DAA: Fixing the pairing based protocols [⋆]

Liqun Chen[1], Paul Morrissey[2] and Nigel P. Smart[2]

[1] Hewlett-Packard Laboratories,
Filton Road,
Stoke Gifford,
Bristol, BS34 8QZ,
United Kingdom.
`liqun.chen@hp.com`

[2] Computer Science Department,
Woodland Road,
University of Bristol,
Bristol, BS8 1UB,
United Kingdom.
{`paulm, nigel`}`@cs.bris.ac.uk`

Abstract. In [17, 18] we presented a pairing based DAA protocol in the asymmetric setting, along with a "security proof". Jiangtao Li has pointed out to us an attack against this published protocol, thus our prior work should not be considered sound.

In this paper we give a repaired version, along with a highly detailed security proof.

A full paper will be made available shortly. However in the meantime we present this paper for the community to check and comment on.

1 Introduction

In [17, 18] we presented a pairing based DAA protocol in the asymmetric setting, along with a "security proof". Jiangtao Li have pointed out to us an attack against this published protocol. Following this we re-examined our security proof and found a number of other major attacks on our scheme, and problems with the "proof". We also found attacks on the symmetric pairing based protocol of [9] and found similar mistakes in the security proof of the symmetric pairing based scheme of [9] (despite it being presented for a different security model).

In trying to repair our protocol we found that the simulation based DAA security model presented in [7] was missing some crucial details. Indeed the proof of the RSA based scheme in [7] does prove security of the protocol, but not with respect to the model as explicitly written in the same paper.

Hence, in this paper we give three contributions:

1. A fully detailed simulation based security model for the DAA protocol.
2. A new asymmetric pairing based DAA protocol.
3. A fully detailed security proof of the new scheme in the new model. In particular we present all details in full so as to avoid the mistakes made in other proofs (which essentially occurred when reduction algorithms for failure events were not worked out in detail).

A full paper will be made available shortly. However in the meantime we present this paper for the community to check and comment on. We would welcome feedback and comments on each of the three contributions, and we wish to apologise to the community for the mistakes in our earlier protocol.

[⋆] The second and third author would like to thank EPSRC and the EU funded project eCrypt for partially supporting the work in this paper. The third author was supported by a Royal Society Wolfson Merit Award. Most of the work in this paper was done whilst the second author was on placement with HP Labs Bristol.

2 Definitions and Security Models of DAA

2.1 The DAA Players

We refer to each of the entities in a DAA scheme as *players*. We first describe the types of players we consider in our model. This set of players is the same set as in [10] and is intended to represent a DAA scheme in which a given TPM wishes to remotely and anonymously authenticate itself to a given verifier. Intuitively, the set of players will consist of a set of users, each comprising a Host and a TPM, a set of issuers, and a set of verifiers to which users want to authenticate their TPM.

We now give a formal description of each of the DAA players:

- A set of users $\mathfrak{U}$ where each $\mathfrak{u}_i \in \mathfrak{U}$ consists of
  - A TPM $\mathfrak{m}_i$ from some set of TPMs $\mathfrak{M}$. Each $\mathfrak{m}_i \in \mathfrak{M}$ has an endorsement key pair $\mathsf{ek}_i = (\mathsf{SK}_i, \mathsf{PK}_i)$ and seed $\mathsf{DAAseed}_i$.
  - A Host $\mathfrak{h}_i$ from some set of hosts $\mathfrak{H}$. Each $\mathfrak{h}_i \in \mathfrak{H}$ will have a counter value cnt_i, a set of commitments $\{\mathsf{comm}\}_i$ and a set of credentials $\{\mathsf{cre}\}_i$.
- A set of issuers $\mathfrak{I}$ where each $\mathfrak{i}_k \in \mathfrak{I}$ has a public and private key pair $(\mathsf{ipk}_k, \mathsf{isk}_k)$ and a long term public value K_k. Each $\mathfrak{i}_k \in \mathfrak{I}$ also maintains a list of rogue TPM internal values, we denote this list by $\mathsf{RogueList}(\mathfrak{i}_k)$.
- A set of verifiers $\mathfrak{V}$. Each verifier $\mathfrak{v}_j \in \mathfrak{V}$ maintains a set of base names $\{\mathsf{bsn}\}_j$ and a list of rogue TPM internal values $\mathsf{RogueList}(\mathfrak{v}_j)$. Each $\mathfrak{v}_j$ may optionally maintain a list of message and signature pairs received (this can be used to trade memory for computation in linking).

We assume that initially the sets $\{\mathsf{comm}\}_i$, $\{\mathsf{cre}\}_i$ are empty for all $\mathfrak{u}_i \in \mathfrak{U}$. In addition we assume that the lists $\mathsf{RogueList}(\mathfrak{i}_k)$ are empty for all $\mathfrak{i}_k \in \mathfrak{I}$, and that the lists $\mathsf{RogueList}(\mathfrak{v}_j)$ and the sets $\{\mathsf{bsn}\}_j$ are empty for all $\mathfrak{v}_j \in \mathfrak{V}$.

It is worth describing the various player parameters and how they relate to each other. Generally, at the time of manufacture, each TPM will have a single endorsement key ek_i embedded into the TPM chip. In addition, each TPM generates a TPM-specific secret $\mathsf{DAAseed}_i$ and stores it in non-volatile memory, this value will never be disclosed or changed by the TPM. We do not consider choosing and assigning the values ek_i and $\mathsf{DAAseed}_i$ in the setup algorithm, since the setup algorithm is run only by an issuer. The $\mathsf{DAAseed}_i$ is generally a 20–byte constant that, together with a given issuer value K_k, allows for the generation and regeneration of a given value of an internal secret key f. Each TPM can have multiple possible values for f (at least one per issuer and possible more if a given issuer has more than one value of K_k). We refer to the set of possible values of f for a given user i as $\{f\}_i$. Since the TPM has limited storage requirements it does not store the current value for f; it regenerates it as required from $\mathsf{DAAseed}_i$. The pair $(\mathsf{cnt}_i, \mathrm{K}_k)$ are unique to a given value of f and can be thought of as an index for a particular f value. For each value of f the TPM will be able to compute many commitments on f. For each value of f, as we will see later, a given issuer could issue multiple credentials. However, each of these credentials will be randomisations of each other and hence the pair $(\mathsf{cnt}_i, \mathrm{K}_k)$ can be thought of as an identifier for a given set of credentials. We assume the Host only stores one such credential for a given f value. The set $\{\mathsf{bsn}\}_j$ is used to achieve user controlled linkability of signatures.

2.2 Formal Definition of a DAA Scheme

Informally, a DAA scheme consists of a system setup algorithm, a protocol for users to obtain credentials, a signing protocol, algorithms for verifying and linking signatures and an algorithm for tagging rogue TPM values. Our definition is similar to that given in [9] but with some modifications. Specifically, we give a single protocol for the joining functionality as opposed to multiple protocols, and our signature functionality is given as a protocol as opposed to an algorithm. Also we have an additional rogue tagging algorithm.

Definition 1 **(Daa Scheme).** Formally, we define a Daa scheme to be a tuple of protocols and algorithms $\mathsf{Daa} = (\mathsf{Setup}, \mathsf{Join}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Link}, \mathsf{RogueTag})$ where:

- $\mathsf{Setup}(1^t)$ is a p.p.t. system setup algorithm. On input 1^t, where t is a security parameter, this outputs a set of system parameters par which contains all of the issuer public keys ipk_k and the various parameter spaces. This algorithm also serves to setup and securely distribute each of the issuer secret keys isk_k.
- $\mathsf{Join}(\mathfrak{u}_i, \mathfrak{i}_k)$ is a three party protocol run between a TPM, a Host and an issuer. In a correct initial run of the protocol with honest players the Host should obtain an additional valid commitment and an additional valid credential. In correct subsequent runs one valid credential should be replaced with another.
- $\mathsf{Sign}(\mathfrak{u}_i, \mathsf{msg}, \mathsf{bsn})$ is a two party protocol run between a TPM and a Host used to generate a signature of knowledge on some message msg with respect to some basename bsn. In a correct run of the protocol with honest players the signature of knowledge will be constructed according to bsn for some specified verifier that will allow the signature to be linked to other signatures with this same verifier and basename, unless $\mathsf{bsn} = \perp$.
- $\mathsf{Verify}(\sigma, \mathsf{msg}, \mathsf{bsn}, \mathsf{ipk}_k)$ is a deterministic polynomial time (d.p.t.) verification algorithm that allows a given verifier to verify a signature of knowledge σ of a credential on a message msg computed with respect to basename bsn and issued by the issuer with public key ipk_k. The verification process will involve checking the signature against the list $\mathsf{RogueList}(\mathfrak{v}_j)$. This algorithm returns either *accept* or *reject*.
- $\mathsf{Link}((\sigma_0, \mathsf{msg}_0), (\sigma_1, \mathsf{msg}_1), \mathsf{bsn}, \mathsf{ipk}_k)$ is a d.p.t. linking algorithm that returns either *linked*, *unlinked* or $\perp$. The algorithm should return $\perp$ if either signature was produced with a rogue key, return *linked* if both are valid signatures on the respective message with respect to the same basename $\mathsf{bsn} \neq \perp$, and return *unlinked* otherwise.
- $\mathsf{RogueTag}(f, \sigma, \mathsf{msg}, \mathsf{bsn}, \mathsf{ipk}_k)$ is a d.p.t. rogue tagging algorithm that returns true if σ is a valid signature for message msg and basename bsn produced using a credential issued by the issuer with public key ipk and using the TPM secret value f and returns false otherwise.

For correctness we require that if

- a user $\mathfrak{u}_i \in \mathfrak{U}$ engages in a run of Join with $\mathfrak{i}_k$, resulting in $\mathfrak{u}_i$ obtaining a commitment comm on a TPM secret value f and a credential cre corresponding to f,
- the user $\mathfrak{u}_i$ then creates two signatures σ_b on two messages msg_b for $b \in \{0, 1\}$ with basename bsn (which could be $\perp$),
- and the secret TPM value used to compute these f is not in $\mathsf{RogueList}$,

then

$$\mathsf{Verify}(\sigma_0, \mathsf{msg}_0, \mathsf{bsn}, \mathsf{ipk}_k) = \mathsf{Verify}(\sigma_1, \mathsf{msg}_1, \mathsf{bsn}, \mathsf{ipk}_k) = \textit{accept}$$

if $\mathsf{bsn} \neq \perp$ then

$$\mathsf{Link}((\sigma_0, \mathsf{msg}_0), (\sigma_1, \mathsf{msg}_1), \mathsf{bsn}, \mathsf{ipk}_k) = \textit{linked},$$

and if $\mathsf{bsn} = \perp$ then

$$\mathsf{Link}((\sigma_0, \mathsf{msg}_0), (\sigma_1, \mathsf{msg}_1), \mathsf{bsn}, \mathsf{ipk}_k) = \textit{unlinked},$$

2.3 The Real/Ideal System based DAA Security Model

In this section we give a detailed description of a slightly modified version of the real/ideal system model [7] for DAA schemes.

Real System Execution for a DAA Scheme. For the real system we model a set of players consisting of $n_{\mathfrak{U}}$ users $\mathfrak{u}_i \in \mathfrak{U}$ each with a host $\mathfrak{h}_i$ and corresponding TPM module $\mathfrak{m}_i$, a set of $n_{\mathfrak{V}}$ verifiers $\mathfrak{v}_j \in \mathfrak{V}$ and a set of $n_{\mathfrak{I}}$ issuers $\mathfrak{i}_k \in \mathfrak{I}$. The honest players in the system receive inputs from and send outputs to the environment Env. Honest players also run cryptographic protocols with each other and perform cryptographic computations themselves according to the description of the DAA scheme. We model an adversary $\mathcal{A}$ as a p.p.t. algorithm that controls a number of corrupt players. Since the adversary controls the set of corrupt players it will arbitrarily interact with other players and Env.

Ideal System Execution for a DAA Scheme. In the ideal system we have the same set of players as in the real system. In addition there exists some trusted third party $\mathcal{T}$. The main difference to the real system is players engage in protocol runs and perform cryptographic computations by passing inputs to $\mathcal{T}$ and receiving outputs from $\mathcal{T}$ rather than performing these themselves according to the scheme.

The trusted third party $\mathcal{T}$ provides the functionality we want from a secure DAA scheme by maintaining a number of lists and making decisions based on these lists. These lists include a list $\mathsf{CorruptTPM}$ of endorsement key and counter pairs, a list of signatures issued $\mathsf{Signatures}$, a list of members $\mathsf{Members}$ and a rogue list $\mathsf{RogueList}$.

We assume whenever a TPM is corrupted it tells $\mathcal{T}$ by sending its index i (which we use as an identifier) to $\mathcal{T}$ who adds this to $\mathsf{CorruptTPM}$. The entries of $\mathsf{Signatures}$ have the form $(\sigma, \mathsf{msg}, \mathsf{bsn}, i, \mathsf{cnt}, \mathfrak{i}_k)$ and each is intended to mean $\mathcal{T}$ computed a signature σ on the message msg on behalf of the user with identifier i using internal secret value, corresponding to cnt, for which this user ran the Join protocol with $\mathfrak{i}_k$, and that is linkable to all other signatures with the same issuer and basename $\mathsf{bsn} \in \{0,1\}^* \cup \perp$ (providing $\mathsf{bsn} \neq \perp$). The entries of $\mathsf{RogueList}$ will contain TPM identifier and counter pairs. The list $\mathsf{Members}$ contains TPM identifier, counter and issuer identifier tuples and is essentially a list of those TPM's that $\mathcal{T}$ has issued credentials to and the issuer on whose behalf these were issued. Intuitively, the list $\mathsf{CorruptTPM}$ is a list of TPM's for which the adversary has complete control and hence knows all values of internal secrets for each value of counter. On the other hand, the list $\mathsf{RogueList}$ will be a list of TPM and internal secret pairs that have been compromised. The adversary may only know a single value of internal secret for a rogue TPM. To this end, $\mathcal{T}$ uses $\mathsf{CorruptTPM}$ to decide if a given identifier and counter pair should be added to $\mathsf{RogueList}$ or not. The trusted third party then performs the following functionality on behalf of players:

Setup. Any corrupted TPM modules inform $\mathcal{T}$ of this by sending their identifier i to $\mathcal{T}$ who adds this to $\mathsf{CorruptTPM}$.

Join. The host $\mathfrak{h}_i$ contacts $\mathcal{T}$ with the identifier i, counter cnt and issuer $\mathfrak{i}_k \in \mathfrak{I}$. Next $\mathcal{T}$ sends cnt to $\mathfrak{m}_i$ and asks if it wants to join with respect to cnt and $\mathfrak{i}_k$. The module $\mathfrak{m}_i$ informs $\mathcal{T}$ of its decision. Then $\mathcal{T}$ sends to $\mathfrak{i}_k$ the pair (i, cnt), and informs $\mathfrak{i}_k$ if $(i, \mathsf{cnt}) \in \mathsf{RogueList}$ or not. Note $\mathcal{T}$ does not add an entry to $\mathsf{RogueList}$. The issuer $\mathfrak{i}_k$ then makes a decision as to whether (i, cnt) can become a member or not and informs $\mathcal{T}$ of this. If $\mathfrak{i}_k$ decides that (i, cnt) can become a member then $\mathcal{T}$ adds $(i, \mathsf{cnt}, \mathfrak{i}_k)$ to $\mathsf{Members}$. Finally $\mathcal{T}$ informs $\mathfrak{h}_i$ of the decision.

Sign. A given host $\mathfrak{h}_i$ requests to sign a message msg with basename bsn using a given pair (i, cnt) by sending to $\mathcal{T}$ a tuple $(\mathsf{msg}, i, \mathsf{cnt}, \mathfrak{i}_k)$.

- If $(i, \mathsf{cnt}, \mathfrak{i}_k) \notin \mathsf{Members}$ for some $\mathfrak{i}_k \in \mathfrak{I}$ then $\mathcal{T}$ denies the request and replies to $\mathfrak{h}_i$ with $\perp$.
- Else if $(i, \mathsf{cnt}, \mathfrak{i}_k) \in \mathsf{Members}$ for some $\mathfrak{i}_k \in \mathfrak{I}$ then $\mathcal{T}$ forwards msg and cnt to $\mathfrak{m}_i$ and asks if it wants to sign with respect to cnt. If so then $\mathcal{T}$ asks $\mathfrak{h}_i$ for a basename with which to produce the signature.
- After receiving bsn the trusted third party $\mathcal{T}$ generates a random σ, adds $(\sigma, \mathsf{msg}, \mathsf{bsn}, i, \mathsf{cnt}, \mathfrak{i}_k)$ to $\mathsf{Signatures}$ and responds to $\mathfrak{h}_i$ with σ.

Verify. A given verifier asks for a verification decision on $(\sigma, \mathsf{msg}, \mathsf{bsn}, \mathfrak{i}_k)$ by submitting this tuple to $\mathcal{T}$.

- If $\mathfrak{i}_k$ is corrupt then $\mathcal{T}$ refuses the request.
- If there does not exist an entry $(\sigma, \mathsf{msg}, \mathsf{bsn}, *, *, \mathfrak{i}_k) \in \mathsf{Signatures}$ then $\mathcal{T}$ replies with *reject*.
- Else if the corresponding pair for this signature $(i, \mathsf{cnt}) \in \mathsf{RogueList}$ then $\mathcal{T}$ informs $\mathfrak{v}_j$ that msg has been signed by a rogue TPM and outputs *reject*.
- Else $\mathcal{T}$ replies with *accept* and the corresponding issuer identity $\mathfrak{i}_k$.

Link. A given verifier $\mathfrak{v}_j$ requests a linkage decision from $\mathcal{T}$ by submitting a tuple $((\sigma_0, \mathsf{msg}_0), (\sigma_1, \mathsf{msg}_1), \mathsf{bsn}, \mathfrak{i}_k)$ to $\mathcal{T}$. If there exists less than two entries on $\mathsf{Signatures}$ of the form $(\sigma_b, \mathsf{msg}_b, \mathsf{bsn}, i, \mathsf{cnt}, \mathfrak{i}_k)$ for $b \in \{0,1\}$, then $\mathcal{T}$ returns $\perp$. Else if any of the entries on $\mathsf{Signatures}$ is such that $(i, \mathsf{cnt}) \in \mathsf{RogueList}$ then $\mathcal{T}$ returns $\perp$. Else if there exists two entries on $\mathsf{Signatures}$ of the form $(\sigma_b, \mathsf{msg}_b, \mathsf{bsn}, i, \mathsf{cnt}, \mathfrak{i}_k)$ for $b \in \{0,1\}$, $(i, \mathsf{cnt}) \notin \mathsf{RogueList}$ for each value of b and $\mathsf{bsn} \neq \perp$ then $\mathcal{T}$ returns *linked*. Otherwise $\mathcal{T}$ returns *unlinked* to $\mathfrak{v}_j$.

$\mathsf{RogueTag}$. When a party wishes to add an entry to $\mathsf{RogueList}$ it submits a tuple $(i, \mathsf{cnt}, \mathfrak{i}_k)$ to $\mathcal{T}$. If $i \in \mathsf{CorruptTPM}$ and $(i, \mathsf{cnt}, \mathfrak{i}_k) \in \mathsf{Members}$ then $\mathcal{T}$ adds (i, cnt) to $\mathsf{RogueList}$ and otherwise does not.

We now discuss some of the properties of this ideal functionality in more detail. Firstly, notice this ideal functionality implements the two most important parts of a joining functionality. Namely, a platform cannot become a member without some interaction with the TPM and without the issuer allowing the same TPM to join. The only real difference with this and the functionality of [7] is that when a module is allowed to join an entry is added to $\mathsf{Members}$ which includes the issuer with which the module joined. Within the functionality one should think of the value of i as an identifier for a given user and a given cnt as one of the internal secrets held; i.e. each value of cnt corresponds to a particular f value in an actual DAA scheme.

In contrast to [7], our functionality splits $\mathsf{Sign}, \mathsf{Verify}$ and Link into separate parts rather than giving them as a whole. As with the model of [7] a host can only sign a message with respect to some counter if it has already been added to the members list with respect to this counter value. Also, a host cannot produce a new signature without interaction with a TPM. We note that a host does not have to decide which verifier it wants to present the signature to at the point of signing.

One of the main differences with our Sign functionality and that of [7] is how the trusted third party implements it using a list of signatures. The entries of this list contain a randomly chosen signature value, the basename used to sign, and the counter value and issuer with which the corresponding join protocol was run. This inclusion of the basename in the signature list entry allows the enforcing of basename verifiability; the ability of a verifier to check a given signature was produced with respect to a given basename. This property is not present in the model of [7] yet is an important property for DAA schemes as a given verifier may have some policy to implement based on basenames used for signatures.

The first thing to notice with our Verify functionality is that if the issuer for which a given signature is presented for verification is corrupt then the trusted third party refuses to deal with it. Indeed, a corrupt issuer can issue credentials to any platform it likes, or itself, regardless of whether such a platform contains a TPM or not. Hence the security model does not cover this case. Notice however that we do allow for the Join protocol to be run with a corrupt issuer and for signatures to be produced on credentials obtained in this manner. This allows us to model anonymity of signatures produced; if an honest TPM obtains a credential from a corrupt issuer then we want that this still remains anonymous and unlinkable where required. We simply disallow signatures produced in this manner to be presented for verification. This assumption that such signatures will never be presented for verification is not contained explicitly in the model of [7], but it is assumed implicitly in the proof contained in [7]. Our model enforces this property explicitly during the Verify ideal functionality.

In addition to this the Verify ideal functionality implements basename verifiability explicitly; to get a verification decision a basename has to be presented to the trusted third party and only if this matches the signature on the list does it give an acceptance decision. This functionality also implements anonymity whilst still preventing signatures produced by a rogue TPM from correctly verifying. Also, notice that if a user presents a signature for verification successfully, then they can tell with which issuer the Join protocol was run. Most importantly, signatures will only verify if an entry was added to $\mathsf{Signatures}$ by the trusted third party. A given signature may also be presented for verification many times as opposed to just once, or may not be presented for verification at all.

User controlled linkability means that correctly verifying signatures with the same $\mathsf{bsn} \neq \perp$ and with the same value of f should link together. The functionality captures this by linking two signatures on the list that use the same basename and counter value. In addition, if the *pairs* of $(\mathsf{bsn}, \mathsf{cnt})$ are different for a given pair of signatures or if $\mathsf{bsn} = \perp$ then signatures will not link together. For honest users, anonymity means that a given cannot be linked to the TPM that computed it. In the ideal functionality signatures are assigned at random, in particular independently of either i or cnt, and verification decisions are also independent of these.

Securely Implementing Functionality. Intuitively, we say a system is secure if its behaviour is computationally indistinguishable from a an ideal DAA system. The formal definition of a secure implementation of a DAA scheme is then as follows.

Definition 2 (Secure Implementation). *A given* DAA *scheme* Daa *is a secure implementation if for every computationally bounded environment* Env *and every* p.p.t. *adversary* $\mathcal{A}$ *there exists some simulator* $\mathcal{S}$*, that controls the same set of players in an ideal–system as* $\mathcal{A}$ *does in a real–system, such that* Env

cannot distinguish whether it is run in the real–system (and interacts with $\mathcal{A}$) or whether it is run in an ideal–system (and interacts with $\mathcal{S}$).

We assume the trusted third party $\mathcal{T}$ in an ideal system is in some sense "invisible" to the environment since otherwise the environment can trivially distinguish real from ideal systems. To this end any protocols for which message are passed through $\mathcal{T}$ appear as if they are passed directly between players and any computations performed by $\mathcal{T}$ on behalf of players appear as if they are performed by the players themselves. In our functionality in particular we require that the signatures chosen by $\mathcal{T}$ have the same shape and structure as in a real world scheme.

3 Pairing Based Cryptography Background

Our main contribution of the paper is a new highly efficient pairing based DAA protocol based on asymmetric pairings, which corrects a flaw in the proposal of [17]. The reason for using asymmetric pairings as opposed to symmetric pairings is that the latter's security levels scale very badly. This poor scaling is due to the embedding degree being bounded by six for supersingular elliptic curves. Thus with the wider acceptance of AES style security levels it has been necessary for pairing protocols to also move to the setting of ordinary elliptic curves, where asymmetric pairings are required. In addition by combining with the latest implementation choices, such as Barreto-Naehrig curves [3], the Ate-pairing [22] and its generalisations, and the use of sextic twists, we can obtain very efficient pairing implementations in the asymmetric setting at high security levels.

Throughout we let $\mathbb{G}_1 = \langle P_1 \rangle, \mathbb{G}_2 = \langle P_2 \rangle$ and $\mathbb{G}_T$ be groups of large prime exponent $q \approx 2^t$ for security parameter t. The groups $\mathbb{G}_1, \mathbb{G}_2$ will be written additively and the group $\mathbb{G}_T$ multiplicatively. If $\mathbb{G}$ is some group then we use the notation $\mathbb{G}^\times$ to mean the non-identity elements of $\mathbb{G}$. If R is some ring or field we take $R^\times$ to mean the non-zero elements of R (non-identity elements for the addition operation).

Definition 3 (Pairing). *A* pairing (or bilinear map) *is a map $\hat{t} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ such that:*

1. *The map $\hat{t}$ is* bilinear. *This means that $\forall P, P' \in \mathbb{G}_1$ and $\forall Q, Q' \in \mathbb{G}_2$ that*
   – $\hat{t}(P + P', Q) = \hat{t}(P, Q) \cdot \hat{t}(P', Q) \in \mathbb{G}_T$.
   – $\hat{t}(P, Q + Q') = \hat{t}(P, Q) \cdot \hat{t}(P, Q') \in \mathbb{G}_T$.
2. *The map $\hat{t}$ is* non-degenerate. *This means that*
   – $\forall P \in \mathbb{G}_1^\times \exists Q \in \mathbb{G}_2$ *such that* $\hat{t}(P, Q) \neq 1_{\mathbb{G}_T} \in \mathbb{G}_T$.
   – $\forall Q \in \mathbb{G}_2^\times \exists P \in \mathbb{G}_1$ *such that* $\hat{t}(P, Q) \neq 1_{\mathbb{G}_T} \in \mathbb{G}_T$.
3. *The map $\hat{t}$ is* computable *i.e. there exist some polynomial time algorithm to compute $\hat{t}(P, Q) \in \mathbb{G}_T$ for all $(P, Q) \in \mathbb{G}_1 \times \mathbb{G}_2$.*

All pairing based DAA protocols, ours included, are based on the pairing based Camenisch-Lysyanskaya signature scheme [14]. This protocol is given by a triple of algorithms, as follows:

– **KeyGeneration:** The private key is a pair $(x, y) \in \mathbb{Z}_q \times \mathbb{Z}_q$, the public key is given by the pair $(X, Y) \in \mathbb{G}_2 \times \mathbb{G}_2$ where $X = xP_2$ and $Y = yP_2$.
– **Signing:** On input of a message $m \in \mathbb{Z}_q$ the signer generates $A \in \mathbb{G}_1$ at random and outputs the signature $(A, B, C) \in \mathbb{G}_1 \times \mathbb{G}_1 \times \mathbb{G}_1$, where $B = yA$ and $C = [x + mxy]A$.
– **Verification:** To verify a signature on a message the verifier checks whether $\hat{t}(A, Y) = \hat{t}(B, P_2)$ and $\hat{t}(A, X) \cdot \hat{t}(mB, X) = \hat{t}(C, P_2)$.

The security of the above signature scheme is related to the hardness of a problem called the bilinear LRSW assumption [14, 24]. To describe this assumption we first define an oracle $\mathcal{O}_{X,Y}(\cdot)$ which on input $f \in \mathbb{Z}_q$ outputs a triple $(A, y \cdot A, (x + fxy)A)$ where $A \leftarrow \mathbb{G}_1, X = x \cdot P_1$ and $Y = y \cdot P_2$. We then have the following definition.

Definition 4 (bilinear LRSW Advantage). *We define the bilinear LRSW advantage $\mathbf{Adv}_{\mathcal{A}}^{\mathrm{bLRSW}}(t)$ of an adversary $\mathcal{A}$ against $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, q, \hat{t})$ as*

$$\Pr\left[\begin{array}{c} x, y \leftarrow \mathbb{Z}_q; X \leftarrow xP_1, Y \leftarrow yP_2; (f, A, B, C) \leftarrow \mathcal{A}^{\mathcal{O}_{X,Y}(\cdot)}(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, X, Y, q, \hat{t}) \\ \wedge \Big(f \notin \mathcal{Q},\ f \in \mathbb{Z}_q^\times,\ A \in \mathbb{G}_1,\ B = y \cdot A,\ C = (x + fxy) \cdot A \Big) \end{array}\right]$$

where $\mathcal{Q}$ is the set of queries that $\mathcal{A}$ made to $\mathcal{O}_{X,Y}(\cdot)$ and $q \approx 2^t$.

We then say a tuple $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, q, \hat{t})$ satisfies the bilinear LRSW assumption if for any p.p.t.adversary $\mathcal{A}$ its advantage $\mathbf{Adv}_{\mathcal{A}}^{\mathrm{bLRSW}}(t)$ is negligible in t.

We also define a very closely related assumption which we refer to as the *blind* bilinear LRSW assumption. To describe this we define an oracle $\mathcal{O}_{X,Y}^{B}(\cdot)$ which on input $F \in \mathbb{G}_1$ outputs a triple $(A, y \cdot A, (xA + rxyF))$ where $r \leftarrow \mathbb{Z}_q$, $A = rP_1 \in \mathbb{G}_1, X = x \cdot P_1 \in \mathbb{G}_1$ and $Y = y \cdot P_2 \in \mathbb{G}_2$. We note this is essentially the same as the oracle $\mathcal{O}_{X,Y}(\cdot)$ since $xA + rxyF = (x + fxy)A$ where $A = rP_1$ for some $r \in \mathbb{Z}_q$ and $F = fP_1$ for some value of $f \in \mathbb{Z}_q$. The only difference is how the requests are made to the oracle; the value of f is "hidden" from the oracle in the blind case. We then have the following definition.

Definition 5 (Blind Bilinear LRSW Advantage). *We define the blind bilinear LRSW advantage* $\mathbf{Adv}_{\mathcal{A}}^{\mathrm{B-bLRSW}}(t)$ *of an adversary* $\mathcal{A}$ *against* $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, q, \hat{t})$ *as*

$$\Pr\left[\begin{array}{c} x, y \leftarrow \mathbb{Z}_q; X \leftarrow xP_1, Y \leftarrow yP_2; (f, A, B, C) \leftarrow \mathcal{A}^{\mathcal{O}_{X,Y}^{B}(\cdot)}(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, X, Y, q, \hat{t}) \\ \wedge \Big(F = fP_1 \notin \mathcal{Q},\ f \in \mathbb{Z}_q^{\times},\ A \in \mathbb{G}_1,\ B = y \cdot A,\ C = (x + fxy) \cdot A\Big) \end{array}\right]$$

where $\mathcal{Q}$ *is the set of queries that* $\mathcal{A}$ *made to* $\mathcal{O}_{X,Y}^{B}(\cdot)$ *and* $q \approx 2^t$.

We then say a tuple $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, q, \hat{t})$ satisfies the blind bilinear LRSW assumption if for any p.p.t. adversary $\mathcal{A}$ its advantage $\mathbf{Adv}_{\mathcal{A}}^{\mathrm{B-bLRSW}}(t)$ is negligible in t.

Notice that the blind bLRSW assumption is at least as strong as the bLRSW assumption and may be slightly stronger. One can easily translate queries made to $\mathcal{O}_{X,Y}(\cdot)$ to queries suitable for $\mathcal{O}_{X,Y}^{B}(\cdot)$ by multiplying a given f by P_1. However, the reverse is only true if one can obtain a given f from $F = fP_1$. As a result, if the B-bLRSW assumption holds then the bLRSW assumption trivially holds but the B-bLRSW assumption may hold *without* the bLRSW assumption holding.

The original bilinear CL signature scheme is given in the symmetric pairing setting (i.e. where $\mathbb{G}_1 = \mathbb{G}_2$), we have chosen the above asymmetric version (i.e. the precise use of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$) so as to reduce the size of the signatures and to have the fastest signing algorithm possible. The key property of this signature scheme is that signatures are re-randomizable without knowledge of the secret key: given (A, B, C) one can re-randomise it by computing (rA, rB, rC) for a random element $r \in \mathbb{Z}_q$.

There is an interesting difference between this signature scheme in the symmetric and the asymmetric settings. In the symmetric setting the signer, on being given two valid signatures (A, B, C) and (A', B', C'), is able to tell that they correspond to a randomisation of a previous signature, without knowing what that message is. He can do this by verifying that $A' = rA, B' = rB$ and $C' = rC$, for some value r, by performing the following steps:

$$\hat{t}(A', B) = \hat{t}(A, B') \text{ and } \hat{t}(A', C) = \hat{t}(A, C').$$

This makes use of the fact that the DDH problem is easy in $\mathbb{G}_1$ in the symmetric setting.

In the asymmetric setting a signer is unable to determine if two signatures correspond to the same message, since in this setting the DDH problem is believed to be hard in $\mathbb{G}_1$. Indeed one can show that an adversary who can tell whether (A', B', C') is a randomisation of (A, B, C), even if the adversary knows x and y, is able to solve DDH in $\mathbb{G}_1$. This difference provides one of the main optimisations of our DAA scheme below. For later use the formal definition of the DDH problem for $\mathbb{G}_1$ is now given:

Definition 6 ($\mathbb{G}_1$-DDH). *We define the* $\mathbf{Adv}_{\mathcal{A}}^{\mathrm{DDH}}(t)$ *of an* $\mathbb{G}_1$*-DDH adversary* $\mathcal{A}$ *against the set of parameters* $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, q, \hat{t})$ *as*

$$\begin{aligned} | \quad & \Pr[x, y, z \leftarrow \mathbb{Z}_q; X \leftarrow xP_1, Y \leftarrow yP_1, Z \leftarrow zP_1; \mathcal{A}(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, X, Y, Z, q) = 1] \\ & - \Pr[x, y \leftarrow \mathbb{Z}_q; X \leftarrow xP_1, Y \leftarrow yP_1; Z \leftarrow \mathbb{G}_1; \mathcal{A}(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, X, Y, Z, q) = 1] \quad | \end{aligned}$$

We then say a tuple $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, q, \hat{t})$ satisfies the DDH assumption for $\mathbb{G}_1$ if for any p.p.t.adversary $\mathcal{A}$ its advantage $\mathbf{Adv}_{\mathcal{A}}^{\mathrm{DDH}}(t)$ is negligible in t. Often this problem in the context of pairing groups is called the *external Diffie–Hellman* problem, or the XDH problem.

We shall require one other problem to be hard. Namely that the discrete logarithm problem in $\mathbb{G}_1$ is hard, even in the presence of an oracle which solves the *static* computational Diffie–Hellman problem for the underlying secret in the discrete logarithm problem. We will call this problem Gap-DLP, since it is similar to the Gap-CDH problem, where one tries to solve the CDH problem with the presence of an oracle to solve DDH. Note, that in general DLP and the general CDH problem are believed to be equivalent, hence such a Gap-DLP problem will be easy if the computational Diffie–Hellman oracle is a general oracle. However, we restrict the oracle to be *static*. Formally we define:

Definition 7 (Gap-DLP). *We define the Gap-DLP advantage* $\mathbf{Adv}_{\mathcal{A}}^{\mathrm{Gap-DLP}}(t)$ *for* $\mathbb{G}_1$ *of an adversary* $\mathcal{A}$ *against* $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, q, \hat{t})$ *as*

$$\Pr\left[x; X \leftarrow xP_1; \mathcal{A}^{\mathcal{O}_x(\cdot)}(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, X, q, \hat{t})\right]$$

where $\mathcal{O}_x$ *is the oracle which on input of* $Y \in \mathbb{G}_1$ *will return* xY *and* $q \approx 2^t$.

We then say a tuple $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, q, \hat{t})$ satisfies the Gap-DLP assumption in $\mathbb{G}_1$ if for any p.p.t. adversary $\mathcal{A}$ its advantage $\mathbf{Adv}_{\mathcal{A}}^{\mathrm{Gap-DLP}}(t)$ is negligible in t.

4 Pairing Based DAA Schemes

We can now give a detailed description of the our new DAA scheme based on asymmetric bilinear maps. Before proceeding we note a general point which needs to be born in mind for each of our following sub-protocols. Every group element received by any party needs to be checked that it lies in the correct group, and in particular does not lie in some larger group which contains the specified group. This is to avoid numerous attacks such as those related to small subgroups etc. In asymmetric pairings this is particularly important since $\mathbb{G}_1$ and $\mathbb{G}_2$ can be considered as distinct subgroups of a large group $\mathbb{G}$. If transmitted elements are actually in $\mathbb{G}$, as opposed to $\mathbb{G}_1$ and $\mathbb{G}_2$, then various properties can be broken such as anonymity and linkability.

Hence, our security proofs implicitly assume that all transmitted group elements are indeed elements of the specified groups, this is a point which is often overlooked in many discussions. For the situation under consideration, namely Type-III pairings [20], efficient methods for checking subgroup membership are given in [19]. Note, we do not count the cost of these subgroup checks in our performance considerations later on, as their relative costs can be quite dependent on the specific groups and security parameters under consideration.

4.1 The Setup Algorithm

To set the system up we need to select parameters for each protocol and algorithm used within the DAA scheme well as the long term parameters for each Issuer. We assume that prior to any system setup each TPM has its private endorsement key SK embedded into it and that each issuer has access to the corresponding public endorsement key PK. We also assume a public key signature scheme has been selected for use with these keys. On input of the security parameter 1^t the setup algorithm executes the following:

1. *Generate the Commitment Parameters* $\mathsf{par}_{\mathrm{C}}$. For this three groups, $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$, of sufficiently large prime order q are selected. Two random generators are selected such that $\mathbb{G}_1 = \langle P_1 \rangle$ and $\mathbb{G}_2 = \langle P_2 \rangle$ along with a pairing $\hat{t} : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$. Next two hash functions $H_1 : \{0,1\}^* \mapsto \mathbb{Z}_q$ and $H_2 : \{0,1\}^* \mapsto \mathbb{Z}_q$ are selected and $\mathsf{par}_{\mathrm{C}}$ is set to be $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{t}, P_1, P_2, q, H_1, H_2)$.
2. *Generate Signature and Verification Parameters* $\mathsf{par}_{\mathrm{S}}$. Three additional hash functions are selected: $H_3 : \{0,1\}^* \mapsto \mathbb{G}_1$, $H_4 : \{0,1\}^* \mapsto \mathbb{Z}_q$ and $H_5 : \{0,1\}^* \mapsto \mathbb{Z}_q$. We set $\mathsf{par}_{\mathrm{S}}$ to be (H_3, H_4, H_5).
3. *Generate the Issuer Parameters* $\mathsf{par}_{\mathrm{I}}$. For each $\mathfrak{i}_k \in \mathfrak{I}$ the following is performed. Two integers are selected $x, y \leftarrow \mathbb{Z}_q$ and the issuer secret key isk_k is assigned to be (x, y). Then the values $X = x \cdot P_2 \in \mathbb{G}_2$ and $Y = y \cdot P_2 \in \mathbb{G}_2$ are computed. The issuer public key ipk_k is assigned to be (X, Y).
   Then an issuer value K_k is computed according to the issuer public values in some predefined manner (we leave the specific details of how this is done as an implementation detail).
   Finally, $\mathsf{par}_{\mathrm{I}}$ is set to be $(\{\mathsf{ipk}_k, \mathrm{K}_k\})$ for each issuer $\mathfrak{i}_k \in \mathfrak{I}$.

4. *Publish Public Parameters.* Finally, the system public parameters par are set to be $(\mathsf{par}_{\mathrm{C}}, \mathsf{par}_{\mathrm{S}}, \mathsf{par}_{\mathrm{I}})$ and are published.

The grouping of system parameters is according to usage. For example the set $\mathsf{par}_{\mathrm{C}}$ contains all system parameters necessary for computing commitments and the set $\mathsf{par}_{\mathrm{V}}$ contains those for any signature verification computations. The group order q is selected so that solving the decisional Diffie–Hellman problem in $\mathbb{G}_1$ takes time 2^t, as does solving the appropriate bilinear LRSW problem with respect to the pairing $\hat{t}$, and as does solving the Gap-DLP problem in $\mathbb{G}_1$.

We do not specify that issuers supply a proof of correctness of their public keys, i.e. that they know the underlying secret key value, or that a given user checks the correctness of the issuer public keys. Instead, during the verification algorithm, the correctness of issuer public keys are verified for any signature produced from a credential issued by a given issuer.

4.2 The Join Protocol

This is a protocol between a given TPM $\mathfrak{m} \in \mathfrak{M}$, the corresponding Host $\mathfrak{h} \in \mathfrak{H}$ and an Issuer $\mathfrak{i} \in \mathfrak{I}$. We first give an overview of how a general Join protocol proceeds. There are 3 main stages to a Join protocol. First the TPM $\mathfrak{m}$ generates some secret message f using the value K_k provided by the issuer and its internal seed DAAseed. We note, for a given issuer a TPM could compute many values of f; one for each value of K_k. The TPM then computes a commitment on this value and passes this to its Host who adds this to the list of commitments for that user and forwards it to the Issuer. In the second stage the issuer performs some checks on the commitment it receives and, if these correctly verify, computes a credential such that the correctness of this credential can be checked by the TPM and Host working together. The final stage of a Join protocol involves the Host and TPM working together to verify the correctness of the credential. In our case the Host first performs some computations and stores some values related to these before passing part of the credential on to the TPM prior to verifying the correctness of the credential and then adding this to the list of credentials for that user.

Our protocol proceeds as shown in Figure 1. The following notes should be kept in mind when examining this protocol.

– We note that the nonce n_I on which the commitment is generated must be one that was sent out by the issuer. Since the issuer may be running many Join protocols at once the host reminds the issuer of the value of n_I it was sent in $\mathsf{comm}_{\mathrm{req}}$ and the issuer then checks this against its records. This checking of the nonce could equally be performed by the server computing a message authentication code tag on the nonce and verifying this upon reciept of it from a host. This method would require more computation from an issuer but require less storage space.
– The commitment used to obtain a given credential is essentially a proof of knowledge of the discrete logarithm of the value F along with a signature on this proof of knowledge. Since only a valid TPM holding the secret key corresponding to a given public endorsement key can produce such a signature the host should not be able to obtain a credential without the aid of a TPM. Since only the TPM knows f then the host should then be unable to produce signatures on this credential without the aid of the same TPM. This of course assumes that a given issuer actually checks the signature and the proof of knowledge of the discrete logarithm of F; if the signature was not checked then a host could select any value of f it wanted and produced a proof of knowledge of this. The host would then be able to compute signatures on this credential that correctly verify. Furthermore, we do not consider the public key signature computations within our performance analysis of the scheme.
– The public key signature scheme used to sign the commitment essentially implements an authenticated channel from the TPM to the issuer. In reality this need not be a signature scheme and can be replaced with any authenticated channel; the security of the overall scheme will then depend upon the security properties of this authenticated channel.
– In contrast with the RSA-based DAA schemes we do not require a relatively complicated proof of knowledge of the correctness of a given commitment. Instead, the proof of knowledge is provided by a very efficient Schnorr signature on the value F computed using the secret key f. The credential computed by $\mathfrak{i}_k$ is then a bilinear CL signature on this commitment.

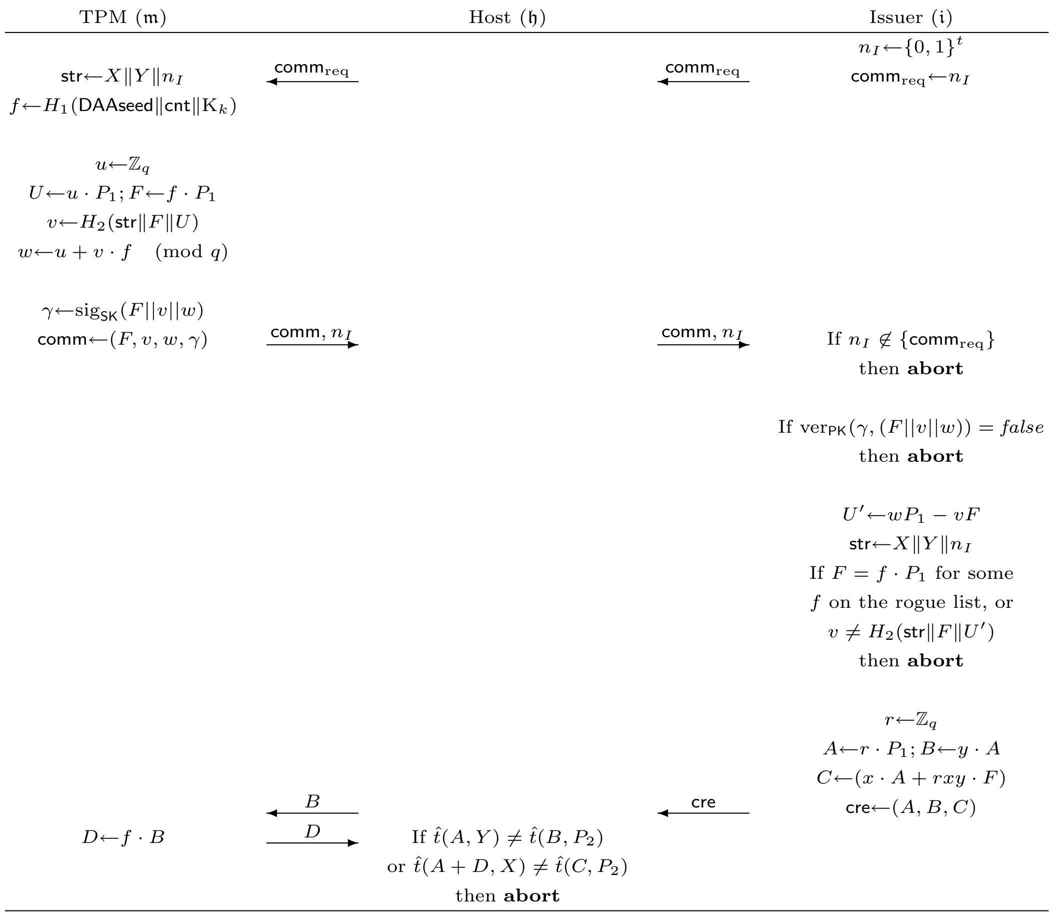
TPM ($\mathfrak{m}$)
Host ($\mathfrak{h}$)
Issuer ($\mathfrak{i}$)
$n_I \leftarrow \{0,1\}^t$
$\mathsf{comm}_{\mathrm{req}} \leftarrow n_I$
$\mathsf{comm}_{\mathrm{req}}$
$\mathsf{comm}_{\mathrm{req}}$
$\mathsf{str} \leftarrow X \| Y \| n_I$
$f \leftarrow H_1(\mathsf{DAAseed} \| \mathsf{cnt} \| \mathrm{K}_k)$
$u \leftarrow \mathbb{Z}_q$
$U \leftarrow u \cdot P_1; F \leftarrow f \cdot P_1$
$v \leftarrow H_2(\mathsf{str} \| F \| U)$
$w \leftarrow u + v \cdot f \pmod q$
$\gamma \leftarrow \mathrm{sig}_{\mathsf{SK}}(F||v||w)$
$\mathsf{comm} \leftarrow (F, v, w, \gamma)$
comm, n_I
comm, n_I
If $n_I \notin \{\mathsf{comm}_{\mathrm{req}}\}$ then **abort**
If $\mathrm{ver}_{\mathsf{PK}}(\gamma, (F||v||w)) = \mathit{false}$ then **abort**
$U' \leftarrow wP_1 - vF$
$\mathsf{str} \leftarrow X \| Y \| n_I$
If $F = f \cdot P_1$ for some f on the rogue list, or $v \neq H_2(\mathsf{str} \| F \| U')$ then **abort**
$r \leftarrow \mathbb{Z}_q$
$A \leftarrow r \cdot P_1; B \leftarrow y \cdot A$
$C \leftarrow (x \cdot A + rxy \cdot F)$
$\mathsf{cre} \leftarrow (A, B, C)$
cre
B
D
$D \leftarrow f \cdot B$
If $\hat{t}(A, Y) \neq \hat{t}(B, P_2)$ or $\hat{t}(A + D, X) \neq \hat{t}(C, P_2)$ then **abort**

Fig. 1. The Join Protocol

- Once a credential is issued from $\mathfrak{i}$, the TPM and the Host verify that this credential is correctly formed. This is to avoid performing computations with a credential that is incorrectly formed. The last part of the protocol therefore performs the verification algorithm from the Camenisch–Lysyanskaya signature scheme.
- Notice that the host does not perform any checks on the correctness of the value D it receives from the TPM. Indeed, since we assume it is harder to break into a TPM than it is to break into a host we do not consider the case of a corrupted TPM and honest host in our security analysis; the host will always trust communications it receives from its corresponding TPM. If the checks on the credential using D fail then the host knows this is due to the credential being incorrectly formed rather than the value of D being incorrectly formed.

4.3 The Signing Protocol

This is a protocol run between a given TPM $\mathfrak{m} \in \mathfrak{M}$ and Host $\mathfrak{h} \in \mathfrak{H}$. The objective of the sign protocol is for m and h to work together to produce a signature of knowledge on some message. The signature should prove knowledge of a discrete logarithm f, knowledge of a valid credential and that this credential was computed for the same value f. We note that the Host will know a lot of the values needed in the computation and will be able to take on a lot of the computational workload. However, if the TPM has not had its internal value of f published (i.e. it is not a rogue module) then the Host will not know f and will be unable to compute the whole signature without the aid of the TPM.

TPM ($\mathfrak{m}$)		Host ($\mathfrak{h}$)
		If $\mathsf{bsn} = \perp$ then $J \leftarrow \mathbb{G}_1$
		else $J \leftarrow H_3(\mathsf{bsn})$
		Either $n_V \leftarrow \{0,1\}^t$ or receive
		$n_V \in \{0,1\}^t$ from the verifier
		$t \leftarrow \mathbb{Z}_q$
		$R \leftarrow t \cdot A; S \leftarrow t \cdot B$
		$T \leftarrow t \cdot C; \beta \leftarrow \hat{t}(S, X)$
$K = f \cdot J; z \leftarrow \mathbb{Z}_q$	$\xleftarrow{h, J, \beta, \mathsf{msg}}$	$h \leftarrow H_4(R \Vert S \Vert T \Vert n_V)$
$L \leftarrow z \cdot J; \tau \leftarrow \beta^z$		
$n_T \leftarrow \{0,1\}^t$		
$c \leftarrow H_5(h \Vert \mathsf{msg} \Vert J \Vert K \Vert L \Vert \tau \Vert n_T)$		
$s \leftarrow z + c \cdot f \pmod q$	$\xrightarrow{(K, c, s, n_T)}$	
		$\sigma \leftarrow (R, S, T, J, K, c, s, n_V, n_T)$

Fig. 2. The Sign Protocol

We let msg denote the message to be signed and bsn denote the base name, both of which may either be chosen/selected by the host, or passed to the host by the verifier. The protocol then proceeds as in Figure 2, so as to produce the signature σ.

Again we provide some notes as to the rationale behind some of the steps:

– During the run of the signature protocol two nonces are used: one from the verifier n_V and one from the TPM n_T. In most applications of the Sign protocol, the signature is generated as a request from the verifier, and the verifier supplies its own value of n_V, to protect against replays of previously requested signatures. If a signature is produced in an off-line manner we allow the Host to generate its own value of n_V. These are used to ensure each signature is different from previous signatures and to ensure no adversarially controlled TPM and Host pair, or no honest TPM and adversarially controlled Host, can predict or force the value of a given signature.
– Prior to running the protocol the Host decides if it wants σ to be linkable to other signatures produced for the same verifier. If it does not want the signature to be linkable to any existing or future signatures then it chooses $\mathsf{bsn} = \perp$. If it decides that it wants the signature to be linked to some previously generated signatures with this verifier then it sets bsn to be the same as that used for the signature it wants to link to. Otherwise, if the Host decides it may want future signatures to be able to be link to this one then it chooses a verifier bsn that it has not used before.
– The use of J and K allows the verifier to identify if the signature was produced by a rogue TPM by computing $f_i \cdot J$ for all f_i values on the rogue list and comparing these to K. This check is performed during the verification algorithm.
– The value t is used to re-randomise the credential to be signed; if the same randomisation of a given credential was used twice then any user in the system can trivially link signatures together. If the original credential (A, B, C) is sent without any re-randomisation then an issuer can trivially link signatures to users. That the issuer cannot link signatures when such re-randomisation is used follows from the earlier mentioned property of the Camenisch–Lysyanskaya signature scheme in the asymmetric setting. Thus it provides two types of linking resistance: It stops any issuer from being able to link a given signature to a given signer (since issuers know the values of r used to compute a credential and without t the credential is sent in the clear), and it stops any player in the system from being able to tell if any two signatures were produced by the same signer (if different values of bsn are used).
– The Host is trusted to keep anonymity because it is assumed that the Host has the motivation to protect privacy and also because the host can always disclose the platform identity anyway. However, the Host is not trusted to be honest for not trying to forge a DAA signature without the aid of TPM.
– The verification of a given signature will require the public key of the issuer that issued the underlying credential, for which (R, S, T) is a randomisation, and the basename on which the signature was computed. Recall in our functional definition these are given along with the signature rather than inside the signature description.

4.4 The Verification Algorithm

This is an algorithm run by a verifier $\mathfrak{v}$. Intuitively the verifier checks that a signature provided proves knowledge of a discrete logarithm f, checks that it proves knowledge of a valid credential issued on the same value of f and that this value of f is not on the list of rogue values. We now describe the details of our Verify algorithm. On input of a signature $\sigma = (R, S, T, J, K, c, s, n_V, n_T)$, a message msg, a basename bsn and an issuer public key $\mathsf{ipk}_k = (X, Y)$ this algorithm performs the following steps:

1. *Check Against Rogue List.* If $K = f_i \cdot J$ for any f_i in the set of rogue secret keys then return *reject*.
2. *Check Correctness of R and S.* If $\hat{t}(R, Y) \neq \hat{t}(S, P_2)$ then return *reject*.
3. *Check J computation.* If $\mathsf{bsn} \neq \perp$ and $J \neq H_3(\mathsf{bsn})$ then return *reject*.
4. *Verify Correctness of Proofs.* This is done by performing the following sets of computations:
   - $\rho_a^\dagger \leftarrow \hat{t}(R, X)$, $\rho_b^\dagger \leftarrow \hat{t}(S, X)$ and $\rho_c^\dagger \leftarrow \hat{t}(T, P_2)$.
   - $\tau^\dagger \leftarrow (\rho_b^\dagger)^s \cdot (\rho_c^\dagger / \rho_a^\dagger)^{-c}$
   - $L^\dagger \leftarrow sJ - cK$.
   - $h^\dagger \leftarrow H_4(R \| S \| T \| n_V)$.

   Finally if $c \neq H_5\left(h^\dagger \| \mathsf{msg} \| J \| K \| L^\dagger \| \tau^\dagger \| n_T\right)$ return *reject* and otherwise return *accept*.

We note the verify algorithm ensures the bsn submitted with the signature is the one used by the TPM to compute K by checking J and K are correctly related to each other, and that $J = H_3(\mathsf{bsn})$ for $\mathsf{bsn} \neq \perp$.

4.5 The Linking Algorithm

This is an algorithm run by a given verifier $\mathfrak{v}_j \in \mathfrak{V}$ which has a set of basenames $\{\mathsf{bsn}\}_j$ in order to determine if a pair of signatures were produced by the same TPM. Signatures can only be linked if they were produced by the same TPM and the user wanted them to be able to be linked together. Formally, on input a tuple $((\sigma_0, \mathsf{msg}_0), (\sigma_1, \mathsf{msg}_1), \mathsf{bsn}, \mathsf{ipk}_k)$ the algorithm performs the following steps:

1. *Verify Both Signatures.* For each signature σ_b, for $b \in \{0, 1\}$ the verifier runs $\mathsf{Verify}(\sigma_b, \mathsf{msg}_b, \mathsf{bsn}, \mathsf{ipk}_k)$ and if either of these returns *reject* then the value $\perp$ is returned.
2. *Compare J and K values.* If $J_0 = J_1$ and $K_0 = K_1$ then return *linked*, else return *unlinked*.

It may be the case that one or both signatures input to the Link algorithm have previously been received and verified by the verifier. Regardless of this we insist that the verifier re-verify these as part of the Link algorithm since the list of rogue TPM values may have been updated since the initial verification.

Also we note the condition that $K_0 = K_1$ ensures only signatures produced with the same basename *and* internal f value can be linked together. Since both signatures correctly verify with bsn this means that in each case the K and J values relate correctly to each other.

Note, our linking algorithm works due to the way that J and K are computed in the signing algorithm. Also note that anyone who knows bsn can link the two signatures, but they cannot link the signatures to the signers.

4.6 The Rogue Tagging Algorithm

The purpose of the rogue tagging algorithm is to ensure that an adversary is not able to tag a given value of TPM internal secret as rogue if the TPM that owns that particular value is not corrupted.

On input a value of f, a signature σ, message msg, basename bsn and issuer public key ipk_k the algorithm proceeds as follows:

1. *Verify the Signature.* If $\mathsf{Verify}(\sigma, \mathsf{msg}, \mathsf{bsn}, \mathsf{ipk}_k) =$ *reject* then the value $\perp$ is returned.
2. *Check (J, K) tuple.* If $K \neq f \cdot J$ then return $\perp$.
3. *Check (R, S, T) tuple.* If $\hat{t}(R + fS, X) \neq \hat{t}(T, P_2)$ then return $\perp$ and otherwise add an entry f to the rogue list.

This algorithm is intended to be run by either a verifier $\mathfrak{v}_j$ or an issuer $\mathfrak{i}_k$ and using its own local list of rogue values. Notice that the final check ensures the underlying for the signature was produced with the same value of f as the remainder of the signature.

5 Security Proof

In this section we give a security analysis of our DAA scheme.

Theorem 1. *Our* DAA *scheme is secure in the random oracle model under the blind bilinear LRSW assumption in* $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, \hat{t})$*, the UF-CMA security of the public key signature scheme used, the hardness of the gap-discrete logarithm problem in* $\mathbb{G}_1$ *and the hardness of the decision Diffie-Hellman problem in* $\mathbb{G}_1$*.*

Proof. We first give an overview of the proof. To prove the theorem we need to show that for every adversary $\mathcal{A}$ in the real system there exists a simulator $\mathcal{S}$ in the ideal system such that the environment cannot tell which system it is in and such that the adversary cannot tell it is in a simulation. We assume a given adversary $\mathcal{A}$ exists and $\mathcal{S}$ has access to $\mathcal{A}$ as a black box. When $\mathcal{S}$ receives messages from $\mathcal{A}$ it has to simulate the behaviour of the honest players in the real system to $\mathcal{A}$ such that $\mathcal{A}$ cannot tell it is run in a simulation. When $\mathcal{S}$ receives messages from $\mathcal{T}$ it has to simulate the behaviour of corrupt players in the ideal system to $\mathcal{T}$. In the real system the simulator allows $\mathcal{A}$ to perform computations and run protocols for the corrupted players and controls the random oracles. In each case the simulator has to respond to queries such that the environment Env cannot tell if it is run in the ideal system with $\mathcal{S}$ or in a real system with $\mathcal{A}$.

To construct the proof we first describe the operation of the simulator $\mathcal{S}$ and its interaction with the environment Env and the adversary $\mathcal{A}$ by describing how $\mathcal{S}$ handles the communications according to combinations of player corruptions. We use the notation (ihM) to describe the communication and computations between a corrupted initiator (lower case i), a corrupt host (lower case h) and an honest TPM (upper case M). This corresponds to a partially corrupted user communicating with a corrupt issuer. We then prove that the simulator will not abort outputting "failure X" for $X \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Each such failure event corresponds to the adversary being able to solve some hard problem that is embedded into the interaction between the adversary and simulator. Finally we prove $\mathcal{A}$ cannot tell it is run in a simulation and the environment cannot distinguish if it is run in a real system with $\mathcal{A}$ or in an ideal system with $\mathcal{S}$ constructed from $\mathcal{A}$.

Answering Random Oracle Queries. To handle random oracle queries $\mathcal{S}$ maintains a number of, initially empty, lists HList_i for $i \in \{1, 2, 3, 4, 5\}$. Each corresponds to a list of input and output pairs used in the simulation of one of the 4 random oracles. Queries to each of the random oracles are answered in the obvious manner.

Simulation of the Setup Algorithm. During the Setup the behaviour of $\mathcal{S}$ is according to the corruption state of the issuer $\mathfrak{i}_k$ running the algorithm. The simulator performs an ideal system setup to $\mathcal{T}$ and a real system setup to $\mathcal{A}$. For the ideal system setup $\mathcal{S}$ informs $\mathcal{T}$ which modules are corrupted by sending i to $\mathcal{T}$ for each one controlled by $\mathcal{S}$. For the real system simulation to $\mathcal{A}$ the simulator performs the following:

1. First $\mathcal{S}$ simulates the generation of the system wide parameters by executing steps 1 to 4 of the Setup algorithm and storing/passing the parameters to $\mathcal{A}$.
2. For each corrupted issuer $\mathfrak{i}_k$ (case (i)) $\mathcal{A}$ will pass the values ipk_k to $\mathcal{S}$ after performing step 3 of Setup. For each honest $\mathfrak{m}_i$ the setup is simulated to $\mathcal{A}$ by $\mathcal{S}$ selecting values for each $\mathsf{DAAseed}_i$ and setting $\mathsf{cnt}_i = 0$. For each corrupted user $\mathfrak{u}_i = (\mathfrak{m}_i, \mathfrak{h}_i)$ the simulator runs the ideal system Join protocol with $\mathfrak{i}_k$ using $\mathsf{cnt}_i = 0$. None of the honest parties in the ideal system will note any of this so this does not give the environment a method of distinguishing the system.
3. For each honest issuer $\mathfrak{i}_k$ (case (I)) the simulator also has to simulate the generation of public and private key pairs to $\mathcal{A}$. This is done as in step 3 of the Setup algorithm and the relevant values stored and passed to $\mathcal{A}$. For each corrupted $\mathfrak{m}_i$ the simulator has to simulate the ideal system setup to $\mathcal{T}$ by setting $\mathsf{cnt}_i = 0$ in the ideal system.

Simulation of the Join Protocol. We are able to distinguish 6 cases of corruption states for players. In each case the simulator essentially runs the protocol correctly and ensures the ideal and real systems are consistent. As already mentioned, we do not consider the case of a corrupt TPM and honest host since we assume it is easier to break into a host than a TPM. This reduces our possible number of cases to consider from 8 to only 6.

Before giving a detailed description of the simulator we first give an informal discussion of the main security properties enforced in a secure joining protocol from the ideal functionality and how these fit into the simulator description. In short, if the adversary is able to violate any of these properties in the real system then the simulator has to abort since it cannot emulate the same behaviour in the ideal system.

Firstly, in the ideal functionality, a host cannot become a member without interaction with a TPM and without an issuer deciding to allow this particular TPM to join. In terms of our new DAA scheme this means that, in the case of an honest TPM but an adversarially controlled host, the adversary should not be able to produce a pair (comm, n_I) claiming to be from a given TPM without the help of this particular TPM. If such a pair is received by the simulator, when acting as an honest issuer, a failure event is raised. We also want that an adversarially controlled module cannot choose the value of n_I used in a commitment itself and that each such n_I issued defines a new commitment. If the simulator, when acting as an honest issuer, receives a pair (comm, n_I) for which it did not send out n_I then it simply aborts as in the protocol. If the same commitment is received with two different n_I values then a failure event is raised. This is regardless of the corruption state of the TPM. One final property present in the ideal functionality and worth mentioning is that an adversary should not be able to produce a credential that verifiers for an honest issuer on its own. Such a violation can only be detected when signatures on such credentials are presented for verification and so we deal with this during the description of the simulator during the verification protocol.

Case (ihm). Since the issuer and user are corrupted the adversary will perform the roles of each player without any interaction with $\mathcal{S}$. Signatures on such credentials will never be allowed to be submitted for verification and so we need not consider this case in the simulation of the verification protocol.

Case (IHM). Since each of the players are honest they will interact with the trusted third party $\mathcal{T}$ in the ideal system. If such a signature is presented to a corrupt verifier in the ideal system then the simulator can simulate such a signature production in the real system. We describe the details of how this is done as a case of the Verify algorithm simulation later.

Case (iHM). The issuer is corrupted but the user is not. The simulator has to play the role of the host and module to $\mathcal{A}$ and has to play the role of the issuer to $\mathcal{T}$. The action here is triggered in the ideal system when $\mathcal{S}$, playing the role of the corrupt issuer, will receive a pair (i, cnt) and information as to whether the module is tagged as a rogue for that pair from $\mathcal{T}$. Before giving $\mathcal{T}$ an answer as to whether it can join the simulator engages in a run of the Join protocol with the adversary in the real system; the adversary will make this decision whilst acting as the corrupt issuer in the real system. The simulator does the following:

– The simulator first plays the role of the honest user $\mathfrak{u}_i$ in the real system by interacting with $\mathcal{A}$ who is playing the role of corrupt issuer $\mathfrak{i}_k$. The simulator begins a run of the real Join protocol. Since $\mathfrak{u}_i$ is honest $\mathcal{S}$ will know $\mathsf{DAAseed}_i$; since the simulator would have selected it during the Setup simulation. During this the simulator correctly computes and stores a value for $\mathsf{comm} = (F, v, w, \gamma)$.
– Eventually $\mathcal{S}$ should receive a credential (if $\mathcal{A}$ decides to continue) from $\mathcal{A}$ in the real system. The simulator then computes the required value of D and checks the correctness of this credential. If this check is passed then, to complete the ideal system Join protocol whilst acting as $\mathfrak{i}_k$, it informs $\mathcal{T}$ that it allows (i, cnt) to become a member. Otherwise $\mathcal{S}$ informs $\mathcal{T}$ that (i, cnt_i) cannot become a member.

Case (Ihm). Here $\mathcal{S}$ has to play the part of the honest issuer $\mathfrak{i}_k$ to $\mathcal{A}$ who controls a fully corrupted user $\mathfrak{u}_i = (\mathfrak{m}_i, \mathfrak{h}_i)$ in the real system. In the ideal system $\mathcal{S}$ plays the part of the corrupt user $\mathfrak{u}_i$ to $\mathcal{T}$. The actions of $\mathcal{S}$ will be triggered in the real system when it receives a request for a $\mathsf{comm}_{\text{req}}$ from $\mathcal{A}$. The decision as to whether this user can join will be made by the honest issuer in the ideal system. The simulator does the following:

– Selects an appropriate $\mathsf{comm}_{\text{req}}$ value and sends this to $\mathcal{A}$. It will then receive a pair (comm, n_I) from $\mathcal{A}$ and performs the necessary checks on comm. If any of these checks fail then $\mathcal{S}$ need do nothing more.
– If n_I was not previous *sent* out by this issuer during a $\mathsf{comm}_{\text{req}}$ request then, as in the protocol description, the simulator aborts this run of the protocol.

– Next the simulator performs the verification checks on (comm, n_I). If any of these checks fail the simulator can ignore the pair.
– If (comm, n_I) has been received by $\mathcal{S}$ before as part of a Join protocol with this issuer, then this is simply a replay of a previous protocol run. The credential computed from this will be a different randomisation of the previously issued credential and such a randomisation could in fact be computed without the issuer. The simulator, acting as the honest issuer in the real system, computes the credential correctly and responds to $\mathcal{A}$ with it. The simulator need do nothing in the ideal system in this case since there will already by a tuple $(i, \mathsf{cnt}, \mathfrak{i}_k)$ on $\mathsf{Members}$ for this particular credential; all $\mathcal{S}$ is doing in this case is re-randomising the credential.
– If (comm, n_I) is new and a pair $(\mathsf{comm}, n_I^\dagger)$ has been received before as part of a Join protocol run, passed each of the checks on it and $n_I^\dagger \neq n_I$ then the adversary must be able to find collisions in H_2 so $\mathcal{S}$ aborts outputting "failure 0".
– If (comm, n_I) is new and a pair $(\mathsf{comm}^\dagger, n_I)$ has been received before during a run of the Join protocol with this issuer where $\mathsf{comm}^\dagger \neq \mathsf{comm}$ then this simply means a different value of cnt was used in the computation of f and hence $\mathsf{comm}^\dagger$. The simulator then, acting as $\mathfrak{h}_i$ in the ideal system, makes a joining request to $\mathcal{T}$ for $(i, \mathsf{cnt}_i + 1, \mathfrak{i}_k)$ and receives a decision from $\mathcal{T}$ as to whether it is allowed to join or not. If not then $\mathcal{S}$ denies the request of $\mathcal{A}$ in the real system and otherwise correctly computes the credential and responds to $\mathcal{A}$ with this. The simulator is able to correctly compute this credential since it knows isk_k; it chose it during the Setup simulation for this issuer.
– If (comm, n_I) is new and no other pair $(\mathsf{comm}^\dagger, n_I)$ or $(\mathsf{comm}, n_I^\dagger)$ with $\mathsf{comm} \neq \mathsf{comm}^\dagger$ and $n_I \neq n_I^\dagger$ has been received by this issuer as part of a Join protocol before then the simulator, acting as $\mathfrak{h}_i$ in the ideal system, makes a joining request to $\mathcal{T}$ for $(i, \mathsf{cnt}_i + 1, \mathfrak{i}_k)$ and receives a decision from $\mathcal{T}$ as to whether it is allowed to join or not. If not then $\mathcal{S}$ denies the request of $\mathcal{A}$ in the real system and otherwise correctly computes the credential and responds to $\mathcal{A}$ with this.

Case (IhM). Here only the host is controlled by the adversary. The simulator will have to play the role of $\mathfrak{h}_i$ in the ideal system to $\mathcal{T}$ and the roles of $\mathfrak{i}_k$ and $\mathfrak{m}_i$ to the adversary in the real system. The action is trigerred when $\mathcal{S}$ receives a request for a $\mathsf{comm}_{\text{req}}$ from $\mathcal{A}$ in the real system and whilst acting as the honest issuer $\mathfrak{i}_k$. Again, the ideal system issuer will decide if this user can join or not. The simulator does the following:

– The simulator, acting as honest issuer $\mathfrak{i}_k$, first generates $n_I \leftarrow \{0,1\}^t$, assigns $\mathsf{comm}_{\text{req}} \leftarrow n_I$ and returns this to $\mathcal{A}$.
– The simulator, acting as honest TPM $\mathfrak{m}_i$, will then receive a value of $\mathsf{comm}'_{\text{req}}$ from $\mathcal{A}$ who is acting as $\mathfrak{h}_i$. We note this may be different to the $\mathsf{comm}_{\text{req}}$ chosen by $\mathcal{S}$ in the previous step. The simulator then correctly computes the TPM side of the join protocol to produce a commitment on n_I using f while acting as $\mathfrak{m}_i$ in the real system. Finally the pair (comm, n_I) is passed to $\mathcal{A}$.
– Next $\mathcal{S}$, acting as $\mathfrak{i}_k$, should receive a pair (comm', n'_I) from $\mathcal{A}$ (again $\mathcal{A}$ may have modified the pair passed to it). If n'_I was not previous *sent* out by this issuer during a $\mathsf{comm}_{\text{req}}$ request then, as in the protocol description, the simulator aborts this run of the protocol.
– If (comm', n'_I) has been received by $\mathcal{S}$ before as part of a Join protocol with this issuer, then this is simply a replay of a previous protocol run. The credential computed from this will be a different randomisation of the previously issued credential and such a randomisation could in fact be computed without the issuer. The simulator, acting as the honest issuer in the real system, computes the credential correctly and responds to $\mathcal{A}$ with it. The simulator need do nothing in the ideal system in this case since there will already by a tuple $(i, \mathsf{cnt}, \mathfrak{i}_k)$ on $\mathsf{Members}$ for this particular credential; all $\mathcal{S}$ is doing is re-randomising the credential.
– Next the simulator, whilst acting as the issuer, performs the verification checks on (comm', n'_I). If any of these checks fail the simulator can ignore the pair. Otherwise the simulator checks if it has sent out comm' whilst acting as $\mathfrak{m}_i$. If not then the host must be able to forge the signature contained within comm' so the simulator aborts outputting "failure 1".
– If (comm', n'_I) is new, from the point of view of the issuer, and a pair $(\mathsf{comm}', n_I^\dagger)$ has been received before as part of a Join protocol run, passed each of the checks on it and $n_I^\dagger \neq n'_I$ then the adversary must be able to find collisions in H_2 so $\mathcal{S}$ aborts outputting "failure 0".

- If (comm', n_I') is new, from the point of view of the issuer, and a pair $(\mathsf{comm}^\dagger, n_I')$ has been received before during a run of the Join protocol with this issuer where $\mathsf{comm}^\dagger \neq \mathsf{comm}$ then this simply means a different value of cnt was used in the computation of f and hence $\mathsf{comm}^\dagger$. The simulator then, acting as $\mathfrak{h}_i$ in the ideal system, makes a joining request to $\mathcal{T}$ for $(i, \mathsf{cnt}_i + 1, \mathfrak{i}_k)$ and receives a decision from $\mathcal{T}$ as to whether it is allowed to join or not. If not then $\mathcal{S}$ denies the request of $\mathcal{A}$ in the real system and otherwise correctly computes the credential and responds to $\mathcal{A}$ with this. The simulator is able to correctly compute this credential since it knows isk_k; it chose it during the Setup simulation for this issuer.
- If (comm', n_I') is new, from the point of view of the issuer, and no other pair $(\mathsf{comm}^\dagger, n_I')$ or $(\mathsf{comm}', n_I^\dagger)$ with $\mathsf{comm}' \neq \mathsf{comm}^\dagger$ and $n_I' \neq n_I^\dagger$ has been received by this issuer as part of a Join protocol before then the simulator, acting as $\mathfrak{h}_i$ in the ideal system, makes a joining request to $\mathcal{T}$ for $(i, \mathsf{cnt}_i + 1, \mathfrak{i}_k)$ and receives a decision from $\mathcal{T}$ as to whether it is allowed to join or not. If not then $\mathcal{S}$ denies the request of $\mathcal{A}$ in the real system and otherwise correctly computes the credential and responds to $\mathcal{A}$ with this.

Case (ihM). Here $\mathcal{S}$, acting as $\mathfrak{m}_i$ in the real system, simulates a correct run of the Join protocol with $\mathcal{A}$. In the ideal system $\mathcal{S}$ plays the roles of $\mathfrak{h}_i$ and $\mathfrak{i}_k$ for an analogous run of the Join protocol. The action here is triggered when $\mathcal{S}$, acting as $\mathfrak{m}_i$ in the real system, receives a value of $\mathsf{comm}_{\mathrm{req}}$ from $\mathcal{A}$. The decision as to whether this user can join is made by the adversary in the real system. The simulator does the following:

- In the ideal system the simulator, acting as the corrupt host $\mathfrak{h}_i$, sends values for identifier, counter corresponding to the value of f to be used in the real system and issuer identifier to $\mathcal{T}$ as a joining request.
- The simulator will then receive a message from $\mathcal{T}$, whilst acting as $\mathfrak{i}_k$ in the ideal system, to ask if the counter and identifier value can become a member. The simulator holds off responding on this for the time being; the adversary will make this decision in the real system.
- In the real system, acting as $\mathfrak{m}_i$, the simulator first computes a correct value of comm according to some value of f and then responds to $\mathcal{A}$ with comm. If the simulator is sent a value of B in the real system, whilst acting as $\mathfrak{m}_i$, then it replies to the trusted third party $\mathcal{T}$ in the ideal system, whilst acting as $\mathfrak{i}_k$, to inform it that $\mathfrak{m}_i$ can become a member in the ideal system.
- The simulator then computes the value for D in the real system and responds to the adversary with this.

Simulation of Sign and Verify. Many parts of the simulation of the Sign and Verify protocol are very closely related and depend upon each other. For this reason, and for greater clarity, we describe these together. In each case we will make it clear exactly which protocol and combination of player corruption is being considered.

For the simulation of the Sign protocol there are 4 possible cases for player corruption combinations. For the Verify protocol there are only two cases of verifier corruption state. However, in each case the behaviour of the simulator will depend upon the corruption state of the user that produced the signature and of the issuer with whom the Join protocol was run. We can then distinguish 6 separate cases where the simulator has to get involved; if the verifier and host have the same corruption state then the verification operation appears as an internal operation of either the adversary or trusted third party. If the verifier is corrupt then the host and hence TPM must be honest and if the verifier is honest then at least the host must be corrupt. This allows us to remove a number of the total possible cases and we describe the operation of $\mathcal{S}$ based on the remaining ones individually.

Before we do describe the details we again give some intuition as to the security properties enforced by the ideal functionality and how these are covered in the description of the simulator. Recall that during the simulation of the join protocol the ideal functionality enforces that a member cannot join if an issuer does not allow it. In real terms this means a user cannot obtain a credential from anywhere other than an issuer. If a credential was computed by the adversary for an honest issuer then the simulator should notice such a credential when a randomisation of it is presented for verification with an honest verifier; it can be recognised either from the value of f used to compute it or by the absence of a corresponding entry on the members list in the ideal system. If this occurs then the simulator raises a failure event.

Other security properties enforced by the ideal functionality include that an adversary cannot forge DAA signatures from an TPM nor can an adversary subvert basename verifiability when computing DAA signatures with an honest TPM. Both of these are covered in the description of the simulator during the verify protocol run with an honest TPM and honest verifier. In each case a failure event is output if the property is broken by the adversary.

We now describe the details of the simulator according to each user corruption case for both Sign and Verify.

Case (HM) of Sign. In this case both host and module are honest hence the user will produce the signature in the real system without any interaction with $\mathcal{S}$. The simulator ensures the ideal system is consistent with the real system when such a signature is presented for verification to a corrupt verifier.

Case $(vHMI)$ of Verify. Here the simulator would not have noticed this signature production in the ideal system since the user is completely honest. In this particular case the issuer i_k is also honest. The action here is triggered when $\mathcal{S}$, acting as the corrupt verifier in the ideal system, receives a signature σ on message msg with respect to bsn from the user $\mathfrak{u}_i$.

Since the user is completely honest this signature should verify correctly when presented to $\mathcal{T}$; if not then $\mathcal{S}$ need do nothing more. The simulator $\mathcal{S}$ then has to produce a signature in the real system on behalf of $\mathfrak{u}_i$ that correctly verifies with the corrupt verifier controlled by $\mathcal{A}$. The simulator will either already have a validly produced credential for the issuer or will simply simulate a run of the Join protocol for this case to obtain one. The simulator then correctly computes both the TPM and host roles in the Sign protocol using the same value of f as in the run of the Join protocol for the credential to be signed. This signature is then passed to the corrupt verifier in the real system for verification.

Case (hm) of Sign. This is similar to the previous case of the Sign protocol simulation. In the real system the adversary will produce the signature internally and without any interaction with the simulator. Consistency with the ideal system is ensured when such a signature is presented for verification to an honest verifier. Again, we deal with this in the description of how $\mathcal{S}$ behaves in the Verify protocol.

Case $(VhmI)$ of Verify. The action of $\mathcal{S}$ here will be triggered when $\mathcal{S}$ receives a signature σ from $\mathcal{A}$ on some message msg with respect to a basename bsn and issuer i_k. The simulator first performs a verification check as in the Verify algorithm description except for the checking for rogue values. If this check fails then $\mathcal{S}$ can just ignore the signature. Otherwise $\mathcal{S}$ performs the rogue check.

If the signature fails the rogue check, the simulator is able to obtain the value of f used from the simulated rogue list SimRogueList. The simulator has to then find which $\mathfrak{m}_i$ to associate the signature to and ensure that in the ideal system any required signature calls are made and the simulated rogue list, SimRogueList, agrees with RogueList maintained by $\mathcal{T}$.

Since i_k is honest and the signature has correctly verified except for the rogue checking then there may have been a run of the Join protocol run with the simulator for case (hmI); such a user would have joined before f was added to RogueList. We know that, since the signature failed the rogue check, $fJ = K$ for the value of f obtained from RogueList. We also know this value of f corresponds to the credential that was signed. If this user obtained the underlying credential correctly then there will be a value of F received by $\mathcal{S}$ while acting as i_k in a Join protocol such that $fP_1 = F$. The simulator can identify which platform f corresponds to by checking if $fP_1 = F$ for each comm received by i_k. Also worth noting is that, since the simulator knows f, it can check if the underlying credential was issued by i_k by checking if there is some tuple (A, B, C) sent out by i_k as part of a Join protocol run such that $\hat{t}(A + fB, X) = \hat{t}(C, P_2)$. We use the value of F received by i_k in such a run since this will identify the specific TPM rather than just link signatures together as would be the case if we used the second check.

If this does not identify the platform then the adversary must have been able to obtain the underlying credential on f without the help of i_k. In this case the simulator aborts outputting "failure 2".

Once the user has been identified the simulator, acting as $\mathfrak{u}_i$ in the ideal system, makes a call to $\mathcal{T}$ as the host to sign msg with respect to the corresponding cnt and bsn values. The simulator then passes the received signature to the corresponding verifier in the ideal system. Upon asking for a verification decision from $\mathcal{T}$ this signature will fail on the rogue check part of verification as the actual signature did.

This is because a rogue tagging request would have been made to $\mathcal{T}$ when f was added to the rogue list in the real system.

If the signature passes the rogue check this means that although the user is corrupted in the real system it has yet to be added to $\mathsf{RogueList}$. In the ideal system this corresponds to a TPM module that has been added to $\mathsf{CorruptTPM}$ but no other player has yet made a rogue tagging request to $\mathcal{T}$ for the particular (i, cnt) pair used in the signature.

In this case the simulator is unable to identify which module has signed the message. Indeed, this should be the case since signatures produced correctly should be unlinkable to a given user. The simulator instead assigns this pair to some corrupted $\mathfrak{m}_i$ and counter value not on the $\mathsf{RogueList}$. For a correctly chosen user the environment will be unable to tell the difference between the systems also due to the anonymity property which is proved in Lemma 3. The simulator does this as follows:

– If there is no module $i \in \mathsf{CorruptTPM}$ with a corresponding cnt such that $(i, \mathsf{cnt}, \mathfrak{i}_k) \in \mathsf{Members}$ yet $(i, \mathsf{cnt}) \notin \mathsf{RogueList}$ then the simulator aborts outputting "failure 2". In this case $\mathcal{A}$ must have been able to obtain a credential that was not issued to it in the real system and use this to produce the signature.
– If $\mathsf{bsn} = \perp$ then $\mathcal{S}$ can select any $i \in \mathsf{CorruptTPM}$ and value of cnt such that $(i, \mathsf{cnt}, \mathfrak{i}_k) \in \mathsf{Members}$ and $(i, \mathsf{cnt}) \notin \mathsf{RogueList}$. Then, acting as the corresponding corrupted host $\mathfrak{h}_i$ in the ideal system, $\mathcal{S}$ initiates the signing of msg with respect to cnt and bsn with $\mathcal{T}$. It will receive a signature and can send this to the corresponding verifier in the ideal system.
– If $\mathsf{bsn} \neq \perp$ then $\mathcal{S}$ has to also ensure it assigns this to a TPM with $i \in \mathsf{CorruptTPM}$ and counter value cnt such that $(i, \mathsf{cnt}) \notin \mathsf{RogueList}$ and such that this links to the correct number of signatures with this same basename. All such signatures from this user and with this issuer will have been built in the ideal system in this manner. Once a suitable user is selected the simulator simply has to build the same linkability of signatures structure. By correctly maintaining this the simulator will always be able to identify a user in the ideal system to assign this signature to. Furthermore, the environment will be unable to distinguish the real from the ideal system since signatures will be unlinkable to users yet linkable to each other in the same shape.
  Once it has found this user and counter value it initiates a signing of msg with respect to bsn and cnt with $\mathcal{T}$. Upon receipt of this signature it can present it to the corresponding verifier in the ideal system.

Case (Hm) of Sign. Since we assume it is harder to break into a TPM than it is to break into a host we do not consider this case.

Case (hM) of Sign. In this case the TPM is honest and the host controlled by the adversary. The simulator has to play the role of honest $\mathfrak{m}_i$ in the real system to $\mathcal{A}$ and corrupt $\mathfrak{h}_i$ in the ideal system to $\mathcal{T}$. The action here will be trigerred when $\mathcal{S}$ receives a tuple $(h, J, \beta, \mathsf{msg})$ from $\mathcal{A}$ in the real system whilst $\mathcal{S}$ is acting as $\mathfrak{m}_i$. If no Join protocol has been successfully finished by $\mathfrak{m}_i$ then $\mathcal{S}$ rejects the request. Before computing the TPM part of the signature the simulator, acting as $\mathfrak{h}_i$ in the ideal system, sends a request to sign to $\mathcal{T}$ for the corresponding $(\mathsf{msg}, i, \mathsf{cnt}, \mathfrak{i}_k)$. If the ideal system TPM does not want to sign the message then the simulator refuses to sign and otherwise, in the real system, correctly computes the TPM part of the Sign protocol for the corresponding value of f used in the Join protocol.

Case $(VhMI)$ of Verify. In this case the verifier is honest and the action of $\mathcal{S}$ will again be triggered when it receives a signature σ from $\mathcal{A}$ on some message msg with respect to a basename bsn and issuer $\mathfrak{i}_k$. The simulator first performs a verification check as in the Verify algorithm description. If this check fails then $\mathcal{S}$ can just ignore the signature.

For the remainder of this case we describe the actions of $\mathcal{S}$ based on what it has previously computed and seen, in terms of the signature and underlying credential, when acting as $\mathfrak{m}_i$.

The $(\sigma, \mathsf{bsn}, \mathsf{msg})$ tuple. If the tuple $(\sigma, \mathsf{bsn}, \mathsf{msg})$ has been previously submitted to $\mathcal{S}$ before whilst acting as $\mathfrak{v}_j$ then $\mathcal{S}$ need do nothing more; this signature should already be assigned to a user in the the ideal system and various checks by $\mathcal{S}$ passed. From here onwards we assume there is some part of this tuple that $\mathfrak{v}_j$ has not seen in any of the signatures it has previously received.

The (R, S, T) tuple. First the simulator checks the tuple (R, S, T) contained within the signature. This should satisfy two things: firstly the underlying credential should correspond to a value of f held by some honest $\mathfrak{m}_i$ and the commitment used to obtain it computed during a run of the Join protocol for the case (hMI). Secondly, the underlying credential should have been computed and issued by $\mathfrak{i}_k$ for some run of Join for the case (hMI) with the same commitment.

During the Join protocol for the case (hMI) the simulator should have computed a commitment using some value of f when acting as an honest $\mathfrak{m}_i$ and then computed a credential on this using the private key for $\mathfrak{i}_k$ whilst acting as $\mathfrak{i}_k$. The simulator will have a list of all such values of f it has used as an honest TPM in the Join protocol for case (hMI) and uses this list to identify which user has produced this signature. It does this by checking if $\hat{t}(R + fS, X) = \hat{t}(T, P_2)$ and $\hat{t}(R, Y) = \hat{t}(S, P_2)$ or not for each value of f on its list. Essentially the simulator is checking if the credential used in the signature is a randomisation of one obtained on the value of f used in the check.

If this condition is not met for one of the f values $\mathcal{S}$ has stored then the adversary must be able to forge a credential from an honest issuer. This is the case since the adversary could have either forged the signature within the commitment and then obtained the credential correctly from the issuer or forged the credential production or both. The case of the credential being obtained correctly and the signature within the commitment being forged would have lead to the simulator aborting with “failure 3” during the Join protocol simulation. As a result we conclude that the adversary must have at least forged the credential. The simulator aborts outputting “failure 2” in this case.

The (J, K, c, s, n_T) tuple. During a simulated run of the Sign protocol for case (hM) the simulator would have received a value of J and computed the remainder of the values K, c, s, n_T in the signature. The simulator knows which $\mathfrak{m}_i$ was used to obtain the credential and the corresponding f for this credential during the Join protocol (otherwise it would have previously aborted outputting a failure event). It then checks its records for $\mathfrak{m}_i$ to see if it has received J and output K, c, s, n_T in response during a Sign protocol run. If it has seen this tuple before we can deduce the value of n_V used by the adversary is the same as the one in the signature. The value of c is identical to one the simulator computed and the signature correctly verified with n_V; if a different value of n_V was used in the adversaries computations then it would have had to find a collision in H_4.

If the simulator has not computed the tuple (J, K, c, s, n_T) then the adversary must be able to compute signatures on valid credentials without the help of $\mathfrak{m}_i$. The simulator identifies how this forgery was computed before outputting a failure event.

First the simulator checks if the value of f, obtained previously, used in the credential computation is such that $K = fJ$. If not then the adversary is able to compute signatures using a value of f' different to that used in the credential computation. In this case the adversary would have been able to somehow subvert the proof of knowledge of f that corresponds to both the credential and the signature. The simulator aborts outputting “failure 3” Otherwise the adversary is able to forge DAA signatures for a value of f unknown to it so the simulator aborts outputting “failure 4”.

The $(\mathsf{bsn}, \mathsf{msg})$ pair. If all of the above checks are passed by the signature then $\mathcal{S}$ needs to ensure the ideal system is consistent with the real system and that basename verifiability has not been subverted. During the case (hM) of Sign a tuple of the form $(\tilde{\sigma}, \mathsf{msg}, \mathsf{bsn}, i, \mathsf{cnt}, \mathfrak{i}_k)$ should have been added to Signatures in the ideal system. The simulator will have a record of all such tuples on this list for this particular user. If $\mathcal{S}$ has no record of such a tuple then the adversary must be able to obtain a signature with one pair $(\mathsf{bsn}', \mathsf{msg}')$ and then use this to produce a correctly verifying signature on $(\mathsf{bsn}, \mathsf{msg})$. To do this the adversary would have to find a collision in either H_3 or H_5 so the simulator aborts outputting “failure 5”. Otherwise the simulator, whilst acting as the corrupt $\mathfrak{h}_i$ in the ideal system, submits the tuple $(\tilde{\sigma}, \mathsf{msg}, \mathsf{bsn}, i, \mathsf{cnt}, \mathfrak{i}_k)$ to $\mathfrak{v}_j$ for verification. This signature will then correctly verify in the ideal system when submitted to $\mathcal{T}$ by $\mathfrak{v}_j$.

Simulation of RogueTag. We assume that RogueTag will only be run by $\mathcal{S}$ when $\mathcal{S}$ is simulating an honest $\mathfrak{v}_j$ or an honest $\mathfrak{i}_k$ in the real system and is given a tuple $(\sigma, \mathsf{bsn}, \mathsf{msg}, \mathsf{ipk}_k, f)$. The simulator first checks this is a valid signature on the parameters given and if this check fails rejects the request.

Next the simulator checks if the value of f submitted corresponds to the values of J and K within the signature; if $K = fJ$ or not. If not then the simulator rejects the request. Otherwise the simulator checks if the value of f submitted corresponds to the underlying credential. If $\hat{t}(R + fS, X) \neq \hat{t}(T, P_2)$ then the adversary must be able to produce a credential with one value f' then produce a signature on this credential using another value f so the simulator aborts outputting "failure 3". Otherwise the simulator checks to see whether it has obtained a credential as an honest TPM using the value of f. It does this by checking if $fP_1 = F$ for any of the values of F it has computed in a run of the Join protocol with either an honest or corrupt host. If so then the adversary must be able to compute discrete logarithms in the group $\mathbb{G}_1$ so the simulator aborts outputting "failure 6". Otherwise we distinguish the behaviour of the simulator based on the corruption state of the issuer with public key ipk_k.

If this issuer is honest then the value of F such that $F = fP_1$ used to obtain the underlying credential should have been received within a comm tuple by the simulator whilst playing the role of this issuer. If not then the adversary must be able to produce credentials without the aid of the issuer so the simulator aborts outputting "failure 2". If such a value of comm was received by the issuer and verified with a public key of an honest TPM then the adversary must have been able to forge signatures used within the commitment so the simulator aborts outputting "failure 1". Otherwise we have the case where the TPM that obtained the credential and produced the signature is corrupt. In this case the simulator adds the value of f to the rogue list and ensures the ideal system is consistent with the real system. It does this by making a rogue tagging request to $\mathcal{T}$ in the ideal system with the corresponding values of i and cnt; it can do this since it is able to identify the TPM in the real system.

If the issuer is corrupt then the simulator will have no way of identifying the corrupt TPM that produced this signature and the underlying credential. It could even be the case that the adversary produced this credential without the aid of a TPM since the issuer is corrupt and can issue credentials to any player it likes. The simulator takes no action in this case; the ideal system will be consistent since the TPM and host must be corrupt and would not have engaged with the trusted third party in the ideal system to be added to the members list. As a result no entry would be added to the ideal system rogue list for this member and counter pair.

The proof the theorem then follows from the description of the simulator $\mathcal{S}$ and Lemmas 1, 2 and 3.

We then have the following two Lemmas.

Lemma 1. *If the blind bilinear LRSW assumption holds for groups $\mathbb{G}_1 = \langle P_1 \rangle, \mathbb{G}_2 = \langle P_2 \rangle, \mathbb{G}_T$ of large prime order q and pairing $\hat{t} : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$, the public key signature scheme used is UF-CMA secure and the gap-discrete logarithm assumption holds in $\mathbb{G}_1$ then $\mathcal{S}$ will abort with negligible probability in the random oracle model.*

Proof. To prove this Lemma we look at each failure event in turn and argue that each will occur with only negligible probability.

Failure 0. For this failure event to occur the simulator must have received two pairs (comm, n_I) and $(\mathsf{comm}, n_I^\dagger)$ such that both pass all of the checks and $n_I \neq n_I^\dagger$. Informally, this means the nonce provided by the issuer has not ensured a new commitment value and, in particular, the adversary must have been able to find a collision in the hash function H_2. We now discuss the details in a more formal manner.

Since $n_I \neq n_I^\dagger$ this means $\mathsf{str} \neq \mathsf{str}^\dagger$ yet they share the same value of v. This means $v = H_2(\mathsf{str} \| F \| U) = H_2(\mathsf{str}^\dagger \| F \| U^\dagger)$ where U may or may not be equal to $U^\dagger$. This would mean a collision in the hash function H_2 but since H_2 is modelled as a random oracle this will happen with negligible probability for sufficiently large q.

Failure 1. This failure event occurs whenever the adversary is able to produce a commitment comm that verifies as coming from an honest $\mathfrak{m}_i$ yet the TPM $\mathfrak{m}_i$ has not computed the commitment. We argue this failure event occurs with negligible probability relative to the UF-CMA security of the digital signature scheme used within the Join protocol.

Informally, to prove this we construct an algorithm which can produce a valid forgery, and hence win the UF-CMA game against the digital signature scheme, with probability at least that of the probability of this failure event occurring. We now describe the details.

We refer to the algorithm against the UF-CMA security of the digital signature scheme as the simulator since it has only minor differences to the description of the simulator in the proof of Theorem 1 and to distinguish it from the adversary in the same proof. At the start of the security game for UF-CMA of the digital signature scheme the simulator is given a public verification key for the signature scheme and a signing oracle that uses some hidden signing key corresponding to this public key. We assume the messages signed are of the form $\mathbb{G}_1||\mathbb{Z}_q||\mathbb{Z}_q$; in reality there will be some encoding of such elements into bit strings. The simulator then sets up the simulation with $\mathcal{A}$ as it does in the proof of Theorem 1. During the setup the simulator selects some TPM module $\mathfrak{m}_i$ for which the host $\mathfrak{h}_i$ is corrupt and for which it hopes this failure event will occur and assigns the public key it was given in the UF-CMA game to be the public endorsement key of this TPM.

The simulator then behaves exactly as in the proof of Theorem 1 except if, when acting as $\mathfrak{m}_i$ in a run of the Join protocol, it is asked for a commitment. In this case it uses the signing oracle provided to it to sign any commitments it sends out to the host.

If this failure event occurs with the honest issuer $\mathfrak{i}_k$ then there must be been some value of commitment sent to $\mathfrak{i}_k$ which the simulator did not ask for a signing query on yet which correctly verifies; the commitment used to obtain the credential which was signed to cause this failure event. We now argue this commitment, which we denote $\mathsf{comm}' = (F', v', w')$, was not submitted as a query to the signature oracle. Firstly, the signature on the credential produced from comm' correctly verified with this issuer hence comm' must have correctly verified with this issuer and a credential issued on it. As a result, if comm' was submitted to the signature oracle by the simulator it would have been during a Join protocol with this issuer. However, the value f' used to obtain F' is different from any which correspond to this issuer and TPM. We know this since the randomised credential in the signature which caused the failure event did not correspond any of the f values for this issuer. Hence comm' is different to any that would have been computed by $\mathfrak{m}_i$ for this issuer and as a result would not have been submitted to the signature oracle by the simulator.

The simulator identifies comm' by checking all those received by $\mathfrak{i}_k$ that verify with the public endorsement key corresponding to $\mathfrak{m}_i$ for the one that $\mathfrak{m}_i$ did not query to the signature oracle and then outputs this as its forgery in the UF-CMA game against the digital signature scheme.

FAILURE 2. For this failure event to occur the adversary must have been able to obtain a credential from an honest issuer without the help of this issuer. The adversary has then been able to produce a signature on this credential that verifies in all but the rogue checking part of the verification. We show this failure event happens with negligible probability relative to the blind bilinear LRSW assumption by constructing an algorithm for the B-bLRSW from the adversary $\mathcal{A}$.

Informally, we construct the algorithm for the B-bLRSW from the simulator by using the oracle $\mathcal{O}^B_{X,Y}(\cdot)$ to produce credentials for unknown x, y. Such credentials are used to compute responses for an honest issuer in Join protocol runs. The algorithm then uses the value of f obtained from the rogue list checking to construct the solution to the B-bLRSW problem. We now describe the details.

In our description we refer to the algorithm for solving the B-bLRSW problem as the simulator since it is only a slight variation on the description of the simulator. We then describe the necessary changes to the previous simulator description to arrive at our algorithm for the B-bLRSW. For the B-bLRSW problem the simulator will be given a tuple $(\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, X, Y, q, \hat{t})$ and an oracle $\mathcal{O}^B_{X,Y}(\cdot)$. It uses $\mathbb{G}_1, \mathbb{G}_2, P_1, P_2, q, \hat{t}$ as the public parameters for the real system setup of the whole scheme and selects one honest issuer in the real system which for it hopes the adversary will compute the credential underlying the signature, without the help of the issuer, which causes this failure event. When hosts ask to run the Join protocol with the issuer the simulator has selected it checks the commitment sent and if it is correct uses $\mathcal{O}^B_{X,Y}(\cdot)$ with input the value of F given in comm to produce the required A, B, C values for the credential. In this way it can correctly simulate the behaviour of this issuer without knowing the values of x or y.

When this failure event occurs the simulator will be able to obtain the (R, S, T) tuple used in the signature and the value of f which this tuple corresponds to from the rogue list.

A credential (R, S, T) will be valid if and only if $\rho_c^\dagger/\rho_a^\dagger = (\rho_b^\dagger)^f$ due to the way the credential is constructed. The verification algorithm computes $\tau^\dagger = (\rho_b^\dagger)^s(\rho_c^\dagger/\rho_a^\dagger)^{-c}$ and for this to verify correctly it must equal β^z. If the two are equal then we get $\rho_c^\dagger/\rho_a^\dagger = (\rho_b^\dagger)^f$ meaning the credential must be a correctly computed one.

Finally, this failure event would not have occurred if fP_1 was submitted as a query to $\mathcal{O}^B_{X,Y}(\cdot)$ hence we can deduce it was not and the tuple (f, R, S, T) is a valid solution to the B-bLRSW problem.

Failure 3. For this failure event to occur the adversary must have somehow obtained a credential with respect to a value f', yet produced a DAA signature on this credential that verifies with respect to a value $f \neq f'$ such that the DAA signature correctly verified. We argue that this will not happen since the DAA signature contains a proof of knowledge of a value of f such that $K = fJ$ and $\hat{t}(R + fS, X) = \hat{t}(T, P_2)$ and that this value of f is the same in both cases. This proof of knowledge consists of the verification of L and τ using the same values of s and c. Thus it is essentially the standard Σ-protocol for equality of logarithms, and the standard soundness argument implies that f must equal f'. More explicitly we could rewind on this failure event, alter the random oracle value c (i.e. fork on c) and then apply the knowledge extractor for the Σ-protocol to extract $f = f'$.

Failure 4. This failure event corresponds to the case $(VhMI)$ of the Verify simulation where $\mathcal{S}$, acting as the verifier $\mathfrak{v}_j$ in the real system has received a signature from the adversary, playing the role of $\mathfrak{h}_i$, which has verified and passed the rogue check.

For this signature the simulator has chosen a value of f as the honest m_i and issued a credential on this as the honest i_k. The simulator has then forged the signature on this credential that corresponds to the value of f used in the credential. We argue this does not happen relative to the gap-DL problem in $\mathbb{G}_1$. Informally, we embed the gap-DL problem into the choice of F in the run of the Join protocol for the credential underlying the signature. We then use the issuer secret parameters to compute the value of D in the same protocol run. If the simulator is asked to produce a signature on this credential we use the Diffie-Hellman oracle provided to produce this. We finally use rewinding on the adversary to extract the underlying value of f used in the forged signatures and hence solve the gap-DL problem. We now describe the details.

We refer to the algorithm against the gap-DL as the simulator since it has only minor differences to the description of the simulator in the proof of Theorem 1 and to distinguish it from the adversary in the same proof. At the start of the game for the gap-DL the simulator is given a description of a group and some generator of that group which will be used as a base for the discrete logarithm problem. It is also given a challenge element of this group and a Diffie-Hellman oracle for the hidden value of f. The simulator then sets up the simulation as in the proof of Theorem 1 but sets the group $\mathbb{G}_1$ to be the group provided in the DL challenge and the element P_1 to be the base/generator of the challenge group. The simulator then runs the simulation as before except for the following. The simulator selects some run of the simulation of the Join protocol for which it hopes a signature will be computed and submitted for verification to cause this failure event. In this simulation of the Join protocol the simulator will receive some value of $\mathsf{comm}_{\mathrm{req}}$ from the adversary. Rather than compute f according to the issuer parameters and compute F from this, the simulator instead performs the following steps:

- Assigns F to be the DL challenge.
- Randomly chooses values for w and v then computes $U = wP_1 - vF$.
- Patches the random oracle for H_2 such that $v = H_2(\mathsf{str}\|F\|U)$.
- Computes a correct signature on this γ and outputs the corresponding commitment comm. Note that this proof of knowledge will pass all the checks the adversary will make on it.

The simulator will then receive some value of B from the adversary and computes $D = ryF$ (the simulator knows y and r since i_k is honest) then returns this to the adversary.

Then, during any simulations of the Sign protocol for this particular value of f the simulator may have to compute a signature on this credential; this will be different to the one that causes this failure event yet the simulator has to produce one that correctly verifies to ensure the adversary cannot tell it is in a simulation. Essentially the simulator has to forge such a signature for the hidden value of f. For this situation to occur the simulator will be passed a value of J from the adversary. The simulator then does the following:

1. First $\mathcal{S}$ selects $c, s \leftarrow \mathbb{Z}_q$. If the value of J to be used is one used before for this particular issuer, $\mathsf{cnt}, \mathsf{comm}_{\mathrm{req}}$ values and TPM, then $\mathcal{S}$ uses the same value of K as before and otherwise computes K

by querying the DH oracle provided on the value of J and records the (J, K) pair to ensure consistency with later signatures forged. The simulator then computes $L = sJ - cK$ and $H = (s - cf)S$ where f is the value used to obtain the credential with the same TPM, $\mathsf{cnt}, \mathsf{comm}_{\mathrm{req}}$ and issuer. The value of S used will be obtained from the random oracle query to H_4 used to obtain h in the signature.

2. The simulator then selects n_T and computes $\tau = \hat{t}(H, X)$ then patches the random oracle for H_5 such that $c = H_5(h\|\mathsf{msg}\|J\|K\|L\|\tau\|n_T)$.
3. The simulator then holds off responding to $\mathcal{A}$ until it simulates the part of $\mathfrak{h}_i$ in the ideal system. In this case, in the ideal system $\mathcal{S}$ has to simulate the part of $\mathfrak{h}_i$ to $\mathcal{T}$. It does this by making a request to $\mathcal{T}$ to sign msg with respect to the current value of cnt for i and $\mathfrak{i}_k$. If $\mathcal{T}$ informs $\mathcal{S}$ that $\mathfrak{m}_i$ is ready to sign and requests a basename then $\mathcal{S}$ looks up the corresponding basename for J from HList_3 then responds to $\mathcal{T}$ with this. The simulator will then receive a signature from $\mathcal{T}$ which it stores and and replies to $\mathcal{A}$, as $\mathfrak{m}_i$ in the real system, with the tuple (K, c, s, n_T) computed in the previous steps.

Such a signature will correctly verify and hence the adversary will be unable to tell it is run in a simulation.

If this failure event occurs for this particular issuer and user combination the simulator will be able to obtain a forged signature σ_1 corresponding to the discrete logarithm of F on a randomisation (R, S, T) of the credential issued during this Join protocol run and using values c_1 and s_1. The simulator makes a note of the random oracle query made to H_5 to obtain c_1. Next the simulator re-starts the adversary using the exact same input and random tape and answering random oracle queries in the same way as before. When the random oracle query used to obtain c_1 is queried the simulator instead responds with a different value c_2. This will result in a new signature forgery σ_2 using values c_2 and s_2 but with the same values for R, S, T and the same values for J, K and L; the execution will only fork at the point of the H_5 random oracle query and will be exactly the same up to this point. We then have $\tau = (\rho_b)^{s_1}(\rho_c/\rho_a)^{-c_1} = (\rho_b)^{s_2}(\rho_c/\rho_a)^{-c_2}$ from which we can extract the underlying f as $(s_1 - s_2) \cdot (c_1 - c_2)^{-1} \pmod q$. Note we have to fork on the H_5 query rather than the H_4 query since we need the value of τ computed to be the same in both runs of the adversary.

Finally, since f corresponds to the discrete logarithm of F and the randomisation (R, S, T) of the credential in which the challenge was embedded we get that f is a valid solution to the gap-DL.

Failure 5. In this case the simulator has worked with the adversary to forge a signature on values $(\mathsf{msg}', \mathsf{bsn}')$ with an unknown value of f which has then correctly verified using a different $(\mathsf{msg}, \mathsf{bsn})$ tuple. If this failure event occurs then the simulator will be able to determine whether the msg or bsn on which it verified are different from those used to compute the signature by either checking its records for a value of msg during a Sign protocol run or by checking the random oracle for H_3. If the value of msg used is different then the adversary has found a collision on H_5. If the value of bsn is different then the adversary has found a collision on H_3; the adversary may have used bsn' in the preimage to H_4 but it must be the case that $J = H_3(\mathsf{bsn})$ for the signature to verify yet if this differs from the original case there must be an entry for the random oracle of H_3 with a preimage of bsn'. In either case this failure event occurs with negligible probability since all hash functions are modelled as random oracles.

Failure 6. Here the adversary is able to obtain the underlying value of f used by an honest TPM in a run of the Join protocol. We show this failure event occurs with negligible probability relative to the gap-DL problem in $\mathbb{G}_1$. When embedding this problem into the simulation we need to ensure the value of commitment computed by the simulator and any signatures computed with respect to the corresponding credential issued are with respect to the same unknown value of f; if the adversary can compute f then it can deduce it is in a simulation and may abort rather than submit the rogue tagging request that causes this failure event. In both cases the DH oracle provided as part of the gap-DL problem is used to embed the problem into the simulation.

We refer to the algorithm against the gap-DL as the simulator since it has only minor differences to the description of the simulator in the proof of Theorem 1 and to distinguish it from the adversary in the same proof. At the start of the game for the gap-DL the simulator is given a description of a group and some generator of that group which will be used as a base for the discrete logarithm problem. It is also given a challenge element of this group and a Diffie-Hellman oracle.

The simulator then sets up the simulation as in the proof of Theorem 1 but sets the group $\mathbb{G}_1$ to be the group provided in the gap-DL challenge and the element P_1 to be the base/generator of the challenge group. The simulator then runs the simulation as before except for the following. The simulator selects some honest TPM, counter value and issuer combination; i.e. a tuple corresponding to a choice of f value for which it hopes the adversary will submit the value of f to cause this failure event. The TPMand host pair can be either (hM) or (HM) and the corruption state of the issuer is unimportant here; we will distinguish the behaviour of the simulator on these two only.

In the simulation of the Join protocol for these cases of host and TPM corruption the simulator will receive some value of $\mathsf{comm}_{\text{req}}$ from the adversary, acting as a corrupt host, or from itself acting as the honest host TPM. Rather than compute f according to the issuer parameters and compute F from this, the simulator instead embeds the gap-DL challenge into the commitment computation as follows:

1. Assigns F to be the gap-DL challenge.
2. Randomly chooses values for w and v then computes $U = wP_1 - vF$.
3. Patches the random oracle for H_2 such that $v = H_2(\mathsf{str}\|F\|U)$.
4. Computes a correct signature on this γ and outputs the corresponding commitment comm. Note that this proof of knowledge will pass all the checks the adversary will make on it.

The simulator will then receive some value of B from either the adversary or itself and computes $D = fB$ by submitting B to the DH oracle provided to it as part of the simulation.

Then, during any simulations of the Sign protocol for this particular value of f the simulator has to forge signatures for the same hidden value of f. If the host is honest then such a forgery will correspond to a $(vHMI)$ run of the Verify protocol and otherwise will correspond to a (hM) run of the Sign protocol. In the first case the simulator will get to choose J and otherwise will be passed a value of J from the adversary. The simulator then does the following:

1. First $\mathcal{S}$ selects $c, s \leftarrow \mathbb{Z}_q$. If the value of J to be used is one used before for this particular issuer, $\mathsf{cnt}, \mathsf{comm}_{\text{req}}$ values and TPM, then $\mathcal{S}$ uses the same value of K as before and otherwise computes K by querying the DH oracle provided on the value of J and records the (J, K) pair to ensure consistency with later signatures forged. The simulator then computes $L = sJ - cK$ and $H = (s - cf)S$ where f is the value used to obtain the credential with the same TPM, $\mathsf{cnt}, \mathsf{comm}_{\text{req}}$ and issuer. The value of S will either have been computed by the simulator (for an honest host) or can be obtained from the random oracle query used to compute h (for a corrupt host).
2. The simulator then selects n_T and computes $\tau = \hat{t}(H, X)$ then patches the random oracle for H_5 such that $c = H_5(h\|\mathsf{msg}\|J\|K\|L\|\tau\|n_T)$.
3. If the host is corrupt then the simulator holds off responding to $\mathcal{A}$ until it simulates the part of $\mathfrak{h}_i$ in the ideal system. In this case, in the ideal system $\mathcal{S}$ has to simulate the part of $\mathfrak{h}_i$ to $\mathcal{T}$. It does this by making a request to $\mathcal{T}$ to sign msg with respect to the current value of cnt for i and $\mathfrak{i}_k$. If $\mathcal{T}$ informs $\mathcal{S}$ that $\mathfrak{m}_i$ is ready to sign and requests a basename then $\mathcal{S}$ looks up the corresponding basename for J from HList_3 then responds to $\mathcal{T}$ with this. The simulator will then receive a signature from $\mathcal{T}$ which it stores and and replies to $\mathcal{A}$, as $\mathfrak{m}_i$ in the real system, with the tuple (K, c, s, n_T) computed in the previous steps.
4. If the host is honest the simulator forms the signature as $\sigma = (R, S, T, J, K, c, s, n_V, n_T)$ and presents the complete signature to $\mathcal{A}$ for verification in the real system.

Finally, since f corresponds to the discrete logarithm of F and the randomisation (R, S, T) of the credential in which the challenge was embedded we get that f is a valid solution to the gap-DL.

This concludes the proof.

Lemma 2. *In an execution of the above system with simulator $\mathcal{S}$ no* p.p.t. *adversary can distinguish if it's run in a simulation of the real system for the scheme or if it is run in an actual execution of the scheme.*

Proof. If no failure events occur then the simulator simply correctly simulates the behaviour of honest players in the real system for values of internal secret parameters that it chooses according to the same distributions as in the protocol description. The simulator only behaves differently from the protocol description if a failure event occurs; in this case a problem is embedded in the simulation. In the previous lemma we showed that each such failure event occurs with negligible probability and hence the adversary can only tell it is in a simulation with negligible probability.

Lemma 3. *In an execution of the above system with simulator $\mathcal{S}$ no computationally bounded environment can distinguish if it's run in the real system with a real adversary or in an ideal system with a simulator provided the adversary used by $\mathcal{S}$ does not abort and the decisional Diffie-Hellman problem is hard in $\mathbb{G}_1$.*

Proof. Lemma 1 argued that each of the failure events occurred with negligible probability and hence an environment cannot distinguish whether it is run in the real or ideal system based on the *actions* of the adversary. What it may be able to do is distinguish which system it is in based on what message flows it sees; what can be *learned* by an adversary. To prove both systems are indistinguishable by an environment we hence show that signatures produced in the real system are unlinkable to each other where they are intended to be since in the ideal system this is always the case. This also proves that signatures are anonymous: an adversary, given a signature it wants to link to a user, could request signatures from each user and try to link them to each other and, if successful, break anonymity.

We first introduce some notation. We let $\sigma(\mathsf{u}_1, f_1, \mathsf{cre}_1(f_1), y_1)$ denote a signature produced by platform u_1 using TPM secret value f_1. This signature is on a credential cre_1 which was issued using issuer secret value y_1 on f_1. We note that if two credentials are the same then this means they will have the same value of f with overwhelming probability for sufficiently large q but two credentials computed with the same value of f may not be the same. In each case we assume any pair of signatures either have different basenames or both have $\mathsf{bsn} = \perp$ since each such pair is intended to be unlinkable.

To prove unlinkability we consider pairs of signatures produced by the same platform and intended to by unlinkable in the real system. For two valid signatures we argue a randomly chosen pair of signatures $\big(\sigma_1(\mathsf{u}_1, f_1, \mathsf{cre}_1(f_1), y_1),\ \sigma_2(\mathsf{u}_1, f_1, \mathsf{cre}_1(f_1), y_1)\big)$ are unlinkable from the point of view of the environment when they are supposed to be if this pair is indistinguishable from a randomly chosen pair of the form $\big(\sigma_3(\mathsf{u}_1, f_1, \mathsf{cre}_1(f_1), y_1),\ \sigma_4(\mathsf{u}_2, f_2, \mathsf{cre}_2(f_2), y_1)\big)$. In other words we argue that the distributions $\big(\sigma_1(\mathsf{u}_1, f_1, \mathsf{cre}_1(f_1), y_1),\ \sigma_2(\mathsf{u}_1, f_1, \mathsf{cre}_1(f_1), y_1)\big)$ and $\big(\sigma_3(\mathsf{u}_1, f_1, \mathsf{cre}_1(f_1), y_1),\ \sigma_4(\mathsf{u}_2, f_2, \mathsf{cre}_2(f_2), y_1)\big)$ are computationally indistinguishable[1]. Note that we only argue indistinguishability for signatures produced with credentials from the *same issuer* since both the adversary and environment can trivially tell apart signatures on credentials from different issuers by running the verification algorithm with different issuer public keys; here we argue that this is all they can learn.

Here we assume it is the environment that is trying to distinguish the distributions. With this in mind we do not allow the environment to see the underlying credentials or the issuer secret values.

we also assume the party that produced the pair of signatures we want to show are unlinkable σ_1 and σ_2, regardless its corruption state, did this in the the correct manner; for known values of f. We argue in 3 stages using a sequence of distributions to show distributions of such pairs are computationally indistinguishable from the target distribution. At each stage we change one component used to produce one of the signatures in a given pair that form a distribution then argue this is computationally indistinguishable from the previous distribution in the sequence. In each step below we underline the changed variable.

1. We first argue that

$$\big(\sigma_1(\mathsf{u}_1, f_1, \mathsf{cre}(f_1), y_1), \sigma_2(\mathsf{u}_1, f_1, \mathsf{cre}(f_1), y_1)\big) \overset{c}{\approx} \big(\sigma_3(\mathsf{u}_1, f_1, \mathsf{cre}(f_1), y_1), \sigma_4(\mathsf{u}_1, \underline{f_2}, \mathsf{cre}(f_1), y_1)\big).$$

   Firstly, we show the signature σ_4 can be produced by the simulator using its powers over the random oracle. The simulator, using a valid credential on f_1, computes J then $K = f_2 J$ and selects s and c at random. It then computes H as $(s - cf_1)F$ and τ as $\hat{t}(H, X)$ where F can be obtained from the randomised credential used in the signature. The simulator then computes L as $sJ - cK$. The simulator can then patch the random oracle for h_5 so that the required c value is output and hence the signature will verify.
   For the first distribution we will have $J_1 \neq J_2$ with overwhelming probability from the assumptions we make on the choice of bsn in each signature. These signatures will contain values $(J_1, K_1 = f_1 J_1)$ and $(J_2, K_2 = f_1 J_2)$. For some value of $\alpha \in \mathbb{Z}_q$ we will have that $J_2 = \alpha J_1$ and hence this distribution will contain a Diffie-Hellman tuple $(f_1 J_1 = K_1, \alpha J_1 = J_2, \alpha f_1 J_1 = K_2)$ in $\mathbb{G}_1$.

[1] We use the notation $\overset{c}{\approx}$ to denote that two distributions are computationally indistinguishable.

For the second distribution we will not have such a Diffie-Hellman tuple since $f_1 \neq f_2$. This is the only difference between the distributions. Hence if the environment can distinguish these two distributions then it can solve the DDH in $\mathbb{G}_1$.

2. Next we argue that

$$\big(\sigma_1(\mathtt{u}_1, f_1, \mathsf{cre}(f_1), y_1), \sigma_2(\mathtt{u}_1, f_2, \mathsf{cre}(f_1), y_1)\big) \overset{c}{\approx} \big(\sigma_3(\mathtt{u}_1, f_1, \mathsf{cre}(f_1), y_1), \sigma_4(\mathtt{u}_1, f_2, \mathsf{cre}(\underline{f_2}), y_1)\big).$$

In this case we have already shown that σ_2 can be produced by the simulator. We do not need to show the signature σ_4 can be can be produced by the simulator since this sort of signature can be validly produced by any party with knowledge of f_2.
We let $\mathsf{cre}_i = (A_i, B_i, C_i)$ be the credential used in the computation of σ_i. We let (E_i, G_i) denote the signature components of σ_i. For the first pair of signature we will have $E_1 \neq E_2$ with overwhelming probability.
These signatures will contain values $(E_1, G_1 = \alpha E_1)$ since $G_1 = t_1 C_1 = (1 + y_1 f_1) x t_1 A = \alpha E_1$ where $\alpha = x(1 + y_1 f_1)$. We will also have $(E_2, G_2 = \alpha E_2)$ since f_1 is used for the second signature also. For some value of $\beta \in \mathbb{Z}_q$ we will have that $E_2 = \beta E_1$ and hence within these two signatures we will have a Diffie-Hellman tuple $(\alpha E_1 = G_1, \beta E_1 = E_2, \alpha\beta E_1 = G_2)$ in $\mathbb{G}_1$.
The second pair of signatures have one signature on a credential produced with a different value of f. In this case the α value in the two signatures will be the same with only negligible probability because of this. The same tuple (G_3, E_4, G_4) is then not a valid Diffie-Hellman tuple. This is the only difference between the two distributions. Notice also that in the first distribution the pair of signatures may be issued on different credentials. They will form a valid Diffie-Hellman tuple so long as the issuer secret values are the same and the f value used to issue the credential is also the same. Hence if the environment can distinguish these two distributions then it can solve the DDH in $\mathbb{G}_1$.
Notice also that this argument covers the case of changing the value of x used by the issuer and so we do not argue this separately.

3. Finally we argue that

$$\big(\sigma_1(\mathtt{u}_1, f_1, \mathsf{cre}(f_1), y_1), \sigma_2(\mathtt{u}_1, f_2, \mathsf{cre}(f_2), y_1)\big) \overset{c}{\approx} \big(\sigma_3(\mathtt{u}_1, f_1, \mathsf{cre}(f_1), y_1), \sigma_4(\underline{\mathtt{u}_2}, f_2, \mathsf{cre}(f_2), y_1)\big).$$

This is trivially true since each of the pairs of signatures are computed in the same way; with different values of f. Neither will have valid Diffie-Hellman tuples.

As a result we conclude

$$\big(\sigma_1(\mathtt{u}_1, f_1, \mathsf{cre}(f_1), y_1), \sigma_2(\mathtt{u}_1, f_1, \mathsf{cre}(f_1), y_1)\big) \overset{c}{\approx} \big(\sigma_3(\mathtt{u}_1, f_1, \mathsf{cre}(f_1), y_1), \sigma_4(\mathtt{u}_2, f_2, \mathsf{cre}(f_2), y_1)\big)$$

which completes the proof.

References

1. M. Backes, M. Maffei and D. Unruh. Zero knowledge in the applied Pi–calculus and automated verification of the direct anonymous attestation protocol. In *IEEE Symposium on Security and Privacy – SSP'08*, 202–215, 2008.
2. S. Balfe, A. D. Lakhani and K. G. Paterson. Securing peer-to-peer networks using trusted computing. In C. Mitchell, editor, *Chapter 10 of Trusted Computing.* IEEE London, 271–298, 2005.
3. P.S.L.M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography – SAC 2005*, Springer-Verlag LNCS 3897, 319–331, 2006.
4. D. Boneh, E. Brickell, L. Chen and H. Shacham. Set signatures. Manuscript, 2003.
5. D. Boneh, E. Brickell and H. Shacham. Set signatures. Manuscript, 2003.
6. E. Brickell. An efficient protocol for anonymously proving assurance of the container of a private key. Submission to the Trusted Computing Group, 2003.
7. E. Brickell, J. Camenisch and L. Chen. Direct anonymous attestation. *Proceedings of the 11th ACM Conference on Computer and Communications Security.* ACM Press, 132–145, 2004.
8. E. Brickell, J. Camenisch and L. Chen. Direct anonymous attestation in context. In C. Mitchell, editor, *Chapter 5 of Trusted Computing.* IEEE London, 143–174, 2005.

9. E. Brickell, L. Chen and J. Li. Simplified security notions for direct anonymous attestation and a concrete scheme from pairings. *Cryptology ePrint Archive.* Report 2008/104, available at `http://eprint.iacr.org/2008/104`.
10. E. Brickell, L. Chen and J. Li. A new direct anonymous attestation scheme from bilinear maps. In *Trusted Computing - Challenges and Applications – TRUST 2008*, Springer-Verlag LNCS 4968, 166–178, 2008.
11. E. Brickell and J. Li. Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. In *Proceedings of the 6th ACM Workshop on Privacy in the Electronic Society (WPES 2007).* ACM Press, 21–30, 2007.
12. J. Camenisch and J. Groth. Group signatures: Better efficiency and new theoretical aspects. In *Security in Communication Networks – SCN 2004.* Springer-Verlag LNCS 3352, 122–135, 2004.
13. J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In *Security in Communications Networks – SCN 2002*, Springer-Verlag LNCS 2576, 268–289, 2003.
14. J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *Advances in Cryptology – CRYPTO 2004*, Springer-Verlag LNCS 3152, 56–72, 2004.
15. J. Camenisch and M. Michels. A group signature scheme based on an RSA-variant. Technical Report RS-98-27, BRICS, University of Aarhus, 1998.
16. L. Chen. A scheme of group signatures with revocation evidence - a proposal for TCG. Submitted to the Trusted Computing Group in May 2003.
17. L. Chen, P. Morrissey and N. P. Smart. Pairings in trusted computing. In *Pairings in Cryptography – Pairing 2008*, Springer-Verlag LNCS 5209, 1–17, 2008.
18. L. Chen, P. Morrissey and N. P. Smart. On proofs of security of DAA schemes. In *Provable Security – ProvSec 2008*, Springer-Verlag LNCS 5324, 167–175, 2008.
19. L. Chen, Z. Cheng and N.P. Smart. Identity-based key agreement protocols from pairings. *Int. Journal of Information Security*, **6**, 213–242, 2007.
20. S. Galbraith, K. Paterson and N.P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, **156**, 3113–3121, 2008.
21. H. Ge and S.R. Tate. A Direct Anonymous Attestation Scheme for Embedded Devices. *Proceedings of Public Key Cryptography – PKC 2007*, Springer-Verlag LNCS 4450, 2007.
22. F. Hess, N.P. Smart and F. Vercauteren. The Eta pairing revisited. *IEEE Transactions on Information Theory*, **52**, 4595–4602, 2006.
23. A. Leung and C. J. Mitchell. Ninja: Non-identity based, privacy preserving authentication for ubiquitous environments. In *Ubiquitous Computing*, Springer-Verlag LNCS 4717, 73–90, 2007.
24. A. Lysyanskaya, R. Rivest, A. Sahai and S. Wolf. Pseudonym systems. In *Selected Areas in Cryptography – SAC 1999*, Springer-Verlag LNCS 1758, 184–199, 1999.
25. B. Smyth, L. Chen and M. Ryan. Direct Anonymous Attestation (DAA): Ensuring privacy with corrupt administrators. In *Security and Privacy in Ad hoc and Sensor Networks – ESAS 2007*, Springer-Verlag LNCS 4572, 218–231, 2007.
26. Trusted Computing Group. TCG TPM specification 1.2. Available at `http://www.trustedcomputinggroup.org`, 2003.