 6

Setup

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

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

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

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

Link

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

Fig. 23. $\mathcal{F}$ for Game 7

Isolated Corrupt TPM
When a TPM $\mathcal{M}_i$ becomes isolated corrupted in the simulated real world, $\mathcal{S}$ defines a local simulator $\mathcal{S}_{\mathcal{M}_i}$ that simulates an honest host with the isolated corrupt $\mathcal{M}_i$. Note that $\mathcal{M}_i$ only talks to one host, who's identity is fixed upon receiving the first message. $\mathcal{S}_{\mathcal{M}_i}$ is defined as follows.

– When $\mathcal{S}_{\mathcal{M}_i}$ receives $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{H}_j)$ as $\mathcal{M}_i$ is isolated corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, sid, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, sid, jsid)$, send $(\mathsf{JOINPROCEED}, sid, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

– When $\mathcal{S}_{\mathcal{M}_i}$ receives $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ as $\mathcal{M}_i$ is isolated corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$, send $(\mathsf{SIGNPROCEED}, sid, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Setup
Unchanged.
Join
Unchanged.
Sign
Honest $\mathcal{M}$, $\mathcal{H}$
Nothing to simulate.
Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

– When $\mathcal{S}$ receives $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$, send $(\mathsf{SIGNPROCEED}, sid, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{H}$, Isolated corrupt $\mathcal{M}$
Nothing to simulate.
Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

– When "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$.
  - Send $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
  - When $\mathcal{S}$ receives $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupt, give "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, sid, ssid)$.

Fig. 24. Simulator for GAME 7

Setup

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

2. `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 $(\mathsf{JOIN}, sid, jsid, \mathcal{H}_j)$ to $\mathcal{M}_i$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, sid, jsid)$ from TPM $\mathcal{M}_i$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = request$ to $delivered$.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{A}$.
   - 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 $\mathcal{I}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   - Else, verify that the provided tracing trapdoor τ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid)$ to $\mathcal{H}_j$.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in $\mathtt{Members}$, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in $\mathtt{Members}$.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\mathtt{DomainKeys}$.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in $\mathtt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

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

Link

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

Fig. 25. $\mathcal{F}$ for Game 8

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Fig. 26. Simulator for Game 8

Setup

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

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

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   – If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`, abort.
   – Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   – Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   – Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   – If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`.
   – Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, **check $\mathsf{CheckTtdHonest}(\tau) = 1$,** and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
   – Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

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

Link

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

Fig. 27. $\mathcal{F}$ for GAME 9

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Fig. 28. Simulator for Game 9

Setup

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

2. `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 $(\mathsf{JOIN}, sid, jsid, \mathcal{H}_j)$ to $\mathcal{M}_i$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, sid, jsid)$ from TPM $\mathcal{M}_i$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = request$ to $delivered$.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{A}$.
   - 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 $\mathcal{I}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
   - Else, verify that the provided tracing trapdoor τ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid)$ to $\mathcal{H}_j$.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in $\mathtt{Members}$, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in $\mathtt{Members}$.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\mathtt{DomainKeys}$.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor τ' registered in $\mathtt{Members}$ or $\mathtt{DomainKeys}$ with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in $\mathtt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

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

Link

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

Fig. 29. $\mathcal{F}$ for Game 10

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Fig. 30. Simulator for GAME 10

Setup

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

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

Sign

5. `Sign Request`. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed`. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor τ' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

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

Link

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

Fig. 31. $\mathcal{F}$ for Game 11

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Fig. 32. Simulator for Game 11

Setup

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

2. `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 $(\mathsf{JOIN}, sid, jsid, \mathcal{H}_j)$ to $\mathcal{M}_i$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, sid, jsid)$ from TPM $\mathcal{M}_i$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = request$ to $delivered$.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{A}$.
   - 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 $\mathcal{I}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   - Else, verify that the provided tracing trapdoor τ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid)$ to $\mathcal{H}_j$.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in $\mathtt{Members}$, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in $\mathtt{Members}$.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\mathtt{DomainKeys}$.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor τ' registered in $\mathtt{Members}$ or $\mathtt{DomainKeys}$ with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in $\mathtt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

7. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i\rangle \in \mathtt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in \mathtt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one τ_i was found.
     - $\mathcal{I}$ is honest and no tuple $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to $\mathtt{VerResults}$ and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

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

Fig. 33. $\mathcal{F}$ for Game 12

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Fig. 34. Simulator for GAME 12

Setup

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

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

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \texttt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor τ' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in `Signed` and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

7. `Verify.` On input $(\mathsf{VERIFY}, sid, m, bsn, \sigma, \texttt{RL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i\rangle \in \texttt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i\rangle \in \texttt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one τ_i was found.
     - $\mathcal{I}$ is honest and no tuple $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
     - $\mathcal{M}_i$ or $\mathcal{H}_j$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle \in \texttt{Signed}$ exists.
     - There is a $\tau' \in \texttt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
   - Add $\langle \sigma, m, bsn, \texttt{RL}, f\rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, sid, f)$ to $\mathcal{V}$.

Link

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

Fig. 35. $\mathcal{F}$ for GAME 13

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Fig. 36. Simulator for Game 13

Setup

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

2. `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 $(\mathsf{JOIN}, sid, jsid, \mathcal{H}_j)$ to $\mathcal{M}_i$.
3. $\mathcal{M}$ `Join Proceed`. On input $(\mathsf{JOIN}, sid, jsid)$ from TPM $\mathcal{M}_i$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = request$ to $delivered$.
   - Output $(\mathsf{JOINPROCEED}, sid, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{A}$.
   - 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 $\mathcal{I}$.
4. $\mathcal{I}$ `Join Proceed`. On input $(\mathsf{JOINPROCEED}, sid, jsid)$ from $\mathcal{I}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - Output $(\mathsf{JOINCOMPLETE}, sid, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, sid, jsid, \tau)$ from $\mathcal{A}$.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
   - Else, verify that the provided tracing trapdoor τ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau\rangle$ into $\mathtt{Members}$ and output $(\mathsf{JOINED}, sid, jsid)$ to $\mathcal{H}_j$.

Sign

5. `Sign Request`. On input $(\mathsf{SIGN}, sid, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in $\mathtt{Members}$, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, sid, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed`. On input $(\mathsf{SIGNPROCEED}, sid, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in $\mathtt{Members}$.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in $\mathtt{DomainKeys}$.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor τ' registered in $\mathtt{Members}$ or $\mathtt{DomainKeys}$ with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in $\mathtt{Signed}$ and output $(\mathsf{SIGNATURE}, sid, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

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

Link

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

Fig. 37. $\mathcal{F}$ for Game 14

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Fig. 38. Simulator for GAME 14

Setup

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

Join

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

Sign

5. `Sign Request.` On input (SIGN, sid, $ssid$, $\mathcal{M}_i$, m, bsn) from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output ($\mathsf{SIGNPROCEED}$, sid, $ssid$, m, bsn) to $\mathcal{M}_i$.
6. `Sign Proceed.` On input ($\mathsf{SIGNPROCEED}$, sid, $ssid$) from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor τ' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in `Signed` and output ($\mathsf{SIGNATURE}$, sid, $ssid$, σ) to $\mathcal{H}_j$.

Verify

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

Link

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

Fig. 39. $\mathcal{F}$ for GAME 15

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Fig. 40. Simulator for Game 15

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

Game 1: This is the real world.

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

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

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

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

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

Game 7: $\mathcal{F}$ now handles the sign queries. When one party creates two signatures with different basenames, $\mathcal{F}$ signs with different keys, showing that the signatures are unlinkable. $\mathcal{S}$ can simulate the real world protocol and block any signatures that would not be successfully generated in the real world. $\mathcal{F}$ may prevent a signature from being output, when the TPM and host did not yet join, or when the signature generated by $\mathcal{F}$ does not pass verification. If the TPM and host did not join, and the host is honest, the real world would also not output a signature, as the host performs this check. The signatures $\mathcal{F}$ generate will always pass verification, as the algorithms that $\mathcal{S}$ set in GAME 4 will only create valid signatures (by completeness of the split signatures, signatures on encrypted messages, and zero-knoweldge proofs). This shows that $\mathcal{F}$ outputs a signature if and only if the real world would outputs a signature.

What remains to show is that the signatures that $\mathcal{F}$ outputs are indistinguishable from the real world signatures. We make this change gradually. First, all signatures come from the real world, and then we let $\mathcal{F}$ gradually create more signatures, until all signatures come from $\mathcal{F}$. Let GAME $7.i.j$ denote the game in which $\mathcal{F}$ creates all signatures for platforms with TPMs $\mathcal{M}_{i'}$ with $i' < i$, lets $\mathcal{S}$ create the signatures if $i' > i$, and for the platform with TPM $\mathcal{M}_i$, the first j distinct basenames are signed. We show that GAME $7.i.j$ is indistinguishable from GAME $7.i.(j+1)$, and by repeating this argument, we have GAME $7 \approx$ GAME 6.

Proof of GAME $7.i.j \approx$ GAME $7.i.(j+1)$ We make small changes to GAME $7.i.j$ and GAME $7.i.(j+1)$, and then show that the remaining difference can be reduced to the key hiding property of the split signatures.

First, we let the NIZK proofs in join and in the signatures be simulated, which is indistinguishable by the zero-knowledge property of the proofs. Second, we encrypt dummy values in join and sign, instead of encrypting *cred* and *gpk*. Under the CPA security of the encryption scheme, this is indistinguishable.[7] Note that the host cannot decrypt his credential while reducing to the CPA security, which means he cannot verify the credential and he cannot later use it to sign. Proof $\pi_{\mathsf{JOIN},\mathcal{I}}$ guarantees that the encrypted credential is valid, so it still aborts when the issuer tries to send a invalid credential. The simulator simulating the honest host can solve the second problem: since GAME 4, the simulator knows the issuer secret key and can therefore create an equivalent credential.

Now, the only remaning difference is the computation of *tag* and *nym*. In GAME $7.i.j$, $\mathcal{S}$ computes these values using the same key as it joined with, and in GAME $7.i.(j+1)$, $\mathcal{F}$ uses a fresh key.

We first show that the difference in *nym* is indistinguishable under the key hiding property of the split signatures. $\mathcal{S}$ simulates the honest host without knowing *gpk*. In the join, it uses a dummy ciphertext and simulates the proof. Signatures with basename $bsn_{j'}$ are handled as follows.

- $j' \leq j$: these signatures are created by $\mathcal{F}$.
- $j' = j+1$: $\mathcal{S}$ gives the challenger of the key hiding game of split signatures message $bsn_{j'}$, giving it the pseudonym for $bsn_{j'}$. As the split signatures are unique, we can use this pseudonym for every signature with $bsn_{j'}$.
- $j' > j+1$: $\mathcal{S}$ uses $\mathcal{O}^{\mathsf{CompleteSign}}$ to compute *tag* and *nym*.

If the bit in the key hiding game is zero, *nym* is computed like in GAME $7.i.j$, and if one, *nym* is computed like in GAME $7.i.(j+1)$, so any environment distin-

[7] Note that $\mathcal{S}$ previously held the trapdoor to the crs encryption key. $\mathcal{S}$ only uses this to extract *gpk* in the join and gives it to $\mathcal{F}$. Since $\mathcal{F}$ does not use this extracted value yet, we can omit these extractions here, and use the CPA property of the encryption scheme.

guishing the different ways to compute *nym* can break the key hiding property of the split signatures.

What remains to show is that using a fresh key for every basename in the computation of *tag* is also indistinguishable. Here we make the same reduction to the key hiding property of split signatures, but now we make a reduction per message that the platform signs with this basename.

Game 8: $\mathcal{F}$ now runs the CheckTtdCorrupt algorithm when $\mathcal{S}$ gives the extracted *gpk* from platforms with a corrupt host. This checks that $\mathcal{F}$ has not seen valid signatures yet that match both this key and existing key. If this happens, we break the key-uniqueness property of the split signatures, so GAME 8 $\approx$ GAME 7.

Game 9: When $\mathcal{F}$ creates fresh domain keys when signing for honest platforms, it checks that there are no signatures that match this key. Since $\mathcal{S}$ instantiated the identify algorithm with the verification algorithm of the split signatures, this would mean there already exists a valid signature under the freshly generated key. Clearly, this breaks the unforgeability-1 property of the split signatures, so GAME 9 $\approx$ GAME 8.

Game 10: $\mathcal{F}$ now performs additional tests on the signatures it creates, and if any fails, it aborts. First, it checks whether the generated signature matches the key it was generated with. With the algorithms $\mathcal{S}$ defined in GAME 4, this always holds. Second, $\mathcal{F}$ checks that there is no other platform with a key that matches this signature. By the key uniqueness property, we cannot have two keys matching one signature. The probability that the same key is registered by someone else is negligible. As the *gpk* is always encrypted, by the CPA security of the encryption scheme, the probability that some other party chooses the same key is the probability that it guessed the key correctly without any further information. The following lemma states that this is infeasible.

Lemma 1. *Let Σ be a secure split signature scheme. No adversary, that chooses ppk on input spar $\leftarrow \Sigma.\mathcal{G}(1^\tau)$, has non-negligible probability of outputting spk, with spk computed as $(spk, ssk_2) \leftarrow \Sigma.\mathsf{CompleteKeyGen}(spar, ppk)$, where the probability is taken over all random choices in $\mathcal{G}$ and* CompleteKeyGen.

Proof. Suppose the adversary can guess spk with nonnegligible probability. Then it has non-negligible probability of finding spk and corresponding secret keys (ssk_1, ssk_2) by running the PreKeyGen *and* CompleteKeyGen *algorithms. Now the adversary can forge signatures, breaking unforgeability-1. As Σ is unforgeable-1, no such adversary can exist.*

Game 11: In verification, $\mathcal{F}$ now checks whether it knows multiple tracing keys that match one signature. As $\mathcal{S}$ instantiated the identify with the verification of split signatures, this cannot happen with nonneglibible probability by the key-uniqueness property of the split signatures, GAME 11 $\approx$ GAME 10.

Game 12: When $\mathcal{I}$ is honest, $\mathcal{F}$ verifying a signature now checks whether the signature matches some key of a platform that joined, and if not, rejects the signature. Under the unforgeability of the signature scheme for encrypted messages, this check will trigger only with negligible probability.

When reducing to the unforgeability of the signature scheme for encrypted messages, we do not know the issuer secret key isk. $\mathcal{S}$ simulating $\mathcal{I}$ therefore simulates proof π in the public key of the issuer. When $\mathcal{S}$ must create a credential while simulating the join protocol, it now uses the signing oracle. From C_2, it can extract gpk using its knowledge of the crs trapdoor. It passes gpk to the signing oracle, along with the ephemeral encryption key epk, which allows simulation without knowing isk. $\mathcal{F}$'s algorithms used to be based on the issuer secret key, which we do not know in this reduciton. We let sig now also use the signing oracle. Instead of encrypting gpk with epk, it passes these two values to the signing oracle, and continues as before. Note that any gpk we pass to the signing oracle is stored in `Members` or `DomainKeys`. Now, when we see a valid signature that does not match any of the gpk values stored, we can extract a forgery: Signatures have structure $(tag, nym, \pi_{\mathsf{SIGN}})$, with

$$\pi_{\mathsf{SIGN}} \leftarrow NIZK\{(gpk, cred) : \mathsf{ESIG.Vf}(ipk, cred, gpk) = 1 \ \wedge \\ \mathsf{SSIG.Vf}(gpk, tag, (0, m, bsn)) = 1 \ \wedge \ \mathsf{SSIG.Vf}(gpk, nym, (1, bsn)) = 1\}$$

If the signature does not match any of the keys (using the $\mathsf{identify}$ algorithm), it means that nym is not a valid split signature under any of the gpk values for which an oracle query has been made. By soundness of the proof, $\mathcal{S}$ can extract a credential on the gpk value used, which will be a forgery. Note that as we perform this extraction only in reductions, online extractability is not required.

As the signature scheme for encrypted messages is unforgeable, we have GAME 12 $\approx$ GAME 11.

Game 13: $\mathcal{F}$ now rejects signatures on message m with basename bsn that match the key of a platform with an honest TPM or honest host, but that platform never signed m w.r.t. bsn. If signatures that would previously have been accepted are now no longer accepted, we can break the unforgeability of the split signatures.

We distinguish three cases: the matching key gpk is found in `Members` and the host is honest, the matchking key is found in `Members` and the host is corrupt, or gpk is found in `DomainKeys`.

[Case 1 – gpk in `Members`, honest host]. We make this change gradually, for each TPM $\mathcal{M}_i$ individually.

$\mathcal{S}$ receives the system parameters, which it puts in the crs. For the case that the matching key is in `Members`, $\mathcal{S}$ gives the TPM's ppk to the challenger when joining. It then has to simulate the host without knowing its part of the secret key. When signing, the host receives the pre-signatures tag' and nym' on messages $(0, m, bsn)$ and $(1, bsn)$ respectively. Now, when a signature σ on message m w.r.t. basename bsn is found that matches the platform's key while the platform never signed m w.r.t. bsn, we can extract a forgery. By soundness of the NIZK proof, we can extract tag with $\mathsf{Vf}(spar, spk, \sigma, (0, m, bsn)) = 1$, and we never queried $\mathcal{O}^{\mathsf{CompleteSign}}$ on $(0, m, bsn)$, giving a forgery.

[Case 2 – gpk in `DomainKeys`*, honest host].* Let GAME 13.$i.j$ denote the game in which $\mathcal{F}$ prevents forgery for keys in `DomainKeys` of the platform with TPM $\mathcal{M}_{i'}$ and $i' < i$, and prevents forgery under the keys in domainkeys with $bsn_{j'}$, $j' < j$ of the platform with TPM $\mathcal{M}_i$ lets $\mathcal{S}$ create the signatures if $i' > i$, and for the platform with TPM $\mathcal{M}_i$, the first j distinct basenames are signed. We show that GAME 13.$i.j$ is indistinguishable from GAME 13.$i.(j+1)$ under unforgeability-1 of the split signatures.

$\mathcal{S}$ receives the system parameters, which it puts in the crs. $\mathcal{S}$ now changes the algorithms it gives to $\mathcal{F}$, such that on input bsn_j, it runs $(ppk, tsk) \leftarrow \mathsf{PreKeyGen}(spar)$ and gives ppk to the challenger. $\mathcal{S}$ receives gpk, for which it does not know the full secret key. When $\mathcal{F}$ wants to sign using gpk, it must create tag and nym without knowing the second part of the secret key. It creates the pre-signature using tsk, and completes the signature using $\mathcal{O}^{\mathsf{CompleteSign}}$. Now, when $\mathcal{F}$ notices a signature on message m w.r.t. basename bsn that the platform never signed, it means it did not query $\mathcal{O}^{\mathsf{CompleteSign}}$ on $(0, m, bsn)$, so we can extract tag which is a forgery on $(0, m, bsn)$.

If the host is corrupt, but the TPM is honest, we can reduce to the unforgeability-2 property of split signatures. We again distinguish between a matching key in `Members` and `DomainKeys`.

[Case 3 – gpk in `Members`*, honest TPM, corrupt host].* We make this change gradually, for each TPM $\mathcal{M}_i$ individually.

$\mathcal{S}$ receives the system parameters, which it puts in the crs. When $\mathcal{S}$ simulates $\mathcal{M}_i$ joining, instead of running $\mathsf{PreKeyGen}$, it uses the ppk as received from the challenger. When $\mathcal{S}$ simulating the issuer receives gpk and π_1 from the platform with $\mathcal{M}_i$, it extracts hsk such that $\mathsf{VerKey}(spar, ppk, spk, hsk) = 1$. Whenever $\mathcal{S}$ must pre-sign using the unknown tsk, it calls $\mathcal{O}^{\mathsf{PreSign}}$. When $\mathcal{F}$ sees a signature matching this platform's key gpk on message m w.r.t. basename bsn that $\mathcal{M}_i$ never signed, extract tag, which is a valid signature on $(0, m, bsn)$ under gpk. Now the unforgeability-2 game is won by submitting $((0, m, bsn), tag, gpk, hsk)$.

Game 14: $\mathcal{F}$ now prevents revocation of platforms with an honest host. Note that revocation requires a gpk value of the platform to be placed on the revocation list. We now show that no environment has nonnegligible probability of entering these values.

For platforms with an honest host, we can remove all information on gpk. First, when we encrypt gpk, tag, or $cred$, we encrypt dummy values instead and simulate the proofs. Second, we can replace the nym values by signatures under different keys, by the key hiding property of the split signatures. Now, the environment must simply guess gpk. By Lemma 1, the probability of guessing the public key of a split signature scheme correctly is negligible, so GAME 14 $\approx$ GAME 13.

Game 15: $\mathcal{F}$ answering linking queries now uses its tracing information to answer the queries. Previously, it compared nym and nym', valid split signatures on bsn under keys gpk and gpk' respectively. If $nym = nym'$, $\mathcal{F}$ answered 1, and otherwise 0.

$\mathcal{F}$ now takes all the *gpk* values it knows and if it finds some *gpk* such that one *nym* is valid under *gpk*, but nym' is not, it outputs that the signatures are not linked. Clearly, in this case we must have $nym \neq nym'$, so the linking decision does not change. If $\mathcal{F}$ finds some *gpk* such that both *nym* and nym' are valid signatures on *bsn* under *gpk*, it outputs that the signatures are linked. By signature uniqueness, we have $nym = nym'$, so again, the linking decision does not change. This shows Game 15 $\approx$ Game 14.