Flexible and Scalable Digital Signatures in TPM 2.0

Liqun Chen
HP Laboratories
Bristol, UK
liqun.chen@hp.com

Jiangtao Li
Intel Corporation
Portland, Oregon, USA
jiangtao.li@intel.com

ABSTRACT

Trusted Platform Modules (TPM) are multipurpose hardware chips, which provide support for various cryptographic functions. Flexibility, scalability and high performance are critical features for a TPM. In this paper, we present the new method for implementing digital signatures that has been included in TPM version 2.0. The core part of this method is a single TPM signature primitive, which can be called by different software programmes, in order to implement signature schemes and cryptographic protocols with different security and privacy features. We prove security of the TPM signature primitive under the static Diffie-Hellman assumption and the random oracle model. We demonstrate how to call this TPM signature primitive to implement anonymous signatures (Direct Anonymous Attestation), pseudonym systems (U-Prove), and conventional signatures (the Schnorr signature). To the best of our knowledge, this is the first signature primitive implemented in a limited hardware environment capable of supporting various signature schemes without adding additional hardware complexity compared to a hardware implementation of a conventional signature scheme.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

TPM; Digital Signatures; Direct Anonymous Attestation

1. INTRODUCTION

A Trusted Platform Module (TPM) is a hardware chip used to provide verifiable attestation and integrity for a computer platform and to also provide support for multiple cryptographic functions necessary to implement data protection in the platform. The implementation of these functions in hardware has an advantage that it provides much better tamper resistance than any software. But the disadvantage of hardware implementation is that it usually costs more and is less flexible than software implementation. Therefore, it is essential to find a good balance between hardware and software implementation, in order to achieve high security using the minimum hardware resources.

TPMs can be made by any chip manufacturer, but to be useful they must all obey standard protocols and use a common interface. The specification of these protocols and interfaces has been developed by an industrial standards body, namely the Trusted Computing Group (TCG). The current version of the TPM specification is 1.2 [30]. This specification is also available as the international standard ISO/IEC 11889 [1]. TPMs implementing version 1.2 of the standard have been embedded in hundreds of millions of computing platforms.

TPM 1.2 only requires support for a small number of cryptographic algorithms: SHA-1, HMAC, RSA signature and encryption, AES[1], one-time-pad with XOR for symmetric encryption, and Direct Anonymous Attestation (DAA). As the TPM becomes widely used, there is a need for more flexible cryptographic algorithms being supported, because different countries and regions around the world have their own standard cryptographic algorithms and therefore their own differing requirements for a cryptographic processor. With a target of providing support for algorithm agility, the TCG has developed a new version of TPM specifications (TPM 2.0), which is now available for public review [31].

One critical challenge when designing the new TPM specification is including multiple cryptographic functionalities and flexibly selective cryptographic algorithms but ensuring that the chip can be made cheaply and still have high performance. In this paper, we describe how the digital signature primitive in TPM 2.0 was developed.

This work is based on two observations: (1) In a complex signature scheme, like DAA [5] and U-Prove [24], which we aim to implement using TPM 2.0, the computation directly using a private key is only a small part of the computation of the signature; (2) the computation using the private key is a self-contained signature operation that can be used by different pieces of software to create different/multiple complicated signatures. Based on these two observations, we designed a small signature primitive, which is implemented within a TPM and can be called by a number of other schemes and protocols. The starting point for this work was the requirement that a TPM be able to support three different types of signature: conventional signatures,

[1]AES is optional for TPM 1.2 according on revision 116 of the TPM 1.2 specification, published on March 3, 2011.

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2516729.

DAA signatures, and U-Prove signatures. All of them can be based on Elliptic Curve Cryptography (ECC). Our goal was to implement these signature schemes without more overhead in hardware than implementing a conventional signature scheme, such as the Schnorr signature scheme [28].

In our scheme the signing functionality is protected by the TPM, but the verification function exists external to the TPM. This is acceptable, because verification does not involve the private key and anybody is able to verify the integrity of a signature for themselves. We transfer a major part of the computation of the signing process to external software and only keep a small core operation in the TPM. The assurance of the scheme is not reduced by moving these computations outside the TPM. The TPM is a principal signer who holds the private key, the external software only plays the role of a helper that does not have the key, and it cannot forge a signature without the aid of the TPM.

The contributions of this paper are summarized as follows:

- We design a new TPM signature primitive, denoted by tpm.sign. It can be seen as a Schnorr type of signature scheme [28]. The special feature is that it has two generators of a group - one is conventional and the other is derived from a hash of an arbitrary integer. Each generator is associated with a Schnorr signature. The signing algorithm combines the two separated signatures into one. We demonstrate how to implement tpm.sign using a few simple TPM 2.0 commands. Most of these commands are also used for other TPM functions; as a result tpm.sign requires very little additional TPM resources.
- We conduct rigorous security analysis of tpm.sign. We prove that tpm.sign is secure under the static Diffie-Hallmen assumption in the random oracle model.
- We present how to use tpm.sign to implement DAA in TPM 2.0. Our implementation supports two different pairing-based DAA schemes using a single interface. The DAA implementation no longer requires any heavy, expensive and single purpose TPM commands as used in TPM 1.2.
- We present how to use tpm.sign to implement U-Prove 1.1 protocol in TPM 2.0. The U-Prove implementation does not require any extra TPM resources, as it simply reuses the TPM commands existing for DAA.

Although the proposed scheme was designed for TPM 2.0, the central idea of the scheme, i.e. using a single hardware-based signature to enable a variety of cryptographic protocols, is not just useful for TPM, but can also benefit other applications; for example, DAA can be used to provide an anonymous announcement system in vehicular networks [15], and U-Prove can use a smartcard or mobile phone for different applications [24].

In the remainder of the paper, we will first introduce the TPM signature primitive, tpm.sign, in §2, including the security proof of this algorithm, which will be followed by implementation details in TPM 2.0 in §3. After that we will present the first application of tpm.sign, which is DAA, in §4; our implementation supports two different pairing-based DAA schemes. We will further demonstrate that tpm.sign can be used as the protected hardware device for Microsoft U-Prove in §5. We will conclude our paper and discuss the future work in §6.

2. THE TPM SIGNATURE PRIMITIVE

2.1 Functional Description

We first describe the TPM signature primitive in TPM 2.0, denoted by tpm.sign. This is a Schnorr type of signature scheme, and has the following three procedures: key generation, signing, and verification. Only the key generation and signing procedures are implemented in TPM. In the rest of the paper, we use the following notation: Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order p and g be a generator. Let $H : \{0,1\}^* \to \mathbb{Z}_p$ and $H_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$ be two collision-resistent hash functions.

Key Generation This procedure selects a random $x \leftarrow \mathbb{Z}_p^*$ and computes $y := g^x$. The secret key is x while the public key is y.

Signing This procedure takes a message $m \in \{0,1\}^*$, a string $str \in \{0,1\}^*$, and a group element $P_1 \in \mathbb{G}$, and calls the following two sub-procedures:

Commit Oracle ($\mathcal{O}_C$) Given P_1, str as input:

1. Verify that $P_1 \in \mathbb{G}$.
2. If $str = \emptyset$, set $P_2 := 1$ (the identity element of $\mathbb{G}$), otherwise, compute $P_2 := H_{\mathbb{G}}(str)$.
3. Choose a random integer $r \leftarrow \mathbb{Z}_p$.
4. Compute $R_1 := P_1^r$, $R_2 := P_2^r$, and $K_2 := P_2^x$ where x is the private key.
5. Output R_1, R_2, and K_2.

Sign Oracle ($\mathcal{O}_S$) Given $c_h, m \in \{0,1\}^*$ as input, where $c_h = (R_1, R_2)$ for simplicity:[2]

1. Compute $c := H(c_h, m)$.
2. Compute $s := r + cx \mod p$ using r from the commit oracle and delete r.
3. Output (c, s).

The signature on m is $(P_1, P_2, R_1, R_2, K_2, c, s)$. It is a signature of knowledge

$$SPK\{(x) : K_1 = P_1^x \wedge K_2 = P_2^x\}(m).$$

Verification This procedure takes the message m, the signature $(P_1, P_2, R_1, R_2, K_2, c, s)$, and $K_1 \in \mathbb{G}$ as input, such that $K_1 = P_1^x$ and $K_2 = P_2^x$. It performs the following steps:

1. If $P_1 = 1$ and $P_2 = 1$, return 0 (invalid).
2. Verify that $H(R_1, R_2, m) = c$.
3. Verify that $R_1 = P_1^s \cdot K_1^{-c}$ and $R_2 = P_2^s \cdot K_2^{-c}$.
4. If any of the verification steps fails, return 0 (invalid), otherwise return 1 (valid).

The verification procedure relies on $K_1 = P_1^x$ and $K_2 = P_2^x$. If they are not guaranteed, then the signature can be easily forged. In the TPM 2.0 implementation of U-Prove and Schnorr signature, K_1 is the public key y. In DAA, the discrete log between P_1 and K_1 can be verified by the verifier through other means. Note that K_2 is a pseudonym

[2]In the TPM implementation of this oracle, c_h could be a hash of R_1, R_2, and other information.

of TPM in the DAA and U-Prove schemes. If the same P_2 is used, K_2 is always the same for a private key x.

Observe that a special case of this signature algorithm is the Schnorr signature. In the signing procedure, choose $P_1 = g$ and $str = \emptyset$ such that $R_1 = g^r$ and $R_2 = K_2 = 1$. In the verification procedure, set $P_1 = g$, $K_1 = y$, and $P_2 = K_2 = 1$. It is easy to see that $R_1 = g^r = g^s \cdot y^{-c} = P_1^s \cdot K_1^{-c}$ holds and the Schnorr signature can be verified successfully.

2.2 Security Notions and Proof

To prove security of tpm.sign, we use the standard security notion for digital signature schemes. We consider *existential forgery* where the goal of the adversary is to produce a valid signature on a message that he chooses. We consider the *chosen-message attack* where the adversary is given adaptive access to signatures on messages of his choice while attempting to forge a signature. In the TPM 2.0 implementation, the signing procedure is implemented using two separate TPM commands, one for each sub-procedure. Besides allowing the adversary to obtain signatures on the messages of his choice, we allow him to directly query the commit oracle $\mathcal{O}_C$ and the sign oracle $\mathcal{O}_S$ of his choice. For example, the adversary can query $\mathcal{O}_C$ a few times before querying $\mathcal{O}_S$, or the adversary can modify c_h to a different value before querying $\mathcal{O}_S$.

Definition 1 (Security Definition). *The* tpm.sign *scheme is secure if an existential forgery is computationally impossible, if the adversary is given adaptive access to signature on messages of his choice and given adaptive access to oracles $\mathcal{O}_C$ and $\mathcal{O}_S$.*

The security of our TPM 2.0 signature scheme depends on the following static Diffie-Hellman (DH) problem.

Definition 2 (Static DH Oracle). *Let $\mathbb{G}$ be a cyclic group of prime order p. Let x be a value in $\mathbb{Z}_p^*$. Given any $P \in \mathbb{G}$, the static DH oracle on x computes and outputs P^x.*

Definition 3 (Static DH Problem). *Let $\mathbb{G}$ be a cyclic group of prime order p. Given $g, h \in \mathbb{G}$ such that $h = g^x$, the static DH problem is to compute x given access to a static DH oracle on x.*

The static DH assumption is that it is computationally infeasible to solve the static DH problem. The static DH assumption is a stronger assumption than the discrete log assumption, as if one can solve the discrete log problem, then he can solve the static DH problem as well. It is believed in the cryptography community that the static DH problem is a computationally hard problem. Several cryptographic protocols rely on the static DH assumption, such as the basic El Gamal encryption [19], Ford-Kaliski server-assisted key generation protocol [21], and Chaum and van Antwerpen's Undeniable Signatures [12]. These protocols provide a static DH oracle on the secret key to the adversary.

To prove security of the tpm.sign scheme, we first review the well-known forking lemma from Pointcheval and Stern [26]. For a signature scheme based on three-pass honest-verifier zero-knowledge identification protocol using the Fiat and Shamir transformation [20], such as the Schnorr signature [28], a signature of a message m is a triple (σ_1, h, σ_2), where σ_1 represents all successive "commitment" of the protocol, $h = H(\sigma_1, m)$ where H is a hash function, and σ_2 represents all successive "answers" of the protocol.

Lemma 1 (The Forking Lemma). *Let $\mathcal{A}$ be a probabilistic polynomial time turning machine. Let Q and R be the number of queries that $\mathcal{A}$ can ask to the random oracle and the number of queries $\mathcal{A}$ can ask to the signer, respectively. Assume that within a time bound T, $\mathcal{A}$ produces a valid signature $(m, \sigma_1, h, \sigma_2)$ with probability $\epsilon \geq 10(R+1)(R+Q)/2^k$. If the triples (σ_1, h, σ_2) can be simulated without knowing the secret key, with an indistinguishable distribution probability, then there is another machine which has control over the machine obtained from $\mathcal{A}$ replacing interaction with the signer by simulation and produces two valid signatures $(m, \sigma_1, h, \sigma_2)$ and $(m, \sigma_1, h', \sigma_2')$ such that $h \neq h'$ in expected time $T' \leq 120686QT/\epsilon$.*

Theorem 2. *The* tpm.sign *scheme is secure under the security definition in Definition 1 in the random oracle model under the static DH assumption.*

Proof. Suppose there is an adversary $\mathcal{A}$ that breaks the tpm.sign scheme above, i.e., $\mathcal{A}$ outputs a forged signature $(m, P_1, P_2, R_1, R_2, K_2, c, s)$ after given access to signatures on messages of its choices and access to oracles $\mathcal{O}_C$ and $\mathcal{O}_S$. We can construct an algorithm $\mathcal{B}$ which makes use of $\mathcal{A}$ to solve the static DH assumption. $\mathcal{B}$ is given a pair $(g, h = g^x)$, where $\mathcal{B}$ can access the static DH oracle on x in polynomial time. That is, $\mathcal{B}$ can sends any $P \in \mathbb{G}$ to the static DH oracle and obtains P^x back. The goal is for $\mathcal{B}$ to compute the private key x. Algorithm $\mathcal{B}$ works as follows.

Key generation: $\mathcal{B}$ 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.

Signature Query: If $\mathcal{A}$ makes a signature query on m with P_1 and str as input, $\mathcal{B}$ computes $P_2 := H_{\mathbb{G}}(str)$ and queries the static DH oracle with P_1 and P_2 and obtains $K_1 = P_1^x$ and $K_2 = P_2^x$ back. $\mathcal{B}$ chooses at random c and s and computes $R_1 := P_1^s \cdot K_1^{-c}$ and $R_2 := P_2^s \cdot K_2^{-c}$. $\mathcal{B}$ patches the random oracle such that $c := H(R_1, R_2, m)$. $\mathcal{B}$ outputs $(m, P_1, P_2, R_1, R_2, K_2, c, s)$ as the signature. Observe that the distribution of this simulated signature by $\mathcal{B}$ is the same as the distribution of a real tpm.sign signature.

Query on $\mathcal{O}_C$: If $\mathcal{A}$ makes a commit query on $\mathcal{O}_C$ to $\mathcal{B}$ with P_1 and str as input, $\mathcal{B}$ computes $P_2 := H_{\mathbb{G}}(str)$ and queries the static DH oracle with P_1 and P_2 and obtains $K_1 = P_1^x$ and $K_2 = P_2^x$, respectively. $\mathcal{B}$ chooses at random c and s, and computes $R_1 = P_1^s \cdot K_1^{-c}$ and $R_2 = P_2^s \cdot K_2^{-c}$. $\mathcal{B}$ outputs (R_1, R_2, K_2). It is easy to see that that this simulation of (R_1, R_2, K_2) is perfect.

Query on $\mathcal{O}_S$: If $\mathcal{A}$ makes a signing query on $\mathcal{O}_S$ to $\mathcal{B}$ with c_h and m as input. $\mathcal{B}$ checks whether (c_h, m) has been queries before. If not, $\mathcal{B}$ sets $c := H_2(c_h, m)$ and outputs (c, s) where c and s were chosen in the commit oracle query. If (c_h, m) has been queried before and $H_2(c_h, m)$ is different from c in the corresponding commitment query, $\mathcal{B}$ returns failure. Clearly, the signature $(m, P_1, P_2, R_1, R_2, K_2, c, s)$ generated from $\mathcal{O}_C$ and $\mathcal{O}_S$ queries can be verified correctly by $\mathcal{A}$. Furthermore, the distribution of this simulated signature by $\mathcal{B}$ is computationally indistinguishable with a real tpm.sign signature.

Forgery: After $\mathcal{A}$ makes the above queries, $\mathcal{A}$ outputs a forged signature $(m, P_1, P_2, R_1, R_2, K_2, c, s)$ that can be successfully verified. Using the forking lemma above (Lemma 1), we can build another simulator $\mathcal{B}$' such that $\mathcal{A}$ outputs two valid tpm.sign signatures $\sigma_1 = (m, P_1, P_2, R_1, R_2, K_2, m, c, s)$ and $\sigma_2 = (m, P_1, P_2, R_1, R_2, K_2, c', s')$. Let $K_1 = P_1^x$. Since both signatures can be verified, we have

$$R_1 \cdot K_1^c = P_1^s, \qquad R_1 \cdot K_1^{c'} = P_1^{s'},$$
$$R_2 \cdot K_2^c = P_2^s, \qquad R_2 \cdot K_2^{c'} = P_2^{s'}.$$

Let $\Delta c = c - c'$ and $\Delta s = s - s'$, we have $K_1^{\Delta c} = P_1^{\Delta s}$ and $K_2^{\Delta c} = P_2^{\Delta s}$, where at least one of P_1 and P_2 is not the identity element. The algorithm $\mathcal{B}$' can compute $x := \Delta s / \Delta c$ where $x = \log_{P_1} K_1 = \log_{P_2} K_2$. In other words, $\mathcal{B}$' can compute the secret key x of the static DH and break the static DH problem.

Observe that there is a loss of efficiency in the reduction, due to possible failure in the query on $\mathcal{O}_S$. Since $c_h = (R_1, R_2)$, if $\mathcal{A}$ follows the signing procedure, the chance of (c_h, m) being queried before is negligible. Therefore, under the static DH assumption, the tpm.sign scheme is secure. □

Although security of the tpm.sign scheme is proved under the static DH assumption, in its implementation of TPM 2.0 there is no obvious way that an adversary can use the TPM as a static DH oracle via$\mathcal{O}_C$. This is because TPM computes $P_2 = H_{\mathbb{G}}(str)$ rather than accepting any P_2 value chosen by the adversary. Whether security of tpm.sign can be proved without the static DH assumption is an open question.

3. IMPLEMENTATION OF tpm.sign

In this section, we present how the tpm.sign scheme is implemented in TPM 2.0. We start with some relevant terms and notation and then explain the commands that are used to implement tpm.sign.

3.1 Terms and Notation

Throughout the paper, we use the notation shown in Table 1. We now introduce a few terms, which will be used to implement tpm.sign.

- Key handle: If a key is associated with multiple commands, the connection between these commands is presented as a key handle that uniquely identifies the key. A key handle is a 32-bit random value. The TPM assigns a key handle when a key is loaded into the TPM. When the key is subsequently used in another command (or multiple commands), the handle is taken as input for this command (or these commands). If more than one key are involved in a command, all handles of these keys are taken as input for the command.
- Key name: The name of an asymmetric key is used for identifying the key externally. It is a message digest of the public portion of the key. It is usually used for computing and verifying the integrity value of the key.
- Key blob: For the reason of limiting TPM resources, most TPM keys are not stored inside of the TPM. Each key is stored outside of the TPM in a format called a key blob, and it is associated with a parent key, say parentK, for the purpose of safe storage and integrity check. For an asymmetric key pair, written as $\mathsf{tk} = (\mathsf{tpk}, \mathsf{tsk})$, the key blob includes the following information: the private part of the key tsk encrypted under the parent key, the public part of the key tpk, and an integrity tag. The tag allows the TPM to verify integrity and authenticity of the key and is achieved by using a message authentication code (MAC). Both the encryption key SK and MAC key MK are derived from parentK by using a key derivation function (KDF). In the rest of this paper, a key blob of tk is denoted as $(\mathsf{tk})^*$ if we do not specify which key the parent key is. The following is an example of a key blob of tk under the parent key parentK:

  $$(\text{SK}, \text{MK}) := \text{KDF}(\mathsf{parentK}),$$
  $$(\mathsf{tk})^* := (\mathsf{tsk})_{\text{SK}} \| \mathsf{tpk} \| \text{MAC}_{\text{MK}}((\mathsf{tsk})_{\text{SK}} \| \mathsf{tpk}.\text{name}).$$

- Encryption blob: Let k_1 and k_2 be two asymmetric public keys of a TPM. To send a message m to the TPM such that only the TPM possessing k_1 and k_2 can release m to the host platform, we use the following encryption blob which encrypts m. The encryption blob has the following format

  $$\text{ENC}_{k_1}(k_2, m) = (seed)_{k_1} \| (m)_{\text{SK}} \| \text{MAC}_{\text{MK}}((m)_{\text{SK}} \| k_2.\text{name}),$$

  where *seed* is a random secret seed value, SK is a symmetric encryption key and MK is a message authentication key, both derived from *seed*. The encryption blob is a KEM-DEM type of encryption over m. To release the value m to the caller, the TPM must be satisfied that both k_1 and k_2 have been loaded in the TPM.

Notation	*Descriptions*
ek	TPM endorsement key pair
epk/esk	public/private portion of ek
tk	asymmetric key created by TPM
tpk/tsk	public/private portion of tk
k.handle	handle of k used for identifying the key internally by a TPM
k.name	name of k used for identifying the key externally
$\text{KDF}(s)$	key derivation function using s as seed
$\text{MAC}_k(m)$	message authentication code of m computed using key k
$\text{ENC}_k(m)$	encryption of m under public key k
$\text{DEC}_k(c)$	decryption of c under private key k
$\text{SIG}_k(m)$	signature of m signed under key k
$(m)_k$	encryption of m under symmetric key k
$(c)_k^{-1}$	decryption of c under symmetric key k
$(k)^*$	key blob of k wrapped by another key
$\text{ENC}_{k_1}(k_2, m)$	encryption blob of m under key k_1 associated with key k_2
$x \| y$	concatenation of x and y
$x \leftarrow S$	x chosen equally at random from a set S
$\|x\|$	bit size of x

Table 1: Notation used in this paper

3.2 The Relevant TPM Commands

All TPM functions are served by using a set of TPM commands. Each command is specified by its input and output,

and an operation between the input and output. Most of the TPM commands have multiple options, regarding to different types of keys and applications. For simplicity, we only explain these options which are related to the tpm.sign implementation and its applications that will be discussed in the later part of the paper. For the same reason, we may also omit some input and output information if they are not relevant to our purposes. The following TPM 2.0 commands are used to implement the tpm.sign scheme.

3.2.1 *Key Generation:* TPM2_Create()

This command is used to generate a TPM asymmetric key pair tk for tpm.sign. The command takes a handle of a parent key (say parentK) and public parameters as input, creates a fresh asymmetric key pair $\mathsf{tk} = (\mathsf{tpk}, \mathsf{tsk})$, and outputs a wrapped key blob, denoted as $(\mathsf{tk})^*$ as described before. In the remaining of the paper, we use

$$(\mathsf{tk})^* \leftarrow \mathsf{TPM2\_Create}() \quad \text{or} \quad (\mathsf{tk})^* \leftarrow \mathsf{Create}()$$

to denote this command and omit the information on how tsk was encrypted by the parent key or which key is the parent key. In the context of tpm.sign, to respond to this command, the TPM performs the following steps:

1. TPM picks a random $x \leftarrow \mathbb{Z}_p$ and computes $y = g^x$, where the values p and g are a part of the public parameters as described in Section 2.
2. TPM sets $\mathsf{tpk} := y$, $\mathsf{tsk} := x$, and $\mathsf{tk} := (\mathsf{tpk}, \mathsf{tsk})$.
3. TPM wraps tk with the parent key and outputs a key blob $(\mathsf{tk})^*$.

A variation of this command is TPM2_CreatePrimary(), in which the private key tsk is derived from a primary seed of the TPM using a key derivation function (KDF). A primary seed is a secret key stored inside of the TPM. As a result, the key tk is an alternative version of the primary seed. The same primary seed can be used to create multiple keys. In order to make each created key unique, some index value(s) shall be used. For simplicity, in the remaining of this paper, we will keep using TPM2_Create() only.

3.2.2 *Load a Key into TPM:* TPM2_Load()

When the TPM creates tk in TPM2_Create(), it does not store a copy of this key internally. In order to use tk, the key has to be loaded into the TPM using the command TPM2_Load(). This command takes as input a parent key handle and a key blob $(\mathsf{tk})^*$, which was created under $(\mathsf{tk})^* \leftarrow$TPM2_Create(). The TPM verifies integrity of the key. If the verification succeeds, the TPM outputs a handle (tk.handle) and the name (tk.name) for the key. In the remaining part of the paper, we use

$$(\mathsf{tk.handle}, \mathsf{tk.name}) \leftarrow \mathsf{TPM2\_Load}((\mathsf{tk})^*)$$
$$\text{or } \mathsf{tk.handle} \leftarrow \mathsf{Load}((\mathsf{tk})^*)$$

to denote this command. After TPM2_Load() has been called, tk is now stored inside the TPM and can be used for future operations.

3.2.3 *Certify a TPM Key:* TPM2_ActivateCredential()

To provision a certificate to the TPM key tk, the command TPM2_ActivateCredential() is used. Suppose that the TPM has an asymmetric endorsement key pair ek, and an authentic copy of the public portion of this key, epk, is accessible to a Certificate Authority (CA). The TPM associated computer platform sends tpk along with epk to the CA for certification. The CA checks that epk is a valid TPM endorsement key, and then computes a certificate (say *cert*) associated with tpk, generates a fresh symmetric encryption key k, computes the encryption blob $\text{ENC}_{\mathsf{epk}}(\mathsf{tpk}, k)$ along with the encrypted certificate $(cert)_k$, and sends the encryption blob and encrypted certificate back to the platform.

The platform uses the TPM2_ActivateCredential() command to let the TPM decrypt and release the symmetric key k, but the TPM will only respond to this command if the TPM has the corresponding esk and the loaded tk. The command has following input: a handle of ek, a handle of tk, and an encryption blob $\text{ENC}_{\mathsf{epk}}(\mathsf{tpk}, k)$ of a secret key k. The encryption blob has the following format

$$\text{ENC}_{\mathsf{epk}}(\mathsf{tpk}, k) = (seed)_{\mathsf{epk}} \| (k)_{\text{SK}} \| \text{MAC}_{\text{MK}}((k)_{\text{SK}} \| \mathsf{tk.name}).$$

In the remaining part of the paper, we use

$$m \leftarrow \mathsf{TPM2\_ActivateCredential}(k_1.\text{handle}, k_2.\text{handle}, c)$$
$$\text{or } m \leftarrow \mathsf{ActivateCredential}(k_1, k_2, c)$$

to denote this command. In the content of tpm.sign m is the symmetric key k, k_1 is ek, k_2 is tk and c is $\text{ENC}_{\mathsf{epk}}(\mathsf{tpk}, k)$. Suppose that the TPM has both keys internally. To respond to this command the TPM performs the following steps:

1. Decrypt $(seed)_{k_1}$ using k_1 to obtain *seed*.
2. Derive SK and MK from *seed*, i.e., $\text{SK} \| \text{MK} := \text{KDF}(seed)$.
3. Retrieve k_2.name and compute $\text{MAC}_{\text{MK}}((k)_{\text{SK}} \| k_2.\text{name})$.
4. Check whether the computed MAC value matches the MAC value in c.
5. If mismatch, return failure; otherwise, decrypt $(k)_{\text{SK}}$ and output k.

3.2.4 *Committing Process:* TPM2_Commit()

As mentioned in Section 2, the signing procedure in tpm.sign includes two phases: committing and signing. The committing process is achieved using the command TPM2_Commit().

This command is specially designed for a number of applications, including DAA (See Section 4) and U-Prove (See Section 5). It causes the TPM to compute the first part of the tpm.sign signature operation. It takes as input a key handle of a signing key tk, a point P_1 in $\mathbb{G}$, a string $\hat{s}$, and an integer $\hat{y}$, where $\hat{s}$ and $\hat{y}$ are used to construct another point P_2 in $\mathbb{G}$, see below for details. The TPM outputs three points R_1, R_2, K_2, and a counter *ctr* to the host, where *ctr* is used for identifying the random value r created by this command. In the remaining part of the paper, we use

$$(R_1, R_2, K_2, ctr) \leftarrow \mathsf{TPM2\_Commit}()$$
$$\text{or } (R_1, R_2, K_2) \leftarrow \mathsf{Commit}()$$

to denote this command. To respond this command the TPM performs the following steps:

1. TPM computes $\hat{x} := H(\hat{s})$ where H is a collision-resistant hash function, and sets $P_2 := (\hat{x}, \hat{y})$.
2. TPM verifies P_1 and P_2 are elements in $\mathbb{G}$.
3. TPM chooses a random integer $r \leftarrow \mathbb{Z}_p$.

4. TPM computes $R_1 := P_1^r$, $R_2 := P_2^r$, and $K_2 := P_2^x$.

5. TPM outputs R_1, R_2, K_2 and ctr while keeping r internally.

Note that some input to this command can be empty. If P_1 is an empty field, then R_1 is not computed. If s and $\hat{y}$ are empty, then R_2 and K_2 are not computed.

Note also that this command is slightly different from the description of tpm.sign in Section 2. In the command, the value P_2 is computed as $P_2 := (H(\hat{s}), \hat{y})$, whereas in the tpm.sign scheme, $P_2 := H_{\mathbb{G}}(str)$. This is because it is expensive for TPM to implement a new hash function $H_{\mathbb{G}}$. Given str, the host will compute $\hat{s}$ an $\hat{y}$ such that $H_{\mathbb{G}}(str) = (H(\hat{s}), \hat{y})$, therefore the TPM does not need to implement $H_{\mathbb{G}}$. Details can be found in Appendix A.

3.2.5 *Signing Process:* TPM2_Sign()

This command causes a TPM to sign a digest of a given message, which is a hash output of the message. The command takes as input a handle of the signing key tk, a message digest c_h, and optionally a counter value ctr, and outputs a signature σ on the message. The counter value ctr is only needed when the sign command is called after executing a commit command Commit(). Standard digital signature algorithms can be used, such as RSA, EC-DSA, or EC-Schnorr signatures. If a conventional signature scheme is used, then there is no need to call the commit command. In the remaining part of the paper, we use

$$\sigma \leftarrow \text{TPM2\_Sign}() \quad \text{or} \quad \sigma \leftarrow \text{Sign}()$$

to denote this command. In the context of tpm.sign, the TPM responds this comment by performing the following steps:

1. TPM computes $c := H(c_h, m)$, where m is the data required to be signed.

2. TPM retrieves r from the commit command based on the ctr value.

3. TPM computes $s := r + c \cdot x \bmod p$ where x is the private key tsk, and deletes r.

4. TPM outputs $\sigma = (c, s)$.

In the description of Commit() and Sign(), we assume an ideal implementation where the random value r created in the commit command is stored inside the TPM. The value r is deleted after the corresponding sign command has been executed. However, for the resource-constrained TPM, such implementation is too expensive. In the TPM 2.0 specification, as suggested by David Wooten [32], the following alternative method is used.

- During each boot of TPM, a random seed value $seed$ is generated and stored. The TPM maintains a maximum counter value $mctr$ (with initial value 0) and a bit table T of $N = 2^n$ entries.

- In each execution of the commit command, the TPM increments $mctr$ and sets $ctr := mctr$. The TPM derives r from $seed$ and ctr, i.e., $r := \text{KDF}(seed, ctr)$. Let i be the least significant n bits of ctr. The TPM sets $T[i] = 1$. In the end of this command, the TPM outputs ctr.

- In the sign command with ctr as input, the TPM first verifies that $mctr - N < ctr \leq mctr$. Let i be the least significant n bits of ctr. The TPM then checks that $T[i] = 1$. If all the verification succeed, the TPM uses the $seed$ and ctr to re-generate r. The TPM then sets $T[i] = 0$.

Using the above method, TPM does not need to save r for each commit command. Observe that the TPM checks $ctr \leq mctr$ and makes sure the same ctr has not been used by the sign command before, an attacker cannot query the future ctr or re-use an old ctr value. In the rest of this paper, we assume r is generated randomly instead of being derived for the purpose of security analysis.

4. APPLICATION 1: DAA IN TPM 2.0

Direct anonymous attestation (DAA) is an anonymous digital signature primitive, providing a balance between user privacy and signer authentication in a reasonable way. In a DAA scheme, there are issuers, signers and verifiers. The role of an issuer is to verify legitimacy of signers and to issue a unique DAA credential to each legitimate signer. A signer proves possession of her credential to a verifier by providing a DAA signature, which reveals neither of the DAA credential nor the signer's identity but allows the verifier to authenticate the signer.

The concept and first concrete scheme of DAA were proposed by Brickell, Camenisch, and Chen [5] for the purposes of remote anonymous attestation of a trusted computing platform. Security of their DAA scheme is based on the Strong RSA problem. In this paper, we call this scheme RSA-DAA for short. This DAA scheme was designed as a TPM function for the TCG and specified in the TPM specification version 1.2 [30]. Since the first introduction of DAA, it has attracted lots of attention from both industry and cryptographic researchers, e.g., [2, 3, 6, 7, 8, 9, 10, 13, 14, 16, 17, 22, 23, 27, 29].

Although the RSA-DAA scheme was adopted by the TCG and included in the TPM 1.2 specification ten years ago, TPM vendors have never been happy with it. The major reason is that the scheme is very expensive. In TPM 1.2 there are two expensive and single purpose commands used to implement the RSA-DAA scheme, namely DAA_Join() and DAA_Sign(). The description of these two commands takes about 10% space of the entire specification of TPM 1.2. Following IBM software TPM 1.2 implementation, DAA takes 6.9% of total code space (see Section 4.6 for the details). In the development of TPM 2.0, the functionality of DAA is still required, as addressing privacy concerns is important in a trusted computing environment. However, both TPM users and vendors want a new DAA implementation with minimum overhead and much higher efficiency than the RSA-DAA scheme. This is one of the motivations of the work described in this paper.

In this section, we demonstrate how to use the tpm.sign scheme to implement DAA, which no longer requires any heavy commands. We first give a general description of DAA, and then show that two existing pairing-based ECDAA schemes ([16, 10]) can be interpreted following this description. After that, we demonstrate that the DAA Join process and Sign process can be implemented using tpm.sign. This application was the main motivation why we designed

tpm.sign, and the result provides evidence that tpm.sign is a flexible and scalable signature algorithm.

4.1 A General Description of DAA

As described in many DAA papers [5, 22, 9, 6, 17, 16, 10, 8], a DAA scheme involves a set of DAA issuers $\mathfrak{I}$, a set of signers $\mathfrak{S}$ and a set of verifiers $\mathfrak{V}$. Each signer comprises a host platform and its associated TPM. A DAA scheme $\mathcal{DAA} = (\mathsf{Setup}, \mathsf{Join}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Link})$ consists of the following five polynomial-time algorithms and protocols:

- Setup: On input of a security parameter 1^t, an issuer $\mathfrak{i} \in \mathfrak{I}$ uses this randomized algorithm to produce its secret key isk, public key ipk and the global public parameters param. param includes a list called RogueList that contains rogue signers' secret keys. RogueList is set as an empty list in Setup and will be updated every time a rogue signer is discovered. Note that how to find a rogue signer is out the scope of a DAA scheme. We will assume that param are publicly known so that we do not need to explicitly provide them as input to other algorithms.
- Join: This protocol is run between an issuer $\mathfrak{i} \in \mathfrak{I}$ and a signer $(\mathfrak{t}, \mathfrak{h}) \in \mathfrak{S}$, where $\mathfrak{t}$ is a TPM and $\mathfrak{h}$ is the corresponding host. The protocol creates the TPM's secret key tsk and its DAA credential cre. The value cre is a signature on tsk under isk and can be verified under ipk. In this paper, we denote this signing and verification algorithms by i.sign and i.verify. The value cre is given to both $\mathfrak{t}$ and $\mathfrak{h}$, but the value tsk is known to $\mathfrak{t}$ only.
- Sign: On input of tsk, cre, a basename bsn (the name string of a verifier $\mathfrak{v} \in \mathfrak{V}$ or a special symbol $\perp$), a message msg to be signed and optionally the verifier's nonce n_V for freshness, $\mathfrak{t}$ and $\mathfrak{h}$ run this protocol to produce a DAA signature σ. More specifically, $\mathfrak{t}$ creates a signature σ_t under tsk and $\mathfrak{h}$ converts σ_t to σ using cre. In this paper, we denote the TPM's signing algorithm and its corresponding verification algorithm by t.sign and t.verify.
- Verify: On input of msg, bsn, and a candidate signature σ for msg and bsn. $\mathfrak{v} \in \mathfrak{V}$ uses this deterministic algorithm to return either **true** (accept) or **false** (reject). Note that if σ was created under a key listed in RogueList, it will be rejected.
- Link: On input of two signatures σ_0 and σ_1, $\mathfrak{v}$ uses this deterministic algorithm to return **linked** (linked), **unlinked** (not linked) or $\perp$ (invalid signatures). Note that, unlike Verify, the result of Link is not relied on RogueList.

We now give a new interpretation of DAA, which will lead us to implement two DAA schemes with the same TPM algorithm, tpm.sign. In our interpretation, a DAA scheme contains the following two underlying signature schemes that intercommunicate to each other:

- i.sign & i.verify: the DAA issuer's signature and verification algorithms. We use cre←i.sign(isk, tsk) to indicate that a DAA credential cre is a signature on the signed message tsk under the signing key isk. The correction of cre can be verified via **accept**←i.verify(ipk, cre, tsk). This signature algorithm has the properties of blinded signing and randomization. By these two properties we mean the following:
  - Given a commit value of tsk rather than tsk, the issuer can create cre. This allows the issuer to sign tsk without knowing this value.
  - Given cre, one can randomize it to obtain another signature cre′ satisfying **accept**←i.verify(ipk, cre′, tsk).
  - The issuer cannot tell whether cre′ and cre are signatures on the same tsk value or not.
- t.sign & t.verify: the TPM's signature and verification algorithms. We use σ_t←t.sign(tsk, bsn, (msg, n_V)) to indicate that a TPM part of the DAA signature is a signature on bsn, msg and n_V under the TPM signing key tsk. This signature has a property of user-controlled-linkability, which has the following meanings:
  - σ_t includes two parts $\sigma_t = (\sigma_{t,\alpha}, \sigma_{t,\beta})$.
  - $\sigma_{t,\alpha}$ is a deterministic signature on bsn under tsk. If two signatures contains the same bsn and $\sigma_{t,\alpha}$ values, they show that these two signatures are signed under the same tsk by the same TPM.
  - $\sigma_{t,\beta}$ is a "conventional" signature on (msg, n_V) under tsk. Depending on a DAA scheme, $\sigma_{t,\beta}$ may only available to the host $\mathfrak{h}$, not the Verifier $\mathfrak{v}$.

A DAA signature is a proof of knowledge of two underlying signatures, cre′ and $\sigma_{t,\alpha}$, in a signature format, denoted by:

$$SPK\{(\mathsf{tsk}, \mathsf{cre}) : \mathsf{cre}' \wedge \sigma_{t,\alpha}\}(\mathsf{msg}, \mathsf{bsn}, n_V).$$

Given the value cre and accessing to the TPM for t.sign, the host is able to compute such a proof. However, without accessing the TPM for t.sign, the host is not able to do so. The tpm.sign algorithm described in Section 2 is an implementation example of t.sign. In the next two subsections, we will explain how the LRSW-DAA scheme [16] and SDH-DAA scheme [10] can be interpreted using the two underlying signatures, i.sign and t.sign. As security of these two DAA schemes have been analyzed in their original work and we only suggest a new way to implement them, we do not repeat the security proof of these two DAA schemes in this paper. Our security analysis of tpm.sign in Section 2.2 proves that the host is not able to forge a tpm.sign signature; this proof indicates that the new implementation does not reduce security level of these two DAA schemes. This follows from the argument that except the host, from the view point of any other entities of the DAA scheme (a DAA Issuer or a DAA Verifier), a DAA signature created following the TPM 2.0 implementation is identical to the one created following the original schemes.

4.2 The LRSW-DAA Scheme

The above general description fits with the LRSW-DAA scheme as follows (we keep the names of parameters as used in [16]):

- $\mathsf{tsk} = f$ and $\mathsf{tpk} = (P_1, F)$, where $F = P_1^f$.
- $\mathsf{cre} = (A, B, C, D, u, v)$: as shown in the join protocol of the DAA scheme in [8]; (A, B, C) is the CL signature [11] on f; (u, v) is a Schnorr signature by the

issuer; as a result, cre is a variety of the CL signature on tpk, that allows the host to verifies it without aid of the TPM.

- $\mathsf{cre}' = (R, S, T, W, h, s)$: as shown in the sign/verify protocol of the DAA scheme in [16]; (h, s) is a Schnorr signature by the TPM. cre' is a blinded CL signature on tsk.
- $\sigma_{t,\alpha} = (J, K)$, where J is equivalent to P_2 and $K = J^f$ equivalent to K_2 in tpm.sign.
- $\sigma_{t,\beta} = (h, s)$.
- $SPK\{(\mathsf{tsk}, \mathsf{cre}) : \mathsf{cre}' \wedge \sigma_{t,\alpha}\}(\mathsf{msg}, \mathsf{bsn}, n_V) = (R, S, T, W, J, K, h, s)$.

4.3 The SDH-DAA Scheme

The above general description fits with the SDH-DAA scheme as follows (again we keep the names of parameters as used in [10]):

- $\mathsf{tsk} = f$ and $\mathsf{tpk} = (h_1, F)$, where $F = h_1^f$.
- $\mathsf{cre} = (A, x)$: cre is a signature on tpk (therefore on tsk) that can be verified by, $e(A, wg_2^x) = e(g_1 F, g_2)$.
- $\mathsf{cre}' = T = Ah_2^a$: cre' is a blind signature on tsk that holds $e(T^{-x} h_1^f h_2^{ax}, g_2) e(h_2, w)^a = e(T, w)/e(g_1, g_2)$.
- $\sigma_{t,\alpha} = (J, K)$, where, similar to the LRSW-DAA scheme, J is equivalent to P_2 and $K = J^f$ equivalent to K_2 in tpm.sign.
- $\sigma_{t,\beta}$ is only available to the host.
- $SPK\{(\mathsf{tsk}, \mathsf{cre}) : \mathsf{cre}' \wedge \sigma_{t,\alpha}\}(\mathsf{msg}, \mathsf{bsn}, n_V) = (J, K, T, c, s_f, s_x, s_a, s_b)$.

4.4 The TPM 2.0 DAA Join Process

There are two options for the TPM 2.0 join process. In the first option, the process is performed in the platform manufacturing line, e.g., the DAA issuer is the Original Equipment Manufacturer (OEM) of the platform. In this case, the issuer knows that it is talking to a genuine TPM. The issuer uses Create() to generate a DAA key, and then creates the corresponding DAA credential for the TPM.

In the second option, the DAA join process is after the platform has been shipped to an end user and the platform has a certified manufacturing endorsement key, ek, which is an asymmetric encryption key and its certificate is available to the DAA issuer. This is a common situation. The Join protocol, as shown in Figure 1, is run among a TPM $\mathfrak{t}$, the corresponding host $\mathfrak{h}$ and a DAA Issuer $\mathfrak{i}$. Suppose that the host and the TPM know which key is a parent key and this key is already available to the TPM. In our description, we assume that this may be the TPM endorsement key ek. The Join protocol has the following four steps. We refer each step to its corresponding line numbers in Figure 1.

1. *Create a DAA key (1-2).* The host asks the TPM to create a DAA key tk, by invoking Create(). The result is a key blob denoted by $(\mathsf{tk})^*$ wrapped by the parent key.
2. *Request a credential (2-7).* The host contacts the issuer to request for a DAA credential by sending public keys epk and tpk. The issuer validates epk and returns back with an encryption blob a of a nonce c using epk and tpk. The host loads the DAA key tk to the TPM by using Load(). The host then asks the TPM to release the nonce c by using ActivateCredential(ek, tk, a), in which the TPM verifies the integrity of the ciphertext a and returns the nonce c.
3. *Prove key possession (8-11).* The host asks the TPM to sign the nonce c from the previous step by using Sign(tk, c) on a conventional signature scheme. The host then sends the signature σ and nonce c back to the issuer. The issuer verifies the nonce c from the previous step and verifies the signature. The issuer creates the corresponding DAA credential $\mathsf{cre} = \mathrm{SIG}_{\mathsf{isk}}(\mathsf{tpk})$ using its secret key isk and the TPM public key tpk. The issuer then creates a session key k, an authentication encryption of cre using k denoted as b, and creates an encryption blob of k denoted as d. The issuer returns back b and d.
4. *Release the credential (11-12).* In order to ask the TPM to release the session key k that was used to encrypt the credential cre, the host uses ActivateCredential once again with input (ek, tk, d). After the host obtains k from the TPM, it decrypts the credential cre and verifies its integrity.

4.5 The TPM 2.0 DAA Sign/Verify Process

The DAA Sign/Verify process involves three entities: a TPM $\mathfrak{t}$, the TPM's host $\mathfrak{h}$ and a verifier $\mathfrak{v}$. Suppose that when the Sign/Verify protocol starts, these entities have already obtained the following necessary key materials and other parameters. The TPM has a private storage key internally and this key is called a parent key, denoted by parentK. The host has access to the DAA issuer's public key ipk and a copy of the TPM's DAA key blob $(\mathsf{tk})^*$ together with the associated credential cre. The verifier has access to the DAA issuer's public key ipk and a list of rogue TPM's private keys RogueList. A DAA scheme allows the verifier to detect whether or not a given DAA signature was created by a key in RogueList. Any signature signed by a key in RogueList will be rejected. We assume that in advance the verifier and host have agreed a basename value bsn, a message msg and a verifier's nonce n_V, all of which will be signed in the protocol. As shown in Figure 2, the Sign/Verify protocol has the following three steps. We refer each step to its corresponding line numbers in Figure 2.

1. *Pre-signing process (1).* The host randomizes the credential cre to obtain cre'. As mentioned in Section 4.1, given an arbitrary pair of cre and cre' values, finding whether or not they are associated to each other is computationally infeasible. The host can make this operation as pre-computation, since it is independent to bsn, msg and n_V.
2. *Signing process (1-6).* The host first loads the key blob $(\mathsf{tk})^*$ into the TPM. The TPM decrypts the blob using the parent key parentK and checks whether the key is valid. If the check passes, the TPM creates a key handle tk.handle and returns it to the host. The

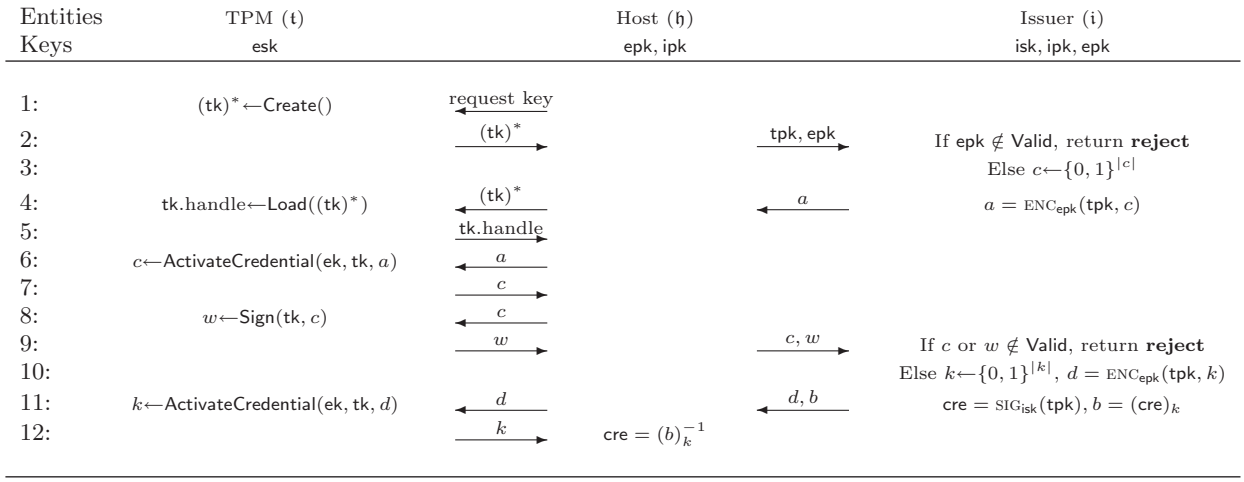

Figure 1: The TPM 2.0 DAA Join protocol.

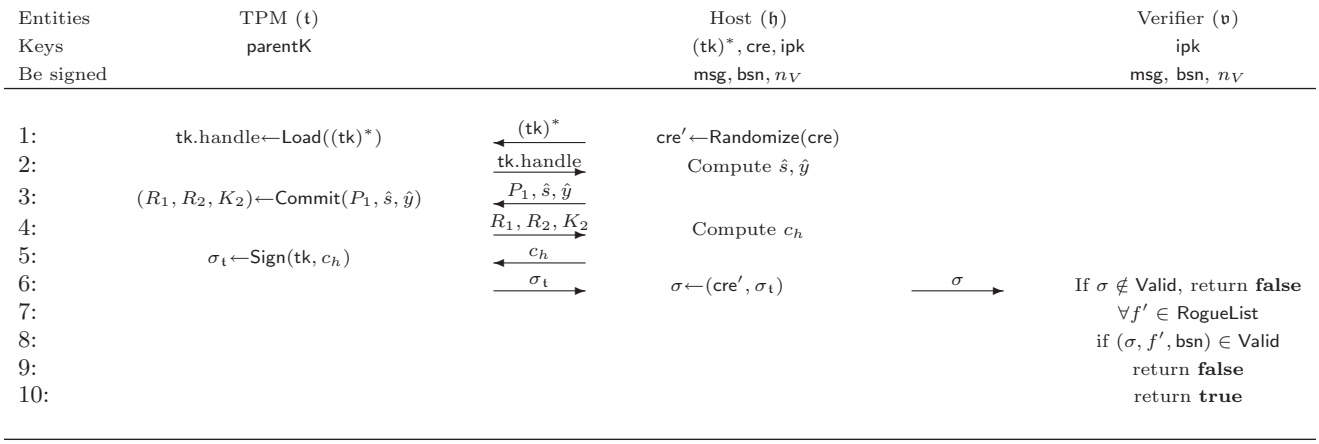

Figure 2: The TPM 2.0 DAA Sign/Verify protocol.

host then computes the values $\hat{s}$ and $\hat{y}$ from bsn and calls the Commit() command with the values $\hat{s}$, $\hat{y}$ and P_1, where P_1 is a part of ipk. The TPM responses the command as Commit Oracle of tpm.sign described in Section 2, and returns the values R_1, R_2 and K_2. The host then computes the value c_h and calls the Sign() command with this value as input. The TPM responses the command as Sign Oracle of tpm.sign described again in Section 2, and returns the value σ_t. The host packs the DAA signature σ from cre′ and σ_t and sends the signature to the verifier.

3. *Verifying process (6-10).* The verifier first verifies the validation of the signature σ, and then checks whether the signature was created by any key in RogueList. If both of the verifications given an appropriate answer, the TPM accepts the signature; otherwise rejects it.

4.6 Comparison with DAA in TPM 1.2

We compare our DAA schemes in TPM 2.0 with the DAA scheme in TPM 1.2 with two aspects: code size and performance. We shall show that our signature primitive achieves significant code and performance savings.

For code size, IBM software TPM 1.2 implementation[3] takes 3.28MB code space, in which 225KB code is used for DAA. In other words, DAA takes 6.9% of total code space. On TPM 2.0 v0.98 software reference implementation released by Microsoft, TPM 2.0 takes 1.61MB code space, in which only 7KB code is used for DAA. This is less than 0.5% of the total code space.

As for performance, DAA join protocol in TPM 1.2 takes 9 modular exponentiations, where modulus is either 1632-bit or 2048-bit, and exponents are more than 200-bit, with two exponents larger than 2048-bit. DAA sign protocol in TPM 1.2 takes 6 modular exponentiations and one of them has exponent more than 2048-bit. In TPM 2.0, the most expensive operation in the DAA join protocol is a conventional signature, including one scalar multiplication on an elliptic curve. The DAA sign operations takes 3 scalar multiplications on an elliptic curve. According to [18], on an Infineon TPM 1.2 chip (revision 1.2.3.16), DAA join takes 56.7 seconds and DAA sign takes 37.7 seconds. We do not have concrete DAA performance numbers on TPM 2.0, as TPM 2.0 chips are not publicly available yet. Based on prelimi-

[3]http://ibmswtpm.sourceforge.net/

nary performance figures from Nationz Technologies[4] on a discrete 40MHz TPM 2.0 chip, a scalar multiplication operation takes only 125ms on a 256-bit prime curve. We estimate that DAA join or sign in TPM 2.0 takes less than 0.5 second.

5. APPLICATION 2: U-PROVE WITH TPM AS A PROTECTED DEVICE

Microsoft U-Prove technology [24] is based on the Stefan Brands pseudonym system [4] and uses the blind signatures and zero-knowledge proofs as the fundamental building blocks. Stefan Brands provided the first description of the U-Prove technology in 2000 [4]. Later Microsoft acquired the technology and published U-Prove 1.1 protocol specification, which were updated to the current revision of U-Prove 1.1 protocol specification (the 2nd version) that was published in April 2013 [24].

In the U-Prove scheme, a U-Prove token is served as a pseudonym for a prover. This token contains a number of attributes of the prover certified by an issuer; and the attributes can be selectively disclosed to a verifier.

The U-Prove 1.1 specification [24] suggests that a U-Prove token can be optionally protected by a trusted hardware device such as a smartcard or a mobile phone. In that case, the U-Prove token cannot be used by the prover without the assistant of this device. The device can protect multiple tokens issued by multiple issuers, but it is too costly to implement all the prover algorithms in the trusted device. U-Prove 1.1 specifies how to split the prover functionality between a trusted device and a prover platform.

U-Prove is not supported by TPM 1.2. Another motivation of this paper is to enable the U-Prove technology in TPM 2.0 such that the TPM can be a trusted device for U-Prove, using the same TPM signature primitive for DAA without any additional overhead. In other words, U-Prove is supported by TPM 2.0 with zero extra cost of the TPM resources. In this section, we briefly review the Microsoft U-Prove technology [24] and describe how to use TPM 2.0 as a protected device for U-Prove 1.1.

5.1 Brief Review of U-Prove 1.1

In U-Prove, there is a U-Prove token served as a pseudonym for the prover. This token contains a number of attributes of the prover certified by the issuer which can be selectively disclosed to a verifier. The prover can decide which attributes to show and which attributes to hide. For each token, there is a public key which aggregates all the information of the token and a signature from the issuer over the public key to certify the token. Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order p. The issuer public key is $(g_0, g_1, \ldots, g_n, g_t) \in \mathbb{G}$ and the corresponding private key is y_0 such that $g_0 = g^{y_0}$. Let $(A_1, \ldots, A_n) \in \{0,1\}^*$ be n attributes of the prover, encoded into the U-Prove token. The public key $h \in \mathbb{G}$ of a U-Prove token has the following form

$$h = (g_0 g_1^{x_1} \cdots g_n^{x_n} g_t^{x_t})^{\alpha}$$

where $(x_1, \ldots, x_n) \in \mathbb{Z}_p$ are the corresponding hash values of attributes $(A_1, \ldots, A_n)$ respectively, x_t is a hash value of the issuer parameters, and $\alpha \in \mathbb{Z}_p$ is the secret key of the prover. A U-Prove token is (h, σ), where σ is a blind signature on the prover public key h issued by the issuer. There are two protocols in U-Prove: issuing token and presenting token.

[4]Private communication with Nationz Technologies.

In the issuing token protocol, both the prover and the issuer agree on the attributes $(A_1, \ldots, A_n)$ of the prover, thus they agree on $\gamma = (g_0 g_1^{x_1} \cdots g_n^{x_n} g_t^{x_t})$. In the end of the issuing token protocol, the prover obtains a blind signature σ on $h = \gamma^{\alpha}$ such that the issuer does not know the secret α. The signature $\sigma = (\sigma_z, \sigma_c, \sigma_r)$ is a signature of knowledge of the following form

$$SPK\{(y_0) : g_0 = g^{y_0} \ \wedge \ \sigma_z = h^{y_0}\}.$$

In the presenting token protocol, the prover selectively discloses a set of her attributes encoded in the public key to a verifier. The goal of this protocol is to prove the integrity and authenticity of the disclosed attribute values. In addition, the prover may commit to the values encoded in some attributes and prove that the commitments are computed correctly. Let $D \subset \{1, \ldots, n\}$ be the indices of disclosed attributes, $U = \{1, \ldots, n\} - D$ to the indices of undisclosed attributes, and $C \subset U$ be indices of committed attributes. In this protocol, for each $i \in C$, the prover chooses o_i randomly from $\mathbb{Z}_p$ and computes a Pedersen commitment [25] of x_i as $c_i = g^{x_i} g_1^{o_i}$. Then the prover computes the following zero-knowledge proof of knowledge

$$SPK\{(\alpha, \{x_i\}_{i\in U}, \{o_i\}_{i\in C}) : \{c_i = g^{x_i} g_1^{o_i}\}_{i\in C} \\ \wedge \ h = (g_0 g_1^{x_1} \cdots g_n^{x_n} g_t^{x_t})^{\alpha}\}(m).$$

The prover sends the U-Prove token (h, σ), disclosed attributes $\{A_i\}_{i\in D}$, commitments of attributes $\{c_i\}_{i\in C}$, and the signature of knowledge to the verifier. The verifier verifies the blind signature σ on h, computes $\{x_i\}_{i\in D}$ from attributes $\{A_i\}_{i\in D}$, and verifies the above signature of knowledge.

5.2 Protocol Details of U-Prove 1.1

In U-Prove 1.1, a U-Prove token can be optionally protected by a hardware device such as a smart card. The presenting token protocol requires both the token secret key α and the presence of the hardware device. Therefore, even if the U-Prove token is stolen by an attacker, it cannot be used unless the device is also physically captured by the attacker. Let $g_d \in \mathbb{G}$ be the public device generator. Let $x_d \in \mathbb{Z}_p$ be the device private key and $h_d = g_d^{x_d}$ be the device public key. The public key h of U-Prove token with device protection has the following form:

$$h = (g_0 g_1^{x_1} \cdots g_n^{x_n} g_t^{x_t} g_d^{x_d})^{\alpha}.$$

The issuing token protocol requires no additional computation from the device as long as the device outputs the device public key h_d to the prover. The prover and the issuer can run the issuing token protocol based on the following agreed value: $\gamma = (g_0 g_1^{x_1} \cdots g_n^{x_n} g_t^{x_t} h_d) = (g_0 g_1^{x_1} \cdots g_n^{x_n} g_t^{x_t} g_d^{x_d})$. The presenting token protocol requires interaction of the device as follows: The device optionally computes $g_s = H_{\mathbb{G}}(str)$ where $H_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$ is a collision-resistent hash function outputting elements in $\mathbb{G}$ and $P_s = g_s^{x_d}$. The value P_s is a pseudonym of the U-Prove token with regard to a specific scope str. The purpose of the (g_s, P_s) pair is similar to (J, K) pair in DAA: used for checking linkability between different U-Prove tokens. For a verifier with a specific scope str, she can link two presenting token protocols from a device, even if two different tokens are used. The device and

the prover then jointly compute the signature of knowledge:

$$SPK\{(\alpha, x_d, \{x_i\}_{i\in U}, \{o_i\}_{i\in C}) : \{c_i = g^{x_i} g_1^{o_i}\}_{i\in C} \\ \wedge\ h = (g_0 g_1^{x_1} \cdots g_n^{x_n} g_t^{x_t} g_d^{x_d})^{\alpha}\ \wedge\ P_s = g_s^{x_d}\}(m, m_d).$$

The device and the prover can compute the presenting token protocol as follows:

1. The prover computes x_i from A_i for $i \in \{1, \ldots, n\}$, and generates $w_0, w_d \leftarrow \mathbb{Z}_p$ at random and $w_i \leftarrow \mathbb{Z}_p$ at random for $i \in U$.
2. The prover sends str to the device.
3. The device computes $w'_d \leftarrow \mathbb{Z}_p$ and $a_d := g_d^{w'_d}$. If $s \neq \emptyset$, the device computes $g_s := H_{\mathbb{G}}(str)$, $a'_p := g_s^{w'_d}$, and $P_s := g_s^{x_d}$. The device returns a_d, a'_p, and P_s.
4. The prover computes

$$SPK\{(\alpha, \{x_i\}_{i\in U}, \{o_i\}_{i\in C}) : \{c_i = g^{x_i} g_1^{o_i}\}_{i\in C} \\ \wedge\ h = (g_0 g_1^{x_1} \cdots g_n^{x_n} g_t^{x_t} h_d)^{\alpha}\}(m, m_d).$$

   as follows: Compute $a = H(h^{w_0}(\prod_{i\in U} g_i^{w_i}) g_d^{w_d} a_d)$ and $a_p = H(g_s^{w_d}\ a'_p)$. For each $i \in C$, choose $o_i, w'_i \leftarrow \mathbb{Z}_p$ and compute $c_i := g^{x_i} g_1^{o_i}$ and $a_i := H(g^{w_i} g_1^{w'_i})$. Compute $m'_d := H(a, D, \{x_i\}_{i\in D}, C, \{c_i, a_i\}_{i\in C}, a_p, P_s, m)$, $c := H(m_d, m'_d)$ and $r_0 := w_0 + c\alpha^{-1} \bmod p$. For each $i \in U$, compute $r_i := w_i - cx_i \bmod p$. For each $i \in C$, compute $r'_i := w'_i - co_i \bmod p$.
5. The prover sends m_d, m'_d to the device.
6. The device computes $c := H(m_d, m'_d)$, and $r'_d := w_d - cx_d \bmod p$, and then outputs r'_d to the prover.
7. The prover computes $r_d := r'_d + w_d \bmod p$.

The signature of the presenting token protocol is: $(\{A_i\}_{i\in D}, a, (a_p, P_s), r_0, \{r_i\}_{i\in U}, r_d, \{c_i, a_i, r'_i\}_{i\in C})$. In the next subsection, we will discuss how to make use of a TPM implementing the TPM 2.0 specification as such a device.

5.3 Using TPM 2.0 as Protected Device

To enable TPM 2.0 as protected device for U-Prove, the prover can compute the presenting token protocol as follows:

1. The prover runs the first step of the presenting token protocol in Section 5.2.
2. The prover loads the device key blob into the TPM and calls the TPM2_Commit() command with $(g_d, \hat{s}, \hat{y})$ as input where $H_{\mathbb{G}}(str) = (H(\hat{s}), \hat{y})$. The prover obtains $(a_d = R_1, a'_p = R_2, P_s = K_2)$.
3. The prover runs Step 4 of the presenting token protocol in Section 5.2.
4. The prover calls the TPM2_Sign() command with (m_d, m'_d) as input and obtains $(c, r'_d = s)$.
5. The prover computes $r_d := r'_d + w_d \bmod p$ and outputs the signature of the protocol $(\{A_i\}_{i\in D}, a, (a_p, P_s), r_0, \{r_i\}_{i\in U}, r_d, \{c_i, a_i, r'_i\}_{i\in C})$.

Since both the TPM2_Commit() and TPM2_Sign() commands have already been used to implement the two DAA schemes as described in Section 4, the implementation of U-Prove in TPM 2.0 does not require any extra TPM resources.

6. CONCLUSION AND FUTURE WORK

We have presented a new digital signature primitive in TPM 2.0 with provable security. The interesting and unique property of this TPM signature primitive is that it can be called by different software programmes, in order to implement different cryptographic schemes and protocols, such as DAA, U-Prove, and Schnorr signatures.

Future work includes: (1) Explore other usages of this TPM signature primitive. (2) Provide a more tight security proof of the TPM signature primitive. In particular we would like to see an answer of the question whether security of this primitive can be proved without the static DH assumption or without the random oracle model. (3) Construct a new tpm.sign scheme which can be proved secure under a weaker assumption but without adding much cost.

Acknowledgement

We thank Ernie Brickell, Chris Newton, Graeme Proudler, Claire Vishik, Monty Wiseman and David Wooten for their useful inputs and discussions. We also thank the TCG TPM working group for their support to this work. We thank Fan Qin and Liu Xin from Nationz Technologies for providing the TPM 2.0 performance data. We appreciate Kevin Butler and anonymous CCS reviewers for their helpful comments.

7. REFERENCES

[1] ISO/IEC 11889:2009 Information technology – Security techniques – Trusted platform module.

[2] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 202–215. IEEE Computer Society, 2008.

[3] D. Bernhard, G. Fuchsbauer, E. Ghadafi, N. P. Smart, and B. Warinschi. Anonymous attestation with user-controlled linkability. *International Journal of Information Security*, 12(3):219–249, 2013.

[4] Stefan A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy*. MIT Press, August 2000.

[5] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 132–145. ACM Press, 2004.

[6] Ernie Brickell, Liqun Chen, and Jiangtao Li. A new direct anonymous attestation scheme from bilinear maps. In *Proceedings of 1st International Conference on Trusted Computing*, volume 4968 of *LNCS*, pages 166–178. Springer, 2008.

[7] Ernie Brickell, Liqun Chen, and Jiangtao Li. Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. *International Journal of Information Security*, 8(5):315–330, 2009.

[8] Ernie Brickell, Liqun Chen, and Jiangtao Li. A (corrected) DAA scheme using batch proof and verification. In *Proceedings of 3rd International Conference on Trusted Systems*, volume 7222 of *LNCS*, pages 304–337. Springer, 2011.

[9] Ernie Brickell and Jiangtao Li. Enhanced Privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. In *Proceedings of the 6th ACM Workshop on Privacy in the Electronic Society*, pages 21–30, October 2007.

[10] Ernie Brickell and Jiangtao Li. A pairing-based DAA scheme further reducing TPM resources. In *Proceedings of 3rd International Conference on Trust and Trustworthy Computing*, volume 6101 of *LNCS*, pages 181–195. Springer, 2010.

[11] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *Advances in Cryptology — CRYPTO '04*, volume 3152 of *LNCS*, pages 56–72. Springer, 2004.

[12] David Chaum and Hans Van Antwerpen. Undeniable signatures. In *Advances in Cryptology — CRYPTO '89*, volume 435 of *LNCS*, pages 212–216. Springer, 1989.

[13] Liqun Chen. A DAA scheme using batch proof and verification. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, volume 6101 of *LNCS*, pages 166–180. Springer, 2010.

[14] Liqun Chen, Paul Morrissey, and Nigel P. Smart. Pairings in trusted computing. In *Proceedings of the 2nd International Conference on Pairing-Based Cryptography*, volume 5209 of *LNCS*, pages 1–17. Springer, 2008.

[15] Liqun Chen, Siaw-Lynn Ng, and Guilin Wang. Threshold anonymous announcement in VANETs. *IEEE Journal on Selected Areas in Communications, Special Issue on Vehicular Communications and Networks*, 2010.

[16] Liqun Chen, Dan Page, and Nigel P. Smart. On the design and implementation of an efficient DAA scheme. In *Proceedings of the 9th Smart Card Research and Advanced Application IFIP Conference*. Springer, 2010.

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

[18] Kurt Dietrich. Anonymous client authentication for transport layer security. In *Communications and Multimedia Security*, volume 6109 of *LNCS*, pages 268–280, 2010.

[19] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology — CRYPTO '84*, volume 196 of *LNCS*, pages 10–18. Springer, 1985.

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

[21] Warwick Ford and Burton S. Kaliski. Server-assisted generation of a strong secret from a password. In *Proceedings of the IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 176–180, 2000.

[22] He Ge and Stephen R. Tate. A direct anonymous attestation scheme for embedded devices. In *Proceeding of 10th International Conference on Practice and Theory in Public Key Cryptography*, volume 4450 of *LNCS*, pages 16–30. Springer, 2007.

[23] Adrian Leung and Chris J. Mitchell. Ninja: Non identity based, privacy preserving authentication for ubiquitous environments. In *Proceedings of 9th International Conference on Ubiquitous Computing*, volume 4717 of *LNCS*, pages 73–90. Springer, 2007.

[24] Microsoft U-Prove Community Technology. U-Prove cryptographic specification version 1.1, 2013. `http://www.microsoft.com/u-prove`.

[25] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology — CRYPTO '91*, volume 576 of *LNCS*, pages 129–140. Springer, 1991.

[26] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, 2000.

[27] Carsten Rudolph. Covert identity information in direct anonymous attestation (DAA). In *Proceedings of the IFIP TC-11 22nd International Information Security Conference (SEC)*, pages 443–448. Springer, 2007.

[28] Claus P. Schnorr. Efficient identification and signatures for smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[29] Ben Smyth, Mark Ryan, and Liqun Chen. Direct anonymous attestation (DAA): Ensuring privacy with corrupt administrators. In *Proceedings of 4th European Workshop on Security and Privacy in Ad-hoc and Sensor Networks*, volume 4572 of *LNCS*, pages 218–231. Springer, 2007.

[30] Trusted Computing Group. TCG TPM specification 1.2, 2003. `http://www.trustedcomputinggroup.org`.

[31] Trusted Computing Group. TCG TPM specification 2.0, 2013. `http://www.trustedcomputinggroup.org/resources/trusted_platform_module_specifications_in_public_review`.

[32] David Wooten. Private communications.

APPENDIX

A. COMPUTATION OF $H_{\mathbb{G}}$

We first describe a method of constructing $H_{\mathbb{G}} : \{0,1\}^* \rightarrow \mathbb{G}$, where $\mathbb{G}$ is an elliptic curve group $E : y^2 = x^3 + ax + b$ over $\mathbb{F}_q$ with cofactor = 1. Given a message $m \in \{0,1\}^*$, $H_{\mathbb{G}}(m)$ can be computed as follows:

1. Set $i := 0$ be a 32-bit unsigned integer.
2. Compute $x := H(i, m)$.
3. Compute $z := x^3 + ax + b \bmod q$.
4. Compute $y := \sqrt{z} \bmod q$. If y does not exist, set $i := i + 1$, repeat step 2 if $i < 2^{32}$, otherwise, report failure.
5. Set $y := \min(y, q - y)$.
6. Output the result as (x, y).

The host platform can help the TPM compute $H_{\mathbb{G}}(m)$, yet the TPM can verify the computation as follows. Given m, the host runs the above algorithm. For a successful execution, let $\hat{s} := (i, m)$ and $\hat{y}$ be the y value in the last step. The host sends $\hat{s}$ and $\hat{y}$ to the TPM. The TPM computes $H_{\mathbb{G}}(m) := (H(\hat{s}), \hat{y})$. This is the first step of the TPM 2.0 commit command.