Direct Anonymous Attestation

Ernie Brickell
Intel Corporation
ernie.brickell@intel.com

Jan Camenisch
IBM Research
jca@zurich.ibm.com

Liqun Chen
HP Laboratories
liqun.chen@hp.com

February 11, 2004

Abstract

This paper describes the direct anonymous attestation scheme (DAA). This scheme was adopted by the Trusted Computing Group as the method for remote authentication of a hardware module, called trusted platform module (TPM), while preserving the privacy of the user of the platform that contains the module. Direct anonymous attestation can be seen as a group signature without the feature that a signature can be opened, i.e., the anonymity is not revocable. Moreover, DAA allows for pseudonyms, i.e., for each signature a user (in agreement with the recipient of the signature) can decide whether or not the signature should be linkable to another signature. DAA furthermore allows for detection of "known" keys: if the DAA secret keys are extracted from a TPM and published, a verifier can detect that a signature was produced using these secret keys. The scheme is provably secure in the random oracle model under the strong RSA and the decisional Diffie-Hellman assumption.

1 Introduction

Consider a trusted hardware module, called the trusted platform module (TPM) in the following, that is integrated into a platform such as a laptop or a mobile phone. Assume that the user of such a platform communicates with a verifier who wants to be assured that the user indeed uses a platform containing such a trusted hardware module, i.e., the verifier wants the TPM to authenticate itself. However, the user wants her privacy protected and therefore requires that the verifier only learns that she uses a TPM but not which particular one – otherwise all her transactions would become linkable to each other. This problem arose in the context of the Trusted Computing Group (TCG). TCG is an industry standardization body that aims to develop and promote an open industry standard for trusted computing hardware and software building blocks to enable more secure data storage, online business practices, and online commerce transactions while protecting privacy and individual rights (cf. [35]). TCG is the successor organization of the Trusted Computing Platform Alliance (TCPA).

In principle, the problem just described could be solved using any standard public key authentication scheme (or signature scheme): One would generate a secret/public key pair, and then embed the secret key into each TPM. The verifier and the TPM would then run the authentication protocol. Because all TPMs use the same key, they are indistinguishable. However, this approach would never work in practice: as soon as one hardware module (TPM) gets compromised and the secret key extracted and published, verifiers can no longer distinguish between real TPMs and fake ones. Therefore, detection of rogue TPMs needs to be a further requirement.

The solution first developed by TCG uses a trusted third party, the so-called privacy certification authority (Privacy CA), and works as follows [33]. Each TPM generates an RSA key pair called an Endorsement Key (EK). The Privacy CA is assumed to know the Endorsement Keys of all (valid) TPMs.

Now, when a TPM needs to authenticate itself to a verifier, it generates a second RSA key pair called an Attestation Identity Key (AIK), sends the AIK public key to the Privacy CA, and authenticates this public key w.r.t. the EK. The Privacy CA will check whether it finds the EK in its list and, if so, issues a certificate on the TPM's AIK. The TPM can then forward this certificate to the verifier and authenticate itself w.r.t. this AIK. In this solution, there are two possibilities to detect a rogue TPM: 1) If the EK secret key was extracted from a TPM, distributed, and then detected and announced as a rogue secret key, the Privacy CA can compute the corresponding public key and remove it from its list of valid Endorsement Keys. 2) If the Privacy CA gets many requests that are authorized using the same Endorsement Key, it might want to reject these requests. The exact threshold on requests that are allowed before a TPM is tagged rogue depends of course on the actual environment and applications, and will in practise probably be determined by some risk-management policy.

This solutions has the obvious drawback that the Privacy CA needs to be involved in every transaction and thus highly available on the one hand but still as secure as an ordinary certification authority that normally operates off-line on the other hand. Moreover, if the Privacy CA and the verifier collude, or the Privacy CA's transaction records are revealed to the verifier by some other means, the verifier will still be able to uniquely identify a TPM.

In this paper, we describe a better solution that was adopted by TCG in the new specification of the TPM [34]. It draws on techniques that have been developed for group signatures [20, 13, 1], identity escrow [27], and credential systems [16, 6]. In fact, our scheme can be seen as a group signature scheme without the capability to open signatures (or anonymity revocation) but with a mechanism to detect rogue members (TPMs in our case). More precisely, we also employ a suitable signature scheme to issue certificates on a membership public key generated by a TPM. Then, to authenticate as a group member, or valid TPM, a TPM proves that it possesses a certificate on a public key for which it also knows the secret key. To allow a verifier to detect rogue TPMs, the TPM is further required to reveal and prove correct of a value $N_V = \zeta^f$, where f is its secret key and ζ is a generator of an algebraic group where computing discrete logarithms is infeasible. As in the Privacy-CA solution, there are two possibilities for the verifier to detect a rogue TPM: 1) By comparing N_V with $\zeta^{\tilde{f}}$ for all $\tilde{f}$'s that are known to stem from rogue TPMs. 2) By detecting whether he has seen the same N_V too many times. Of course, the second method only works if the same ζ is used many times. However, ζ should not be a fixed system parameter as otherwise the user gains almost no privacy. Instead, ζ should either be randomly chosen by the TPM each time when it authenticates itself or every verifier should use a different ζ and change it with some frequency. Whether a verifier allows a TPM to choose a random base ζ and, if not, how often a verifier changes its ζ is again a question of risk management and policies and is outside the scope of this paper. However, we assume in the following that in case ζ is not random, it is derived from the verifier's name, e.g., using an appropriate hash function.

Because the TPM is a small chip with limited resources, a requirement for direct anonymous attestation was that the operations carried out on the TPM be minimal and, if possible, be outsourced to (software that is run on) the TPM's host. Of course, security must be maintained, i.e., a (corrupted) host/software should not be able to authenticate without interacting with the TPM. However, privacy/anonymity needs only be guaranteed if the host/software is not corrupted: as the host controls all the communication of the TPM to the outside, a corrupted host/software can always break privacy/anonymity by just adding some identifier to each message sent by the TPM. In fact, our scheme satisfies an even stronger requirement: when the corrupted software is removed, the privacy/anonymity properties are restored.

As our scheme employs the Camenisch-Lysyanskaya signature scheme [7] and the respective discrete logarithms based proofs to prove possession of a certificate, unforgeability of certificates holds under the strong RSA assumption and privacy and anonymity is guaranteed under the decisional Diffie-Hellman assumption. Furthermore, we use the Fiat-Shamir heuristic to turn proofs into signatures and thus our security proofs also assume random oracles.

As already mentioned, our setting shares many properties with the one of group signatures [20, 13, 1], identity escrow [27], and credential systems [16, 6] and we employ many techniques [1, 6, 7] used in these schemes. However, unlike those schemes, the privacy/anonymity properties do not require that the issuer uses so-called safe primes. We achieve this by a special sub-protocol when issuing credentials. This rids us of the necessity that the issuer proves that his RSA modulus is a safe-prime product which makes the setup of those schemes rather inefficient.

Finally, Brickell's direct proof method [4] addresses the same problem as we do. Its security is based on the Bounded Decision Diffie-Hellman assumption and a new assumption called the *Interval RSA* assumption stating that given e and n it is hard to find a pair (x, y) such that $x \equiv y^e \pmod{n}$ and x lies in some small specific interval. Besides this non-standard assumption, the direct proof methods is far less efficient than ours (it uses cut-and-choose proofs) and does not provide security against corrupted TPM's when issuing certificates, i.e., the signature scheme used to issue certificates is not shown to be secure against adaptive chosen message attacks.

2 Formal Specification of Direct Anonymous Attestation and Security Model

This section provides the formal model of direct anonymous attestation (DAA). As in [6], we use an ideal-system/real-system model to prove security based on security models for multi-party computation [14, 15] and reactive systems [30, 31].

We summarize the ideas underlying these models. In the real system there are a number of players, who run some cryptographic protocols with each other, an adversary $\mathcal{A}$, who controls some of the players, and an environment $\mathcal{E}$ that 1) provides the player $\mathcal{U}_i$ with inputs and 2) arbitrarily interacts with $\mathcal{A}$. The environment provides the inputs to the honest players and receives their outputs and interacts arbitrarily with the adversary. The dishonest players are subsumed into the adversary.

In the ideal system, we have the same players. However, they do not run any cryptographic protocol but send all their inputs to and receive all their outputs from an ideal all-trusted party $\mathcal{T}$. This party computes the output of the players from their inputs, i.e., applies the functionality that the cryptographic protocols are supposed to realize.

A cryptographic protocol is said to implement securely a functionality if for every adversary $\mathcal{A}$ and every environment $\mathcal{E}$ there exists a simulator $\mathcal{S}$ controlling the same parties in the ideal system as $\mathcal{A}$ does in the real system such that the environment can not distinguish whether it is run in the real system and interacts with $\mathcal{A}$ or whether it is run in the ideal system and interacts with the simulator $\mathcal{S}$.

We now specify the functionality of direct anonymous attestation. We distinguish the following kinds of players: the issuer $\mathcal{I}$, a trusted platform module (TPM) $\mathcal{M}_i$ with identity $\mathtt{id}_i$, a host $\mathcal{H}_i$ that has TPM $\mathcal{M}_i$ "built in," the rogue detection oracle $\mathcal{O}$ announcing which TPMs are rogue, and verifiers $\mathcal{V}_j$.

The ideal system all-trusted party $\mathcal{T}$ supports the following operations:

Setup: Each player indicates to $\mathcal{T}$ whether or not it is corrupted. Each TPM $\mathcal{M}_i$ sends its unique identity $\mathtt{id}_i$ to $\mathcal{T}$ who forwards it to the respective host $\mathcal{H}_i$.

Join: The host $\mathcal{H}_i$ contacts $\mathcal{T}$ and requests to become a member w.r.t. to a counter value `cnt`. Thus $\mathcal{T}$ sends the corresponding TPM $\mathcal{M}_i$ the counter value `cnt` and asks it whether it wants to become a member w.r.t. counter value `cnt`. Then, $\mathcal{T}$ asks the issuer $\mathcal{I}$ whether the platform with identity `id` and counter value `cnt` can become a member. If $\mathcal{M}_i$ was tagged rogue w.r.t. some counter value, $\mathcal{T}$ also tell $\mathcal{I}$ this. If the issuer agrees, $\mathcal{T}$ notifies $\mathcal{H}_i$ that it has become a member.

DAA-Sign/Verify: A host $\mathcal{H}_i$ wants to sign a message m with respect to some basename $\mathtt{bsn} \in \{0,1\}^* \cup \{\perp\}$ and some counter value `cnt` for some verifier $\mathcal{V}_j$. So $\mathcal{H}_i$ sends m, `bsn` and `cnt` to $\mathcal{T}$. If $\mathcal{H}_i/\mathcal{M}_i$

are not a member w.r.t. `cnt`, then $\mathcal{T}$ denies the request. Otherwise, $\mathcal{T}$ forwards m and `cnt` to the corresponding $\mathcal{M}_i$ and asks it whether it wants to sign. If it does, $\mathcal{T}$ tells $\mathcal{H}_i$ that $\mathcal{M}_i$ agrees and asks it w.r.t. which basename `bsn` it wants to sign (or whether it wants to abort). If $\mathcal{H}_i$ does not abort, $\mathcal{T}$ proceeds as follows

- If $\mathcal{M}_i$ has been tagged rogue w.r.t. `cnt`, $\mathcal{T}$ lets $\mathcal{V}_j$ know that a rogue TPM has signed m.
- If $\texttt{bsn} = \perp$ then $\mathcal{T}$ informs $\mathcal{V}_j$ that m has been signed w.r.t. `bsn`.
- If $\texttt{bsn} \neq \perp$ then $\mathcal{T}$ checks whether $\mathcal{H}_i/\mathcal{M}_i$ have already signed a message w.r.t. `bsn` and `cnt`. If this is the case, $\mathcal{T}$ looks up the corresponding pseudonym P in its database; otherwise $\mathcal{T}$ chooses a new random pseudonym $P \in_R \{0,1\}^{\ell_\sigma}$ (the quantity ℓ_σ is a security parameter). Finally, $\mathcal{T}$ informs $\mathcal{V}_j$ that the platform with pseudonym P has signed m.

Rogue Tagging: $\mathcal{O}$ tells $\mathcal{T}$ to tag of the platform with identity `id` w.r.t. `cnt` as a rogue. If the TPM with identity `id` is not corrupted, $\mathcal{T}$ denies the request. Otherwise, $\mathcal{T}$ marks the TPM with identity `id` as rogue w.r.t. counter value `cnt`.

Let us discuss our model. Note that the ideal system model captures both unforgeability and anonymity/pseudonymity. More precisely, a signature can only be produced with the involvement of a TPM that is a member and is not tagged rogue. Furthermore, signatures involving the same TPM w.r.t. the same basename are linkable to each other via a pseudonym P, but if they are done w.r.t. different basenames or no basename then they cannot be linked. (If a signature is done w.r.t. a basename we have $\texttt{bsn} \in \{0,1\}^*$ and if its done w.r.t. no basename we set $\texttt{bsn} = \perp$.) These two properties hold, regardless of whether or not the corresponding host is corrupted. Anonymity/pseudonymity is only guaranteed if both the host and the TPM are honest, as a dishonest party can always announce its identity and the messages it signed.

The way we have modeled rogue-tagging of a TPM has no effect on old messages but only causes verifiers to reject messages signed by a TPM that is tagged rogue. In the cryptographic protocol, however, an (honest) verifier would in principle be able to identify the messages a rogue TPM signed before it gets tagged as rogue. For simplicity, we do not model this and thus an honest verifier in the real system will not consider this information.

In the model, we assumed that no two verifiers use the same basename `bsn`. It they do, we consider them to be the same entity. But of course, a verifier can use several different basenames.

We also assume that TPM has a unique identity `id` with respect to which is can identify itself. However, the issuer can always add new hosts/TPMs to the system and hence the number of hosts/TPMs is not fixed. As all these hosts/TPMs have some external identifier, we do not model such additions but just assume that there is always another host/TPM if we need one. Also, in real life, the issuer will decide whether or not a given TPM is allowed to become a member based upon some policy (e.g., the issuer might only allow a specific TPM to join w.r.t., e.g., ten different counter values) and based upon a list of the identities of the valid TPMs.

In our security proofs we make two assumptions about the adversary: First, we assume that the rogue-tagging oracle $\mathcal{O}$ is always controlled by the adversary. The rationale behind this is that the rogue-tagging oracle models the case where a rogue hardware module from a platform is found, its keys are extracted and published on a rogue list. Thus the adversary "controls" this oracle as the adversary can decide when such a rogue platform is found. Note that this assumption strengthens the model. Second, we will not consider corrupted TPMs embedded into honest hosts. The reason is that we assume that at some point all platforms, i.e., hosts and TPM are genuine. Once the platforms get shipped, they might get compromised. As it is easier to compromise a platform than the TPM, we assume that whenever a TPM is corrupted, then so is the platform.

3 Preliminaries

3.1 Notation

Let $\{0,1\}^\ell$ denote the set of all binary strings of length ℓ. We often switch between integers and their representation as binary strings, e.g., we write $\{0,1\}^\ell$ for the set $[0, 2^\ell - 1]$ of integers. Moreover, we often use $\pm\{0,1\}^\ell$ to denote the set $[-2^\ell + 1, 2^\ell - 1]$.

We need some notation to select the high and low order bits of an integer. Let $\mathtt{LSB}_u(x) := x - 2^u \lfloor \frac{x}{2^u} \rfloor$ and $\mathtt{CAR}_u(x) := \lfloor \frac{x}{2^u} \rfloor$. Let $(x_k \dots x_0)_b$ denote the binary representation of $x = \sum_{i=0}^{k} 2^i x_i$, e.g., $(1001)_b$ is the binary representation of the integer 9. Then $\mathtt{LSB}_u(x)$ is the integer corresponding to the u least signification bits of (the binary representation of) x, e.g., $\mathtt{LSB}_4(57) = \mathtt{LSB}_4((111001)_b) = (1001)_b = 9$, and $\mathtt{CAR}_u(x)$ is the integer obtained by taking the binary representation of x and right-shifting it by u bits (and cutting of those bits), e.g., $\mathtt{CAR}_4(57) = \mathtt{CAR}_4((111001)_b) = (11)_b = 3$. Also note that $x = \mathtt{LSB}_u(x) + 2^u \mathtt{CAR}_u(x)$.

3.2 Protocols to Prove Knowledge of and Relations among Discrete Logarithms

In our scheme we will use various protocols to prove knowledge of and relations among discrete logarithms. To describe these protocols, we use notation introduced by Camenisch and Stadler [13] for various proofs of knowledge of discrete logarithms and proofs of the validity of statements about discrete logarithms. For instance,

$$PK\{(\alpha, \beta, \gamma) : y = g^\alpha h^\beta \ \wedge \ \tilde{y} = \tilde{g}^\alpha \tilde{h}^\gamma \ \wedge \ (u \leq \alpha \leq v)\}$$

denotes a "*zero-knowledge* P*roof of* K*nowledge of integers* α*,* β*, and* γ *such that* $y = g^\alpha h^\beta$ *and* $\tilde{y} = \tilde{g}^\alpha \tilde{h}^\gamma$ *holds, where* $v < \alpha < u$," where $y, g, h, \tilde{y}, \tilde{g}$, and $\tilde{h}$ are elements of some groups $G = \langle g \rangle = \langle h \rangle$ and $\tilde{G} = \langle \tilde{g} \rangle = \langle \tilde{h} \rangle$. The convention is that Greek letters denote the quantities the knowledge of which is being proved, while all other parameters are known to the verifier. Using this notation, a proof protocol can be described by just pointing out its aim while hiding all details.

In the random oracle model, such protocols can be turned into signature schemes using the Fiat-Shamir heuristic [23, 32]. We use the notation $SPK\{(\alpha) : y = g^\alpha\}(m)$ to denote a signature obtained in this way.

Proof of knowledge of the discrete logarithm modulo a composite. In this paper we apply such *PK*'s and *SPK*'s to the group of quadratic residues modulo a composite n, i.e., $G = \mathrm{QR}_n$, where n is a safe-prime product. This choice for the underlying group has some consequences. First, the protocols are proofs of knowledge under the strong RSA assumption [24]. Second, the largest possible value of the challenge c must be smaller than the smallest factor of G's order. Third, soundness needs special attention in the case that the verifier is not equipped with the factorization of n because then deciding membership in QR_n is believed to be hard. Thus the prover needs to convince the verifier that the elements he presents are indeed quadratic residues, i.e., that the square roots of the presented elements exist. This can in principle be done with a protocol by Fiat and Shamir [23]. However, often it is sufficient to simply execute $PK\{(\alpha) : y^2 = (g^2)^\alpha\}$ or $PK\{(\alpha) : y = \pm g^\alpha\}$ instead of $PK\{(\alpha) : y = g^\alpha\}$. The quantity α is defined as $\log_{g^2} y^2$, which is the same as $\log_g y$ in case y is in QR_n.

Other proofs in a fixed group. A proof of knowledge of a representation of an element $y \in G$ with respect to several bases $z_1, \dots, z_v \in G$ [18] is denoted $PK\{(\alpha_1, \dots, \alpha_v) : y = z_1^{\alpha_1} \cdot \ldots \cdot z_v^{\alpha_v}\}$. A proof of equality of discrete logarithms of two group elements $y_1, y_2 \in G$ to the bases $g \in G$ and $h \in G$, respectively, [17, 19] is denoted $PK\{(\alpha) : y_1 = g^\alpha \wedge y_2 = h^\alpha\}$. Generalizations to prove equalities among representations of the elements $y_1, \dots, y_w \in G$ to bases $g_1, \dots, g_v \in G$ are straightforward [13].

A proof of knowledge of a discrete logarithm of $y \in G$ with respect to $g \in G$ such that $\log_g y$ that lies in the integer interval $[a, b]$ is denoted by $PK\{(\alpha) : y = g^\alpha \ \wedge \ \alpha \in [a, b]\}$. Under the strong RSA assumption and if it is assured that the prover is not provided the factorization of the modulus (i.e., is not provided the order of the group) this proof can be done efficiently [3] (it compares to about six ordinary $PK\{(\alpha) : y = g^\alpha\}$.)

Proofs of knowledge of equality in different groups. The previous protocol can also be used to prove that the discrete logarithms of two group elements $y_1 \in G_1, y_2 \in G_1$ to the bases $g_1 \in G_1$ and $g_2 \in G_2$ in *different* groups G_1 and G_2 are equal [5, 11]. Let the order of the groups be q_1 and q_2, respectively. This proof can be realized only if both discrete logarithms lie in the interval $[0, \min\{q_1, q_2\}]$. The idea is that the prover commits to the discrete logarithm in some group, say $G = \langle g \rangle = \langle h \rangle$ the order of which he does not know, and then executes $PK\{(\alpha, \beta) : y_1 \stackrel{G_1}{=} g_1^\alpha \wedge y_2 \stackrel{G_2}{=} g_2^\alpha \wedge C \stackrel{G}{=} g^\alpha h^\beta \wedge \alpha \in [0, \min\{q_1, q_2\}]\}$, where C is the commitment. This protocol generalizes to several different groups, to representations, and to arbitrary modular relations.

3.3 Cryptographic Assumptions

We prove security under the strong RSA assumption and the decisional Diffie-Hellman assumption.

Assumption 1 (Strong RSA Assumption). *The strong RSA (SRSA) assumption states that it is computational infeasible, on input a random RSA modulus n and a random element $u \in \mathbb{Z}_n^*$, to compute values $e > 1$ and v such that $v^e \equiv u \pmod{n}$.*

The tuple (n, u) generated as above is called an *instance* of the *flexible* RSA problem.

Assumption 2 (DDH Assumption). *Let Γ be an ℓ_Γ-bit prime and ρ is an ℓ_ρ-bit prime such that $\rho | \Gamma - 1$. Let $\gamma \in \mathbb{Z}_\Gamma^*$ be an element of order ρ. Then,for sufficiently large values of ℓ_Γ and ℓ_ρ, the distribution $\{(\delta, \delta^a, \delta^b, \delta^{ab})\}$ is computationally indistinguishable from the distribution $\{(\delta, \delta^a, \delta^b, \delta^c)\}$, where δ is a random element from $\langle\gamma\rangle$, and a, b, and c are random elements from $[0, \rho - 1]$*

3.4 The Camenisch-Lysyanskaya Signature Scheme

The direct anonymous attestation scheme is based on the Camenisch-Lysyanskaya (CL) signature scheme [8, 29]. Unlike most signature schemes, this one is particularly suited for our purposes as it allows for efficient protocols to prove knowledge of a signature and to retrieve signatures on secret messages efficiently using discrete logarithm based proofs of knowledge [8, 29]. As we will use somewhat different (and also optimized) protocols for these tasks than those provided in [8, 29], we recall the signature scheme here and give an overview of discrete logarithm based proofs of knowledge in the following subsection.

Key generation. On input 1^k, choose a special RSA modulus $n = pq$, $p = 2p' + 1$, $q = 2q' + 1$. Choose, uniformly at random, $R_0, \ldots, R_{L-1}, S, Z \in \mathrm{QR}_n$. Output the public key $(n, R_0, \ldots, R_{L-1}, S, Z)$ and the secret key p. Let ℓ_n be the length of n.

Message space. Let ℓ_m be a parameter. The message space is the set $\{(m_0, \ldots, m_{L-1}) : m_i \in \pm\{0, 1\}^{\ell_m}\}$.

Signing algorithm. On input $m_0, \ldots, m_{L-1}$, choose a random prime number e of length $\ell_e > \ell_m + 2$, and a random number v of length $\ell_v = \ell_n + \ell_m + \ell_r$, where ℓ_r is a security parameter. Compute the value A such that $Z \equiv R_0^{m_0} \ldots R_{L-1}^{m_{L-1}} S^v A^e \pmod{n}$. The signature on the message $(m_0, \ldots, m_{L-1})$ consists of (e, A, v).

Verification algorithm. To verify that the tuple (e, A, v) is a signature on message $(m_0, \ldots, m_{L-1})$, check that $Z \equiv A^e R_0^{m_0} \ldots R_{L-1}^{m_{L-1}} S^v \pmod{n}$, and check that $2^{\ell_e} > e > 2^{\ell_e - 1}$.

Theorem 1 ([8]). *The signature scheme is secure against adaptive chosen message attacks [26] under the strong RSA assumption.*

The original scheme considered messages in the interval $[0, 2^{\ell_m} - 1]$. Here, however, we allow messages from $[-2^{\ell_m} + 1, 2^{\ell_m} - 1]$. The only consequence of this is that we need to require that $\ell_e > \ell_m + 2$ holds instead of $\ell_e > \ell_m + 1$.

4 The Direct Anonymous Attestation Scheme

High-Level Description. The basic idea underlying the direct anonymous attestation scheme is similar to the one of the Camenisch-Lysyanskaya anonymous credential system [6, 8, 29] : A trusted hardware module (TPM) chooses a secret "message" f, obtains a Camenisch-Lysyanskaya (CL) signature (aka attestation) on it from the issuer via a secure two-party protocol, and then can convince a verifier that it got attestation anonymously by a proof of knowledge of an attestation. To allow the verifier to recognize rogue TPMs, a TPM must also provide a pseudonym N_V and a proof that the pseudonym is formed correctly, i.e., that it is derived from the TPM's secret f contained in the attestation and a base determined by the verifier. We discuss different ways to handle rogue TPMs later.

Before providing the actual protocols, we first expand on the basic idea and explain some implementation details. First, we split the TPM's secret f into two ℓ_f-bit messages to be signed and denote the (secret) messages by f_0 and f_1 (instead of m_0 and m_1). This split allows for smaller primes e as their size depends on the size of the messages that get signed. Now, the two-party protocol to sign secret messages is as follows (cf. [8]). First, the TPM sends the issuer a commitment to the message-pair (f_0, f_1), i.e., $U := R_0^{f_0} R_1^{f_1} S^{v'}$, where v' is a value chosen randomly by the TPM to "blind" the f_i's. Also, the TPM computes $N_I := \zeta_I^{f_0 + f_1 2^{\ell_f}} \mod \Gamma$, where ζ_I is a quantity derived from the issuer's name, and sends U and N_I to the issuer. Next, the TPM convinces the issuer that U and N_I correctly formed (using a proof of knowledge a representation of U w.r.t. the bases R_0, R_1, S and N_I w.r.t. ζ_I) and that the f_i's lie in $\pm\{0,1\}^{\ell_f + \ell_{\mathcal{H}} + \ell_{\varnothing} + 2}$, where ℓ_f, $\ell_{\mathcal{H}}$, and $\ell_{\varnothing}$ are security parameters. This interval is larger than the one from which the f_i's actually stem because of the proof of knowledge we apply here. If the issuer accepts the proof, it compares N_I with previous such values obtained from this TPM to decide whether it wants to issue a certificate to TPM w.r.t. N_I or not. (The issuer might not want to grant too many credentials to the TPM w.r.t. different N_I, but should re-grant a credential to a N_I it has already accepted.) To issue a credential, the issuer chooses a random ℓ_v-bit integer v'' and a random ℓ_e-bit prime e, signs the hidden messages by computing $A := \left(\frac{Z}{US^{v''}}\right)^{1/e} \mod n$, and sends the TPM (A, e, v''). The issuer also proves to the TPM that she computed A correctly. The signature on (f_0, f_1) is then $(A, e, v := v' + v'')$, where v should be kept secret by the TPM (for f_0 and f_1 to remain hidden from the issuer), while A and e can be public.

A TPM can now prove that it has obtained attestation by proving that it got a CL-signature on some values f_0 and f_1. This can be done by a zero-knowledge proof of knowledge of values f_0, f_1, A, e, and v such that $A^e R_0^{f_0} R_1^{f_1} S^v \equiv Z \pmod{n}$. Also, the TPM computes $N_V := \zeta^{f_0 + f_1 2^{\ell_f}} \mod \Gamma$ and proves that the exponent here is related to those in the attestation, where $\zeta \in \langle\gamma\rangle$, i.e., the subgroup of $\mathbb{Z}_\Gamma^*$ of order ρ.

As mentioned in the introduction, the base ζ is either chosen randomly by the TPM or is generated from a basename value $\mathtt{bsn}_V$ provided by the verifier. The value N_V serves two purposes. The first one is rogue-tagging: If a rogue TPM is found, the values f_0 and f_1 are extracted and put on a blacklist. The verifier can then check N_V against this blacklist by comparing it with $\zeta^{\hat{f}_0 + \hat{f}_1 2^{\ell_f}}$ for all pairs $(\hat{f}_0, \hat{f}_1)$ on the black list. Note that i) the black list can be expected to be short, ii) the exponents $\hat{f}_0 + \hat{f}_1 2^{\ell_f}$ are small (e.g., 200-bits), and iii) batch-verification techniques can be applied [2]; so this check can be done efficiently. Also note that the blacklist need not be certified, e.g., by a certificate revocation

agency: whenever f_0, f_1, A, e, and v are discovered, they can be published and everyone can verify whether (A, e, v) is a valid signature on f_0 and f_1. The second purpose of N_V is its use as a pseudonym, i.e., if ζ is not chosen randomly by the TPM but generated from a basename then, whenever the same basename $\mathtt{bsn}_V$ is used, the TPM will provide the same value for N_V. This allows the verifier to link different transactions made by the same TPM while not identifying it, and to possibly reject a N_V if it appeared too many times. By defining how often a different basename is used (e.g., a different one per verifier and day), one obtains the full spectrum from full identification via pseudonymity to full anonymity. The way the basename is chosen,the frequency it is changed, and the threshold on how many times a particular N_V can appear before a verifier should reject it, is a question that depends on particular scenarios/applications and is outside of the scope of this document.

We finally note that, whenever possible, multiple proofs by a party are merged in to a single one. Furthermore, the Fiat-Shamir heuristic [23] is used to turn a proof into a "signature of knowledge".

As mentioned before, some operations of the TPM can be outsourced to the host in which the TPM is embedded. In particular, operation that are related to hiding the TPM's identity but not to the capability of producing a proof-signature of knowledge of an attestation can be preformed on the host. The rationale behind this is that the host can always reveal a TPM's identity.

Security Parameters. We employ the security parameters ℓ_n, ℓ_f, ℓ_e, ℓ'_e, ℓ_v, $\ell_\varnothing$, $\ell_{\mathcal{H}}$, ℓ_r, ℓ_Γ, and ℓ_ρ, where ℓ_n (2048) is the size of the RSA modulus, ℓ_f (104) is the size of the f_i's (information encoded into the certificate), ℓ_e (368) is the size of the e's (exponents, part of certificate), ℓ'_e (120) is the size of the interval the e's are chosen from, ℓ_v (2536) is the size of the v's (random value, part of certificate), $\ell_\varnothing$ (80) is the security parameter controlling the statistical zero-knowledge property, $\ell_{\mathcal{H}}$ (160) is the output length of the hash function used for the Fiat-Shamir heuristic, ℓ_r (80) is the security parameter needed for the reduction in the proof of security, ℓ_Γ (1632) is the size of the modulus Γ, and ℓ_ρ (208) is the size of the order ρ of the sub group of $\mathbb{Z}^*_\Gamma$ that is used for rogue-tagging (the numbers in parentheses are our proposal for these parameters). We require that:

$$\ell_e > \ell_\varnothing + \ell_{\mathcal{H}} + \max\{\ell_f + 4\ ,\ \ell'_e + 2\}\ , \qquad \ell_v > \ell_n + \ell_\varnothing + \ell_{\mathcal{H}} + \max\{\ell_f + \ell_r + 3\ ,\ \ell_\varnothing + 2\}\ , \qquad \ell_\rho = 2\ell_f$$

The parameters ℓ_Γ and ℓ_ρ should chosen such that the discrete logarithm problem in the subgroup of $\mathbb{Z}^*_\Gamma$ of order ρ with Γ and ρ being primes such that $2^{\ell_\rho} > \rho > 2^{\ell_\rho - 1}$ and $2^{\ell_\Gamma} > \Gamma > 2^{\ell_\Gamma - 1}$, has about the same difficulty as factoring ℓ_n-bit RSA moduli (see [28]).

Finally, let $\mathcal{H}$ be a collision resistant hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{\ell_{\mathcal{H}}}$.

Setup for Issuer. This section describes how the issuer chooses its public key and secret key. The key generation also produces a non-interactive proof (using the Fiat-Shamir heuristic) that the keys were chosen correctly. The latter will guarantee the security requirements of the host (resp., its user), i.e., that privacy and anonymity of signatures will hold.

1. The issuer chooses a RSA modulus $n = pq$ with $p = 2p' + 1$, $q = 2q' + 1$ such that p, p', q, q' are all primes and n has ℓ_n bits.

2. Furthermore, it chooses a random generator g' of QR_n (the group of quadratic residues modulo n).

3. Next, it chooses random integers $x_0, x_1, x_z, x_s, x_h, x_g \in [1, p'q']$ and computes

$$g := g'^{x_g} \bmod n \qquad h := g'^{x_h} \bmod n \qquad S := h^{x_s} \bmod n \tag{1}$$

$$Z := h^{x_z} \bmod n \qquad R_0 := S^{x_0} \bmod n \qquad R_1 := S^{x_1} \bmod n\ . \tag{2}$$

4. It produces a non-interactive proof *proof* that R_0, R_1, S, Z, g, and h are computed correctly, i.e., that $g, h \in \langle g' \rangle$, $S, Z \in \langle h \rangle$, and $R_0, R_1 \in \langle S \rangle$. We refer to Appendix A for the details of this proof.

5. It generates a group of prime order: Choose random primes ρ and Γ such that $\Gamma = r\rho + 1$ for some r with $\rho \nmid r$, $2^{\ell_\Gamma - 1} < \Gamma < 2^{\ell_\Gamma}$, and $2^{\ell_\rho - 1} < \rho < 2^{\ell_\rho}$. Choose a random $\gamma' \in_R \mathbb{Z}_\Gamma^*$ such that $\gamma'^{(\Gamma-1)/\rho} \not\equiv 1 \pmod{\Gamma}$ and set $\gamma := \gamma'^{(\Gamma-1)/\rho} \bmod \Gamma$.

6. Finally, it publishes the public key $(n, g', g, h, S, Z, R_0, R_1, \gamma, \Gamma, \rho)$ and the proof *proof* and stores $p'q'$ as its secret key.

Let $H_\Gamma(\cdot)$ and $H(\cdot)$ be two collision resistant hash functions $H_\Gamma(\cdot) : \{0,1\}^* \to \{0,1\}^{\ell_\Gamma + \ell_\varnothing}$ and $H(\cdot) : \{0,1\}^* \to \{0,1\}^{\ell_\mathcal{H}}$.

Verification of the Issuer's Public Key. An issuer's public key can verified as follows.

1. Verify the proof *proof* that $g, h \in \langle g' \rangle$, $S, Z \in \langle h \rangle$, and $R_0, R_1 \in \langle S \rangle$ as stated in Appendix A.

2. Check whether Γ and ρ are primes, $\rho \mid (\Gamma - 1)$, $\rho \nmid \frac{\Gamma-1}{\rho}$, and $\gamma^\rho \equiv 1 \pmod{\Gamma}$.

3. Check whether all public key parameter have the required length.

If R_0, R_1, S, Z, g, and h are not formed correctly, it could potentially mean that the security properties for the TPM/host do not hold. However, it is sufficient if the platform/host (i.e., owner of the TPM) verifies the proof that R_0, R_1, g, and h are computed correctly once. In principle, it is just sufficient if one representative of platform users checks this proof. Also, if γ does not generate a subgroup of $\mathbb{Z}_\Gamma^*$, the issuer could potentionally use this to link different signatures.

As we shall see, it is not important for the security of the platform (i.e., the anonymity/pseudonymity properties) that n is a product of two safe primes.

Join Protocol. Let $PK_I := (n, g', g, h, S, Z, R_0, R_1, \gamma, \Gamma, \rho)$ be the public key of the issuer and let PK'_I be a long-term public key of the issuer used to authenticate PK_I for the DAA. Let $\zeta_I \equiv (H_\Gamma(1 \| \mathtt{bsn}_I))^{(\Gamma-1)/\rho} \bmod \Gamma$, where $\mathtt{bsn}_I$ is the issuers long-term basename.

The input to the TPM is $(n, R_0, R_1, S, \rho, \Gamma)$.

We assume that the TPM and the issuer have established a one-way authentic channel, i.e., the issuer needs to be sure that it talks to the right TPM. Note that authenticity of the channel is enough, i.e., we do not require secrecy, in fact we even assume the host reads all messages and may choose not to forward some messages sent by the issuer to the TPM. In fact, it is sufficient that the value U be transmitted authentically from the TPM to the issuer. In setting of TCG, one can achieve this using the Endorsement Key (EK) pair was issued to the TPM (see Appendix B).

Let `DAAseed` be the seed to compute the f_0 and f_1 secret of the TPM. Also let `cnt` be the current value of the counter keeping track of the number of times the TPM has run the Join protocol (however, the TPM is allows to re-run the Join protocol with the same `cnt` value many times).

1. The TPM and host verify that PK_I is authenticated by PK'_I.

2. The host computes $\zeta_I := (H_\Gamma(1 \| \mathtt{bsn}_I))^{(\Gamma-1)/\rho} \bmod \Gamma$ and sends ζ_I to the TPM.

3. The TPM checks whether $\zeta_I^\rho \equiv 1 \pmod{\Gamma}$.

4. Let $i := \lfloor \frac{\ell_\rho + \ell_\varnothing}{\ell_\mathcal{H}} \rfloor$ (i will be 1 for values of the parameters selected in Section 4).
   The TPM computes

$$f := H(H(\texttt{DAAseed} \| H(PK'_I)) \| \texttt{cnt} \| 0) \| \ldots \| H(H(\texttt{DAAseed} \| H(PK'_I)) \| \texttt{cnt} \| i) \pmod{\rho}, \tag{3}$$

$$f_0 := \texttt{LSB}_{\ell_f}(f), \qquad f_1 := \texttt{CAR}_{\ell_f}(f) \ , \tag{4}$$

$$v' \in_R \{0,1\}^{\ell_n + \ell_\varnothing} \ , \tag{5}$$

$$U := R_0^{f_0} R_1^{f_1} S^{v'} \bmod n \ , \qquad N_I := \zeta_I^{f_0 + f_1 2^{\ell_f}} \bmod \Gamma \tag{6}$$

   and sends U and N_I to the issuer.

5. The issuer checks for all (f_0, f_1) on the rogue list whether $N_I \stackrel{?}{\neq} (\zeta^{f_0 + f_1 2^{\ell_f}}) \pmod{\Gamma}$. The issuer also checks this for the N_I this platform had used previously. If the issuer finds the platform to be rogue, it aborts the protocol.

6. The TPM proves to the issuer knowledge of f_0, f_1, and v': it executes as prover the protocol

$$SPK\{(f_0, f_1, v') : \quad U \equiv \pm R_0^{f_0} R_1^{f_1} S^{v'} \pmod{n} \ \wedge \ N_I \equiv \zeta_I^{f_0 + f_1 2^{\ell_f}} \pmod{\Gamma} \ \wedge \\ f_0, f_1 \in \{0,1\}^{\ell_f + \ell_\varnothing + \ell_\mathcal{H} + 2} \ \wedge \ v' \in \{0,1\}^{\ell_n + \ell_\varnothing + \ell_\mathcal{H} + 2}\}(n_t \| n_i)$$

   with the issuer as the verifier. This protocol is implemented as follows, where some non-critical operations are performed by the host and not be the TPM.

   (a) The TPM picks random integers $r_{f_0}, r_{f_1} \in_R \{0,1\}^{\ell_f + \ell_\varnothing + \ell_\mathcal{H}}$, $r_{v'} \in_R \{0,1\}^{\ell_n + 2\ell_\varnothing + \ell_\mathcal{H}}$, and computes $\tilde{U} := R_0^{r_{f_0}} R_1^{r_{f_1}} S^{r_{v'}} \bmod n$ and $\tilde{N}_I := \zeta_I^{r_{f_0} + r_{f_1} 2^{\ell_f}} \bmod \Gamma$, and sends $\tilde{U}$ and $\tilde{N}_I$ to the host.

   (b) The issuer chooses a random string $n_i \in \{0,1\}^{\ell_\mathcal{H}}$ and sends n_i to the host.

   (c) The host computes $c_h := H(n \| R_0 \| R_1 \| S \| U \| N_I \| \tilde{U} \| \tilde{N}_I \| n_i)$ and sends c_h to the TPM

   (d) The TPM chooses a random $n_t \in \{0,1\}^{\ell_\varnothing}$ and computes

$$c := H(c_h \| n_t) \in [0, 2^{\ell_\mathcal{H}} - 1] \ .$$

   (e) The TPM computes

$$s_{f_0} := r_{f_0} + c \cdot f_0 \ , \qquad s_{f_1} := r_{f_1} + c \cdot f_1 \ , \qquad s_{v'} := r_{v'} + c \cdot v' \ . \tag{7}$$

   (f) TPM sends $(c, n_t, s_{f_0}, s_{f_1}, s_{v'})$ to the host.

   (g) The host forwards $(c, n_t, s_{f_0}, s_{f_1}, s_{v'})$ to the issuer.

   (h) Issuer verifies the proof as follows. It computes

$$\hat{U} := U^{-c} R_0^{s_{f_0}} R_1^{s_{f_1}} S^{s_{v'}} \bmod n \quad \text{and} \quad \hat{N}_I := N_I^{-c} \zeta_I^{s_{f_0} + 2^{\ell_f} s_{f_1}} \bmod \Gamma \tag{8}$$

   and checks

$$c \stackrel{?}{=} H(H(n \| R_0 \| R_1 \| S \| U \| N_I \| \hat{U} \| \hat{N}_I \| n_i) \| n_t) \in [0, 2^{\ell_\mathcal{H}} - 1] \ , \tag{9}$$

$$s_{f_0}, s_{f_1} \stackrel{?}{\in} \{0,1\}^{\ell_f + \ell_\varnothing + \ell_\mathcal{H} + 1} \ , \qquad \text{and} \qquad s_{v'} \stackrel{?}{\in} \{0,1\}^{\ell_n + 2\ell_\varnothing + \ell_\mathcal{H} + 1} \ . \tag{10}$$

7. The issuer chooses $\hat{v} \in_R \{0,1\}^{\ell_v - 1}$ and a prime $e \in_R [2^{\ell_e - 1}, 2^{\ell_e - 1} + 2^{\ell'_e - 1}]$ and computes $v'' := \hat{v} + 2^{\ell_v - 1}$ and
$$A := \left(\frac{Z}{US^{v''}}\right)^{1/e} \bmod n \ .$$

8. To convince the host that A was correctly computed, the issuer as prover runs the protocol
$$SPK\{(d): \quad A \equiv \pm\left(\frac{Z}{US^{v''}}\right)^{d} \pmod{n}\}(n_h)$$
with the host:
   (a) The host chooses a random integer $n_h \in \{0,1\}^{\ell_\varnothing}$ and sends n_h to the issuer.
   (b) The issuer randomly chooses $r_e \in_R [0, p'q']$ and computes
$$\tilde{A} := \left(\frac{Z}{US^{v''}}\right)^{r_e} \bmod n \ , \quad c' := H(n \| Z \| S \| U \| v'' \| A \| \tilde{A} \| n_h) \ , \quad \text{and} \quad s_e := r_e - c'/e \bmod p'q' \tag{11}$$
   and sends c', s_e, and (A, e, v'') to the host.
   (c) The host verifies whether e is a prime and lies in $[2^{\ell_e - 1}, 2^{\ell_e - 1} + 2^{\ell'_e - 1}]$, computes $\hat{A} := A^{c'}\left(\frac{Z}{US^{v''}}\right)^{s_e} \bmod n$, and checks whether $c' \stackrel{?}{=} H(n \| Z \| S \| U \| v'' \| A \| \hat{A} \| n_h)$.

9. The host forwards v'' to the TPM.

10. The TPM receives v'', sets $v := v'' + v'$, and stores (f_0, f_1, v).

Our protocol differs from the one provided in [8] to sign a committed message mainly in that our protocol requires the issuer to prove that A lies in $\langle h \rangle$. The host can conclude that $A \in \langle h \rangle$ from the proof the issuer provides in Step 8, the fact that $A^e U S^{v''} \equiv Z \pmod{n}$, and the proofs the issuer provides as part of the setup that $S, R_0, R_1, Z \in \langle h \rangle$. We refer to the security proof of Theorem 2, Equations (39) and (40), for the details of this. The reason for requiring $A \in \langle h \rangle$ is to assure that later, in the signing algorithm, A can be statistically hidden in $\langle h \rangle$. Otherwise, an adversarial signer could compute A for instance as $b\left(\frac{Z}{US^{v''}}\right)^{1/e} \bmod n$ using some b such that $b^e = 1$ and $b \notin \langle h \rangle$. As a DAA-signature contains the value $T_1 = Ah^w$ for a random w (see DAA-signing protocol), and adversarial signer would be able to link T_1 to A, e.g., by testing $T_1 \in \langle h \rangle$. Prior schemes such as [1, 6, 8] have prevented this by ensuring that n is a safe-prime product and then made sure that all elements are cast into QR_n. However, proving that a modulus is a safe-prime product is rather inefficient [10] and hence the setup of these schemes is not really practical. We note that the proof in Step 8 is zero-knowledge only if the issuer has chosen n as a safe-prime product. This is a property that the issuer is interested in, and not the TPM, host, or users of a platform.

Because our security proof requires rewinding to extract f_0, f_1, and v' from an adversarial TPM, the join protocol can only be run sequentially, i.e., not in parallel with many TPMs. At some loss of efficiency, this drawback could be overcome for instance by using the verifiable encryption [12] of these values.

DAA-Signing Protocol. We now describe how a platform can prove that is got an anonymous attestation credential and at the same time authenticate a message. Thus, the platform gets as input a message m to DAA-sign. We remark that in some cases the message m will be generated by the TPM and might not be known to the host. That is, the TPM signs an Attestation Identity Key (AIK), an RSA public key that is has generated, which the TPM will later to sign its internal registers.

Let $n_v \in \{0,1\}^{\ell_{\mathcal{H}}}$ be a nonce and $\mathtt{bsn}_V$ a basename value provided by the verifier. Let b be a byte describing the use of the protocol, i.e., $b = 0$ means that the message m is generated by the TPM and $b = 1$ means that the message m was input to the TPM.

The input to the protocol for the TPM is m, $(n, R_0, R_1, S, \Gamma, \rho)$ and (f_0, f_1, v), and the host's input to the protocol is m, the certificate (A, e) and $(n, g, g', h,\ R_0, R_1, S, Z, \gamma, \Gamma, \rho)$.

The signing algorithm is as follows.

1. (a) Depending on the verifier's request (i.e., whether $\mathtt{bsn}_V \neq \perp$ or not), the host computes ζ as follows
$$\zeta \in_R \langle\gamma\rangle \qquad \text{or} \qquad \zeta := (H_\Gamma(1\|\mathtt{bsn}_V))^{(\Gamma-1)/\rho} \bmod \Gamma$$
and sends ζ to the TPM.
   (b) The TPM checks whether $\zeta^\rho \equiv 1 \pmod{\Gamma}$.
2. (a) The host picks random integers $w, r \in \{0,1\}^{\ell_n+\ell_\varnothing}$ and computes $T_1 := Ah^w \bmod n$ and $T_2 := g^w h^e (g')^r \bmod n$.
   (b) The TPM computes $N_V := \zeta^{f_0+f_1 2^{\ell_f}} \bmod \Gamma$ and sends N_V to the host.
3. Now, the TPM and host together produce a "signature of knowledge" that T_1 and T_2 commit to a certificate and N_V was computed using the secret key going with that certificate. That is, they compute the "signature of knowledge"

$$\begin{aligned} &SPK\{(f_0, f_1, v, e, w, r, ew, ee, er) : \\ &\quad Z \equiv \pm T_1^e R_0^{f_0} R_1^{f_1} S^v h^{-ew} \pmod{n} \ \wedge\ T_2 \equiv \pm g^w h^e g'^r \pmod{n} \ \wedge \\ &\quad 1 \equiv \pm T_2^{-e} g^{ew} h^{ee} g'^{er} \pmod{n} \ \wedge\ N_V \equiv \zeta^{f_0+f_1 2^{\ell_f}} \pmod{\Gamma} \ \wedge \\ &\quad\quad f_0, f_1 \in \{0,1\}^{\ell_f+\ell_\varnothing+\ell_{\mathcal{H}}+2} \ \wedge\ (e - 2^{\ell_e}) \in \{0,1\}^{\ell'_e+\ell_\varnothing+\ell_{\mathcal{H}}+1}\}(n_t\|n_v\|b\|m)\ . \end{aligned}$$

   Most of the secrets involved are actually known by the host; in fact only the values involving f_0, f_1, and v need to be computed by the TPM, as the reader can see below.

   (a) i. The TPM picks random integers
$$r_v \in_R \{0,1\}^{\ell_v+\ell_\varnothing+\ell_{\mathcal{H}}}, \qquad r_{f_0}, r_{f_1} \in_R \{0,1\}^{\ell_f+\ell_\varnothing+\ell_{\mathcal{H}}},$$
   and computes
$$\tilde{T}_{1t} := R_0^{r_{f_0}} R_1^{r_{f_1}} S^{r_v} \bmod n \qquad \tilde{r}_f := r_{f_0} + r_{f_1} 2^{\ell_f} \bmod \rho \qquad \tilde{N}_V := \zeta^{\tilde{r}_f} \bmod \Gamma\ .$$
   The TPM sends $\tilde{T}_{1t}$ and $\tilde{N}_V$ to the host.

   ii. The host picks random integers
$$\begin{aligned} r_e \in_R \{0,1\}^{\ell'_e+\ell_\varnothing+\ell_{\mathcal{H}}}\ , &\qquad r_{ee} \in_R \{0,1\}^{2\ell_e+\ell_\varnothing+\ell_{\mathcal{H}}+1}\ , \\ r_w, r_r \in_R \{0,1\}^{\ell_n+2\ell_\varnothing+\ell_{\mathcal{H}}}\ , &\qquad r_{ew}, r_{er} \in_R \{0,1\}^{\ell_e+\ell_n+2\ell_\varnothing+\ell_{\mathcal{H}}+1} \end{aligned}$$
   and computes
$$\tilde{T}_1 := \tilde{T}_{1t} T_1^{r_e} h^{-r_{ew}} \bmod n\ ,\quad \tilde{T}_2 := g^{r_w} h^{r_e} g'^{r_r} \bmod n\ ,\quad \tilde{T}'_2 := T_2^{-r_e} g^{r_{ew}} h^{r_{ee}} g'^{r_{er}} \bmod n\ . \tag{12}$$

   (b) i. Host computes
$$c_h := H((n\|g\|g'\|h\|R_0\|R_1\|S\|Z\|\gamma\|\Gamma\|\rho)\|\zeta\|(T_1\|T_2)\|N_V\|(\tilde{T}_1\|\tilde{T}_2\|\tilde{T}'_2)\|\tilde{N}_V)\|n_v) \in [0, 2^{\ell_{\mathcal{H}}}-1]\ .$$
   and sends c_h to the TPM.

ii. The TPM chooses a random $n_t \in \{0,1\}^{\ell_\varnothing}$ and computes

$$c := H(H(c_h \| n_t) \| b \| m) \in [0, 2^{\ell_\mathcal{H}} - 1] \ .$$

and sends c, n_t to the host.

(c) i. The TPM computes (over the integers)

$$s_v := r_v + c \cdot v \ , \qquad s_{f_0} := r_{f_0} + c \cdot f_0 \ , \quad \text{and} \qquad s_{f_1} := r_{f_1} + c \cdot f_1$$

and sends (s_v, s_{f_0}, s_{f_1}) to the host.

ii. The host computes (over the integers)

$$s_e := r_e + c \cdot (e - 2^{\ell_e - 1}) \ , \qquad s_{ee} := r_{ee} + c \cdot e^2 \ , \qquad s_w := r_w + c \cdot w \ ,$$
$$s_{ew} := r_{ew} + c \cdot w \cdot e \ , \qquad s_r := r_r + c \cdot r \ , \qquad s_{er} := r_{er} + c \cdot e \cdot r \ .$$

4. The host outputs the signature

$$\sigma := (\zeta, (T_1, T_2), N_V, c, n_t, (s_v, s_{f_0}, s_{f_1}, s_e, s_{ee}, s_w, s_{ew}, s_r, s_{er})) \ . \tag{13}$$

Let us give some intuition about why this signature should convince a verifier that N_V links to secrets that were certified. The first term in the *SPK* can be rewritten as $Z \equiv (\pm\frac{T_1}{h^{ew/e}})^e R_0^{f_0} R_1^{f_1} S^v \pmod n$, if e divides we (see proof of security). The second two terms show that e indeed divides we, so the rewriting works. Therefore, because the last terms that show that the f_i's and e satisfy the length requirements, $(\frac{T_1}{h^{ew/e}}, e, v)$ is a valid certificate for f_0 and f_1. The remaining term shows that N_V is based on the same f_0 and f_1.

The main difference of this DAA-signing protocol to the signature generation in prior schemes such as [1, 6, 8] is that here we have distributed the necessary operation to two parties, the TPM and the host. Basically, the TPM only produces the proof-signature $SPK\{(f_0, f_1, v) : (Z/A^e) \equiv R_0^{f_0} R_1^{f_1} S^v \bmod n \ \wedge \ N_V \equiv \zeta^{f_0 + f_1 2^{\ell_f}} \pmod \Gamma\}(n_t \| n_v \| b \| m)$, which the host then extends to the full DAA-signature (note that (Z/A^e) is known to the host). Note that the *SPK* produced by the TPM is not anonymous (even with a random ζ) as the value (Z/A^e) would fully identify the TPM. While it is intuitively obvious that the host cannot generate such signatures on its own or turn one by a TPM into one on a different message, we provide a formal proof of this in Section 5

Verification Algorithm. Verification of a signature

$$\sigma = (\zeta, (T_1, T_2), N_V, c, n_t, (s_v, s_{f_0}, s_{f_1}, s_e, s_{ee}, s_w, s_{ew}, s_r, s_{er})) \tag{14}$$

an a message m w.r.t. the public key $(n, g, g', h, R_0, R_1, S, Z, \gamma, \Gamma, \rho)$ is as follows

1. Compute

$$\hat{T}_1 := Z^{-c} T_1^{s_e + c2^{\ell_e - 1}} R_0^{s_{f_0}} R_1^{s_{f_1}} S^{s_v} h^{-s_{ew}} \bmod n \ , \qquad \hat{T}_2 := T_2^{-c} g^{s_w} h^{s_e + c2^{\ell_e - 1}} g'^{s_r} \bmod n \ , \tag{15}$$

$$\hat{T}_2' := T_2^{-(s_e + c2^{\ell_e - 1})} g^{s_{ew}} h^{s_{ee}} g'^{s_{er}} \bmod n \ , \quad \text{and} \qquad \hat{N}_V := N_V^{-c} \zeta^{s_{f_0} + s_{f_1} 2^{\ell_f}} \bmod \Gamma \ . \tag{16}$$

2. Verify that

$$c \stackrel{?}{=} H(H(H((n \| g \| g' \| h \| R_0 \| R_1 \| S \| Z \| \gamma \| \Gamma \| \rho) \| \zeta \| (T_1 \| T_2) \| N_V \| (\hat{T}_1 \| \hat{T}_2 \| \hat{T}_2') \| \hat{N}_V \| n_v) \| n_t) \| b \| m) \ , \tag{17}$$

$$N_V, \zeta \stackrel{?}{\in} \langle \gamma \rangle \ , \quad s_{f_0}, s_{f_1} \stackrel{?}{\in} \{0,1\}^{\ell_f + \ell_\varnothing + \ell_\mathcal{H} + 1} \ , \quad \text{and} \quad s_e \stackrel{?}{\in} \{0,1\}^{\ell_e' + \ell_\varnothing + \ell_\mathcal{H} + 1} \ . \tag{18}$$

3. If ζ was not chosen at random but derived from a verifier's basename, then check whether

$$\zeta \stackrel{?}{\equiv} (H_\Gamma(1\|\mathtt{bsn}_V))^{(\Gamma-1)/\rho} \pmod{\Gamma} \ .$$

4. For all (f_0, f_1) on the rogue list check whether $N_V \stackrel{?}{\neq} (\zeta^{f_0+f_1 2^{\ell_f}}) \pmod{\Gamma}$.

The check $N_V, \zeta \stackrel{?}{\in} \langle\gamma\rangle$ can be done by raising N_V and ζ to the order of γ (which is ρ) and checking whether the result is 1. In case ζ is random, one can apply so called batch verification techniques (cf. [2]) to obtain a considerable speed-up of the verification step 4. Also note that the involved exponents are relatively small. Finally, if ζ is not random, one could precompute $\zeta^{f_0+f_1 2^{\ell_f}}$ for all (f_0, f_1) on the rogue list.

On Rogue Tagging. When a certificate (A, e, v) and values f_0 and f_1 are found (e.g., on the Internet or embedded into some software), they should be distributed to all potential verifiers. These verifiers can then check whether $A^e R_0^{f_0} R_1^{f_1} S^v \equiv Z \pmod{n}$ holds and then put f_0 and f_1 on their list of rogue keys. Note that this does not involve a certificate revocation authority.

Inputs and Outputs of the Parties. To actually meet the specifiation of DAA we gave in Section 2, it remains to define the outputs of the parties in the real system. The hosts do not output anything. The issuer outputs the identity of a platform after it has joined; if a platform that has been tagged rogue tries to join the protocols, it outputs the identity of this platform and the relevant counter value. Finally, when a verifier sees a valid signature (this excludes signatures generated by a rogue TPM), it produces its output as follows: (1) If the signature was produced by a rogue TPM, it outputs m together with the note that m was signed by a rogue. (2) If the signature was produced w.r.t. $\mathtt{bsn}_V = \perp$, it just outputs the message. (3) If the signature was produced with a ζ derived from a basename (i.e., w.r.t. $\mathtt{bsn}_V \neq \perp$), it looks in its database whether it already assigned a pseudonym P to the pair (ζ, N_V); otherwise it chooses new random pseudonym $P \in_R \{0,1\}^{\ell_\sigma}$ and assigns it the the pair (the quantity ℓ_σ is a security parameter). In either case, it outputs P and m.

5 Security Proofs

The following theorem establishes the security of our scheme.

Theorem 2. *The protocols provided in Section 4 securely implement a secure direct anonymous attestation system under the decisional Diffie-Hellman assumption in $\langle\gamma\rangle$ and the strong RSA assumption.*

We refer to Appendix C for the actual security proofs and give here only a very high-level overview of the proof. The proof of Theorem 2 consists of the description of a simulator and arguments that the environment cannot distinguish whether it is run in the real system, interacting with $\mathcal{A}$ and the real parties, or in the ideal system, interacting with $\mathcal{S}$ and the ideal parties. Recall that the simulator interacts with $\mathcal{T}$ on behalf of the corrupted parties of the ideal system, and simulates the real-system adversary $\mathcal{A}$ towards the environment $\mathcal{E}$.

The simulator, which has black box access to the adversary, basically runs the real system protocol in the same way as an honest party would, apart from the DAA-signing protocol, where the simulator is just told by $\mathcal{T}$ that some party signed a message (possible w.r.t. a pseudonym) but it does not know which party signed. Thus the simulator just chooses a random N_V and then forges a signature by using the zero-knowledge simulator of the *SPK* proof and the power over the random oracle. Also, if the simulator notes that the adversary signed some message, it chooses some corrupted host and tells $\mathcal{T}$ that this host has signed on behalf of a party. The simulator will fail in cases where the adversary manages

to forge a signature, to sign on behalf of a honest TPM/host, or to tag an honest TPM as a rogue. We show that these cases cannot occur only under the strong RSA and the discrete logarithm assumption. We then show that if the simulator does not fail then, under the decisional Diffie-Hellman assumption, the environment will not be able to tell whether or not it is run in the real system interacting with the adversary or the ideal system interacting with the simulator.

6 Conclusion

The protocols we describe could be extended im many ways. For instance, only minor changes would be necessary to allow the issuer to use any RSA modulus instead of only safe-prime products. However, one would need to make a small order assumption [22]. Another extension would to guaranteed anonymity/pseudonymity to the host (in fact to its user) even if the TPM deviates from the protocol (cf. [19]). Finally, as we have already mentioned in Section 4, one could extend the join protocol in such a way that the system can also be proved secure if issuer runs the protocol concurrently with different TPMs.

References

[1] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A practical and provably secure coalition-resistant group signature scheme. In M. Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, volume 1880 of *LNCS*, pages 255–270. Springer Verlag, 2000.

[2] M. Bellare, J. A. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In K. Nyberg, editor, *Advances in Cryptology — EUROCRYPT '98*, volume 1403 of *LNCS*, pages 236–250. Springer Verlag, 1998.

[3] F. Boudot. Efficient proofs that a committed number lies in an interval. In B. Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 431–444. Springer Verlag, 2000.

[4] E. Brickell. An efficient protocol for anonymously providing assurance of the container of a private key. Submitted to the Trusted Computing Group, Apr. 2003.

[5] E. F. Brickell, D. Chaum, I. B. Damgård, and J. van de Graaf. Gradual and verifiable release of a secret. In C. Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, volume 293 of *LNCS*, pages 156–166. Springer-Verlag, 1988.

[6] J. Camenisch and A. Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In B. Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 93–118. Springer Verlag, 2001.

[7] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In M. Yung, editor, *Advances in Cryptology — CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer Verlag, 2002.

[8] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In S. Cimato, C. Galdi, and G. Persiano, editors, *Security in Communication Networks, Third International Conference, SCN 2002*, volume 2576 of *LNCS*, pages 268–289. Springer Verlag, 2003.

[9] J. Camenisch and M. Michels. A group signature scheme with improved efficiency. In K. Ohta and D. Pei, editors, *Advances in Cryptology — ASIACRYPT '98*, volume 1514 of *LNCS*, pages 160–174. Springer Verlag, 1998.

[10] J. Camenisch and M. Michels. Proving in zero-knowledge that a number n is the product of two safe primes. In J. Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, volume 1592 of *LNCS*, pages 107–122. Springer Verlag, 1999.

[11] J. Camenisch and M. Michels. Separability and efficiency for generic group signature schemes. In M. Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *LNCS*, pages 413–430. Springer Verlag, 1999.

[12] J. Camenisch and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. In D. Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *LNCS*, pages 126–144, 2003.

[13] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In B. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume 1296 of *LNCS*, pages 410–424. Springer Verlag, 1997.

[14] R. Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Weizmann Institute of Science, Rehovot 76100, Israel, June 1995.

[15] R. Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[16] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, Oct. 1985.

[17] D. Chaum. Zero-knowledge undeniable signatures. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, volume 473 of *LNCS*, pages 458–464. Springer-Verlag, 1991.

[18] D. Chaum, J.-H. Evertse, and J. van de Graaf. An improved protocol for demonstrating possession of discrete logarithms and some generalizations. In D. Chaum and W. L. Price, editors, *Advances in Cryptology — EUROCRYPT '87*, volume 304 of *LNCS*, pages 127–141. Springer-Verlag, 1988.

[19] D. Chaum and T. P. Pedersen. Wallet databases with observers. In E. F. Brickell, editor, *Advances in Cryptology — CRYPTO '92*, volume 740 of *LNCS*, pages 89–105. Springer-Verlag, 1993.

[20] D. Chaum and E. van Heyst. Group signatures. In D. W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, volume 547 of *LNCS*, pages 257–265. Springer-Verlag, 1991.

[21] R. Cramer and V. Shoup. Signature schemes based on the strong RSA assumption. *ACM Transactions on Information and System Security*, 3(3):161–185, 2000.

[22] I. Damgård and M. Koprowski. Practical threshold RSA signatures without a trusted dealer. In B. Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 152–165. Springer Verlag, 2001.

[23] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO '86*, volume 263 of *LNCS*, pages 186–194. Springer Verlag, 1987.

[24] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In B. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume 1294 of *LNCS*, pages 16–30. Springer Verlag, 1997.

[25] R. Gennaro, S. Halevi, and T. Rabin. Secure hash-and-sign signatures without the random oracle. In J. Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, volume 1592 of *LNCS*, pages 123–139. Springer Verlag, 1999.

[26] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, Apr. 1988.

[27] J. Kilian and E. Petrank. Identity escrow. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume 1642 of *LNCS*, pages 169–185, Berlin, 1998. Springer Verlag.

[28] A. K. Lenstra and E. K. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[29] A. Lysyanskaya. *Signature schemes and applications to cryptographic protocol design*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 2002.

[30] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *Proc. 7th ACM Conference on Computer and Communications Security*, pages 245–254. ACM press, Nov. 2000.

[31] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 184–200. IEEE Computer Society, IEEE Computer Society Press, 2001.

[32] D. Pointcheval and J. Stern. Security proofs for signature schemes. In U. Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, volume 1070 of *LNCS*, pages 387–398. Springer Verlag, 1996.

[33] Trusted Computing Group. Trusted computing platform alliance (tcpa) main specification, version 1.1a. Republished as Trusted Computing Group (TCG) main specifcation, Version 1.1b, Available at `www.trustedcomputinggroup.org`, 2001.

[34] Trusted Computing Group. Trusted platfrom module (TPM) main specification, version 1.2. Available at `www.trustedcomputinggroup.org`, 2001.

[35] Trusted Computing Group website. `www.trustedcomputinggroup.org`.

A Generation and Verification of a Proof that Public Key Elements lie in the right Subgroups

This section describes the steps to be performed by the issuer to prove that it computed R_0, R_1, S, Z, g, and h correctly, i.e., that $g, h \in \langle g' \rangle$, $S, Z \in \langle h \rangle$, and $R_0, R_1 \in \langle S \rangle$. It also describes the steps for a host to verify the proof and thus to establish that the privacy and anonymity properties of the scheme will hold.

Generating the proof.

1. The prover chooses random

$$\tilde{x}_{(g,1)}, \ldots, \tilde{x}_{(g,\ell_{\mathcal{H}})} \in_R [1, p'q'] \qquad \tilde{x}_{(h,1)}, \ldots, \tilde{x}_{(h,\ell_{\mathcal{H}})} \in_R [1, p'q'] \tag{19}$$

$$\tilde{x}_{(s,1)}, \ldots, \tilde{x}_{(s,\ell_{\mathcal{H}})} \in_R [1, p'q'] \qquad \tilde{x}_{(z,1)}, \ldots, \tilde{x}_{(z,\ell_{\mathcal{H}})} \in_R [1, p'q'] \tag{20}$$

$$\tilde{x}_{(0,1)}, \ldots, \tilde{x}_{(0,\ell_{\mathcal{H}})} \in_R [1, p'q'] \qquad \tilde{x}_{(1,1)}, \ldots, \tilde{x}_{(1,\ell_{\mathcal{H}})} \in_R [1, p'q'] \tag{21}$$

and computes

$$\tilde{g}_{(g,i)} := g'^{\tilde{x}_{(g,i)}} \bmod n \qquad \tilde{h}_{(h,i)} := g'^{\tilde{x}_{(h,i)}} \bmod n \tag{22}$$

$$\tilde{S}_{(s,i)} := h^{\tilde{x}_{(s,i)}} \bmod n \qquad \tilde{Z}_{(z,i)} := h^{\tilde{x}_{(z,i)}} \bmod n \tag{23}$$

$$\tilde{R}_{(0,i)} := S^{\tilde{x}_{(0,i)}} \bmod n \qquad \tilde{R}_{(1,i)} := S^{\tilde{x}_{(1,i)}} \bmod n \tag{24}$$

for $i = 1, \ldots, \ell_{\mathcal{H}}$.

2. The prover computes

$$c := H(n, g', g, h, S, Z, R_0, R_1, \tilde{g}_{(g,1)}, \ldots, \tilde{g}_{(g,\ell_{\mathcal{H}})}, \tilde{h}_{(h,1)}, \ldots, \tilde{h}_{(h,\ell_{\mathcal{H}})}, \\ \tilde{S}_{(s,1)}, \ldots, \tilde{S}_{(s,\ell_{\mathcal{H}})}, \tilde{Z}_{(z,1)}, \ldots, \tilde{Z}_{(z,\ell_{\mathcal{H}})}, \tilde{R}_{(0,1)}, \ldots, \tilde{R}_{(0,\ell_{\mathcal{H}})}, \tilde{R}_{(1,1)}, \ldots, \tilde{R}_{(1,\ell_{\mathcal{H}})}) \ . \tag{25}$$

3. The prover computes

$$\hat{x}_{(g,i)} := \tilde{x}_{(g,i)} - c_i x_g \bmod p'q' \qquad \hat{x}_{(h,i)} := \tilde{x}_{(h,i)} - c_i x_h \bmod p'q' \tag{26}$$

$$\hat{x}_{(s,i)} := \tilde{x}_{(s,i)} - c_i x_s \bmod p'q' \qquad \hat{x}_{(z,i)} := \tilde{x}_{(z,i)} - c_i x_z \bmod p'q' \tag{27}$$

$$\hat{x}_{(0,i)} := \tilde{x}_{(0,i)} - c_i x_0 \bmod p'q' \qquad \hat{x}_{(1,i)} := \tilde{x}_{(1,i)} - c_i x_1 \bmod p'q' \tag{28}$$

for $i = 1, \ldots, \ell_{\mathcal{H}}$, where c_i is the i-th bit of c.

4. The prover publishes

$$\mathit{proof} := (c, \hat{x}_{(g,1)}, \ldots, \hat{x}_{(g,\ell_{\mathcal{H}})}, \hat{x}_{(h,1)}, \ldots, \hat{x}_{(h,\ell_{\mathcal{H}})}, \\ \hat{x}_{(s,1)}, \ldots, \hat{x}_{(s,\ell_{\mathcal{H}})}, \hat{x}_{(z,1)}, \ldots, \hat{x}_{(z,\ell_{\mathcal{H}})}, \hat{x}_{(0,1)}, \ldots, \hat{x}_{(0,\ell_{\mathcal{H}})}, \hat{x}_{(1,1)}, \ldots, \hat{x}_{(1,\ell_{\mathcal{H}})},) \ .$$

as proof that $g, h \in \langle g' \rangle$, $S, Z \in \langle h \rangle$, and $R_0, R_1 \in \langle S \rangle$.

This proof uses binary challenges (the c_i's) and it is easy to see that it is indeed a zero-knowledge proof that R_0 and R_1 lie in $\langle S \rangle$, that Z and S lie in $\langle h \rangle$, and that g and h lie in $\langle g' \rangle$

Verifying the proof. A proof

$$\mathit{proof} = (c, \hat{x}_{(g,1)}, \ldots, \hat{x}_{(g,\ell_{\mathcal{H}})}, \hat{x}_{(h,1)}, \ldots, \hat{x}_{(h,\ell_{\mathcal{H}})}, \hat{x}_{(s,1)}, \ldots, \hat{x}_{(s,\ell_{\mathcal{H}})}, \hat{x}_{(z,1)}, \ldots, \hat{x}_{(z,\ell_{\mathcal{H}})}, \\ \hat{x}_{(0,1)}, \ldots, \hat{x}_{(0,\ell_{\mathcal{H}})}, \hat{x}_{(1,1)}, \ldots, \hat{x}_{(1,\ell_{\mathcal{H}})})$$

that R_0, R_1, S, Z, g, and h are correctly formed is verified as follows

1. Compute

$$\hat{g}_{(g,i)} := g^{c_i} g'^{\hat{x}_{(g,i)}} \bmod n \qquad \hat{h}_{(h,i)} := h^{c_i} g'^{\hat{x}_{(h,i)}} \bmod n \tag{29}$$

$$\hat{S}_{(s,i)} := S^{c_i} h^{\hat{x}_{(s,i)}} \bmod n \qquad \hat{Z}_{(z,i)} := Z^{c_i} h^{\hat{x}_{(z,i)}} \bmod n \tag{30}$$

$$\hat{R}_{(0,i)} := R_0^{c_i} S^{\hat{x}_{(0,i)}} \bmod n \qquad \hat{R}_{(1,i)} := R_1^{c_i} S^{\hat{x}_{(1,i)}} \bmod n \tag{31}$$

for $i = 1, \ldots, \ell_{\mathcal{H}}$, where c_i is the i-th bit of c.

2. Verify

$$c \stackrel{?}{=} H(n, g', g, h, S, Z, R_0, R_1, \hat{g}_{(g,1)}, \ldots, \hat{g}_{(g,\ell_{\mathcal{H}})}, \hat{h}_{(h,1)}, \ldots, \hat{h}_{(h,\ell_{\mathcal{H}})}, \\ \hat{S}_{(s,1)}, \ldots, \hat{S}_{(s,\ell_{\mathcal{H}})}, \hat{Z}_{(z,1)}, \ldots, \hat{Z}_{(z,\ell_{\mathcal{H}})}, \hat{R}_{(0,1)}, \ldots, \hat{R}_{(0,\ell_{\mathcal{H}})}, \hat{R}_{(1,1)}, \ldots, \hat{R}_{(1,\ell_{\mathcal{H}})}) \ . \tag{32}$$

B Authenticating a TPM w.r.t. an Endorsement Key

In the join protocol, the issuer must be sure that the value U stems from the TPM that owns a given endorsement public key EK. In this paper we just assume that the issuer receives U in an authentic manner, the following protocol could be used to achieve this.

(a) The issuer chooses a random $n_e \in \{0,1\}^{\ell_\varnothing}$ and encrypts n_e under the EK and sends the encryption to the TPM.

(b) The TPM decrypts this and thereby retrieves some string n_e. Then, the TPM computes $a_U := H(U\|n_e)$ and sends a_U to the issuer.

(c) The issuer verifies if $a_U = H(U\|n_e)$ holds.

This protocol should be executed between the steps 4 and 6 of the join protocol. This protocol is in the same spirit as the protocol to "authenticate" the AIK in the TCG TPM 1.1b specification.

C Security Proofs

Proof. We first describe how to obtain a simulator $\mathcal{S}$ such that the environment $\mathcal{E}$ cannot distinguish whether it is run in the real system, interacting with $\mathcal{A}$ and the real parties, or in the ideal system, interacting with $\mathcal{S}$ and the ideal parties.

Recall that the simulator interacts with $\mathcal{T}$ on behalf of the corrupted parties of the ideal system, and simulates the real-system adversary $\mathcal{A}$ towards the environment $\mathcal{E}$. The simulator $\mathcal{S}$ is given $\mathcal{A}$ as a black box. The simulator $\mathcal{S}$ will use $\mathcal{A}$ to simulate the conversations of $\mathcal{E}$ with $\mathcal{A}$. That is, the simulator will forward all messages from $\mathcal{E}$ to $\mathcal{A}$ and all messages from $\mathcal{A}$ to $\mathcal{E}$.

As we are in the random oracle model, the simulator has full control of the random oracle, i.e., the simulator plays the random oracle towards $\mathcal{A}$. Whenever $\mathcal{S}$ gets a call from $\mathcal{A}$ to the random oracle, $\mathcal{S}$ is free to answer it in any way subject to the only constraint that it cannot give two different answers to the same query.

We now describe how $\mathcal{S}$ handles the different operations of the system. These operations are triggered either by requests/messages from $\mathcal{T}$ to any of the corrupted party (which are thus received by $\mathcal{S}$) or then by requests/messages from $\mathcal{A}$ to any of the honest parties.

The simulator $\mathcal{S}$ will need to handle the operations differently depending on which parties are corrupted. Accordingly, we name the cases as follows. A capital letter denotes that the corresponding party is not corrupted and a small letter denotes that is it corrupted. For instance, (IMh) denotes the case where the issuer and the TPM are not corrupted, but the host is. If a party is not listed, then the simulator handles this case independently of whether or not this party is corrupted.

Ideal System Setup: For all parties controlled by $\mathcal{S}$, it indicates to $\mathcal{T}$ that they are corrupted.

Simulation of the Real System's Setup: Case (i). If the issuer is corrupted, $\mathcal{S}$ receives the issuer's public key $(n, g, g', h, R_0, R_1, S, Z, \gamma, \Gamma, \rho)$ from $\mathcal{A}$. For every honest platform, the simulator $\mathcal{S}$ chooses a random seed `DAAseed`. If there exists a corrupted platform, then let $\hat{U}$ be some corrupted platform and $\mathcal{S}$ runs the ideal-system join protocol for $\hat{\mathcal{H}}$ with counter value 0. Note that no honest party in the ideal system will note any of this.

Case (I). If the issuer is not corrupted, $\mathcal{S}$ runs the key generation of the issuer's and sends the thereby obtained public key $(n, g, g', h, R_0, R_1, S, Z, \gamma, \Gamma, \rho)$ to $\mathcal{A}$ as the public key of the issuer. Note that in this case $\mathcal{S}$ knows the factorization of n. Also, for every dishonest platform $\mathcal{H}_i$, the simulator $\mathcal{S}$ initializes the counter value $\mathtt{cnt}_i$ with 0.

Simulation of the Join Protocol/Operation: Players in this operation are a host, a TPM and the issuer. We distinguish six cases, depending on whether or not the issuer, the host, and the TPM are corrupted (recall that we do not consider corrupted TPMs with uncorrupted hosts).

Case (ihm): Issuer and the platform (host and TPM) are corrupted. There is nothing $\mathcal{S}$ has to do, i.e., it won't even notice as this is basically an internal transaction of A.

Case (IHM): Issuer and platform are not corrupted. $\mathcal{S}$ does not have to do anything (again, it won't even notice as this operation does not trigger a call from $\mathcal{T}$ to $\mathcal{S}$).

Case (iHM): Issuer is corrupted but not the platform. In this case $\mathcal{S}$ gets a request from $\mathcal{T}$ that platform with identity $\mathtt{id}_i$ wants to join w.r.t. counter value $\mathtt{cnt}_i$. Thus, $\mathcal{S}$ has to play the ideal-system issuer towards $\mathcal{T}$ and to simulate the real-system platform $\mathcal{H}_i/\mathcal{M}_i$ towards $\mathcal{A}$. I.e., $\mathcal{S}$ runs the real-system join protocol as platform $\mathcal{H}_i/\mathcal{M}_i$ with $\mathcal{A}$ as issuer using the counter value $\mathtt{cnt}_i$. If the join protocol finishes successfully and $\mathcal{S}$ obtains a certificate (A, e, v) from $\mathcal{A}$ for values f_0 and f_1, it stores these and informs $\mathcal{T}$ that the platform is allowed to join. If the protocol failes, $\mathcal{S}$ informs $\mathcal{T}$ that platform is not allowed to join.

Case (Ihm): Issuer is not corrupted but the platform is.

In this case $\mathcal{S}$ gets a request from $\mathcal{A}$, as real-system platform with identity $\mathtt{id}_i$, to run the join protocol.

Thus $\mathcal{S}$ plays the issuer and runs the join protocol with the adversary as platform as follows. It runs the join protocol until Step 8b (i.e., it does not send (A, e, v') yet). If the protocol is aborted before this step, the simulator does not need to do anything further for this query. If $\mathcal{S}$ has not seen N_I sent by $\mathcal{A}$ during the join protocol, it increments the counter $\mathtt{cnt}_i$, stores $\mathtt{cnt}_i$ and N_I, and informs $\mathcal{T}$ that $\mathcal{H}_i$ wants to join w.r.t. counter value $\mathtt{cnt}_i$. If $\mathcal{S}$ has seen the N_I sent by $\mathcal{A}$, it looks up the counter value $\mathtt{cnt}_i$ stored with N_I, and informs $\mathcal{T}$ as $\mathcal{H}_i$ that it wants to join w.r.t. counter value $\mathtt{cnt}_i$. Next, $\mathcal{S}$ plays its role as ideal-system TPM $\mathcal{M}_i$ and sends $\mathtt{id}_i$ and waits until $\mathcal{T}$ tells it (addressed as $\mathcal{H}_i$) whether it was allowed to join. If the answer is positive, $\mathcal{S}$ finishes the join protocol with $\mathcal{A}$; otherwise it aborts the protocol.

Case (IhM): Issuer and TPM are not corrupted, but the host is.

This case is similar to the one above (case (Ihm)), with the exception that $\mathcal{S}$ also runs the TPM part of the join protocol with $\mathcal{A}$ and will need to send $\mathtt{id}_i$ as TPM $\mathcal{M}_i$ to $\mathcal{T}$ (which it gets as corrupted host $\mathcal{H}_i$ from $\mathcal{T}$ during the setup phase of the ideal system).

Case (ihM): Issuer and host are corrupted but not the TPM.

In this case, $\mathcal{S}$ gets a request from $\mathcal{A}$ that host $\mathcal{H}_i$ wants to use TPM $\mathcal{M}_i$ in a join protocol w.r.t. the counter value $\mathtt{cnt}_i$. So $\mathcal{S}$ triggers a request to $\mathcal{T}$ that $\mathcal{H}_i$ wants to join w.r.t. counter value $\mathtt{cnt}_i$. As the issuer is corrupted, $\mathcal{S}$ gets a request from $\mathcal{T}$ whether the platform with identity $\mathtt{id}_i$ can join. Next, $\mathcal{S}$ runs the TPM part of the join protocol with the adversary. If the protocol finished successfully $\mathcal{S}$, i.e., $\mathcal{S}$ received a value v' from $\mathcal{A}$ for values f_0 and f_1 it chose, it stores these and, acting as corrupted issuer, informs $\mathcal{T}$ that the platform is allowed to join. Otherwise, $\mathcal{S}$ informs $\mathcal{T}$ that platform is not allowed to join.

Simulation of the DAA-Sign Protocol/Operation: The simulator will only note that this operation is going on when the host is corrupted and the TPM is not. That is, when the adversary engages with the (honest) TPM in the signing protocol.

Case (hM): Request from corrupted host to non-corrupted TPM to sign a message. So $\mathcal{S}$ gets a call from $\mathcal{A}$ addressed as TPM $\mathcal{M}_i$ to sign message m w.r.t. counter value $\mathtt{cnt}_i$ and then proceeds as follows.

1. If no join protocol has been successfully finished by $\mathcal{M}_i$ w.r.t. counter value $\mathtt{cnt}_i$, simulator $\mathcal{S}$ rejects the request.
2. (a) $\mathcal{S}$ receives a ζ from $\mathcal{A}$. It looks up ζ in its records. If ζ is present, it looks up the N_V used with ζ before, otherwise it chooses a random $N_V \in_R \langle\zeta\rangle$.
   (b) $\mathcal{S}$ sends N_V to $\mathcal{A}$.
3. $\mathcal{S}$ now forges the TPM's part of the signature as follows.
   (a) $\mathcal{S}$ picks random integers

$$s_v \in_R \{0,1\}^{\ell_v+\ell_\varnothing+\ell_\mathcal{H}}, \qquad s_{f_0}, s_{f_1} \in_R \{0,1\}^{\ell_f+\ell_\varnothing+\ell_\mathcal{H}},$$

   (b) $\mathcal{S}$ picks a random $c \in \{0,1\}^{\ell_\mathcal{H}}$.
   (c) $\mathcal{S}$ computes

$$\tilde{T}_{1t} := Z^{-c} R_0^{s_{f_0}} R_1^{s_{f_1}} S^{s_v} \bmod n \ , \tag{33}$$

$$\tilde{N}_V := N_V^{-c} \zeta^{s_{f_0}+s_{f_1} 2^{\ell_f}} \bmod \Gamma \ . \tag{34}$$

   and sends $\tilde{T}_{1t}$ and $\tilde{N}_V$ to $\mathcal{A}$.
   (d) $\mathcal{S}$ receives c_h from $\mathcal{A}$, chooses a random n_t, and patches the random oracle such that

$$c = H(c_h \| n_t \| b \| m) \ .$$

4. $\mathcal{S}$ request as $\mathcal{H}_i$ from $\mathcal{T}$ that m be signed w.r.t. counter value $\mathtt{cnt}_i$. If $\mathcal{T}$ informs $\mathcal{S}$ that $\mathcal{M}_i$ is ready to sign and asks for a basename, $\mathcal{S}$ postpones the answer to $\mathcal{T}$ ($\mathcal{S}$ will answer this call only once it has seen the signature from $\mathcal{A}$, see "Simulation of the DAA-Verify operation" below) but sends $\mathcal{A}$ the values c, n_t, s_{f_0}, s_{f_1}, and s_v.

Simulation of the DAA-Verify Operation/Algorithm: This action can be triggered in two different ways. First, $\mathcal{S}$ (as ideal-system verifier $\mathcal{V}_j$) can be informed by $\mathcal{T}$ that some honest platform signed m w.r.t. a basename $\mathtt{bsn}$ (and possibly a pseudonym P). This is case (H) below.

Second, $\mathcal{S}$ (as real-system verifier $\mathcal{V}_j$) can receive a real-system signature from the adversary. This is case (h).

Note that we need not distinguish whether or not the issuer is corrupted.

Case (H): The platform is not corrupted (by assumption, the TPM is honest if the host is).

In this case $\mathcal{S}$ gets a notification from $\mathcal{T}$ that some platform has signed m w.r.t. $\mathtt{bsn}$ and, if $\mathtt{bsn} \neq \perp$, pseudonym P.

$\mathcal{S}$ proceeds as follows to simulate a signature in the real-system using the power over the random oracle.

1. (a) If $\mathtt{bsn} = \perp$, simulator $\mathcal{S}$ picks a random $\zeta \in \langle\gamma\rangle$ and picks a random $N_V \in_R \langle\zeta\rangle$.
   (b) If $\mathtt{bsn} \neq \perp$, $\mathcal{S}$ looks up P in its records. If P is present, it looks up (ζ, N_V) used with P before, otherwise $\mathcal{S}$ sets $\zeta := (H_\Gamma(1\|\mathtt{bsn}))^{(\Gamma-1)/\rho} \bmod \Gamma$, chooses a random $N_V \in_R \langle\zeta\rangle$, and stores (ζ, N_V) with P.
   (c) $\mathcal{S}$ picks a random $T_1 \in_R \langle h\rangle$ and $T_2 \in_R \langle g'\rangle$.
2. $\mathcal{S}$ now forges a signature w.r.t. ζ, N_V, T_1, and T_2 as follows.

(a) $\mathcal{S}$ picks random integers

$$\begin{aligned} s_v &\in_R \{0,1\}^{\ell_v+\ell_\varnothing+\ell_{\mathcal{H}}}, & s_{f_0}, s_{f_1} &\in_R \{0,1\}^{\ell_f+\ell_\varnothing+\ell_{\mathcal{H}}}, \\ s_e &\in_R \{0,1\}^{\ell'_e+\ell_\varnothing+\ell_{\mathcal{H}}}, & s_{ee} &\in_R \{0,1\}^{\ell'_e+\ell_e+\ell_\varnothing+\ell_{\mathcal{H}}+1}, \\ s_w &\in_R \{0,1\}^{\ell_n+2\ell_\varnothing+\ell_{\mathcal{H}}} & s_{ew} &\in_R \{0,1\}^{\ell_e+\ell_n+2\ell_\varnothing+\ell_{\mathcal{H}}+1} \\ s_r &\in_R \{0,1\}^{\ell_n+2\ell_\varnothing+\ell_{\mathcal{H}}} & s_{er} &\in_R \{0,1\}^{\ell_e+\ell_n+2\ell_\varnothing+\ell_{\mathcal{H}}+1} \end{aligned}$$

(b) $\mathcal{S}$ picks a random $c \in \{0,1\}^{\ell_{\mathcal{H}}}$.

(c) $\mathcal{S}$ computes

$$\tilde{T}_1 := Z^{-c} T_1^{s_e + c2^{\ell_e - 1}} R_0^{s_{f_0}} R_1^{s_{f_1}} S^{s_v} h^{-s_{ew}} \bmod n \ , \tag{35}$$

$$\tilde{T}_2 := T_2^{-c} g^{s_w} h^{s_e + c2^{\ell_e - 1}} (g')^{s_r} \bmod n \ , \tag{36}$$

$$\tilde{T}'_2 := T_2^{-(s_e + c2^{\ell_e - 1})} g^{s_{ew}} h^{s_{ee}} (g')^{s_{er}} \bmod n \ , \quad \text{and} \tag{37}$$

$$\tilde{N}_V := N_V^{-c} \zeta^{s_{f_0} + s_{f_1} 2^{\ell_f}} \bmod \Gamma \ . \tag{38}$$

(d) $\mathcal{S}$ chooses a random n_t and patch the random oracle such that

$$c = H(H((n\|g\|g'\|h\|R_0\|R_1\|S\|Z\|\gamma\|\Gamma\|\rho)\|\zeta\|(T_1\|T_2)\|N_V\|(\tilde{T}_1\|\tilde{T}_2\|\tilde{T}'_2)\|\tilde{N}_V\|)\|n_t\|n_v\|b\|m)) \ .$$

3. $\mathcal{S}$ sends $\sigma := (\zeta, (T_1, T_2), N_V, c, n_t, (s_v, s_{f_0}, s_{f_1}, s_e, s_{ee}, s_w, s_{ew}, s_r, s_{er}))$ as signature on m to $\mathcal{A}$.

Case (h): $\mathcal{S}$ obtains a (new) signature from $\mathcal{A}$ on a message m w.r.t. N_V, ζ, and basename `bsn`.

First, $\mathcal{S}$ checks the validity of the signature, except the rogue check. If these checks fail, $\mathcal{S}$ can just ignore the signature. Otherwise, $\mathcal{S}$ proceeds with the rogue check, i.e., checks whether $\zeta^{f_0 + f_1 2^{\ell_f}} = N_V$ holds for any of the f_0 and f_1 on the rogue-list.

- If there is a matching (f_0, f_1) pair, $\mathcal{S}$ checks whether this pair is assigned to any $\mathtt{id}_i$ w.r.t. some counter value $\mathtt{cnt}_i$. (If the pair is assigned, then $\mathtt{id}_i$ is marked rogue w.r.t. the counter value $\mathtt{cnt}_i$). If there is such an $\mathtt{id}_i$, $\mathcal{S}$ triggers a call to $\mathcal{T}$ as $\mathcal{H}_i$ to sign m w.r.t. the counter value $\mathtt{cnt}_i$ and `bsn` and the processing of the operation is finished. (Note that $\mathtt{id}_i$ will be the identity of a corrupted TPM and thus also $\mathcal{H}_i$ is corrupted.)
  If there is no such $\mathtt{id}_i$, $\mathcal{S}$ checks whether it has already seen the (ζ, N_V) pair appearing in the signature. As this can only happen if the issuer is corrupt, $\mathcal{S}$ is free to choose any corrupt TPM $\mathcal{M}_i$ and a counter value $\mathtt{cnt}_i$ such that $\mathcal{H}_i/\mathcal{M}_i$ is not yet a member w.r.t. $\mathtt{cnt}_i$, trigger the calls to $\mathcal{T}$ to make them a member, then the call to tag them rogue, and finally the calls as $\mathcal{H}_i$ to sign m w.r.t. the counter value $\mathtt{cnt}_i$ and `bsn`.
- If there is no matching (f_0, f_1) pair, the signature comes from a platform that is (not yet) tagged as rogue. So $\mathcal{S}$ has to figure out to which platform it should associate this signature. To this end $\mathcal{S}$ checks whether it has already seen the (ζ, N_V) pair appearing in the signature.
  - If (ζ, N_V) was used before, $\mathcal{S}$ proceeds as follows.
    - If $\mathcal{S}$ used (ζ, N_V) as honest $\mathcal{M}_i$ in the simulation of the DAA-sign, and $\mathcal{S}$ owes $\mathcal{T}$ a reply whether or not to proceed with a signature on the message m, $\mathcal{S}$ replies to $\mathcal{T}$ (as host $\mathcal{H}_i$) that it indeed wants to sign m (cf. Case (hM) Part I).
    - If $\mathcal{S}$ used (ζ, N_V) as honest $\mathcal{M}_i$in the simulation of the DAA-sign, but $\mathcal{S}$ (as host $\mathcal{H}_i$) does not owe $\mathcal{T}$ a reply whether or not to proceed with a signature on the message m, $\mathcal{S}$ stops outputting "failure 1".

* Otherwise $\mathcal{S}$ looks in its database which host $\mathcal{H}_i$, TPM $\mathcal{M}_i$, and counter value $\mathtt{cnt}_i$ it assigned to that pair previously and then initiates and executes the signing of m w.r.t. $\mathtt{bsn}$ and $\mathtt{cnt}_i$ with $\mathcal{T}$ on behalf of $\mathcal{H}_i$.

– If (ζ, N_V) is new to $\mathcal{S}$, it has to assign a "free" TPM $\mathcal{M}_i$ and counter value $\mathtt{cnt}_i$ pair to (ζ, N_V). As $\mathcal{S}$ has not seen (ζ, N_V), this means that the involved TPM must be corrupted.
  * If $\mathtt{bsn} = \perp$ then $\mathcal{S}$ can just select any corrupted $\mathcal{M}_i$ and counter value $\mathtt{cnt}_i$ (note that by assumption the corresponding host is then corrupted, too) such that $\mathcal{M}_i$ is not tagged as rogue w.r.t. counter value $\mathtt{cnt}_i$. Then, as host $\mathcal{H}_i$, simulator $\mathcal{S}$ initiates the signing of message m w.r.t. $\mathtt{bsn}$ and counter value $\mathtt{cnt}_i$ with $\mathcal{T}$.
  * If $\mathtt{bsn} \neq \perp$ then $\mathcal{S}$ select any corrupted $\mathcal{M}_i$ and counter value $\mathtt{cnt}_i$ such that $\mathcal{M}_i$ is not tagged as rogue w.r.t. counter value $\mathtt{cnt}_i$ and such that $\mathcal{S}$ has not yet triggered a signature call to $\mathcal{T}$ as $\mathcal{H}_i$ w.r.t. $\mathtt{bsn}$ and $\mathtt{cnt}_i$. If it finds such a "free" $(\mathcal{M}_i, \mathtt{cnt}_i)$ pair, it initiates the signing of message m w.r.t. $\mathtt{bsn}$ and counter value $\mathtt{cnt}_i$, as host $\mathcal{H}_i$. If there is no such free $(\mathcal{M}_i, \mathtt{cnt}_i)$ pair and the issuer is not corrupt, the adversary must be able to forge signatures and so $\mathcal{S}$ stops outputting "failure 2". However, if the issuer is corrupt, $\mathcal{S}$ can simply generate such a pair by letting some corrupt TPM and host join w.r.t. some free counter value $\mathtt{cnt}_i$.

Simulation of the Rogue-Tagging Operation: By assumption the rogue-tagging oracle is always corrupted. Thus, $\mathcal{S}$ obtains f_0, f_1, s, A and e from $\mathcal{A}$.

If $R_0^{f_0} R_1^{f_1} S^s A^e$ does not equal Z, simulator $\mathcal{S}$ just ignores the values. Next, $\mathcal{S}$ checks whether f_0 and f_1 correspond to a N_I or N_V that $\mathcal{S}$ used with an honest TPM $\mathcal{M}_i$. If this is the case, $\mathcal{A}$ computed the discrete logarithm of this N_I or N_V and so $\mathcal{S}$ stops outputting "failure 3". Otherwise, $\mathcal{S}$ proceeds as follows. We now distinguish whether or not the issuer is corrupted.

Case (I): the issuer is not corrupted. In this case $\mathcal{S}$ looks up to which platform $\mathcal{H}_i$ and counter value $\mathtt{cnt}_i$ the values f_0 and f_1 correspond to, using the N_I retrieved from the join protocol and then tells $\mathcal{T}$ this platform should be tagged rogue w.r.t. counter value $\mathtt{cnt}_i$. If $\mathcal{S}$ finds no N_I that corresponds to f_0 and f_1, the adversary must have forged a membership certificate and so $\mathcal{S}$ stops outputting "failure 4".

Case (i). The issuer is corrupted. In this case $\mathcal{S}$ looks whether there is already a N_V that corresponds to the values f_0 and f_1, it looks up the related $\mathtt{id}_i$ and counter value $\mathtt{cnt}_i$ and tells $\mathcal{T}$ to tag them as rogue. (There might be several matching N_V's even belonging to different $\mathtt{id}_i$. However, it it's safe to take anyone of them.) If there is no matching N_V, simulator $\mathcal{S}$ just stores f_0 and f_1 without assigning the pair a $\mathtt{id}_i$ and a counter value $\mathtt{cnt}_i$. (It can be the case that the matching N_V is only later produced by the adversary and so $\mathcal{S}$ can trigger the rogue tagging only then, see "Simulation of the DAA-Sign/Verify operation".)

This concludes the description of the simulator $\mathcal{S}$. It remains to argue that $\mathcal{S}$ is such that the environment cannot distinguish whether it is run in the ideal system or in the real system.

We now argue that the simulator described above works, i.e., that under the decisional Diffie-Hellman assumption in $\mathbb{Z}_\Gamma^*$ and under the strong RSA assumption, it will not stop outputting "failure X" and that the environment can not distinguish whether or not it is run in the real system or in the ideal system.

- Failure 1: If this failure occurs, the adversary has forged a signature w.r.t. an N_V that the simulator chose. Because the signature is based on a proof of knowledge of the discrete logarithm of N_V, we can extract $\log_\zeta N_V$ by using rewinding on the adversary and the power over the random oracle. Thus, by again using the power over the random oracle to set ζ to a given target value,

we can reduce an adversary that produces a failure of this type to one that compute discrete logarithms. Note in such a reduction we would loose a factor related to the number of oracle calls.

- Failure 2: This failure can only occur if the issuer is honest and the adversary produces signatures w.r.t. given basenames but with more different N_V's than there are TPM's that were made members by the issuer. This means that the adversary must be able to produce signatures w.r.t. a pair (N_V, ζ) such that $\log_\zeta N_V = \log_{\zeta_I} N_I$ for to any N_I for which the issuer (simulator) completed the join protocol. We will show in Lemma 3 that this is not possible under the strong RSA assumption.

- Failure 3: The adversary produced f_0 and f_1 such that N_V or N_I equals $\zeta^{f_0+f_1 2^{\ell_f}}$ for some ζ and N_V or N_I. However, the simulator choose all the N_V randomly, i.e., such that it did not know $\log_\zeta N_V$. Similarly, the simulator could also have chosen N_I at random; this would not change the distribution of the other values sent to the adversary. It is now straightforward to show that the adversary could be used to solve the discrete logarithm problem in $\mathbb{Z}^*_\Gamma$. Note that this reduction is tight because of the random-self-reducibility of the discrete logarithm problem.

- Failure 4: This is a similar failure as failure 3 and can only occur if the issuer is honest and the forged a membership certificate (A, e, v) with f_0 and f_1 such that that do not correspond to a N_I for which the issuer (simulator) completed the join protocol. We will show in Lemma 3 that this is not possible under the strong RSA assumption.

- It remains to argue that the environment and adversary cannot distinguish whether they are run in the real system or in the ideal system. To this end note that the simulator performs all operations as the respective players would in the real system with the exceptions of the operations related to sign protocol. Also note that the simulator will patch the random oracle such that its outputs are uniformly random. When simulating an honest TPM, the simulator chooses N_V, $\tilde{N}_V$, $\hat{T}_{1t}$, s_v, s_{f_0}, and s_{f_1} and, when simulating an honest host, $\mathcal{S}$ also chooses T_1, T_2, s_e, s_{ee}, s_w, s_{we}, s_r, and s_{re} not as specified by the protocol. (We note that $\hat{T}_1$, $\hat{T}_2$, and $\hat{T}'_2$ are a function of these values and the output of the random oracle.) Thus we need to argue that the adversary and environment cannot distinguish whether these values are chosen as the simulator does or as specified by the sign protocol.

  It is easy to see that all s_i's are distributed statistically close in both cases if $\ell_\varnothing$ is sufficiently large, i.e., the statistical difference is at most $2^{-(\ell_\varnothing - 1)}$.

  Next, consider the distribution of N_V's. The simulator chooses all N_V's at random whereas in the real system the $\log_\zeta N_V$ will be the same for a given TPM and counter value. However, under the decisional Diffie-Hellman assumption, no adversary can distinguish the two distributions provided that the modulus Γ and the size ρ of the subgroup these elements come from are sufficiently large.

  Consider the distribution of $\tilde{N}_V$'s: it is easy to see that $\tilde{r}_f (= r_{f_0} + r_{f_1} 2^{\ell_f} \bmod \rho)$ with r_{f_i} chosen as specified in the protocol is distributed statistically close to $s_{f_0} + s_{f_1} 2^{\ell_f} \bmod \rho)$ with s_{f_i} as chosen by the simulator. Now as $N_V \in \langle \zeta \rangle$, the $\tilde{N}_V$'s are distributed (computationally) indistinguishably in the real and ideal system.

  Next, consider the values T_1 and T_2. The simulator chooses them randomly from $\langle h \rangle$ and $\langle g' \rangle$, respectively. We now show that an honest host computes T_1 as Ah^w and T_2 as $g^w h^e (g')^r$ and hence they are distributed statistically close to random elements of $\langle h \rangle$ and $\langle g' \rangle$, respectively, if $A \in \langle h \rangle$ and $g, h \in \langle g' \rangle$ as w and r are chosen such that $w \bmod |h|$ and $r \bmod |g'|$ are distributed statistically close to $[0, |h|]$ and $[0, |g'|]$. We show that $A \in \langle h \rangle$ and $g, h \in \langle g' \rangle$ will indeed be the case, even is the issuer was corrupted. First, the issuer proves in the set-up phase that the elements R_0 and R_1 lie in the group generated by S, that S and Z lie in the group generated by

h, and that g, and h lie in the subgroup generated by g'; Second, $A \in \langle h \rangle$ is ensured by the the proof the issuer give in Step 8 of the Join protocol and the check of the host that e is a sufficiently large prime. That is, from that proof one can deduce that

$$A^{\Delta c'} \equiv \left(\frac{Z}{US^{v''}}\right)^{\Delta s_e} \pmod{n} \tag{39}$$

holds for some $\Delta c' \in [-2^{\ell_{\mathcal{H}}+1}, 2^{\ell_{\mathcal{H}}+1}]$ and Δs_e (this can be derived from two accepting answers to different challenges c' but the same value $\tilde{A}$). We also know that $A^e \equiv \frac{Z}{US^{v''}} \pmod{n}$ holds and that there are exits integers a_1 and a_2 such that $ea_1 + \Delta c' a_2 = 1$ as $\Delta c' \nmid e$ (e is prime and $\Delta c'$ is smaller than e). Thus we have

$$A \equiv A^{a_1 e + \Delta c' a_2} \equiv \left(\frac{Z}{US^{v''}}\right)^{a_1 + a_2 \Delta s_e} \pmod{n} \tag{40}$$

and therefore A must lie in the group generated by h.

□

Lemma 3. *Under the strong RSA assumption, there exists no adversary that does not control the issuer but can make the simulator output failure 2 or 4 in the above simulation, provided that the join protocol is run sequentially.*

To prove this lemma, we use the following lemma which is a generalization of a Theorem stated by Camenisch and Shoup [12] and which used before implicitly in the analysis of many schemes based on the strong RSA assumption (e.g., [9, 25, 21, 1, 6]).

Lemma 4. *Let n be a safe prime product and $z \in \mathrm{QR}_n$. Let $r_i \in_R [n, n^2]$ and e_i be integers that have no factor smaller than $2^{\ell_e - 1}$, and $x_i := z^{e_i r_i}$ for $(i = 1, \ldots, u)$. Then, for any algorithm that obtains $(n, x_1, \ldots, x_u, e_1, \ldots, e_u)$, but not any other values related to the r_i's, and then outputs s_i's and c, and t such that $t^c = \prod x_i^{s_i}$ and $|c| < 2^{\ell_e - 1}$, we have that either $c | s_i$ holds for all i, or then, with probability at least $1/2$ we can compute integers $e > 1$ and v such $v^e = z$.*

The reason this lemma holds is that the x_i reveal at most r_i modulo the order of z and $c \nmid e_i$ if $e_i > 1$ and thus, when considering $t^c \equiv z^{\sum e_i r_i s_i} \pmod{n}$, c must either divide all s_i or one can get an equation of the form $\hat{t}^c = z^u$ such that $c \nmid u$ and then compute a non-trivial root of z.

Furthermore, we related our proofs to the proof of security are based on the security of the CL signature scheme [8]. Thus there exists a simulator/oracle $\mathcal{S}_{CL}$ with the following behavior:

Generation of Public key : On input an instance $(n, z \in \mathrm{QR}_n)$ of the flexible RSA problem the simulator outputs a public key $(n, R_0, \ldots, R_{u-1}, S, Z)$, such that $R_0, \ldots, R_{u-1}, S, Z$ satisfy the properties of the x_i's in Lemma 4.

Adaptive Signature Queries: On input a vector of messages $\vec{m}_i = (m_{(i,0)}, \ldots, m_{(i,u-1)})$, with $m_i \in \pm\{0,1\}^{\ell_m}$, the simulator/oracle $\mathcal{S}_{CL}$ outputs a signature A_i, e_i, v_i such that $A_i^{e_i} = R_0^{m_{(i,0)}} \cdots R_{u-1}^{m_{(i,u-1)}} S^{v_i} Z$, $e_i \in [2^{\ell_e - 1}, 2^{\ell_e - 1} + 2^{\ell'_e - 1}]$ is a prime and $(v_i - 2^{\ell_v}) \in \{0,1\}^{\ell_v}$, where it is required that $\ell_e > \ell_m + 1$. (In the original proof, primes in $[2^{\ell_e - 1}, 2^{\ell_e}]$ where considered [8]. Clearly, the proof also holds when requiring the simulator to output only primes from $[2^{\ell_e - 1}, 2^{\ell_e - 1} + 2^{\ell'_e - 1}]$.)

Forgery: On input a signature (A, e, v) on a vector of messages $\vec{m}$ with $\vec{m} \neq \vec{m}_i$ for all i, it outputs a solution to the instance $(n, z \in \mathrm{QR}_n)$ of the flexible RSA problem.

We are now ready to prove Lemma 3.

Proof of Lemma 3. We describe and algorithm $\mathcal{S}'$ that, given an adversary $\mathcal{A}$ and an environment $\mathcal{E}$ that makes the simulator § describe above output failure 2 or 4, even though $\mathcal{A}$ does not control the issuer, but we may assume that controls all TPM and hosts, can solve a random instance $(n, z \in \mathrm{QR}_n)$ of the flexible RSA problem.

The algorithm $\mathcal{S}'$ will interact with $\mathcal{A}$, $\mathcal{E}$, and $\mathcal{T}$ as follows.

Simulation of Real-System Setup. $\mathcal{S}'$ has to generate the public key of the issuer for the given instance of the flexible RSA problem. To this end $\mathcal{S}'$ employs the simulator $\mathcal{S}_{CL}$ as described above on input this instance and receive R_0, R_1, S, Z. Then $\mathcal{S}'$ chooses $g := z^{r_g} \bmod n$, $g' := z^{r_{g'}} \bmod n$ and $h := z^{r_h} \bmod n$ for $r_g, r_{g'}, r_h \in_R [n, n^2]$. $\mathcal{S}'$ also chooses two primes Γ and ρ such that $\rho | \Gamma - 1$ and a generator γ of the subgroup of $\mathbb{Z}_\Gamma^*$ of order ρ. Moreover, $\mathcal{S}'$ forges all the proofs that the elements g, h, R_0, R_1, S, Z using the power over the random oracle and standard zero-knowledge simulation techniques. $\mathcal{S}'$ send these proofs and the public key $(n, g, g', h, R_0, R_1, S, Z, \gamma, \Gamma, \rho)$ to $\mathcal{A}$.

Simulation of the Join Protocol. $\mathcal{S}'$ gets a request from $\mathcal{A}$, as real-system platform with identity $\mathtt{id}_i$, to run the join protocol.

Thus $\mathcal{S}'$ plays the issuer and runs the join protocol with the adversary as platform as follows.

It runs the join protocol as follows. First, $\mathcal{S}'$ receives U and N_I from the $\mathcal{A}$. Next, $\mathcal{S}'$ runs the Step 6 of the join protocol with $\mathcal{A}$ once, and uses standard rewinding techniques and the power over the random oracle to obtain two transcripts of the join protocol w.r.t. the same values for U, N_I, $\tilde{U}$, and $\tilde{N}_I$ but different values for $(c, s_{f_0}, s_{f_1}, s_{v'})$, i.e.,

$$(c^{(0)}, s_{f_0}^{(0)}, s_{f_1}^{(0)}, s_{v'}^{(0)}) \quad \text{and} \quad (c^{(1)}, s_{f_0}^{(1)}, s_{f_1}^{(1)}, s_{v'}^{(1)}) \tag{41}$$

such that $c^{(0)} \neq c^{(1)}$. Let $\Delta c = c^{(0)} - c^{(1)}$ and $\Delta s_i = s_i^{(1)} - s_i^{(0)}$. From the verification Equations (9) and (8) we derive that

$$U^{\Delta c} \equiv R_0^{\Delta s_0} R_1^{\Delta s_1} S^{\Delta s_{v'}} \pmod{n} \tag{42}$$

Now, due to Lemma 4, we have that Δc divides the Δs_i's and $\Delta s_{v'}$. Let $\hat{s}_i := \Delta s_i / \Delta c$ and $\hat{s}_{v'} := \Delta s_{v'} / \Delta c$. Thus we have

$$U \equiv b_U R_0^{\hat{s}_0} R_1^{\hat{s}_1} S^{\hat{s}_{v'}} \pmod{n} \tag{43}$$

for some b_U such that $b_U^{\Delta c} \equiv 1 \pmod{n}$. Now b_U must be ± 1 for this to happen because if it was not ± 1 then either $\phi(n)/4 | \Delta c$ or $\gcd(b, n) > 1$ and we could factor and thus solve the flexible RSA instance. Moreover, because of the check-terms (9) we have that

$$\hat{s}_i \in \{0,1\}^{\ell_f + \ell_\varnothing + \ell_{\mathcal{H}} + 2} \quad \text{and} \quad \hat{s}_{v'} \in \{0,1\}^{\ell_n + 2\ell_\varnothing + \ell_{\mathcal{H}} + 2} \tag{44}$$

Next, if $\mathcal{S}'$ has not seen N_I sent by $\mathcal{A}$ during the join protocol, it increments the counter $\mathtt{cnt}_i$, stores $\mathtt{cnt}_i$ and N_I, and informs $\mathcal{T}$ that $\mathcal{H}_i$ wants to join w.r.t. counter value $\mathtt{cnt}_i$. If $\mathcal{S}'$ has seen the N_I sent by $\mathcal{A}$, it looks up the counter value $\mathtt{cnt}_i$ stored with N_I, and informs $\mathcal{T}$ as $\mathcal{H}_i$ that it wants to join w.r.t. counter value $\mathtt{cnt}_i$. Next, $\mathcal{S}'$ plays its role as ideal-system TPM $\mathcal{M}_i$ and sends $\mathtt{id}_i$ and waits until $\mathcal{T}$ tells it (addressed as $\mathcal{H}_i$) whether it was allowed to join. If the answer is positive, $\mathcal{S}'$ asks the $\mathcal{S}_{CL}$ for a signature on the message tuple $(\hat{s}_0, \hat{s}_1)$ and receives $(\hat{A}, \hat{v}, \hat{e})$ such that

$$Z \equiv \hat{A}^{\hat{e}} R_0^{\hat{s}_0} R_1^{\hat{s}_1} S^{\hat{v}} \pmod{n} \tag{45}$$

Note that because of $\hat{s}_i \in \{0,1\}^{\ell_f + \ell_\varnothing + \ell_{\mathcal{H}} + 2}$, the message tuple satisfies the length requirement $\ell_e > \ell_m + 1$ on the message tuple by the oracle as we required $\ell_e > \ell_f + \ell_\varnothing + \ell_{\mathcal{H}} + 4$ and $\hat{s}_i \in \{0,1\}^{\ell_f + \ell_\varnothing + \ell_{\mathcal{H}} + 2}$.

$\mathcal{S}'$ sets

$$v'' := \hat{v} - \hat{s}_{v'} \ .$$

Note that the oracle $\mathcal{S}_{CL}$ gives $\mathcal{S}'$ a random v of length ℓ_v. As $\hat{s}_{v'} \in \{0,1\}^{\ell_n+2\ell_\varnothing+\ell_\mathcal{H}+2}$ and $\ell_v > \ell_n + \ell_f + \ell_\varnothing + \ell_\mathcal{H} + \ell_r + 3 > \ell_n + 2\ell_\varnothing + \ell_\mathcal{H} + 2$ by assumption, the value $v'' = v - \hat{s}_{v'}$ is statistically close to a random v'' of length ℓ_v as the issuer would choose it in the join protocol if $\ell_r + \ell_f$ are sufficiently large (e.g., $\ell_r + \ell_f > 80$).

$\mathcal{S}'$ sets

$$e := \hat{e} \quad \text{and} \quad A := \hat{A} b_U \bmod n \ .$$

It is easy to verify that

$$A^e \equiv \frac{Z}{US^{v''}} \pmod{n}$$

holds (note that e is odd), i.e., that A satisfies the same equation as it would when is was generated by the issuer. Thus $\mathcal{S}'$ can send (A, e, v'') to the $\mathcal{A}$. Next $\mathcal{S}'$ runs the rest of the join protocol, where it again uses the power over the random oracle and standard zero-knowledge simulation techniques to forge the proof in Step 8. Finally, $\mathcal{S}'$ stores $(f_0, f_1, \mathtt{cnt}_i)$ in its record for $\mathcal{H}_i/\mathcal{M}_i$.

Note that rewinding here is ok, as we assume that the join protocol is run only sequentially.

Simulation of the DAA-Sign Operation/Protocol Unlike as in the case with the simulator $\mathcal{S}$, here this case does not occur as we assumed that all platforms are corrupted.

Simulation of the DAA-Verify Operation/Algorithm If $\mathcal{S}'$ receives a valid signature by the adversary $\mathcal{A}$, it checks whether the value N_V contained in the signature equals $\zeta^{f_0+f_1 2^{\ell_f}}$ modulo Γ for any (f_0, f_1) retrieved in the join protocol. As soon as $\mathcal{S}'$ finds a signature for which there is no such pair, $\mathcal{S}'$ rewinds the adversary to the point in time when he made the call to the random oracle that lead to the c that is part of the signature and provides the adversary with a different c. By a standard argument, this will yield us a second signature, i.e., $\mathcal{S}'$ can extract two signatures w.r.t. the same values for $(\zeta, (T_1, T_2), N_V, (\tilde{T}_1, \tilde{T}_2, \tilde{T}_2'), \tilde{N}_V)$, but with different values for c and possibly for $(s_v, s_{f_0}, s_{f_1}, s_e, s_{ee}, s_w, s_{ew})$, i.e., two signatures

$$(\zeta, (T_1, T_2), N_V, (\tilde{T}_1, \tilde{T}_2, \tilde{T}_2'), \tilde{N}_V, c^{(0)}, (s_v^{(0)}, s_{f_0}^{(0)}, s_{f_1}^{(0)}, s_e^{(0)}, s_{ee}^{(0)}, s_w^{(0)}, s_{ew}^{(0)}, s_r^{(0)}, s_{er}^{(0)}))$$

and

$$(\zeta, (T_1, T_2), N_V, (\tilde{T}_1, \tilde{T}_2, \tilde{T}_2'), \tilde{N}_V, c^{(1)}, (s_v^{(1)}, s_{f_0}^{(1)}, s_{f_1}^{(1)}, s_e^{(1)}, s_{ee}^{(1)}, s_w^{(1)}, s_{ew}^{(1)}, s_r^{(1)}, s_{er}^{(1)}))$$

such that $c^{(0)} \neq c^{(1)}$. Let $\Delta c = c^{(0)} - c^{(1)}$ and $\Delta s_i = s_i^{(1)} - s_i^{(0)}$. From the verification Equations (15-16) and the last one of (18) one can derive the following equations

$$Z^{\Delta c} \equiv T_1^{\Delta s_e + \Delta c 2^{\ell_e}} R_0^{\Delta s_{f_0}} R_1^{\Delta s_{f_1}} S^{\Delta s_v} h^{-\Delta s_{ew}} \pmod{n} \tag{46}$$

$$T_2^{\Delta c} \equiv g^{\Delta s_w} h^{\Delta s_e + \Delta c 2^{\ell_e}} g'^{\Delta s_r} \pmod{n} \tag{47}$$

$$T_2^{\Delta s_e + \Delta c 2^{\ell_e - 1}} \equiv g^{\Delta s_{ew}} h^{\Delta s_{ee}} g'^{\Delta s_{er}} \pmod{n} \tag{48}$$

$$\Delta s_{f_0}, \Delta s_{f_1} \in \pm \{0,1\}^{\ell_f + \ell_\varnothing + \ell_\mathcal{H} + 2} \tag{49}$$

$$\Delta s_e \in \pm \{0,1\}^{\ell_e' + \ell_\varnothing + \ell_\mathcal{H} + 2} \ . \tag{50}$$

From Lemma 4 and Equation (47) it follows that Δc divides both Δs_w and Δs_e. Let $\hat{s}_w = \Delta s_w / \Delta c$ and $\hat{s}_e = \Delta s_e / \Delta c$. Raising Equation (48) to Δc and using Equation (47) we get

$$(g^{\Delta s_w} h^{\Delta s_e + \Delta c 2^{\ell_e - 1}} g'^{\Delta s_r})^{(\Delta s_e + \Delta c 2^{\ell_e - 1})} \equiv g^{\Delta c \Delta s_{ew}} h^{\Delta c \Delta s_{ee}} g'^{\Delta c \Delta s_{er}} \pmod{n} \ . \tag{51}$$

It's not hard to see that

$$\Delta s_w(\Delta s_e + \Delta c 2^{\ell_e - 1}) = \Delta c \Delta s_{ew}$$

must holds as otherwise we could solve the given flexible RSA instance (via retrieving a multiple of the $\phi n/4$ which enables us to compute roots). From this and the definition of $\hat{s}_w$ (i.e., the fact that $\Delta c | \Delta s_w$), we have

$$\Delta s_{ew} = \hat{s}_w(\Delta s_e + \Delta c 2^{\ell_e - 1}) = \hat{s}_w \Delta c(\hat{s}_e + 2^{\ell_e}) \tag{52}$$

with which we can rewrite Equation 46 as follows (also using the definition of $\hat{s}_e$)

$$\Big(\frac{h^{\hat{s}_w}}{T_1}\Big)^{\Delta c(\hat{s}_e + 2^{\ell_e})} \equiv R_0^{\Delta s_{f_0}} R_1^{\Delta s_{f_1}} S^{\Delta s_v} Z^{-\Delta c} \pmod{n} \ . \tag{53}$$

Plugging this equation into Lemma 4, we can derive that Δc divides the Δs_{f_i}'s and Δs_v. Let $\hat{s}_{f_i} = \Delta s_{f_i}/\Delta c$ and $\hat{s}_v = \Delta s_v/\Delta c$. (Actually, from Lemma 4 we can only derive that Δc divides the $\Delta s_{f_i} e_{R_i}$; but, as e_{R_i} are primes that are bigger that Δc, the claim follows.) Now we can conclude from Equation (53) that

$$Z \equiv b_Z R_0^{\hat{s}_{f_0}} R_1^{\hat{s}_{f_1}} S^{\hat{s}_v} \Big(\frac{T_1}{h^{\hat{s}_w}}\Big)^{(\hat{s}_e + 2^{\ell_e})} \pmod{n} \tag{54}$$

for some b_Z such that $b_Z^{\Delta c} = 1$. Now b_Z must be ± 1 for this to happen, because otherwise either $\phi(n)/4 | \Delta c$ or $(b, n) > 1$ and we could factor and thus solve the flexible RSA instance. Let $\hat{b} = 1$ if $\hat{s}_e$ is even and $\hat{b} = -1$ otherwise. Thus we have

$$Z \equiv R_0^{\hat{s}_{f_0}} R_1^{\hat{s}_{f_1}} S^{\hat{s}_v} \Big(\frac{\hat{b} T_1}{h^{\hat{s}_w}}\Big)^{(\hat{s}_e + 2^{\ell_e})} \pmod{n} \tag{55}$$

Moreover, from Equations (49) and (50) it follows that

$$\hat{s}_{f_i} \in [-2^{\ell_f + \ell_\varnothing + \ell_{\mathcal{H}} + 2} + 1, 2^{\ell'_e + \ell_\varnothing + \ell_{\mathcal{H}} + 2} - 1] \text{ and} \tag{56}$$

$$(\hat{s}_e + 2^{\ell_e}) \in [2^{\ell_e} - 2^{\ell'_e + \ell_\varnothing + \ell_{\mathcal{H}} + 2} + 1, 2^{\ell_e} + 2^{\ell'_e + \ell_\varnothing + \ell_{\mathcal{H}} + 2} - 1] \ . \tag{57}$$

We now consider the equations which are the ones involving N_V. From the verification Equation (17) and the second equation of (16) we can derive that

$$N_V^{\Delta c} \equiv \zeta^{\Delta s_{f_0} + \Delta s_{f_1} 2^{\ell_f}} \pmod{\Gamma} \ . \tag{58}$$

We have already established that Δc divides the Δs_{f_i}'s. Furthermore the verification checks (18) guarantees that N_V and ζ lie in the group generated by γ Thus we have

$$N_V \equiv \zeta^{\hat{s}_{f_0} + \hat{s}_{f_1} 2^{\ell_f}} \pmod{\Gamma} \ . \tag{59}$$

This means that we have

$$\hat{s}_{f_0} + \hat{s}_{f_1} 2^{\ell_f} \not\equiv f_0 + f_1 2^{\ell_f} \pmod{\rho}$$

for all (f_0, f_1) that we had extracted from $\mathcal{A}$ via the join protocol. Hence $(\Big(\frac{\hat{b} T_1}{h^{\hat{s}_w}}\Big), (\hat{s}_e + 2^{\ell_e}), \hat{s}_v)$ constitutes a forged signature on the message tuple $(\hat{s}_{f_0}, \hat{s}_{f_1})$. Thus $\mathcal{S}'$ can compute this and feed this message-signature pair to the $\mathcal{S}_{CL}$ oracle to get an solution to the instance of the flexible RSA problem.

Rogue-Tagging Operation. We just receive the f_0, f_1, v, A, and e from $\mathcal{A}$ and feed this as message-signature pair to the $\mathcal{S}_{CL}$ oracle to get an solution to the instance of the flexible RSA problem!

This concludes the proof.

□