Algorithm Agility – Discussion on TPM 2.0 ECC Functionalities

Liqun Chen[1,2] and Rainer Urian[3]

[1] Hewlett Packard Laboratories, liqun.chen@hpe.com
[2] University of Surrey, liqun.chen@surrey.ac.uk
[3] Infineon Technologies AG, rainer.urian@infineon.com

Abstract. The TPM 2.0 specification has been designed to support a number of Elliptic Curve Cryptographic (ECC) primitives, such as key exchange, digital signatures and Direct Anonymous Attestation (DAA). In order to meet the requirement that different TPM users may favor different cryptographic algorithms, each primitive can be implemented from multiple algorithms. This feature is called *Algorithm Agility*. For the purpose of performance efficiency, multiple algorithms share a small set of TPM commands. In this paper, we review all the TPM 2.0 ECC functionalities, and discuss on whether the existing TPM commands can be used to implement new cryptographic algorithms which have not yet been addressed in the specification. We demonstrate that four asymmetric encryption schemes specified in ISO/IEC 18033-2 can be implemented using a TPM 2.0 chip, and we also show on some ECDSA variants that the coverage of algorithm agility from TPM 2.0 is limited. Security analysis of algorithm agility is a challenge, which is not responded in this paper. However, we believe that this paper will help future researchers analyze TPM 2.0 in more comprehensive methods than it has been done so far.

Keywords: algorithm agility, elliptic curve cryptography, trusted platform module

1 Introduction

Trusted Platform Module (TPM) is an international standard for a tamper-resistant crypto processor. TPM's technical specification is developed by a computer industry standard body called Trusted Computing Group (TCG). The first broadly used TPM specification is TPM version 1.2 [29], which was released in 2003. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) standardized this specification as ISO/IEC 11889 in 2009 [1].

The TPM 1.2 specification only supported a small number of cryptographic algorithms: RSA encryption and digital signatures, SHA-1 hash function, HMAC message authentication code and Direct Anonymous Attestation (DAA) based on the RSA problem. This fixed algorithm coverage was not satisfactory to worldwide TPM users. Besides an obvious reason that SHA-1 is no longer suitable for

digital signatures based on the attack in [31], people from different countries and regions may favor different cryptographic algorithms, especially elliptic curve cryptography. This required the TCG to revise the TPM specification.

As a reaction, the TCG now continuously revises the TPM specification, and the biggest step was to move from TPM 1.2 to TPM 2.0. The latest TPM 2.0 release is Trusted Platform Module Library Specification Revision 01.16 released in October 2014 [30]. ISO/IEC standardized this specification as ISO/IEC 11889 in December 2015 [2] and meanwhile the previous 2009 edition [1] was withdrawn.

Among many important modifications from TPM 1.2, the most attractive change from the authors of this paper's view point is that the TPM 2.0 supports **Algorithm Agility**, which means that each cryptographic primitive can be used by multiple cryptographic algorithms. This is managed by using the TCG Algorithm Registry [28].

Although algorithm agility is a well received property, the performance efficiency is still one of the most important requirements in the development of the TPM 2.0 family. In order to achieve a balance between algorithm agility and high performance, the TPM 2.0 specification allows a set of TPM commands to be shared by multiple algorithms.

The new cryptographic functionalities from TPM 1.2 include a number of Elliptic Curve Cryptographic (ECC) primitives, such as Elliptic Curve (EC) based digital signatures, key exchange and DAA. This paper is focused on discussing the algorithm agility for TPM 2.0 ECC functionalities. We aim to make the following contributions:

1. Find whether the existing TPM 2.0 commands can be used to implement new cryptographic algorithms which have not yet addressed in the current specification [2, 30]. We demonstrate that four asymmetric encryption schemes specified in ISO/IEC 18033-2 [4] can be implemented using a TPM 2.0 chip. We also show that the coverage of algorithm agility from TPM 2.0 is limited.
2. Show some obstacles one faces when implementing an algorithm in such way that it will be usable from many different standards.
3. Provide a concise description of the ECC functionalities in TPM 2.0, which is easier to follow by cryptographic researchers than the specification [2, 30].

In the literature, there are many papers aimed to analyze security features of a TPM. They all focus on individual cryptographic algorithms or functions; for example, analyzing privacy-CA solution [15, 19] and DAA [10, 33, 13]. To the best of our knowledge, a comprehensive security analysis of multiple TPM functionalities, such as algorithm agility, does not exist. This is a big challenge. We notice that one reason why this has not happened yet is because the TPM specification is not reader friendly for cryptographic researchers, with the evidence that the specification [2, 30] is over 1500 pages long. Although this paper does not aim to provide a complete response to this challenge, we believe that the content of this paper will help the future cryptographic researcher analyst TPM 2.0 in more comprehensive methods than it has been done so far.

The remaining part of this paper is arranged as follows. In the next section, we will review the existing TPM 2.0 ECC functionalities, which include a short

overview of the TPM's key handling method and commands, and the TPM ECC related commands. In Section 3 we list the EC-based cryptographic algorithms and protocols which were already addressed in the TPM 2.0 specification. In Section 4, we will discuss a number of asymmetric encryption algorithms, which can be implemented by using the existing TPM 2.0 ECC functionalities, although they have not been mentioned in the specification yet. In this section we also show the limitation of TPM algorithm agility by variants of ECDSA signature algorithms. In Section 5, taking an example of the EC-based Schnorr digital signature scheme, we will further discuss on some issues in compatibility. Section 6 will share our considerations about TPM performance. We will conclude the paper in Section 7 with an open question on how to define and prove security notions for the TPM 2.0 algorithm agility property.

2 Overview of the TPM 2.0 ECC Functionalities

In this section we give an overview of the Elliptic Curve Cryptographic (ECC) functionalities which are specified in the TPM 2.0 specification [2, 30]. Because the TCG TPM 2.0 specification [30] is still under development, the information used in this section is based on a version of the TPM library published by ISO/IEC in 2015 [2]. We introduce a set of major TPM 2.0 commands that are used to implemented the TPM 2.0 ECC functions. After that we list all the EC-based cryptographic algorithms that are mentioned in the specification.

2.1 Introduction to TPM Keys

To describe ECC keys in the TPM 2.0 environment, we use the notation shown in Table 1.

TPM Key Structures In the TPM 2.0 environment, TPM keys are arranged with key hierarchies. For the reason of limiting TPM resources, keys are normally stored outside. Each key except a root key is associated with a parent key, $\mathsf{parentK}$, and the top parent key is a root key.

Let an ECC key be denoted by `tk` with a private potion `tsk` and a public potion `tpk`. Some system parameters about an ECC key, known by a TPM, include coefficients of the curve, a field modulus of the curve, an order of group elements on the curve and a generator of the group. For simplicity, we use `tpk` to cover all of these parameters. Each key `tk` is associated a key name denoted by `tk.name`, key blob denoted by `tk.blob` and key handle denoted by `tk.handle`, which have the following meanings.

- *Key name:* `tk.name` is a message digest of `tpk` and the key's attributes. It is usually used for verifying the integrity of the key.
- *Key blob:* Each TPM key stored outside of the TPM is in a format of a key blob; `tk.blob` includes the following information: `tsk` encrypted under its $\mathsf{parentK}$, `tpk`, and an integrity tag. The tag allows the TPM to verify

Notation	*Descriptions*
`tk`	ECC key created by TPM
`tpk`/`tsk`	public/private portion of `tk`
parentK	a key used to introduce another key
k.`name`	name of key k used for identifying the key externally
k.`blob`	key blob of key k wrapped by its parentK
k.`handle`	handle of key k used for identifying the key internally by a TPM
$\text{KDF}(s)$	key derivation function using s as seed
$\text{MAC}_k(m)$	message authentication code of m computed using key k
$(m)_k$	encryption of m under symmetric key k
$x \| y$	concatenation of x and y
$\mathbb{G}_p$	an elliptic curve group of prime order p
G	a generator of $\mathbb{G}_p$

Table 1. Notation used in this paper

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). The following is a brief description of `tk.blob`:

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

– *Key handle:* If `tk` is associated with multiple commands, the connection between these commands is presented as `tk.handle` that uniquely identifies the key. `tk.handle` is assigned by the TPM when `tk` is loaded into the TPM. Such a key handle is a 4 byte (word) value. The first byte designates the handle type and the remaining three bytes are uniquely referring the key. After the loading command, when `tk` is subsequently used in another command (or multiple commands), the handle is taken as input for each command. If more than one key are involved in a command, all handles of these keys are taken as input for the command.

The usage of each ECC key are classified by three key base attributes as *restricted*, *sign* and *decrypt*. Table 2 shows valid combinations.

The *sign* attribute is used to allow the key to perform signing operations, e.g. this key can be used for the TPM2_Sign() command.

The *decrypt* attribute is used to allow the key to perform decryption operations, e.g. this key can be used for the TPM2_ECDH_ZGen() command.

The *restricted* attribute needs a bit more explanation. Let's first explain *restricted sign* keys. The TPM can be used to sign externally given messages or to sign internally generated data. For instance, the TPM2_Quote() command signs

sign	*decrypt*	*restricted*	*description*
0	0	0	no key, user defined data blob
0	0	1	not allowed
0	1	0	a decryption key but may not be a parentK
0	1	1	may be a parentK
1	0	0	a key for signing external date
1	0	1	a key for signing TPM generated data only
1	1	0	a general-purpose key
1	1	1	not supported

Table 2. Key base attributes

the values of some platform configuration registers and the TPM2_Certify() command signs a TPM generated key. A verifier must be assured that the signatures actually have been performed by those commands on internal TPM data. To do this, the TPM puts a special tag word called TPM_GENERATED_VALUE in the message header of the signature. This tag proves to the verifier that the signature belongs to TPM created data. If the signing key has the *restricted* attribute, the TPM will only sign an externally given message by TPM2_Sign(), if the message does not start with the TPM_GENERATED_VALUE tag. This protects that the TPM2_Sign() command cannot be misused to fake a platform attestation.

The *restricted decryption* attribute is mainly used for the parent key to protect a key blob. Here it must be targeted that only the TPM can decrypt the key blob. The *restricted* attribute protects this key from being used for general purpose decryption commands (e.g. TPM2_ECDH_ZGen()). If the key would not be *restricted* an attacker could simply use the TPM2_ECDH_ZGen() command to decrypt the key blob.

In addition to the base attributes there are other key attributes. We will not go into detail here but only mention the most important ones.

- userWithAuth and adminWithPolicy: they control authorization of the key.
- fixedTPM and fixedParent: they control if the key can be duplicated under another parent key of the same TPM or another one.

2.2 TPM 2.0 Key Handling Commands

All TPM functions are served by using a set of TPM commands. 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 implementation of the TPM ECC functions 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.

Generate a key: TPM2_Create(). An ECC key `tk` can be generated by using this command. The command takes a handle of a parent key (say parentK)

that has already been loaded into the TPM and public parameters about the curve, algorithm identifier and so on as input, creates a fresh ECC key pair $\texttt{tk} = (\texttt{tpk}, \texttt{tsk})$, and outputs a wrapped key blob, `tk.blob` as described before.

In the context of the ECC functions, to respond to this command, the TPM performs the following steps:

1. TPM picks a random $x \leftarrow \mathbb{Z}_p$ and computes $Y = [x]G$, where the values p and G are a part of the public parameters dependent on the ECC algorithms that will be discussed in the next subsection.
2. TPM sets $\texttt{tpk} := Y$, $\texttt{tsk} := x$, and $\texttt{tk} := (\texttt{tpk}, \texttt{tsk})$.
3. TPM wraps `tk` with the parent key and outputs a key blob `tk.blob`.

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 a root key of the key hierarchy. The same primary seed can be used to create multiple root keys. In order to make each created key unique, some index value(s) shall be used. Primary keys will be used internally as root keys which protect a key hierarchy of ordinary keys. They will normally not be used for cryptographic services and we therefore ignore them in the remaining of this paper.

Load a key into TPM: TPM2_Load(). When `tk` is created in TPM2_Create(), it is not stored inside of the TPM. 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 `tk.blob`. The TPM verifies integrity of the key by checking the validation of the blob under the parent key, optionally also verifies the user authorization and the attributes consistence. If all the verification succeeds, the TPM outputs a handle `tk.handle` and the name `tk.name` for the key. After TPM2_Load() has been called, `tk` is now stored inside the TPM and can be used for future operations.

Load an external key to TPM: TPM2_LoadExternal(). An external key that is not part of a TPM key hierarchy can also be loaded into the TPM. This will normally be a public key only. For example, if a signature verification is the purpose, then the public verification key will be loaded into the TPM with this command.

2.3 TPM 2.0 ECC Commands

Commit an ephemeral secret for signing: TPM2_Commit(). Several EC-based signature schemes are implemented using two phases: committing and signing. The committing process is achieved using the command TPM2_Commit(). It takes as input a key handle of a signing key `tk`, a point P_1 in $\mathbb{G}_p$, a string $\hat{s}$, and an integer $\hat{y}$, where $\hat{s}$ and $\hat{y}$ are used to construct another point P_2 in $\mathbb{G}_p$,

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. 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 $\mathbb{G}_p$.
3. TPM chooses a random integer $r \leftarrow \mathbb{Z}_p$.
4. TPM computes $R_1 := [r]P_1$, $R_2 := [r]P_2$, and $K_2 := [\mathtt{tsk}]P_2$.
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 $\hat{s}$ and $\hat{y}$ are empty, then R_2 and K_2 are not computed. If all the three elements P_1, $\hat{s}$ and $\hat{y}$ are empty, then $R_1 = [r]G$, where G is a long-term base in the curve parameters and was used in creating `tk`.

Sign: TPM2_Sign(). This command can be used as a one-phase signing operation or the second phase of the two-phase signing protocols. It 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 TPM2_Commit(). Standard digital signature algorithms can be used, such as RSA, ECDSA, or ECSchnorr signatures. If a conventional signature scheme is used, then there is no need to call the commit command. In the context of a two-phase signing protocol, the TPM responds to this command by performing the following steps:

1. TPM retrieves r from the commit command based on the ctr value.
2. TPM computes $s := r + c \cdot \mathtt{tsk} \bmod p$ and deletes r.
3. TPM outputs s.

Note: Recently Xi et. al. [33] and Camenisch et. al. [13] reported an issue in the security proof of [16], that requires to a modification of the scheme in [2] by adding the nonce n_t. Note also that the nonce n_t is in another version of EC-DAA schemes specified in ISO/IEC 20008-2 [6], so this issue does not require such a modification to ISO/IEC 20008-2. This modification has of course implications also to other protocols which rely on the ECDAA functionality. Here is the modified sign algorithms.

1. TPM created a nonce $n_t \rightarrow \mathbb{Z}_p$.
2. TPM computes $c := H(c_h, n_t)$.
3. TPM retrieves r from the commit command based on the ctr value.
4. TPM computes $s := r + c \cdot \mathtt{tsk} \bmod p$ and deletes r.
5. TPM outputs $\sigma = (n_t, s)$.

The TPM 2.0 specification also contains commands which perform a signature over TPM internally stored date. For instance, TPM2_Quote() is used to sign platform configuration registers and TPM2_Certify() will sign another TPM stored key. We will not go into detail of those commands.

Compute an ephemeral key: TPM2_ECDH_KeyGen(). This command takes as input the public portion of a loaded key including an EC point P in the curve, chooses an element d uniformly at random from the space of the ECC private key, computes $Q := [d]P$ and outputs P and Q. The TPM does not record or output the value d. Since the operation can be performed by software, no authorization is required to use the loaded key and the key may be either *sign* or *encrypt*.

Compute a static DH key: TPM2_ECDH_ZGen(). This command takes as input a loaded key with the private portion d along with the corresponding public parameters, and an elliptic curve point P. The TPM first verifies whether P matches with the public parameters. If the verification passes, the TPM computes $Z := [d]P$ and outputs Z. Since this operation uses the private portion of an ECC key, authorization of the key is required. The attributes of the key is the *restricted* attribute CLEAR and the *decrypt* attribute SET.

Commit an ephemeral secret for key exchange: TPM2_EC_Ephemeral(). This command takes as input the public parameters for an ECC key with the elliptic curve point G, generates an ephemeral private portion of an ECC key r by using a counter technique as used in TPM2_Commit(), and computes a public key $P := [r]G$. The value of P is returned to the caller along with the counter value associated with r.

Compute a DH key: TPM2_ZGen_2Phase(). This command takes as input a scheme selector and the counter value returned by TPM2_EC_Ephemeral() along with the corresponding public parameters, recreates r and regenerates the associated public key. After that the TPM will "retire" the r value so that it will not be used again. This command can be used to achieve multiple key exchange protocols, which may have different operations. The scheme selector is used to tell the TPM which key exchange protocols should be implemented.

The TPM 2.0 specification also contains the TPM2_ActivateCredential() command which uses an ECC decryption algorithm internally. This command cannot be used for decryption of general purpose data. Therefore, we will not go into the detail of this commands.

3 Known ECC Use Cases for the TPM 2.0

The TPM 2.0 specification [2] supports three ECC primitives: conventional digital signatures, anonymous digital signatures that is called direct anonymous attestation, and Diffie-Hellman (DH) key exchange.

3.1 Conventional Digital Signatures

The following three conventional digital signature algorithms are mentioned in the TPM 2.0 specification [2].

1. ECDSA. The specification does not explain any details about this algorithm, but simply referring it to ISO/IEC 14888-3 [3]. This algorithm is originally described in NIST Fips 186-3 [23]. It is also defined in numerous other specifications, e.g. BSI TR-03111 [12].
2. ECSchnorr. The specification specifies an implementation of the EC Schnorr signature scheme, which is assigned as the TPM_ALG_ECSCHNORR scheme. The scheme includes the EC Schnorr signing operation and signature verification operation. The reference for the EC Schnorr signature scheme given in the TPM 2.0 specification is ISO/IEC 14888-3 [3].
3. SM2. The specification specifies the SM2 digital signature scheme, which is the Chinese EC-based signature scheme, originally published as the Chinese National Standard [27]. This digital signature scheme has recently been adopted by ISO/IEC and the process of adding it into an amendment of ISO/IEC 14888-3 [3] is in progress.

3.2 Direct Anonymous Attestation (DAA)

One of the main purposes of a TPM chip is to attest the state of the platform configuration to some verifier. This is basically been done by signing the values of platform configuration registers inside the TPM. It is an important privacy requirement that two attestations shall not be linkable. In the pre-DAA epoch, this has been accomplished by using a privacy certification authority (privacy CA). This basically worked in the following way. For each attestation, the TPM contacts the privacy CA and requests a new key, the "Attestation Identity Key" (AIK) together with a corresponding X.509 certificate. This enables privacy, because the verifier always sees a different public key. If the verifier gets two attestations, then he cannot tell if they came from two different TPMs or form the same one. In this case, the attestations are *unlinkable* from the verifier.

The downside of this approach is that the privacy CA is involved in every attestation. Furthermore this CA can link two signatures from the same TPM and can find which TPM was the signer. Therefore TGC were looking for a solution which didn't need the privacy CA: In the TCG history, DAA was the only cryptographic primitive that was designed to meet the TCG special privacy requirement. DAA is an anonymous digital signature. A DAA protocol accomplishes unlinkability by randomizing the signatures and associated certificates.

The first RSA DAA scheme was introduced by Brickell, Camenisch and Chen [10] for the TPM 1.2 specification [29]. The TPM 2.0 specification has been designed to support a new family of Elliptic Curve (EC) based DAA protocols. The TPM 2.0 specification [2] supports two different DAA protocols which

are based on pairings over elliptic curves. The first [17] is based on Camenisch-Lysyanskana (CL) credentials [14] and the second one [11] is based on sDH credentials [9]. The paper from Chen and Li [16] shows how both DAA protocols can be used with a TPM 2.0 chip.

3.3 DAA with attributes (DAA-A)

Chen and Urian [18] have recently preposed an extension of DAA by adding multiple attributes. This protocol is related to the U-Prove protocol but has a significant advantage over it: In contrast to the U-prove protocol, DAA-A is *multi-show unlinkable*. The DAA-A protocol comes in two variants, which correspond to the respective ECDAA protocols:

- the CL DAA-A protocol which corresponds to the CL ECDAA protocol.
- the sDH DAA-A protocol which corresponds to the sDH ECDAA (aka Epid) protocol.

In a nutshell, the DAA-A protocol works as follows: Each attribute value will be encoded as an exponent for an ECC key which is normally stored on the host but can also be for better security stored on the TPM. The DAA-A Issuer defines the list of attributes which shall be used in a DAA-A credential. According to the minimum disclosure principle, the TPM/host shall only reveal a minimum set of attributes to the Verifier. The TPM/host decide on each individual attestation what attributes they will reveal to the Verifier and what attributes they will hide form him. The revealed attributes will be sent by the TPM/host to the Verifier as part of the DAA-A Sign protocol. The correctness of the hidden attributes will be proved to the Verifier by a zero-knowledge proof.

Attributes can be stored on the host or on the TPM. TPM hosted attributes are stored as conventional signature keys. The DAA-A scheme uses the TPM2_Sign() and TPM2_Commit() commands, specified in ISO/IEC 11889 [2], as sub-protocols to aid in the generation of the DAA-A signature (see [18] for details).

Due to the proposed change of the TPM2_Sign() command for ECDAA (see 2.3), the integration of DAA-A with this command must also be changed. We leave how to handle this new adaption as an open problem, and from our point of view this problem is not trivial.

3.4 U-Prove

The U-Prove protocol [25] from Microsoft is an attribute based protocol with user controlled selective disclosure. The paper of Chen and Li [16] shows how U-Prove can be integrated with a TPM 2.0 chip. But the U-Prove protocol has the severe drawback that it is not multi-show unlinkable. The reason for this is that the authentication token of the U-Prove protocol is signed by a Schnorr-like signature and the signature value can be used as a correlation handle. To be unlinkable, a U-Prove token may only be used once.

3.5 Key Exchange

The following Diffie-Hellman (DH) based key exchange schemes in the EC setting are mentioned in the TPM 2.0 specification [2]. Interestingly this technique is called "secret sharing" in the TPM 2.0 specification. Secret sharing has been broadly used as a different cryptographic protocol, in which multiple entities each holds a share of a common secret and a number of these entities can work together and retrieve such a secret. In order to avoid any confusion, we name this technique "key exchange" throughout the paper.

1. One-Pass DH. The specification specifies the one-pass DH key exchange scheme and refers it to NIST SP800-56A [26].
2. Two-Pass DH. The specification specifies the two-pass DH key exchange scheme, which is also from NIST SP800-56A [26].
3. ECMQV. The specification specifies the two-pass DH key exchange scheme, which is known as EC-based MQV [26].
4. SM2 key exchange. The specification specifies the two-pass DH key exchange scheme from the SM2 family, the Chinese National Standard on ECC [27].

4 New ECC Use Cases for TPM 2.0

In this section we will discuss how cryptographic protocols can be used with a TPM 2.0, although they have not been mentioned in the specification yet. First we will show that a TPM can nicely be integrated in asymmetric encryption schemes. Then we will show the limitation of TPM integration by discussing some variants of signature algorithms.

4.1 Asymmetric Encryption

Based on the TPM 2.0 specification [2, 30], ECC is not used directly for encryption. It is well-known that in ECC, a key exchange functionality is used to establish a symmetric key from an ECC key, and then a symmetric algorithm is used for data encryption, which is known as the hybrid encryption, i.e., Key Encapsulation Mechanism and Data Encapsulation mechanism (KEM-DEM). The TPM 2.0 specification does not specify any KEM-DEM scheme. In this section, we demonstrate how to use TPM 2.0 to implement the ElGamal based KEM schemes in the ECC setting from ISO/IEC 18033-2 [4]. For the performance consideration (as a TPM chip is not efficient for data encryption/decryption compared with software), a DEM scheme would likely be implemented by software, and therefore we do not discuss it in this paper.

Generally speaking, a KEM consists of three algorithms:

– A key generation algorithm $\mathsf{KEM.KeyGen}()$ that takes as input the public system parameters par and outputs a public-key/private-key pair (pk, sk).

- An encryption algorithm KEM.Encrypt() that takes as input (pk, par) and outputs a secret-key/ciphertext pair (K, C).
- A decryption algorithm KEM.Decrypt() that takes as input (sk, C, par) and outputs K.

The public system parameters par depend on the particular scheme, and in the ECC setting they should include an elliptic curve defined over a given finite field, a subgroup of the elliptic curve group with a prime order q and a generator G, a hash function HASH() and a key derivation function KDF(). For simplicity, we omit other items in par.

ISO/IEC 18033-2 [4] specifies three ElGamal-based KEM schemes in the ECC setting. Respectively, they are ECIES (Elliptic Curve Integrated Encryption Scheme) based on the work of Abdalla, Bellare, and Rogaway [7, 8], PSEC (Provably Secure Elliptic Curve encryption) based on the work of Fujisaki and Okamoto [22] and ACE (Advanced Cryptographic Engine) based on the work of Cramer and Shoup [21, 20]. Recently a new submission of the ElGamal-based KEM scheme in the ECC setting [24] has been adopted by ISO/IEC and an amendment of ISO/IEC 18033-2 specifying this new scheme is in progress. This scheme is called FACE (Fast ACE).

Table 3 shows the algorithms in these four KEMs. Note that we have changed the notation used in ISO/IEC 18033-2 for the purpose of this paper, because we want to demonstrate that the same set of TPM functions can be used for all the three KEMs.

By using a TPM 2.0 chip to operate a KEM, we mean that the TPM generates a public-key/private-key pair, stores the key pair in the TPM protected environment and uses the private-key to decrypt a secret key, which is used for the DEM operation in a later stage. For the best use of the TPM, we only make use of the TPM for the operations involving the private-key and leaves other operations, such as the KEM.Encrypt() algorithm, the KDF() function and the HASH() function, to the software. With this consideration, these three KEM schemes can be implemented by using the same TPM ECC functionalities.

Now, let us see how to implement the KEM.KeyGen() and KEM.Decrypt() algorithms in ECIES and PSEC using a number of TPM 2.0 commands, which were introduced in Section 2.2. We assume that a caller enabling to run these software operations mentioned before has authorization to use the TPM commands as follows.

1. In the KEM.KeyGen() algorithm, The caller first chooses an existing TPM key as a parent key parentK. If the key is stored outside of the TPM, the caller uses the TPM2_Load() command to load the key and receives a key handle parentK.handle from the TPM. In order to generate the public-key/private-key pair `tk` $= (pk, sk)$, where $sk = x$ and $pk = Y = [x]G$, the caller calls TPM2_Create(), that takes as input the public system parameters par along with parentK.handle, generates the key pair `tk` and a key blob `tk.blob`, and outputs the blob. Recall that `tk.blob` includes sk encrypted under parentK, pk and a tag to check integrity.

	KEM.KeyGen(par)	KEM.Encrypt(pk, par)	KEM.Decrypt(sk, C, par)
ECIES [7, 8]	$x \in [1, q)$ $Y = [x]G$ $sk \leftarrow x$ $pk \leftarrow Y$ Return (pk, sk)	$r \in [1, q)$ $C_0 = [r]G$, $C = C_0$ $D = [r]Y$ $K = \text{KDF}(C_0 \| \| D)$ Return (K, C)	$C_0 = C$ $D = [x]C_0$ $K = \text{KDF}(C_0 \| \| D)$ Return K
PSEC [22]	$x \in [0, q)$ $Y = [x]G$ $sk \leftarrow x$ $pk \leftarrow Y$ Return (pk, sk)	$seed \in \{0,1\}^{seedLen}$ $t = u \| \| K = \text{KDF}(0 \| \| seed)$ $r = u \mod q$ $C_0 = [r]G$, $D = [r]Y$ $E = \text{KDF}(1 \| \| C_0 \| \| D)$ $C = C_0 \| \| (seed \oplus E)$ Return (K, C)	Parse $C = C_0 \| \| F$ $D = [x]C_0$ $E = \text{KDF}(1 \| \| C_0 \| \| D)$ $seed = F \oplus E$ $t = \text{KDF}(0 \| \| seed) = u \| \| K$ $r = u \mod q$ Return K, if $C_0 = [r]P$ Otherwise, return Fail
ACE [21, 20]	$x_1, x_2, x_3, x_4 \in [0, q)$ $Y_1 = [x_1]G$ $Y_2 = [x_2]G$ $Y_3 = [x_3]G$ $Y_4 = [x_4]G$ $sk \leftarrow (x_1, x_2, x_3, x_4)$ $pk \leftarrow (Y_1, Y_2, Y_3, Y_4)$ Return (pk, sk)	$r \in [0, q)$ $C_0 = [r]G$ $D_1 = [r]Y_1$, $D_4 = [r]Y_4$ $\alpha = \text{HASH}(C_0 \| \| D_1)$ $r' = \alpha \cdot r \mod q$ $E = [r]Y_2 + [r']Y_3$ $C = C_0 \| \| D_1 \| \| E$ $K = \text{KDF}(C_0 \| \| D_4)$ Return (K, C)	Parse $C = C_0 \| \| D_1 \| \| E$ $\alpha = \text{HASH}(C_0 \| \| D_1)$ $t = x_2 + x_3 \cdot \alpha \mod q$ If $D_1 \neq [x_1]C_0 \vee E \neq [t]C_0$ return Fail Otherwise calculate $D_4 = [x_4]C_0$ $K = \text{KDF}(C_0 \| \| D_4)$ Return K
FACE [24]	$a_1, a_2 \in [0, q)$ $G_1 = [a_1]G$ $G_2 = [a_2]G$ $x_1, x_2, y_1, y_2 \in [0, q)$ $C = [x_1]G_1 + [x_2]G_2$ $D = [y_1]G_1 + [y_2]G_2$ $sk \leftarrow (x_1, x_2, y_1, y_2)$ $pk \leftarrow (G_1, G_2, C, D)$ Return (pk, sk)	$r \in [0, q)$ $U_1 = [r]G_1$ $U_2 = [r]G_2$ $\alpha = \text{HASH}(U_1 \| \| U_2)$ $r' = \alpha \cdot r \mod q$ $V = [r]C + [r']D$ $K \| T = \text{KDF}(V)$ $C = U_1 \| \| U_2 \| \| T$ Return (K, C)	Parse $C = U_1 \| \| U_2 \| \| T$ $\alpha = \text{HASH}(U_1 \| \| U_2)$ $t_1 = x_1 + y_1 \cdot \alpha \mod q$ $t_2 = x_2 + y_2 \cdot \alpha \mod q$ $V = t_1 U_1 + t_2 U_2$ $K \| T' = \text{KDF}(V)$ Return K, if $T = T'$ Otherwise, return Fail

Table 3. Four KEMs in ISO/IEC 18033-2 [4] and ISO/IEC 18033-2/AMD1 [5]

2. In the KEM.Decrypt() algorithm, the caller first loads the key pair `tk` into the TPM using TPM2_Load() that will return a tk.handle. The caller then calls TPM2_ECDH_ZGen() with the input tk.handle and the value $C_0 = [r]P$. The TPM will computes and outputs $D = [x]C_0$. The caller can take care of the remaining operations using software to obtain the secret key K, and in PSEC the K value can be obtained if the necessary check $C_0 = [r]P$ passes; otherwise the caller will get a Fail message.

In the ACE KEM, the key pair $tk = (pk, sk)$ consists of four private-key values $sk = (x_1, x_2, x_3, x_4)$ and four corresponding public-key values $pk = (Y_1, Y_2, Y_3, Y_4)$. The caller can treat them as four independent key pairs $tk_1 = (Y_1, x_1)$, $tk_2 = (Y_2, x_2)$, $tk_3 = (Y_3, x_3)$ and $tk_4 = (Y_4, x_4)$. In the KEM.KeyGen() algorithm, the caller runs the operation in the first bullet four times each obtain-

ing one key pair tk_i for $i = [1, 4]$. In the KEM.Decrypt() algorithm, again the caller calls the same TPM commands in the second bullet four times, each with C_0 as input but loading a different key pair tk_i to obtain (D_1, D_2, D_3, D_4). Obviously the caller can verify the value E since $E = D_2 + [\alpha]D_3$ and $\alpha = \mathsf{Hash}(C_0||D_1)$. By following the remaining part of the decryption algorithm, the caller can verify the ciphertext C and retrieve the secret key K if the verification succeeds or obtain a Fail message if the verification fails.

The FACE scheme first might need some explanation regarding the value T. The KDF function for this scheme does not only generate the bits for a key K, but instead generates some additional bits for the so called *Tag* value T. The size of the Tag value are defined in the system parameters.

The FACE KEM algorithm has four public keys and four private keys. But the private and public keys are not directly related. In the KEM.KeyGen() algorithm, one has to calculate two public points G_1 and G_2 without corresponding private keys. To use a TPM here, one can use two invocations of the TPM2_ECDH_KeyGen() TPM command to generate them as ephemeral points. Then four private keys x_1, x_2, y_1, y_2 must be generated and two further public keys: $C = [x_1]G_1 + [x_2]G_2$ and $D = [y_1]G_1 + [y_2]G_2$. The TPM calculates the intermediate points $X_1 = [x_1]G_1$, $X_2 = [x_2]G_2$, $Y_1 = [y_1]G_1$, $Y_2 = [x_2]G_2$ with four TPM2_ECDH_ZGen() command calls. The host then finalises the calculation by adding the points to get $C = X_1 + X_2$ and $D = Y_1 + Y_2$. In the KEM.Decrypt() algorithm, the receiver has to calculate $t_1 = x_1 + \alpha y_1$ and $t_2 = x_2 + \alpha y_2$ and then $V = [t_1]U_1 + [t_2]U_2$. In order to use the TPM, one must a bit rearrange the equations. The TPM calculates the intermediate points $X_1 = [x_1]U_1$, $X_2 = [x_2]U_2$, $Y_1 = [y_1]U_1$, $Y_2 = [y_2]U_2$ with four TPM2_ECDH_ZGen() command calls. The host then finalises the calculation by computing the point $V = X_1 + X_2 + [\alpha](Y_1 + Y_2)$.

4.2 Limitations of Algorithmic Agility

The ECDSA algorithm implemented in the TPM is described in NIST Fips 186-3 [23]. It is also defined in numerous other specifications, e.g. ISO/IEC 14888-3 [3] and BSI TR-03111 [12]. Despite this standard ECDSA scheme, [3] also describes three further national schemes:

- EC-GDSA (Elliptic Curve German Digital Signature Algorithm)
- EC-KCDSA (Elliptic Curve Korean Certificate-based Digital Signature Algorithm)
- EC-RDSA (Elliptic Curve Russian Digital Signature Algorithm)

It would be nice if the current TPM 2.0 specification could also be integrated in those schemes. But this seems to be impossible. Generally speaking, in order to integrate a TPM for implementing an algorithm, one has to split the algorithm into two parts in such a way that the TPM can calculate one part, and the host can calculate the remaining part. It is thereby crucial that the host only performs the operation that needs the public keys only. Every operation involing

the private key must be done by the TPM. Such a splitting can be done easily if the underling primitive is as simple as an ECDH point multiplication. This was the case in the KEM schemes above. But the ECDSA-type of signature schemes require to make more complicated operations on the private key. For instance, there is no obvious way to calculate $[x^{-1}]G$ by using the existing TPM commands in which the public key is formed as $Y = [x|G$.

Now, instead of implementing each algorithm separately on a TPM, a suggestion for a future TPM related research could be to split the different signature algorithms in simple "atomic" pieces, where the private key parts can be easily implemented on a TPM.

5 Compatibility Issue in Algorithm Agility

Algorithm compatibility is crucial for algorithm agility. However it is a common practice in cryptographic standardization to ignore this. That means, different standards for the same cryptographic protocol often use different and incompatible implementation choices. This will not be an issue if the TPM has been considered at the time when the cryptographic protocol is designed. But it will be a problem if the TPM shall be used to enhance the security for an already existing cryptographic system.

The TCG noticed this especially for the elliptic curve based Schnorr signature scheme. Therefore, they decided to revise the current Schnorr implementation in the TPM 2.0 specification in order to optimize interoperability. By the date of writing this paper, the final version of this revision has not been done. The following discussion shows the problems one faces by trying to reach a maximum amount of interoperability.

The public system parameters *par* for the EC-based digital signature scheme also depend on the particular scheme, and they should include an elliptic curve defined over a given finite field, a subgroup of the elliptic curve group with a prime order q and a generator G, and a hash function HASH(). We use x to denote the private key and Y for the public key. For simplicity, again we omit other items in *par*.

The Schnorr signature algorithm basically consists of the following steps:

1. Choose a random value r and calculate the point $R = [r]G$.
2. Calculate the signature value c by hashing the x-coordinate R_x of the point R and a given message M, c = HASH(M, R_x). See the discussion below for the different choices how this hashing can be done on the bit level.
3. Calculate the signature value s. Here we have the two choices to calculate either (a) $s = r + c \cdot x \mod q$ or (b) $s = r - c \cdot x \mod q$.
4. Return the signature (c, s).

The Schnorr signature verification algorithm to the signature (c, s) for the message M consists of the following steps:

1. Calculate the point R'. Here we must use the correct version corresponding to the sign variant, i.e. either (a) $R' = [s]G - [c]Y$ or (b) $R' = [s]G + [c]Y$.

2. Calculate the signature value c' by hashing the x-coordinate R'_x of the point R' and message M as $c' = \text{HASH}(M, R'_x)$.
3. Return Accept, if $c = c'$ or Reject, otherwise.

Note that the different calculation variants for the signature value s can easily be transformed into each other by inverting the s value, i.e. if (c, s) is a signature for variant (a), then $(c, -s)$ will be a signature for variant (b) (and vice-versa).

Let us now discuss the different hash calculation variants. The first decision to make is the bit encoding of the value R_x. Since R_x is an element of the finite field with q elements, it can be encoded as a byte string of length $\lceil \log_{256}(q) \rceil$. Let this be the default encoding. This encoding can contain leading zero bytes. As an alternative encoding one can strip down those leading zeroes from the default encoding. Let the $\text{TRZ}(x)$ denote the function which truncates the leading zeroes from the default encoded byte string x. The next choice we must make is in which order the message M and the value R_x will enter the HASH function, i.e. either as $h = \text{HASH}(M||R_x)$ or as $h = \text{HASH}(R_x||M)$. The next choice regards the truncation of the value h. This is necessary only if the bit size λ of the hash result is bigger than the bit size l of the binary encoding of the number q, i.e. $l = \lceil \log_2 q \rceil$. Here one has the choices to either leave the value as it is or truncate the $\lambda - l$ least significant bits of h. Let us denote this truncation of a bit string x as $\text{TRH}(x)$. As a last choice, we can now set $c = h$ or reduce h first to $h' = h \mod q$ and set $c = h'$.

For comparison, in Table 4, we list the three existing implementations of the EC Schnorr signature scheme in ISO/IEC 14888-3 [3], ISO/IEC 11889 [2] and BSI TR-03111 [12] respectively, and a new implementation proposed by the TCG recently [32].

	Sign(par, x, M)	Verify(par, Y, s, c, M)
ISO/IEC 14888-3 [3]	$r \in [1, q)$, $R = [r]G$ $c = \text{HASH}(R_x\|\|M)$ $s = r + c \cdot x \mod q$	$R' = [s]G - [c]Y$ $c' = \text{HASH}(R'_x\|\|M)$ Accept, iff $c = c'$
BSI TR-03111 [12]	$r \in [1, q)$, $R = [r]G$ $c = \text{TRH}(\text{HASH}(M\|\|R_x))$ $s = r - c \cdot sk \mod q$	$R' = [s]G + [c]Y$ $c' = \text{TRH}(\text{HASH}(M\|\|R'_x))$ Accept, iff $c = c'$
ISO/IEC 11889 [2]	$r \in [1, q)$, $R = [r]G$ $c = \text{HASH}(M\|\|(\text{TRZ}(R_x \bmod q)) \bmod q$ $s = r + c \cdot x \mod q$	$R' = [s]G - [c]Y$ $c' = \text{HASH}(M\|\|(\text{TRZ}(R'_x \bmod q)) \bmod q$ Accept, iff $c = c'$
New TCG proposal [32]	$r \in [1, q)$, $R = [r]G$ $c = \text{TRH}(\text{HASH}(R_x\|\|M))$ $s = r + c \cdot x \mod q$	$R' = [s]G - [c]Yk$ $c' = \text{TRH}(\text{HASH}(R'_x\|\|M)$ Accept, iff $c = c'$

Table 4. Different EC Schnorr implementation variants

6 Performance considerations

TPM chips are optimized to provide a high level of hardware security. This means they have to be resistant against sophisticated physical attacks, like fault injection or side channel leakage. Security certifications according to Common Criteria or FIPS give evidence for this security level. TPM chips are also required to be cost optimized devices. This implies that they will be somewhat restricted regarding processor speed and memory resources. However the performance of TPM chips is continuously increasing due to higher clock frequencies, sophisticated cryptographic co-processors and firmware optimizations.

As a TPM chip is normally invoked by the software stack of an multitasking operating system, the performance also depends on that software part. It is therefore difficult to provide meaningful performance measurements for TPM chips.

The bottom line is that a host CPU is faster but provides no hardware security while the TPM chip is slower but provides a far high level of hardware security. Due to this performance/security asymmetry, it is very important to cleverly split the algorithm between the host CPU and the TPM chip. The TPM should only perform the operations involving the private key.

7 Conclusion with an Open Question

In this paper, we have shown that a TPM 2.0 chip is a reasonably powerful cryptographic engine, which can potentially achieve more than what have be specified in its published specification [2]. This benefits from the property of algorithm agility. However, the algorithm agility has made the environment much more complex than these algorithms individually implemented and analyzed in their original security proof. Therefore, it is a real challenge to make a sound security analysis for the entire TPM/host system. This paper has not done anything in this topic. We finish this paper with an open question: How to define the security notion of algorithm agility? On the other words, whether it is possible and then how to build a security model for TPM 2.0 ECC functionalities and to prove it?

References

1. ISO/IEC 11889:2009 (all parts) Information technology – Trusted platform module.
2. ISO/IEC 11889:2015 (all parts) Information technology – Trusted platform module library.
3. ISO/IEC 14888-3:2016 Information technology – Security techniques – Digital signatures with appendix – Part 3: Discrete logarithm based mechanisms.
4. ISO/IEC 18033-2:2006 Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers.
5. ISO/IEC 18033-2/amd1 Encryption algorithms – Part 2: Asymmetric ciphers – Amendment 1.

6. ISO/IEC 20008-2:2013 Information technology – Security techniques – Anonymous digital signatures – Part 2: Mechanisms using a group public key.
7. Michel Abdalla, Mihir Bellare, and Phillip Rogaway. DHAES: an encryption scheme based on the Diffie- Hellman problem. Cryptology ePrint Archive, Report 1999/007, 1999. http://eprint.iacr.org.
8. Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In *In Topics in Cryptology - CT-RSA 2001*, volume 2045 of *LNCS*, pages 143–158. Springer, 2001.
9. Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *Advances in Cryptology — EUROCRYPT '04*, volume 3027 of *LNCS*, pages 56–73. Springer, 2004.
10. 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.
11. 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.
12. BSI. Technical Guideline TR-03111, Elliptic Curve Cryptography, v2.0. BSI, 2012. `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03111/BSI-TR-03111_pdf.html`.
13. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. In *Public-Key Cryptography – PKC 2016*, volume 9615 of *LNCS*, pages 234–264. Springer, 2016.
14. 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.
15. Liqun Chen, Ming-Feng Lee, and Bogdan Warinschi. Security of the enhanced TCG privacy-CA solution. In *Proc. 6th International Symposium on Trustworthy Global Computing (TGC 2011)*, volume 7173 of *LNCS*, pages 121–141. Springer, 2011.
16. Liqun Chen and Jiangtao Li. Flexible and scalable digital signatures in tpm 2.0. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, pages 37–48. ACM Press, 2013.
17. 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.
18. Liqun Chen and Rainer Urian. DAA-A: Direct anonymous attestation with attributes. In *TRUST 2015*, volume 9229 of *LNCS*, pages 228–245. Springer.
19. Liqun Chen and Bogdan Warinschi. Security of the TCG privacy-CA solution. In *Proc. 6th IEEE/IFIP International Symposium on Trusted Computing and Communications (TrustCom 2010)*, pages 609–616. IEEE Press, 2010.
20. Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. Cryptology ePrint Archive, Report 2001/108, 2001. http://eprint.iacr.org.
21. Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology-Crypto '98*, pages 13–25. Springer, 1998.
22. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology — CRYPTO '99*, volume 1666 of *LNCS*, pages 537–554. Springer, 1999.

23. Patrick Gallagher, Deputy Director Foreword, and Cita Furlani Director. Fips pub 186-3 federal information processing standards publication digital signature standard (dss), 2009.
24. Kaoru Kurosawa and Le Trieu Phong. Kurosawa-Desmedt key encapsulation mechanism, revisited and more. In *AFRICACRYPT 2014*, volume 8469 of *LNCS*, pages 51–68. Springer, 2014.
25. Microsoft U-Prove Community Technology. U-Prove cryptographic specification version 1.1, 2013. `http://www.microsoft.com/u-prove`.
26. National Institute of Standards and Technology. Recommendation for pair-wise key estabishment schemes using discrete logarithm cryptography. Special Publication 800-56A, March 2007.
27. Chinese National Standards. Public key cryptographic algorithm SM2 based on elliptic curves – Part 2: digital signature algorithm.
28. TCG. TCG algorithm registry. Committee Draft, January 7, 2016.
29. Trusted Computing Group. TCG TPM specification 1.2, 2003. `http://www.trustedcomputinggroup.org`.
30. Trusted Computing Group. TCG TPM library 2.0, 2014. `http://www.trustedcomputinggroup.org/tpm-library-specification/`.
31. Xiaoyun Wang, Yiqun Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Crypto 2005*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005.
32. David Wooten. Final schnorr algorithm. email to TCG TPMWG, 2016.
33. Li Xi, Kang Yang, Zhenfeng Zhang, and Dengguo Feng. DAA-related APIs in TPM 2.0 revisited. In *TRUST 2014*, volume 8564 of *LNCS*, pages 1–18. Springer, 2014.