DAA-related APIs in TPM2.0 Revisited

Li Xi

Trusted Computing and Information Assurance Laboratory
Institute of Software, Chinese Academy of Sciences
Beijing 100080, China
xili@is.iscas.ac.cn

Abstract. In TPM2.0, a single signature primitive is proposed to support various signature schemes including Direct Anonymous Attestation (DAA), U-Prove and Schnorr signature. This signature primitive is implemented by several APIs which can be utilized as a static Diffie-Hellman oracle. In this paper, we measure the practical impact of the SDH oracle in TPM2.0 and show the security strength of these signature schemes can be weakened by 14-bit. We propose a novel property of DAA called forward anonymity and show how to utilize these DAA-related APIs to break forward anonymity. Then we propose new APIs which not only remove the Static Diffie-Hellman oracle but also support the forward anonymity, thus significantly improve the security of DAA and the other signature schemes supported by TPM2.0. We prove the security of our new APIs under the discrete logarithm assumption in the random oracle model. We prove that DAA satisfy forward anonymity using the new APIs under the Decision Diffie-Hellman assumption. Our new APIs are almost as efficient as the original APIs in TPM2.0 specification and can support LRSW-DAA and SDH-DAA together with U-Prove as the original APIs.

Keywords: TPM2.0; Direct Anonymous Attestation; API

1 Introduction

Direct anonymous attestation(DAA) is a special group signature scheme that enables remote authentication of a trusted platform which contains a valid TPM[1] while preserving the platform's privacy. Basically, DAA protocol allows a trusted platform called signer to sign arbitrary message and send it to a verifier who can be convinced the message is indeed signed by a valid TPM without leaking the signer's identity. A RSA-based direct anonymous attestation (DAA) is proposed by Brickell *et al.* [2]. This RSA-based DAA is adopted by TCG and included in TPM1.2 specification[3]. After that several ECC-based DAA[4–8] are proposed to achieve better performance and shorter signature length. Some of them are now supported by the latest TPM2.0 specification[1]. In TPM2.0, a single TPM signature primitive [9] which can support various signature schemes including DAA and U-Prove is implemented by several DAA-related application programming interfaces (APIs).

An interesting feature of DAA is to provide differing degrees of privacy. While DAA signatures can be totally anonymous, a pseudonymous DAA signature can be linked to another signature by using a specific basename. A DAA signature signed by a trusted platform contains a ticket $t = (J,K = J^k) \in G \times G$, where G is a cylic group, k is the DAA secret key of the trusted platform and $J = \texttt{hash}(basename)$ if $basename \neq \perp$. This ticket is used for linking and rogue tagging: given two signatures, if the two tickets in signatures are the same, then these two signatures are linked. In some DAA schemes, including the scheme adopted by the TPM1.2 specification, the TPM simply gets input J and output J^k to the host, thus can be used as a static Diffie Hellman oracle which significantly reduces the security strength of DAA[10]. A security fix is proposed and is adopted by the TPM2.0 specification[1]. Now in TPM2.0 specification, TPM gets *basename* as input instead of J and calculates $J = \texttt{hash}(basename)$ by itself. It is believed that there is no obvious way that TPM can be used as a static DH oracle even though the security proof of the DAA-related APIs in TPM2.0 specification is still based on the static DH assumption[9]. Unfortunately, Tolga Acar *et al.* [11] show these DAA-related APIs can still be used as static DH oracle. Moreover, we find the security proof [9] is not correct either, which is rather disturbing as DAA is one of the few complex cryptographic protocols deployed in real life. Hundreds of millions of computers have been equipped with TPM.

Another important feature of DAA is that the signer, i.e. the trusted platform, is split into two parts: the TPM part and the host part. The TPM is a low speed hardware chip with high security; the host normally is a X86-based PC equipped with powerful CPU but is easy to corrupt. While the security definitions of user-controlled traceability and non-frameability of DAA give the adversary the ability to compromise the host, all the previous definitions and analyses of anonymity of DAA [12, 13] consider a setting that the host and TPM are both honest. This is easy to understand, because if the host of a trusted platform is already corrupted when signing, it can easily reveal its identity together with the signature, so anonymity of the platform can not be preserved. However, as host is easier to compromise than TPM, the host part of a trusted platform which is honest when signing can be controlled by the adversary later. The adversary can then utilize the APIs provided by the TPM to find out if a given signature was signed by this TPM previously, thus breaks the anonymity. For example, consider a adversary who has gathered DAA signatures sent to a service provider (this is quite reasonable as DAA signatures are not confidential, moreover the adversary can be a malicious service provider itself), by corrupting the host part of a specific user, he may be able to trace all the previous actions of this user, even if the DAA signatures produced by this user are totally anonymous.

1.1 Contribution

In this paper, we provide the following main contributions:

1. We measure the practical impact of the SDH oracle in TPM2.0. We analyze the Barreto-Naehrig (BN) elliptic curves [14] defined in ISO/IEC 15946-5[15] which are recommended by the TPM2.0 specification and show the security strength of DAA can be significantly reduced by about 14-bit in practice.
2. We propose a new property called forward anonymity of DAA. This property assures that even if the host of a trusted platform is corrupted, the anonymity of DAA signatures signed by the platform previously will not be broken. We propose a security definition of forward anonymity based on interactive game and shows attacks against forward anonymity both in the original DAA schemes and in the implementations of these DAA schemes using APIs in TPM2.0 specification.
3. We propose new APIs which not only remove the Static Diffie-Hellman oracle but also support the forward anonymity thus significantly improve the security of DAA. We present *correct* security proof of our new APIs under the discrete logarithm assumption in the random oracle model. We prove that both LRSW-DAA[7] and SDH-DAA[16] satisfy forward anonymity using the new APIs under the Decision Diffie-Hellman assumption. Our new APIs are almost as efficient as the original APIs in TPM2.0 specification and can still support LRSW-DAA and SDH-DAA together with U-Prove[17] as the original APIs.

2 Background

In this section, we briefly introduce the static Diffie Hellman assumption and the DAA-related APIs in TPM2.0.

2.1 Static DH assumption

Definition 1. *Static DH oracle Let G be a cyclic group of prime order n, x is a value in Z_n^*. Given any $p \in G$ as input, the static DH oracle on x outputs p^x.*

Definition 2. *Static DH problem. Let G be a cyclic group of prime order n, x is a value Z_n^*. Given $g, h = g^x \in G$, the static DH problem is to compute x given access to a static DH oracle on x.*

The static DH assumption is that for large n, it is computational infeasible to solve the static DH problem. It is direct to see that static DH assumption is stronger then discrete logarithm assumption. While the static DH problem is still believed as a computational hard problem, the study of Brown and Gallant[18] shows that static DH problem is easier to solve than the discrete log problem:

Theorem 1. *[18] Given a cyclic group $G = \langle g \rangle$, the order of which is $n = uv+1$, a group member $h = g^x$, there is an algorithm that (1) asks the static DH oracle on x u times, (2) performs at most $2(\sqrt{u} + \sqrt{v})$ scalar multiplications in G and 10 simple arithmetic operations on numbers no larger than n and (3) outputs x.*

2.2 The DAA-related APIs in TPM2.0

We briefly recall the DAA protocol and DAA-related APIs in TPM2.0, for details of these APIs, we refer the reader to [9]. The DAA protocol consists of two subprotocols: the join protocol and the sign protocol. In the join protocol, the host use `TPM2_Create` to create the DAA key tsk, the public key is tpk. The host asks for the DAA credential corresponds to tsk by sending tpk to the issuer thus tsk will not be revealed. Using tpk the issuer generates the credential for tsk.

In the sign protocol, given a basename bsn, the host works together with the TPM by calling `TPM2_Commit` and `TPM2_Sign` to generate the DAA signature. In high level, the final DAA signature can be seen as consists of two parts: a ticket $(J, K) = (H_G(bsn), H_G(bsn)^{tsk})$ which is used for linking, and the information that proves (1) the trusted platform has a valid DAA credential and (2) the ticket and the credential are bound with the same DAA secret key tsk. Notice in complex signatures such as DAA, the computation which needs the secret key is only a small part. This part which is generate by `TPM2_Commit` and `TPM2_Sign` is actually a self-contained signature primitive

3 Static DH oracle in TPM2.0

Tolga Acar *et al.* [11] show these DAA-related APIs can be used as static DH oracle. In `TPM2_Commit` API, the input are a group member $P_1 \in \mathbb{G}$ which is a cyclic group with prime order n, and a string $str \in \{0,1\}^*$ that represents the basename, TPM calculates $P_2 = H_G(str)$, the output is $R_1 = P_1^r$, $R_2 = P_2^r$, $K_2 = P_2^x$, x is the DAA secret key. In `TPM2_Sign` API, the TPM get a input c_h, m from the host, calculate $c = H(c_h, m)$, and output (c, s), $s = r + cx$.

Notice that there is no restriction on the first input P_1, if the host is corrupted, he can send whatever he wants. Now the host (adversary) can get $R_1 = P_1^r$ from `TPM2_Commit` and he can then get $(c, s = r + cx)$ from `TPM2_Sign`. Thus he can calculate $Mid = P_1^{cx} = P_1^s / P_1^r$, then he can calculate $P_1^x = Mid^{1/c}$ ($1/c \bmod n$ is easy to calculate as n is a public prime number). Thus TPM can still be used as a static DH oracle, and we believe this is the reason that the security proof can only make through under static DH assumption[9].

3.1 Practical impact of the Static DH oracle

Theoretically, according to result of [18], for $p = uv + 1$, in the worst case, when there exist $u \approx p^{1/3}$, the adversary who controls the host can query TPM u times and uses about $2(\sqrt{u} + \sqrt{v}) \approx O(p^{1/3})$ group operations to solve the static DH problem. For a 256bit p, the adversary can query the TPM about 2^{85} times then solve the discrete log problem with $O(2^{85})$ computations instead of $O(2^{128})$ computations. However, as TPM is a low speed device, and according to the algorithm in [18], the static DH query should be asked sequentially, i.e. the adversary has to obtain g^{k^n} then ask the static DH oracle for $g^{k^{n+1}}$, where k is the DAA key, it may be impractical for the adversary to ask for too many static DH oracles using the known technology.

According to preliminary performance figures in [9], on a discrete 40MHz TPM2.0 chip, a scalar multiplication operation on a 256-bit prime curve takes only 125ms, and according to our benchmark, a scalar multiplication on a 256-bit prime curve takes less than 0.5 ms on a X86-based PC equipped with a Intel i7-3770M CPU at 4x3.4GHz, so it takes less than 130ms to get the answer to a static DH query using the method described in above subsection. The adversary can then ask the static DH oracle about $2.43 \times 10^8 > 2^{27}$ times in one year. So the security strength can be significantly weakened by about 14bit, which means by using 256bit BN curve, the security strength is now only 114 bit instead of the assumed 128 bit.

Notice that in order to utilize the Brown-Gallant algorithm, the adversary has to find a large $u|n-1$, n is the order of the elliptic curve. In the TPM2.0 specification, it is recommended that DAA be implemented using the Barreto-Naehrig (BN) elliptic curve [14] as defined in ISO/IEC 15946-5[15]. We give the factorizations of $n-1$ for the BN curves given in ISO/IEC 15946-5[15] in table 1.

BN256	$2 \cdot 2 \cdot 3 \cdot 7 \cdot 7 \cdot 189239 \cdot 24818737 \cdot 6192533153 \cdot 53176290319 \cdot 127328277910133303695654392417046642892297$
BN224	$2 \cdot 2 \cdot 3 \cdot 13 \cdot 43 \cdot 3539 \cdot 3099193 \cdot 118621 \cdot 21529517 \cdot 105380711 \cdot 247994786597 \cdot 5490314800167041813327$
BN192	$2 \cdot 2 \cdot 3 \cdot 269 \cdot 124427 \cdot 923526871 \cdot 15942266405279489963 \cdot 1061479012505267222401$
BN160	$2 \cdot 2 \cdot 3 \cdot 12132793 \cdot 164442871007 \cdot 448873741399 \cdot 13599345810651634 9$

Table 1. Factorizations of $n-1$ for BN curves

As shown by the table 1, every BN curve in the international standard ISO/IEC 15946-5 which is adopted by TPM2.0 specification has a large $u|n-1$ which is close to 2^{27}. For example, the 160bit BN curve which is supposed to provide 80bit security strength now only provide poorly 66bit security strength which may be easy to break in nowadays. Moreover, as static DH assumption is a non-standard assumption and has not been studied enough, we do not have the confidence that more efficient algorithms will not be found. If a more efficient algorithm is found, the firmware of TPM may need to be updated which is hard to implement. So obviously, the safest solution is to design new DAA-related APIs that can be proved secure under a weaker assumption, for example, the discrete logarithm assumption.

4 Forward Anonymity

In this section we introduce the notion of forward anonymity and show how the adversary can break forward anonymity both in the original DAA schemes and the implementation of DAA using APIs in TPM2.0.

All the previous definitions and analyses of anonymity of DAA consider a setting that the host and TPM are both honest. However, we find a DAA protocol which is proved to be secure under former definitions of anonymity may not be able to resist the following attack: the whole platform is honest when signing a signature, after the signature is signed, the adversary wants to find out whether this signature is signed by the platform, so he corrupts the host and gains information stored in the host and the ability to directly communicate with the TPM. Of course, we assume an honest trusted platform will wipe out all the one-time information used in signing after the signature is produced, including the random number and the signature. With these capabilities, the adversary may be able to find out if the signature was signed by the platform before.

As we know, the host is easier to corrupt than TPM, thus it isn't enough to rely on the security of host to achieve anonymity under high level security requirement. It is promising to guarantee anonymity of signatures only under the assumption that the TPM is honest which is reasonable because TPM is designed to resist software attacks and some kinds of physic attacks.

So we propose forward anonymity. Informally, the notion of forward anonymity requires that the following property is held in the DAA scheme: even after a adversary compromised the host of a trusted platform, he finds it hard to find out if a previous signed anonymous DAA signature(with basename $=\perp$) is signed by this trusted platform as long as the TPM is not corrupted. Notice that generally we can not expect a pseudonymous signature to remain anonymous after the host is compromised, because the adversary with ability to communicate with TPM can always generate a new signature using the same basename as the pseudonymous signature. Thus using function `link`, the adversary can decide if this pseudonymous signature is generate by the platform.

The notion of forward anonymity is defined via a game played by a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ as follows, for simplicity, we assume that for each trusted platform there will be only one DAA secret key:

Initial: $\mathcal{C}$ runs Setup and gives the resulting issuer's secret key `isk` to $\mathcal{A}$. $\mathcal{C}$ publishes the public parameters on a public channel.

Phase 1: The adversary makes the following requests to $\mathcal{C}$:

- Join. $\mathcal{A}$ submits a TPMs identity id to $\mathcal{C}$, who acts as the trusted platform with identity id and executes Join protocol with $\mathcal{A}$.
- Sign. $\mathcal{A}$ submits a TPMs identity id, a message m and a basename bsn to $\mathcal{C}$, who acts as the trusted platform with identity id and execute Sign protocol with $\mathcal{A}$ using message m and basename bsn.
- API. $\mathcal{A}$ submits a TPMs identity id along with the name of the API he wants to use, for example, `TPM2_Commit`, and the data used in calling the API of his choice to $\mathcal{C}$, who acts as the TPM with identity id and responds with the output of the API. Also the information stored inside the host part of this trusted platform will be output to the adversary.
- Corrupt. $\mathcal{A}$ submits a TPMs identity id to $\mathcal{C}$, who response with the DAA secret key created by this TPM.

Challenge: At the end of Phase 1, $\mathcal{A}$ chooses two TPMs' identities id_0 and id_1, submits the two identities and a message m of his choice to $\mathcal{C}$. $\mathcal{A}$ must not have made any API query or Corrupt query on either id_0 or id_1. For simplicity, we assume that $\mathcal{A}$ has already made Join query on id_0 and id_1, $\mathcal{C}$ chooses a bit b uniformly at random, produce a signature as platforms with identity id_b using m and $bsn = \perp$, then output signatures to the adversary.

Phase 2: The adversary can do what he can in phase 1 except that he can not make any Corrupt query on either id_0 or id_1, notice now he can make API query using id_0 and id_1.

Guess: $\mathcal{A}$ returns a bit b', the advantage of $\mathcal{A}$ is $\mathbf{Adv}(\mathcal{A}) = |\mathbf{Pr}(b = b') - 1/2|$. We say that a DAA scheme satisfies forward anonymity if for any probabilistic polynomial-time adversary $\mathcal{A}$, $\mathbf{Adv}(\mathcal{A})$ is negligible.

In both LRSW-DAA [7] and SDH-DAA[16], even the totally anonymous DAA signature contains a tuple (J, K), $K = J^k$, which is used for rogue tagging. When signing a totally anonymous signature, the host will choose a random string str, then TPM will calculate $J = \texttt{hash}(str)$ and $K = J^k$. Notice that it is unnecessary to include the random string str in the signature, and to achieve forward anonymity, it actually should not be include in the signature. Otherwise after the host is compromised, the adversary can use the string str and the API `TPM2_Commit` to reconstruct the tuple (J, K), thus can decide whether the signature is signed by this TPM.

Attacks against Forward Anonymity In the original LRSW-DAA, $(c, s) = PK\{(k) : K_1 = P_1^k \wedge K_2 = P_2^k\}(m, m_h)$ is generated by a single procedure, the random commitments R_1, R_2 are not outputted to the host. In TPM2.0, this procedure is splitted into two parts: `TPM2_Commit` and `TPM2_Sign` in order to support various signature, especially the SDH-DAA. However our analysis in section 3 shows splitting the procedure causes the TPM can be used as static DH oracle. In TPM2.0, DAA protocols can not satisfy forward anonymity because given a challenge DAA signature which contains a ticket (J, K), the adversary utilize the TPM as the static DH oracle and get $K' = J^k$, k is the DAA secret key. By checking whether $K' = K$, the adversary can find out whether the signature is signed by this TPM.

It is worth noting that in the original LRSW-DAA[7] (of course we move the calculation $J = \texttt{hash}(basename)$ inside the TPM according to [10]), even though now there is no obvious way that TPM being used as a static DH oracle, the forward anonymity still can not be preserved. Given a TPM and a challenge DAA signature which contains a ticket $(\hat{J}, \hat{K})$, $\hat{K} = \hat{J}^{k^*}$ where k^* is the DAA secret key this signature is signed under, the adversary generates a basename bsn and output $(\hat{J}, bsn)$ to the TPM. The TPM will output a ticket $(J = H_G(bsn), K = J^k)$ and a proof of knowledge $(c, s) = PK\{(k) : K = J^k \wedge K' = \hat{J}^k\}$, now K' is not known by the adversary. Notice $(c, s) = PK\{(k) : K = J^k \wedge K' = \hat{J}^k\}$ can only be verified by two pair $(J, K = J^k)$ and $(\hat{J}, K' = \hat{J}^k)$. The adversary can now use the ticket (J, K) and the ticket $(\hat{J}, \hat{K})$ in the challenge DAA signature to verify (c, s), if (c, s) is verified, then the challenge DAA signature is signed by this TPM. This attack works because TPM does not check the first entry of

input $(\hat{J}, bsn)$. So our analyses of static DH oracle and forward anonymity both show there should be restriction on the first parameter of input ($P_1 \in G_1, bsn \in \{0,1\}^*$).

5 The new DAA-related API

5.1 Fix the DAA-related APIs to satisfy forward anonymity and to remove the Static DH oracle

Our target is to revise the APIs without adding much cost and retain the capabilities of the APIs, i.e. the revised APIs should still support LRSW-DAA, SDH-DAA and U-Prove. As pointed out above, there should be restriction on the first input of the `TPM2_Commit`. Actually, we step a little further, we will bind $P_1 \in G$ which is the first input of the `TPM2_Commit` with the DAA key $(g, y = g^x)$: now P_1 can only be g or a fixed group member $P \in G_1$ which is bound to the DAA key. Before using `TPM2_Commit` and `TPM2_Sign` to generate DAA signatures, if the P_1 needed by `TPM2_Commit` is different from g, then the host should first output the P_1 to the TPM, then the TPM will bind P_1 with the DAA key. After binding, the API `TPM2_Commit` can only use the *same* bound group member P_1 or g as input.

We propose a new command `TPM2_DAAbind` to bind $P_1 \in G_1$ with the DAA secret key k. Let the public key of k is $(g, y = g^k)$, the input of this command is a pair (P_1, K_1) together with prove of knowledge $PK\{(r) : K_1 = y^r \wedge P_1 = g^r\}$ which is generate by the issuer (the join protocol of LRSW-DAA in TPM2.0 is different from the original scheme[7], it actually adopted the join protocol in [8]: the issuer generates a DAA credential together with a proof of knowledge). The TPM checks the signature prove of knowledge $PK\{(r) : K_1 = h^r \wedge P_1 = g^r\}$ and bind P_1 with the DAA key. Only after the DAA key is bound with a group member $P_1 \neq g$, the host can call the API `TPM2_Commit` using the *fixed* P_1 as a input.

Now we describe the new TPM signature primitive, denoted by `tpm.sign`*.

Key Generation (`TPM2_Create`): The TPM generates a random number k and computes $y = g^k$. The secret key is k, the public key is y.

DAA Binding (`TPM2_DAAbind`): Given a pair (P_1, K_1) and a proof of knowledge $PK\{(r) : P_1 = g^r \wedge K_1 = y^r\}$, the TPM verifies the proof of the knowledge $PK\{(r) : P_1 = g^r \wedge K_1 = y^r\}$, if it is right, then bind k with P_1.

Signing:

Commit Oracle(`TPM2_Commit`)Given $P_1 \in G, l \in Z_P, str \in \{0,1\}^*$ as input:

1. Verify that if $P_1 = g$ or P_1 has been bound with k, if both not, abort.
2. If $str = \emptyset$, set $P_2 = 1$, otherwise, compute $P_2 := H_G(str)$
3. Choose a random integer $r \leftarrow Z_p$.
4. Compute $R_1 = P_1^{lr}$,$R_2 = P_2^r$, and $R_2 = P_2^k$ where k is the private key.

5. Output R_1, R_2, K_2 and a counter ctr. The random number r is bound with ctr, the counter is then increased by 1.

Sign Oracle(`TPM2_Sign`) Given c_h, m and a counter number ctr as input.

1. Generate a nonce n_T, compute $c := H(c_h, m, n_T)$.
2. According to the counter ctr finds the corresponding r, compute $s := r + ck \bmod p$, delete r.
3. Output(c, s, n_T).

Verification: Given a signature $(m, m_h, P_1' = P_1^l, P_2, R_1, R_2, K_2, c, s, n_T)$, $K_1 = P_1'^k$ and a collision free function F, the verification proceeds as follows:

1. Verify that $P_1 \neq 1$.
2. Verify that $H(F(R_1, R_2, m_h), m, n_T) = c$.
3. Verify that $R_1 = P_1'^s \cdot K_1^{-c}$ and $R_2 = P_2^s \cdot K_2^{-c}$.

Our `TPM2_Commit` is slightly different from the original one, as now $R_1 = P_1^{lr}, l$ is an input number. This difference guarantees that the implementations of DAA protocols and U-Prove using our new APIs is as efficient as using the original APIs which will be discussed thoroughly in section 6.

5.2 Security Proof of the New `tpm.sign`*

We first point out the mistake in the security proof of the original DAA-related APIs in TPM2.0[9] then we present *correct* security proof of our new APIs under the discrete logarithm assumption in the random oracle model.

Mistake in the security proof of DAA-related APIs in TPM2.0 In the proof[9], the simulator $\mathcal{B}$ does not provide the hash query H for the adversary $\mathcal{A}$. Actually, after making a `TPM2_Commit` query, $\mathcal{A}$ can first query H n times using arbitrary pairs (c_h, m_i), $i \in [0, n]$ and gets answers $h_i = H(c_h, m_i)$, then call the `TPM2_Sign` using one pair (c_h, m_j), $j \in [0, n]$, however, the simulation of `TPM2_Sign` have to output the (c, s) which is already fixed in the simulation of `TPM2_Commit`. So now $c = H(c_h, m_j) \neq h_j$ which means the simulation fails.

In our new API `TPM2_Sign`, the TPM generates a nonce n_T. By adding a nonce n_T, we fix this problem: now $c = H(c_h, m_j, n_T)$ and $\mathcal{A}$ can not obtain n_T before calling `TPM2_Sign` as n_T is a newly generated nonce.

Security Proof of the our new APIs We prove the security of our new `tpm.sign`* using the standard security notion of signature schemes which is existential unforgeability under adaptive chosen message attacks (EUF-CMA). In EUF-CMA model, the attacker is allowed to query the signing oracle adaptively. In our case, it means the attacker is allowed to call the APIs adaptively, i.e, he can call `TPM2_DAAbind` (bind oracle), `TPM2_Commit` (commit oracle), `TPM2_Sign` (sign oracle) as he wishes.

Definition 3. *The* `tpm.sign` *scheme is said to be existentially unforgeable under adaptive chosen message attacks if there is no probabilistic polynomial-time adversary $\mathcal{A}$ with non-negligible advantage in the following game played with a challenger $\mathcal{C}$:*

Initial: $\mathcal{C}$ *runs Setup and call* `TPM2_Create` *to create the secret key.* $\mathcal{C}$ *sends systems public parameters params and public key to* $\mathcal{A}$

Queries: The adversary $\mathcal{A}$ *adaptively makes API queries as he wishes.*

Forgery: The adversary $\mathcal{A}$ *produces a pair* (m^*, σ^*)*, notice* m^* *should not be called in* `TPM2_Sign` *queries. The adversary* $\mathcal{A}$ *wins if* σ^* *is a valid signature.*

Our security proof is based on the well-known forking lemma [19] which applies to signatures with the form (σ_1, h, σ_2). Here σ_1 are random commitments; $h = H(\sigma_1, m)$ where H is a hash function. In `tpm.sign` primitives, there is a slight difference: $h = H(F(\sigma_1, m_1), m_2)$ where F is a collision free function. It is direct to see the forking lemma still holds for `tpm.sign`, actually, we can see $H(F)$ as a new hash function, as F is collision free.

Theorem 2. *The new* `tpm.sign`* *is existentially unforgeable under adaptive chosen message attacks in the random oracle model under the DL assumption.*

Proof: If there is an adversary $\mathcal{A}$ that breaks the new `tpm.sign`* scheme, i.e. $\mathcal{A}$ outputs a forged signature $(m, m_h, P_1, P_2, R_1, R_2, K_2, c, s, n_T)$ after given arbitrary access to `TPM2_DAAbind`, `TPM2_Commit` and `TPM2_Sign`, then there exists an algorithm $\mathcal{B}$ which utilize $\mathcal{A}$ to solve the DL assumption. $\mathcal{B}$ is given a pair $(g, h = g^x)$, and $\mathcal{B}$ wants to compute x. Algorithm $\mathcal{B}$ works as follows:

key generation(`TPM2_Create`): $\mathcal{B}$ sets h as the public key and outputs it to $\mathcal{A}$ and sets $log_g h$ as the corresponding private key x, although $\mathcal{B}$ does not know x.

Bind Query(`TPM2_DAAbind`): Given a pair (P, K) and a proof of knowledge $PK\{(r) : P = g^r \wedge K = h^r\}$. $\mathcal{B}$ verifies the proof of the knowledge $PK\{(r) : P = g^r \wedge K = h^r\}$, if it is right, stores (P, K). Due to the soundness of proof of knowledge, we have $K = P^x$.

Hash Query: There are two hash functions modelled as random oracles: H_G and H.

- H_G: Given a input str, if the str is not queried before, $\mathcal{B}$ generates a random number r, calculated $H_G(str) := g^r$, store $(H_G(str) = g^r, r)$ in the hash list of H_G, if the str has been queried, return the former answer.
- H: Given a input x, if x has been queried before, return the former answer. If x has not been queried, choose a random number r and add the (x, r) to H's hash list.

Commit Query(`TPM2_Commit`): If $\mathcal{A}$ makes a commit query with (P_1, l, str) as input, $\mathcal{B}$ first check if P_1 is equal to g or has been bound to the DAA key using `TPM2_DAAbind`, if not, returns fail. $\mathcal{B}$ calls the H_G oracle to get $P_2 = H_G(str) = g^r$ and the random number r. Now $\mathcal{B}$ knows $K_1 = P_1^x$ and $K_2 = h^r = P_2^x$. $\mathcal{B}$ chooses at random c and s and computes $R_1 := P_1^{ls} \cdot K_1^{-lc}$ and $R_2 := P_2^s \cdot K_2^{-c}$. $\mathcal{B}$ outputs (R_1, R_2, K_2) and a counter number ctr. Then $ctr = ctr + 1$. It is direct to see this simulation of commit query is perfect.

Sign Query(`TPM2_Sign`): If $\mathcal{A}$ makes a sign query on m, the input are (c_h, m) and a counter number ctr, if ctr is not queried before, $\mathcal{B}$ generates a random

number n_T, and sets $c := H_2(c_h, m, n_T)$, store $((c_h, m, n_T), c)$ into the hash list of H, then output the (c, s, n_T), ctr is marked as used. Notice the failure only occurs if $H(c_h, m, n_T)$ has been queried before and the answer is $c' \neq c$. However, it is direct to see the chance this failure happens is negligible, because n_T is a newly generated nonce by the TPM.

Forgery: $\mathcal{A}$ produces a signature $(m, m_h, \tilde{P_1}, \tilde{P_2}, \tilde{R_1}, \tilde{R_2}, \tilde{K_2}, c, s, n_T)$. Using the forking lemma, we can use $\mathcal{A}$ to output two signature $\sigma_1 = (m, m_h, \tilde{P_1}, \tilde{P_2}, \tilde{R_1}, \tilde{R_2}, \tilde{K_2}, c, s, n_T)$ and $\sigma_2 = (m, m_h, \tilde{P_1}, \tilde{P_2}, \tilde{R_1}, \tilde{R_2}, \tilde{K_2}, c', s', n'_T)$. We have $\tilde{R_1} \cdot \tilde{K_1}^{c} = \tilde{P_1}^{s}$ and $\tilde{R_1} \cdot \tilde{K_1}^{c'} = \tilde{P_1}^{s'}$. So $\tilde{K_1}^{c-c'} = \tilde{P_1}^{s-s'}$, thus we can calculate the discrete logarithm $x = (s - s')/(c - c')$. Notice calculating $x = (s - s')/(c - c')$ does not need to know $\tilde{K_1} = \tilde{P_1}^{x}$.

Therefore, under the discrete logarithm assumption, the new `tpm.sign`* is secure.

6 Applications and Implementation of the new TPM.Sign APIs

Now we show how to use our modified DAA APIs to implement LRSW-DAA[7], SDH-DAA[16]and U-Prove[17]. Then we present how our APIs can be implemented, particularly `TPM2_DAAbind`. Details about how to use the original APIs to implement LRSW-DAA, SDH-DAA and U-Prove can be found in [9].

6.1 Applications of the New APIs: DAAs and U-Prove

The LRSW-DAA protocol

Join: The host calls `TPM2_Create` to get the `tpk` $= g^{tk}$. Based on `tpk` the issuer sends the DAA credential (A, B, C, D) which is a CL-LRSW signature on tk together with a proof of knowledge $\sigma_I = PK\{(r) : B = g^r \wedge D = \texttt{tpk}^r\}$. The host binds B with the DAA key tk by calling `TPM2_DAAbind`(B, D, σ_I) and stores the credential (A, B, C, D).

Sign: Given a nonce n_V, a message m, the host generate a random number $l \in Z_p$, calls `TPM2_Commit`(B, l, bsn). The TPM first checks that B is bound with tk, then outputs $R_1 = B^{lr}, R_2 = H_G(bsn)^r, K = H_G(bsn)^{tk}$. The host then uses the random number l to randomize the credential: $(R, S, T, W) = (A^l, B^l, C^l, D^l)$, notice that $R_1 = S^r$. The host calculate $c_h = H_3(R, S, T, W, J = H_G(bsn), K, R_1, R_2, n_V)$ and calls `TPM2_Sign`(c_h, m) to get a TPM signature $(c, s_f, n_T) = PK\{(k) : W = S^{tk} \wedge K = J^{tk}\}$. The final DAA signature is $(R, S, T, W, J, K, c, s_f, n_v, n_T)$.

The SDH-DAA protocol

Join: The join process is almost the same as LRSW-DAA except that there is no need for the host to execute `TPM2_DAAbind` because when signing the first input of `TPM2_Commit` will always be g which is part of the DAA public key.

Sign: Given a nonce n_V, a message m, the host calls `TPM2_Commit`$(g, 1, bsn)$. The TPM first checks that g is part of the public key, then outputs $R_1 = g^r, R_2 = H_G(bsn)^r, K = H_G(bsn)^{tk}$. The host generate random numbers $a, r_x, r_a, r_b \leftarrow Z_p$, calculate $b = ax \ mod \ p$, $T = Ah_2^a$, then uses R_2 to generate the random commitments used in the final proof of knowledge: $R_2' = e(R_2 T^{-r_x} h_2^{r_b}, g_2) e(h_2, w)^{r_a}$. The host calculate $c_h = H_3(J = H_G(bsn), K, T, R_1, R_2', n_V)$ and calls `TPM2_Sign`(c_h, m) to get a TPM signature (c, s_f, n_T). The host calculate $s_x = r_x + cx$, $s_a = r_a + ca$, $s_b = r_b + cb$, the final DAA signature is $(J, K, T, c, s_f, s_x, s_a, s_b, n_v, n_T)$.

U-Prove using the New APIs U-Prove is a pseudonym system based on the blind signatures and zero-knowledge proofs. In the U-Prove 1.1 specification, it is proposed that a U-Prove token can be protected by a hardware device. By using a hardware device, the leaked U-Prove token still can not be used unless the hardware device is also controlled by the adversary. Moreover, the hardware device can produce a ticket which is the same as in the DAA protocol to provide user-controlled linkability: the tickets generated by the same device key and basename are the same.

Using our new APIs to protect the U-Prove token is almost the same as using the original APIs[9]. When the input of the original `TPM2_Commit` is (g_d, str), the input of our new API `TPM2_Commit` is now $(g_d, 1, str)$. There is no need to execute `TPM2_DAAbind` as g_d is part of the TPM public key.

6.2 Implementation and efficiency analysis

We have two targets: first the runtime performance of various protocols using the new `tpm.sign`[8] should still be as good as using the original `tpm.sign`, second the revised API should still be easy to be implemented in the TPM, as TPM is just a cheap chip.

Notice our newly added API actually have little influence on the run time performance of protocols including various DAA schemes and U-Prove. The host has only to run the new command `TPM2_DAAbind` once after the join protocol and then he can call `TPM2_Commit` and `TPM2_Sign` an arbitrary number of times. The revise API `TPM2_Commit` is almost as efficient as the original `TPM2_Commit`: in the new API `TPM2_Commit`(P_1, l, str), the only operation added is now the TPM need to calculate $l \cdot r$ first, where $l, r \in Z_p^*$. Calculating $l \cdot r$ takes much less time than calculating a point multiplication in an elliptic curve. In our new API `TPM2_Sign`, the only operation added is generating a nonce n_T which is also very efficient. The computational workload of the host part using our APIs is the same as using the original APIs.

Implementing the new command `TPM2_DAAbind` will not add much cost to TPM as it only uses operation in G_1 which has already be implemented in TPM for supporting `TPM2_Commit`. Moreover, TPM2.0 supports ECC Schnorr signature validation while the verification of $PK\{(r) : P_1 = g^r \wedge K_1 = y^r\}$ in `TPM2_DAAbind` is almost the same as the verification of Schnorr signature. The

binding of P_1 with the DAA key is also easy to implement. Notice TPM2.0 is able to protect the integrity of the key object by using a key hierarchy, so what we need to do is just adding the P_1 into the key object. Now the key object contains a new entry called `daabind`. When the object (DAA key) is not bound to any $P_1 \in G_1$, `daabind` equals to zero; if the object has been bound to a P_1, `daabind` equals to P_1. The detail of `TPM2_DAAbind` is as follows, we adopt the notation in [9]:

1. Given a DAA key pair $\mathtt{tk} = (\mathtt{tpk} = (\mathtt{g}, \mathtt{h}), \mathtt{tsk})$, where `tpk` is the public key and `tsk` is the secret key, before `TPM2_DAAbind` is executed, the key blob of `tk` is $\mathtt{tk}^* = (\mathtt{tsk})_{SK} \parallel \mathtt{tpk} \parallel MAC_{MK}((\mathtt{tsk})_{SK} \parallel \mathtt{tpk}.name)$. $\mathtt{tpk}.name$ is a message digest of the public portion of `tk`. The integrity of the key object `tk` is protected by a message authentication code (MAC) using a MAC key MK and the secret key `tsk` is encrypted by a secret key SK, both SK and MK are derived from the parent key of `tk`: $(SK, MK) = KDF(\mathtt{parentK})$. Thus the secrecy and integrity of `tk` is protected by the key hierarchy.
2. Given a pair (P, K) and a proof of knowledge $PK\{(r) : P = g^r \wedge K = h^r\}$. TPM verifies the proof of the knowledge $PK\{(r) : P = g^r \wedge K = h^r\}$, if it is right, generates a new key blob $\mathtt{tk}_n^*$ which binds `tk` with P as follows. First TPM pads the `tpk`: $\mathtt{tpk}_n \leftarrow \mathtt{tpk} \parallel P$, then generate the new key name $\mathtt{tpk}_n.name = \mathtt{hash}(\mathtt{tpk}_n)$, finally generate the MAC for binding: $MAC_{MK}((\mathtt{tsk})_{SK} \parallel \mathtt{tpk_n}.name)$. TPM outputs the new key blob $\mathtt{tk}_n^* = (\mathtt{tsk})_{SK} \parallel \mathtt{tpk_n} \parallel MAC_{MK}((\mathtt{tsk})_{SK} \parallel \mathtt{tpk}_n.name)$.

7 Forward Anonymity in the New TPM.Sign scheme

In this section, we prove that both LRSW DAA and SDH DAA satisfy forward anonymity using the new APIs under the Decision Diffie-Hellman assumption.

Theorem 3. *Under the $G_1 - DDH$ assumption, the implementation of LRSW-DAA using the new $\mathtt{tpm.sign}^*$ satisfies forward anonymity. More specifically, if there is an adversary A that succeeds with a non-negligible probability to break the forward anonymity game, then there is a polynomial-time algorithm B that solves the $G_1 - DDH$ problem with a non-negligible probability.*

Proof: If there exists a adversary $\mathcal{A}$ that breaks the forward anonymity of LRSW-DAA, then we can build a polynomial-time simulator $\mathcal{B}$ that breaks the $G_1 - DDH$ problem as follows. The input to $\mathcal{B}$ is a tuple $(u, v = u^a, w = u^b, z = u^c) \in G_1 \times G_1 \times G_1 \times G_1$, where (a, b) are independent uniform random elements in Z_p, and either $c = ab$ or c is also a independent uniform random element in Z_p. By interacting with $\mathcal{A}$, $\mathcal{B}$ wants to find out whether c equals ab or c is a random element.

We first give a overview of the security proof. $\mathcal{B}$ first select a special trusted platform S^*, the secret key f of which is $a = log_u v$, however $\mathcal{B}$ does not know a. $\mathcal{B}$ creates the other trusted platform by honestly executing the Join protocol with $\mathcal{A}$. $\mathcal{B}$ uses the pair (u, v) to simulate the answers to queries about the trusted

platform S^*. In the challenge phase, if $\mathcal{A}$ select S^* as one of the two challenge platform S_1, S_2, then $\mathcal{B}$ choose the bit b so that $S^* = S_b$, and calculate the challenge signature sig_c using the pair (w, z), so if $log_w z = log_u v$ then sig_c is a valid DAA signature signed by S^* and $\mathcal{A}$ should have advantage in deciding b, if $log_w z \neq log_u v$ then sig_c is actually a valid DAA signature signed under the DAA secret key $b^{-1}c$ which is independent of S_1 and S_2, so $\mathcal{A}$ can no have any advantage guessing b or may simply abort the game. So $\mathcal{B}$ can utilize $\mathcal{A}$ to judge whether $c = ab$. $\mathcal{B}$ works as follows:

Setup: choose (G_1, G_2) as a bilinear group pair of prime order p with generator $g_1 = u$ and g_2 respectively and a bilinear paring $e : G_1 \times G_2 \rightarrow G_T$. The gpk is $G_1, G_2, G_T, e, g_1 = u, g_2, p, H_G, H, H_3$, in which H_G is used by TPM in `TPM2_Commit`, H is used by TPM in `TPM2_Sign`, H_3 is used by host to generate c_h. $\mathcal{B}$ choose two random number $x, y \leftarrow Z_P$ as the issuer's secret key, and calculate $X = g_2^x, Y = g_2^y$ as the issuer's public key. $\mathcal{B}$ sends issuer's secret key and gpk to $\mathcal{A}$.

Hash queries: H_G and H are modeled as random oracles. $\mathcal{B}$ response to the hash queries about H_G and H the same as in the proof of theorem 2.

`TPM2_Create`: Given a TPM's identity T_i, If $T_i \neq T^*$ then $\mathcal{B}$ honestly execute this API, i.e. generates a secret key x and sets the public key $Y = u^x$; if $T_i = T^*$, $\mathcal{B}$ sets v as the public key and outputs it to $\mathcal{A}$ and sets $log_u v$ as the corresponding private key x, although B does not know x.

`TPM2_DAAbind`: $\mathcal{B}$ act the same as in the proof of theorem 2 .

`TPM2_Commit` queries: Given a TPM's identity T_i, $P_1 \in G_1, bsn \in \{0,1\}^*, l \in Z_p$ as input, If $T_i \neq T^*$ then $\mathcal{B}$ honestly execute this API using the DAA secret key; if $T_i = T^*$, $\mathcal{B}$ act the same as in the proof of theorem 2.

`TPM2_Sign` queries: Given a TPM's identity T_i, if $T_i \neq T^*$ then $\mathcal{B}$ honestly execute this API using the DAA secret key; if $T_i = T^*$, $\mathcal{B}$ act the same as in the proof of theorem 2.

Join queries: Given a trusted platform's identity T_i, $\mathcal{B}$ honestly execute the Join protocol with $\mathcal{A}$ by calling `TPM2_Create`, `TPM2_Commit` and `TPM2_Sign` queries described above, $\mathcal{B}$ stores the DAA credential obtained from $\mathcal{A}$.

Sign queries: Given a trusted platform's identity T_i, a message m, a nonce n_V, a basename bsn, $\mathcal{B}$ extracts the credential (A, B, C, D) then execute the Sign protocol by calling the `TPM2_Commit` query and `TPM2_Sign` query together to generate a valid DAA signature

Corrupt queries: Given a trusted platform's identity T_i, If $T_i \neq T^*$ then $\mathcal{B}$ outputs the DAA secret key. If $T_i = T^*$ then output "**Abort 1**".

Challenge: In the challenge phase, $\mathcal{A}$ output a message m, nonce n_V, two trusted platforms' identities T_1 and T_2. If $T^* \notin \{T_1, T_2\}$, then $\mathcal{B}$ quits and outputs "**Abort 2**". Otherwise, $\mathcal{B}$ chooses bit b such that $T_b = T^*$, and generate the challenge signature σ^* using w, z as follows:

1. $\mathcal{B}$ chooses a random $r \leftarrow Z_p$ and sets $J := w^r$ and $K := z^r$.
2. $\mathcal{B}$ generate the CL-LRSW credential $(A, B, C, D) \leftarrow (w, w^y, w^x \cdot v^{xy}, v^y)$ for (w, v). It is direct to see (A, B, C, D) is a valid CL-LRSW signature for $log_w v$.

3. Using a random number $l \leftarrow Z_p$, $\mathcal{B}$ calculate $(U, S, T, W) = (A^l, B^l, C^l, D^l)$.
4. $\mathcal{B}$ chooses at random c and s and computes $R_1 := S^s \cdot W^{-c}$ and $R_2 := J^s \cdot K^{-c}$. $\mathcal{B}$ calculate $c_h = H_3(R, S, T, W, n_V, R_1, R_2, J, K)$, generate a nonce n_T and set $H(c_h, m, n_T) = c$. If $H(n_T, c_h, m)$ has been queried before, $\mathcal{B}$ quits and outputs "**Abort 0**".

The final challenge signature is $(R, S, T, W, J, K, c, s, n_T, n_V)$. It is direct to see if $log_w(z) = log_u(v)$ then the challenge signature is a valid signature of trusted platform T^*; if not, then the challenge signature is a valid signature corresponds to DAA secret key $log_w(z)$ which is independent of $\{T_1, T_2\}$, so $\mathcal{A}$ can not have any advantage in deciding b.

Output. In the end, $\mathcal{A}$ outputs $b \in \{0, 1\}$ as the guess for b or aborts without any output. If $b = b$, then $\mathcal{B}$ outputs 1, which means that $z = u^{ab}$. Otherwise $\mathcal{B}$ outputs 0, which means that z is a random element in G_1.

We now analysis the probability that $\mathcal{B}$ does not abort in the above game. There are three case that $\mathcal{B}$ may abort, we discuss each case as follows:

Abort 0 The chance that this type abortion happens is $O(1/p)$, where p is a large prime, so the probability is negligible.

Abort 1 Notice that $\mathcal{A}$ can not corrupt all the trusted platforms, so the probability that this abortion does not happen is at least $1/q_j$.

Abort 2 This abortion does not happen if in the challenge phase, $\mathcal{A}$ chooses T^* as one of the $\{T_1, T_2\}$, so the probability that this abortion does not happen is at least $1/q_j$.

So the the probability that $\mathcal{B}$ does not abort in the above game is at least $1/q_j$, (if the **Abort 2** does not happen, then **Abort 1** should not happen, as the trusted platforms using in challenge phase must not be corrupted).

Let ε be the advantage of $\mathcal{A}$ in breaking the forward anonymity game. Assume that $\mathcal{B}$ does not abort when simulating the above game, if $z = u^{ab}$, then $\mathcal{B}$ simulate the game perfectly, so $\mathcal{A}$ will still have the same advantage in winning the forward anonymity game. If z is a random member in G_1, then the challenge signature is a valid signature corresponds to DAA secret key $log_w(z)$ which is independent of $\{T_1, T_2\}$, so $\mathcal{A}$ can not have any advantage in deciding b. So if $\mathcal{B}$ does not abort, then he can break the $G_1 - DDH$ assumption with probability at least $\varepsilon/2$.

Theorem 4. *Under the $G_1 - DDH$ assumption, the implementation of SDH-DAA using the new* `tpm.sign`* *satisfies forward anonymity. More specifically, if there is an adversary $\mathcal{A}$ that succeeds with a non-negligible probability to break the forward anonymity game, then there is a polynomial-time algorithm $\mathcal{B}$ that solves the $G_1 - DDH$ problem with a non-negligible probability.*

Proof: The basic idea of this proof is analogous to that of theorem 3.

8 Conclusion

In TPM2.0, a single signature primitive is proposed to support various signature schemes including Direct anonymous attestation(DAA), U-Prove and Schnorr signature. In order to to support various signature schemes, this signature primitive is splitted into two parts (`TPM2_Commit` and `TPM2_Sign`) and there is no restriction on the input of `TPM2_Commit`. However, this gives too much ability to the outside, thus these APIs can be utilized as static Diffie-Hellman oracle and forward anonymity can not be satisfied. We propose new APIs which not only remove the Static Diffie-Hellman oracle but also support the forward anonymity thus significantly improve the security of DAA and the other signature schemes supported by TPM2.0. Our new APIs are almost as efficient as the original APIs in TPM2.0 specification and can support LRSW-DAA and SDH-DAA together with U-Prove as the original APIs. We believe our research actually shows the importance of reducing the potential attack surface, i.e limiting the ability provided to the outside, the ability should be just sufficient to be functional.

References

1. Group, T.C.: Tcg tpm specification 2.0, https://www.trustedcomputinggroup.org (2012)
2. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Proceedings of the 11th ACM conference on Computer and communications security, ACM (2004) 132–145
3. Group, T.C.: Tcg tpm specification 1.2, https://www.trustedcomputinggroup.org (2003)
4. Brickell, E., Chen, L., Li, J.: A new direct anonymous attestation scheme from bilinear maps. In: Trusted Computing-Challenges and Applications, Springer Verlag (2008) 166–178
5. Chen, X., Feng, D.: Direct anonymous attestation for next generation tpm. Journal of Computers **3**(12) (2008) 43–50
6. Chen, L., Morrissey, P., Smart, N.P.: Daa: Fixing the pairing based protocols. Technical report, Cryptology ePrint Archive, Report 2009/198 (2009)
7. Chen, L., Page, D., Smart, N.: On the design and implementation of an efficient daa scheme. In: Smart Card Research and Advanced Application, Springer-Verlag (2010) 223–237
8. Brickell, E., Chen, L., Li, J.: A (corrected) daa scheme using batch proof and verification. In: Trusted Systems. Springer (2012) 304–337
9. Chen, L., Li, J.: Flexible and scalable digital signatures in tpm 2.0. In: CCS '13:Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM (2013) 37–48
10. Brickell, E., Chen, L., Li, J.: A static diffie-hellman attack on several direct anonymous attestation schemes. In: Trusted Systems. Springer (2012) 95–111
11. Acar, T., Nguyen, L., Zaverucha, G.: A tpm diffie-hellman oracle. Technical report, Cryptology ePrint Archive: Report 2013/667 (2013)
12. Brickell, E., Chen, L., Li, J.: Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. International Journal of Information Security **8**(5) (2009) 315–330

13. Chen, L.: A daa scheme requiring less tpm resources. In: 5th International Conference on Information Security and Cryptology. Volume 6151 of LNCS., Springer (2011) 350–365
14. Barreto, P.S., Naehrig, M.: Pairing-friendly elliptic curves of prime order. In: Selected areas in cryptography, Springer (2006) 319–331
15. ISO/IEC: Iso/iec 15946-5:2009 information technology – security techniques – cryptographic techniques based on elliptic curves – part 5: Elliptic curve generation
16. Brickell, E., Li, J.: A pairing-based daa scheme further reducing tpm resources. In: Trust and Trustworthy Computing. Springer (2010) 181–195
17. Microsoft: U-prove cryptographic specification v1.1, http://www.microsoft.com/u-prove (2013)
18. Brown, D.R., Gallant, R.P.: The static diffie-hellman problem. IACR Cryptology ePrint Archive **2004** (2004) 306
19. Pointcheval, D., Stern, J.: Security arguments for digital signatures and blind signatures. Journal of cryptology **13**(3) (2000) 361–396