Type-Based Analysis of Protected Storage in the TPM

Jianxiong Shao, Dengguo Feng, and Yu Qin

Trusted Computing and Information Assurance Laboratory,
Institute of Software, Chinese Academy of Sciences
{shaojianxiong,feng,qin_yu}@tca.iscas.ac.cn

Abstract. The Trusted Platform Module (TPM) is designed to enable trustworthy computation and communication over open networks. The TPM provides a way to store cryptographic keys and other sensitive values in its shielded memory and act as *Root of Trust for Storage* (RTS). The TPM interacts with applications via a predefined set of commands (an API). In this paper, we give an abstraction model for the TPM 2.0 specification concentrating on Protected Storage part. With identification and formalization of their secrecy properties, we devise a type system with asymmetric cryptographic primitives to statically enforce and prove their security.

Keywords: TPM, Trusted computing, Type system, API analysis.

1 Introduction

The Trusted Platform Module (TPM) is a system component designed to establish trust in a platform by providing protected storage, robust platform integrity measurement, secure platform attestation and other security mechanisms. The TPM specification is an industry standard [12] and an ISO/IEC standard [11] coordinated by the Trusted Computing Group. The TPM is separate from the system on which it reports (the host system) and the only interaction is through the interface (API) predefined in its specification.

In the last few years, several papers have appeared to indicate vulnerabilities in the TPM API designs. These attacks highlight the importance of formal analysis of the API commands specifications. A number of efforts have analyzed secrecy and authentication properties of protocols using model checkers, theorem provers, and other tools. Backes *et al.* used ProVerif to obtain the first mechanized analysis of DAA protocol[2]. In [6], a TPM impersonation attack was discovered when sharing authdata between users are allowed. Lin described an analysis of various fragments of the TPM API using Otter and Alloy[10]. In [9], an analysis of the TPM API was described by using finite state automata. Delaune *et al.* used the tool ProVerif to analyze the API commands and rediscover some known attacks and some new variations on them[7].

Most of the established work on formal analysis of TPM API commands and protocols focus on original TPM 1.2 specification, whose latest revision [12] is in

S. Qing et al. (Eds.): ICICS 2013, LNCS 8233, pp. 135–150, 2013.

2006. However, Trusted Computing Group (TCG) has published the TPM 2.0 specification on their website in 2012. The new version of the TPM specification has several changes from previous versions especially on the protected storage part. In this paper, we conduct a formal analysis of the protected storage part of API commands in the TPM 2.0 specification w.r.t secrecy property. A formal security proof of secrecy, in the presence of a Dolev-Yao attacker who have complete control over all the existent sessions, is first proposed based on a core type system statically enforcing API security.

Our present work extends the line of research by exploring a language-based, static analysis technique that allows for proving the security of key management API commands. In [4], Centenaro *et al.* devise a language to specify PKCS#11 key management APIs at a fine granularity. We utilize their abstraction of key templates in our model but devise a new type system to check information flow properties for cryptographic operations in security APIs. Moreover, we devise a new set of assignment commands to specify the internal functions according to Trusted Platform Module Library (TPML) 2.0, Part 4: Supporting Routines.

For the core type system, although Centenaro *et al.* considered the integrity (they call it trust) for keys, the key with high integrity in their model must be with high confidentiality. It cannot be used to formalize asymmetric cryptographic primitives since the public key should be with high integrity but low confidentiality. In our model, we devise a new type system with more specific types for asymmetric cryptographic primitives. Actually in this sense our result is more in the line of [13], in which Keighren *et al.* proposed a type system based on the principles of information flow to investigate a much stronger property *noninterference* for a general model. Yet they gave no language to express the internal commands and did not consider the integrity level. We apply the types of keys from [8,1,14], in which the types of the payload are determined by the types of the key. We also consider the integrity level, which is different from [13].

The paper is organized as follows. In section 2 we give a brief introduction to the protected storage part of the TPM 2.0 specification and describe the simple language for modeling TPM commands. In section 3 we introduce the core type system statically enforcing API security. In section 4 we apply the type system to our model of the TPM API commands, which we prove to be secure. We conclude in section 5.

2 Overview of the TPM Protected Storage

Trusted Platform Module (TPM) is defined as the Root of Trust for Storage (RTS) by TCG, since the TPM can be trusted to prevent inappropriate access to its memory, which we call Shielded Locations. TPM protections are based on the concept of Protected Capabilities and Protected Objects. A Protected Capability is an operation that must be performed correctly for a TPM to be trusted. A Protected Object is data (including keys) that can be accessed to only by using Protected Capabilities. Protected Objects in the TPM reside in Shielded Locations. The size of Shielded Locations may be limited. The effective

memory of the TPM is expanded by storing Protected Objects outside of the TPM memory with cryptographic protections when they are not being used and reloading if necessary.

2.1 Protected Storage Hierarchy

In the TPM 2.0 specification, the TPM Protected Objects are arranged in a tree structure, which is called Protected Storage Hierarchy. A hierarchy is constructed with storage keys as the connectors to which other key objects or connectors may be attached. A Storage Key, acting as a parent, protects its children when those objects are stored out of the TPM. Storage keys should be used in the process of creation, loading, duplication, unsealing, and identity activation. However, such keys cannot be used in the cryptographic support functions.

When creating a new object on the device, two commands are needed. In the command `TPM2_Create()`, a loaded storage key should be provided as the parent and a loadable creation blob protected by it is created. The keys used in this protection are derived from a seed in the sensitive area of the parent object. Then the command `TPM2_Load()` may load the creation blob into the TPM with a handle returned. The new key can be used by reference to its handle.

We focus on the process of duplication, which needs three commands. Duplication allows an object to be a child of an additional parent key. In the command `TPM2_Duplicate()`, a loaded object for duplication and its new parent handle should be provided and a duplication blob is returned. The duplication blob is protected in a similar way to the creation blob except that the seed is random and protected by new parent's asymmetric methods to guarantee that only the new parent may load it. In this way, the storage key must be asymmetric. In the command `TPM2_Import()`, the duplication blob is transformed to a loadable blob, which can be loaded in `TPM2_Load()`.

An object might be connected to another hierarchy by two ways. One is to duplicate it directly by the process above. The other is to duplicate one of its ancestors and it can be loaded by its creation blob. The hierarchy attributes of an object, *FixedParent* and *FixedTPM*, indicate how the object can be connected to another hierarchy. An object with *FixedParent* SET means it cannot be duplicated directly and with *FixedTPM* SET means all of its ancestors have *FixedParent* SET. Thus an object with *FixedParent* CLEAR must have *FixedTPM* CLEAR. The attribute *FixedTPM* of an object depends on *FixedTPM* in its parent and *FixedParent* in itself. The hierarchy attributes setting matrix is in Table 1

The consistency of the hierarchy settings is checked by internal function `PublicAttributesValidation()` in object templates (when creating) and in public areas for loaded objects (when loading) or duplicated objects (when importing). The root of a hierarchy is denoted as the Primary Object which is protected by keys derived from a Primary Seed and its attributes. The Primary Object can be seen as a child object of a virtual object with *FixedTPM* SET.

Table 1. Allowed Hierarchy Settings

Parent's *FixedTPM*	Object's *FixedParent*	Object's *FixedTPM*
CLEAR	CLEAR	CLEAR
CLEAR	SET	CLEAR
SET	CLEAR	CLEAR
SET	SET	SET

2.2 Object Structure Elements

According to the TPM 2.0 specification, each object has two components: public area and sensitive area. The former contains the fields `objectAttributes` and `type`. For an asymmetric key object, the public key should also be contained in the public area. The sensitive area includes an authorization value (`authValue`), a secret value used to derive keys for protection of its child (`seedValue`), and the private key (`sensitive`) dependant on the type of the object.

For the public area, the attributes of the object (`objectAttributes`) are in 5 classes: hierarchy, usage, authorization, creation, and persistence. The hierarchy attributes have been discussed above.

The usage of an object is determined by three attributes: *restricted*, *sign*, and *decrypt*. An object with only *decrypt* SET may use the key in its sensitive area to decrypt data blobs that have been encrypted by that key (for symmetric key) or the public portion of the key (for asymmetric key). Thus we call it Decryption Key Object. An object with both *decrypt* and *restricted* SET is used to protect the other objects when they are created or duplicated. A restricted decryption key is often referred to as a Storage Key Object. An object with *sign* SET may perform signing operation and with both *sign* and *restricted* SET may only sign a digest produced by the TPM. This two kinds of objects corresponds to the secure platform attestation. On the viewpoint of the protected storage, they act the same way as the Decryption Key Objects and could not be used as the Storage Key Objects. It is the same case for a legacy key with both *sign* and *decrypt* SET. It is not allowed for an object with all the three attributes SET. Thus we divide all the objects into two groups: `Decryption Key Object` and `Storage Key Object` which correspond to the leaf node and the branch node.

For the sensitive area, `seedValue` is required for `Storage Key Objects`. It is an obfuscation value for `Decryption Key Object`.

3 Modeling the TPM APIs

3.1 A Language for Modeling TPM Commands

In this section, we extend the work of [4] to get an imperative language which is more suitable to specify the protected storage part of TPM 2.0 APIs.

Values and Expressions. Let $\mathcal{C}$ and $\mathcal{F}$ respectively denote the set of atomic constant and fresh values with $\mathcal{C} \bigcap \mathcal{F} = \emptyset$. The former specifies any public data, including the templates of the key objects and the usage of the key derivation function (kdf). The latter is used to model the generation of new fresh values such as the sensitive values and the seed values of the key objects. We introduce the extraction operator $f \leftarrow \mathcal{F}$ in [4] to represent the extraction of the first 'unused' value f from $\mathcal{F}$. It is obvious that the extracted values are always different. We define the values in Table 2. For the sake of readability, we let $\tilde{v}$ denote a tuple $(v_1, \cdots, v_k)$ of values.

Table 2. Definition of Values and Expressions

$v, v', h ::=$	values	$e ::=$	expressions
val	atomic fresh value	x, y	variables
tmp	template	$kdf(usg, x)$	key diversification
usg	$\{STORAGE, INTEGRITY\}$	$ek(x)$	encryption key
$kdf(usg, v)$	key diversification	$senc(x, \tilde{y})$	sym encryption
$senc(v', \tilde{v})$	sym encryption	$aenc(x, \tilde{y})$	asym encryption
$ek(v)$	encryption key	$hmac(x, \tilde{y})$	hmac computation
$aenc(v', \tilde{v})$	asym encryption		
$hmac(v', \tilde{v})$	hmac computation		

We use template to describe the properties of the key objects. Denoted by tmp, a template is represented as a set of attributes. Set an attribute for a key object is to include such an attribute in its template set. Formally, a template tmp is a subset of $\{W, E, A, S, N, F\}$. First, two attributes are used to identify the groups of key objects: W (wrap) for `Storage Key Object`; E (encryption) for `Decryption Key Object`. Second, we use A (Asymmetric) and S (Symmetric) to specify the field `type` in the public area of the key object. Third, for the hierarchy attributes *FixedTPM* and *FixedParent*, we use N (*Non-FixedParent*) to denote *FixedParent* CLEAR and F to denote *FixedTPM* SET. We do not specify the other attributes since they are irrelevant to the protected storage hierarchy. As in section 2, (W, E), (A, S), (N, F), and (W, S) are on the list of conflicting attribute pairs. Actually, the allowable combination of the attributes can only be of the form $\{W, A, N/F\}$, $\{E, A, N/F\}$, and $\{E, S, N/F\}$ where N/F means N, F or neither. We have 3 kinds of key objects which is denoted by `mode`: the Storage Key Object, the Symmetric Decryption Key Object, and the Asymmetric Decryption Key Object. We abstract such restrictions and focus on a particular set of all allowable templates of keys denoted by $\wp$, which we call the security policy. In our model, $\wp$ contains the above 9 possible templates.

Constant value $usg \in \{STORAGE, INTEGRITY\}$ is a label to specify the usage of the key derived from the *Seed* stored in a Storage Key Object. $STORAGE$ means a symmetric key and $INTEGRITY$ means an HMAC key. We use $kdf(usg, v)$ to denote a new key obtained via key derivation function from label usg and a seed value v. $senc(v', \tilde{v})$ performs symmetric encryption

on a tuple of values $\tilde{v}$. $ek(v)$ denotes the public encryption key value corresponding to the private key v and can be published. Notice that we model a cryptographic scheme where the encryption key can be recovered from the corresponding decryption key, which means decryption keys should be seen as key-pairs themselves. $aenc(v', \tilde{v})$ and $hmac(v', \tilde{v})$ denote, respectively, the asymmetric encryption and the HMAC computation of the tuple $\tilde{v}$ with the key v'.

As in [4], we use a set of expressions to manipulate the above values. Table 2 gives the formalization of expressions which are similar to those of values. Expressions are based on a set of variables $\mathcal{V}$. We introduce the memory environment $\mathbf{M} : x \mapsto v$ in [4] to denote the evaluation of variables. For simplicity, we let $\tilde{x}$ denote a tuple $(x_1, \cdots, x_n)$ of variables and $\mathbf{M}(\tilde{x}) = \tilde{v}$ the evaluation $\mathbf{M}(x_1) = v_1, \cdots, \mathbf{M}(x_n) = v_n$. Expression e in an environment $\mathbf{M}$ evaluating to v is denoted by $e \downarrow^{\mathbf{M}} v$. It is trivial to derive the semantics of evaluation for the expressions in Table 2.

Handle-Map. In the TPM 2.0 specification, objects are referenced in the commands via handles. We use a key handle-map $\mathbf{H} : h \mapsto (tmp, v_s, v_k)$ from a subset of atomic fresh values $\mathcal{F}$ to tuples of templates, seed values and key values. We do not consider the Authentication mechanisms in the TPM. This corresponds to a worst-case scenario in which attackers may gain access to all keys available in the TPM without knowing their values. Thus, for the sensitive area, we only need to model the field `seedValue` by v_s and `sensitive` by v_k. The type of sensitive values v_k and v_s is dependant on the template tmp.

API Commands and Semantics. We devise a set of internal functions according to the supporting routines in Trusted Platform Module Library 2.0 for object and hierarchy.

An API is specified as a set $\mathcal{A} = \{c_1, \cdots, c_n\}$ of commands. Each command contains a binding of values to variables and a sequence of inner execution of clauses as follows:

$c ::= \lambda \tilde{x}.p$

$p ::= \varepsilon |\ x := e |\ \texttt{return}\ \tilde{y} |\ p_1; p_2 |\ (x_t, x_s, x_k) := \texttt{checkTemplate}\ (y_h, tmp) |$
$\mathrm{x_k} := \texttt{genKey}\ (y_t) | x_s := \texttt{genSeed}\ (y_t) | x_h := \texttt{ObjectLoad}\ (y_s, y_k, y_t) |$
$(\mathrm{x_{pA}}, \mathrm{x_{inA}}) := \texttt{PAV}\ (y_{pA}, y_{inA}) | \tilde{x} := f$

$f ::= \texttt{sdec}\ (y_k, y_c) |\ \texttt{adec}\ (y_k, y_c) |\ \texttt{checkHMAC}\ (y_k, y_{hmac}, \tilde{y}_v).$

All of the free variables (variables that have no evaluation) in clauses p appear in input parameters $\tilde{x} = (x_1, \cdots, x_n)$. We will only focus on the API commands in which `return` $\tilde{y}$ can only occur as the last clause. Intuitively, ε denotes the empty clause; $x := e$ is an evaluation of variable x; $p_1; p_2$ recursively specifies the sequential execution of clauses. `chechTemplate` retrieves k_s, k_v, and tmp' of a key object loaded on the device, given its handle by requiring the template to match some pattern tmp. `genKey` and `genSeed` generate a new key value or seed value, given its template y_t. `ObjectLoad` loads a new key object with its sensitive values and an allowable template. `PAV` checks the hierarchy attributes in the template

y_{pA} of the parent object should be compatible with that in the template y_{inA} of an input object according to Table 1. The other three internal functions f are cryptography operations provided by the TPM and cannot be used directly by user applications. `sdec` and `adec` respectively specify the symmetric and asymmetric decryption. The decrypting function fails (ie. is stuck) if the given key is not the right one. `checkHAMC` checks whether $y_{hmac} = hamc(y_k, \tilde{y}_v)$ and if so, $\tilde{y}_v$ is evaluated to $\tilde{x}$, or otherwise, it fails. A call to an API command $c = \lambda(x_1, \cdots, x_k).p$, written as $c(v_1, \cdots, v_k)$, binds variables $x_1, \cdots, x_k$ to values $v_1, \cdots, v_k$, executes p and outputs the value given by `return` $\tilde{y}$.

For convenience, it is required that all the variables on the left side of the assignment clauses may appear only once. This does not limit the capability of our model since the repeated variables can be rewrite to different names.

An API command c working on a configuration contains a memory environment $\mathbf{M}$ and a key handle-map $\mathbf{H}$, which is denoted as $\langle \mathbf{M}, \mathbf{H}, p \rangle$. Operation semantics are expressed as follows.

$$\frac{e\downarrow_M v}{\langle M,H,x:=e\rangle\rightarrow\langle M\cup[x\mapsto v],H,\varepsilon\rangle}$$

$$\frac{H(M(y_h))=(v_t,v_s,v_k),tmp\subseteq v_t}{\langle M,H,(x_t,x_s,x_k):=checkTemplate(y_h,tmp)\rangle\rightarrow\langle M\cup[x_t\mapsto v_t,x_s\mapsto v_s,x_k\mapsto v_k],H,\varepsilon\rangle}$$

$$\frac{v_k\leftarrow\mathcal{F},M(y_t)\in\wp}{\langle M,H,x_k:=genKey(y_t)\rangle\rightarrow\langle M\cup[x_k\mapsto v_k],H,\varepsilon\rangle},\ \frac{v_s\leftarrow\mathcal{F},M(y_t)\in\wp}{\langle M,H,x_s:=genSeed(y_t)\rangle\rightarrow\langle M\cup[x_s\mapsto v_s],H,\varepsilon\rangle}$$

$$\frac{v_h\leftarrow\mathcal{F},M(y_t)\in\wp}{\langle M,H,x_h:=ObjectLoad(y_s,y_k,y_t)\rangle\rightarrow\langle M\cup[x_h\mapsto v_h],H\cup[v_h\mapsto(M(y_t),M(y_s),M(y_k))],\varepsilon\rangle}$$

$$\frac{y_k\downarrow_M k,y_c\downarrow_M senc(k,\tilde{v})}{\langle M,H,\tilde{x}:=sdec(y_k,y_c)\rangle\rightarrow\langle M\cup[\tilde{x}\mapsto\tilde{v}],H,\varepsilon\rangle},\ \frac{y_k\downarrow_M k,y_c\downarrow_M aenc(ek(k),\tilde{v})}{\langle M,H,\tilde{x}:=adec(y_k,y_c)\rangle\rightarrow\langle M\cup[\tilde{x}\mapsto\tilde{v}],H,\varepsilon\rangle}$$

$$\frac{M(y_{pA}),M(y_{inA})\in\wp,F\in M(y_{pA})\Rightarrow N/F\in M(y_{inA}),F\notin M(y_{pA})\Rightarrow F\notin M(y_{inA})}{\langle M,H,x_{inA}:=PAV(y_{pA},y_{inA})\rangle\rightarrow\langle M\cup[x_{inA}\mapsto M(y_{inA})],H,\varepsilon\rangle}$$

$$\frac{y_k\downarrow_M k,\tilde{y}_v\downarrow_M\tilde{v},y_{hmac}\downarrow_M HMAC(k,\tilde{v})}{\langle M,H,\tilde{x}:=checkHMAC(y_k,y_{hmac},\tilde{y}_v)\rangle\rightarrow\langle M\cup[\tilde{x}\mapsto\tilde{v}],H,\varepsilon\rangle}$$

$$\frac{\langle M,H,p_1\rangle\rightarrow\langle M',H',\varepsilon\rangle}{\langle M,H,p_1;p_2\rangle\rightarrow\langle M',H',p_2\rangle},\ \frac{\langle M,H,p_1\rangle\rightarrow\langle M',H',p'_1\rangle}{\langle M,H,p_1;p_2\rangle\rightarrow\langle M',H',p'_1;p_2\rangle}$$

$$\frac{a=\lambda\tilde{x}.p,\langle M_e\cup[\tilde{x}\mapsto\tilde{v}],H,p\rangle\rightarrow\langle M',H',return\ e\rangle,e\downarrow_{M'}v}{a(\tilde{v})\downarrow_{H,H'}v}$$

We explain the second rule and the other rules are similar. For the function `checkTemplate`, it evaluates y_h in $\mathbf{M}$, finds the key referenced by the handle $\mathbf{M}(y_h)$, and checks whether $tmp \subseteq v_t$. If so, it may store the key object in the tuple variables (x_t, x_s, x_k), noted $\mathbf{M} \cup [x_t \mapsto v_t, x_s \mapsto v_s, x_k \mapsto v_k]$. The last rule is standard for API calls on a configuration. The API command are executed and the returned value is given as the output value of the call. Notice that we cannot observe the memory used internally by the device. The only exchanged data are input parameters and the returned value. This is the foundation for the attacker model.

3.2 Attacker Model and API Security

The attacker is formalized in a classic Dolev-Yao style. The knowledge of the attacker is denoted as a set of values derived from known values V with his capability. Let V be a finite set of values, The knowledge of the attacker $\mathcal{K}(V)$ is defined as the least superset of V such that $v, v' \in \mathcal{K}(V)$ implies

(1) $(v, v') \in \mathcal{K}(V)$
(2) $senc(v, v') \in \mathcal{K}(V)$

(3) $aenc(v, v') \in \mathcal{K}(V)$
(4) if $v = senc(v', v'')$, then $v'' \in \mathcal{K}(V)$
(5) if $v = aenc(ek(v'), v'')$, then $v'' \in \mathcal{K}(V)$
(6) $kdf(v, v') \in \mathcal{K}(V)$
(7) $hmac(v, v') \in \mathcal{K}(V)$

API commands can be called by attackers in any sequences and with any parameters in his knowledge. The returned values will be added to his set of known values and enlarge his knowledge. Formally, An attacker configuration is denoted as $\langle \mathbf{H}, V \rangle$ and has a reduction as follows:

$$\frac{c \in \mathcal{A}, v_1 \cdots v_k \in \mathcal{K}(V), c(v_1, \cdots, v_k) \downarrow^{\mathbf{H},\mathbf{H}'} v}{\langle \mathbf{H}, V \rangle \rightarrow_{\mathcal{A}} \langle \mathbf{H}', V \cup \{v\} \rangle}$$

The set of initial known values V_0 contains all the atomic constant values in $\mathcal{C}$. For all Asymmetric key value $v'' \in \mathcal{F}, ek(v'') \in V_0$. The set of initial handle-map $\mathbf{H}_0$ is empty. In our model, $\rightarrow_{\mathcal{A}}^*$ notes multi-step reductions.

The main property of the Protected Storage Hierarchy required by TPM 2.0 specifications is secrecy. More specifically, the value of private keys loaded on a TPM should never be revealed outside the secure device, even when exposed to a compromised host system.

Formally, the sensitive keys available on the TPM should never be known by the attacker, as well as the seed in a Storage Key. The definition of *Secrecy of API commands* follows.

Definition 1 (Secrecy). Let $\mathcal{A}$ be an API. $\mathcal{A}$ is secure if for all reductions of attacker configuration $\langle \emptyset, V_0 \rangle \rightarrow_{\mathcal{A}}^* \langle \mathbf{H}, V \rangle$, we have

Let g be a handle in $\mathbf{H}$ such that $\mathbf{H}(g) = (tmp, v_s, v_k)$ and $F \in tmp$. Then, $v_s, v_k \notin \mathcal{K}(V)$.

The language in section 2.2 can be used to model the TPM 2.0 API commands of protected storage part. We give a brief specification on them and conclude they preserve secrecy in section 4.

4 Type System

4.1 A Core Type System

In this section, we present a type system to statically enforce secrecy in API commands. At first, we introduce the concept of security level [8], a pair $\sigma_C \sigma_I$, to specify the levels of confidentiality and integrity. We consider two levels: $High(H)$ and $Low(L)$. Intuitively, values with high confidentiality cannot be read by the attackers while data with high integrity should not be modified by the attackers.

While it is safe to consider a public value as secret, low integrity cannot be promoted to high integrity. Otherwise, data from the attackers may erroneously be considered as coming from a secure device. Therefore, we have the confidentiality and integrity preorders: $L \sqsubseteq_C H$ and $H \sqsubseteq_I L$. We let σ_C and σ_I, range over $\{L, H\}$, while we let σ range over the pairs $\sigma_C \sigma_I$ with $\sigma_C \sigma_I \sqsubseteq \sigma_C' \sigma_I'$ iff

$\sigma_C \sqsubseteq_C \sigma_C^{'}$ and $\sigma_I \sqsubseteq_I \sigma_I^{'}$. It gives the standard four-point lattice. Formally, type syntax T is as follows:

$$T ::= \sigma | \rho^{\sigma} | SeedK^{\sigma}[] | \phi K^{\sigma}[\tilde{T}],$$

where

$$\sigma ::= \sigma_C \sigma_I = LL | LH | HL | HH$$
$$\rho ::= Unwrap | Dec | Sym | Any$$
$$\phi ::= \rho | Wrap | Enc | hmac.$$

Each type has an associated security level denoted by $\mathcal{L}(T)$. For basic types we trivially have $\mathcal{L}(\sigma) = \sigma$. As expected, we have $\mathcal{L}(\rho K^{\sigma}[\tilde{T}]) = \sigma$ and $\mathcal{L}(\rho^{\sigma}) = \sigma$. It is nature to define $\mathcal{L}_C(T)$ and $\mathcal{L}_I(T)$ for confidentiality and integrity levels.

In type syntax T, σ is the type for general data at such level. ρ^{σ} is the type of templates. label ρ specifies the `mode` of the key object which depends on its template. $Unwrap$ denotes the Storage Key Object; Dec denotes the Asymmetric Decryption Key Object; Sym denotes the Symmetric Decryption Key Object; Any is the top mode including all the three modes. All templates are public. Yet the templates with F are generated by the TPM and cannot be forged. Thus they have a security level LH. The other templates with attribute N or without any hierarchy attributes may be forged by the attackers via the process of duplication or loading. Thus they have a security level LL. The types are as follows:

$$\frac{W, A, F \in tmp}{\vdash tmp : Unwrap^{LH}}, \frac{E, A, F \in tmp}{\vdash tmp : Dec^{LH}}, \frac{E, S, F \in tmp}{\vdash tmp : Sym^{LH}},$$

$$\frac{W, A \in tmp, F \notin tmp}{\vdash tmp : Unwrap^{LL}}, \frac{E, A \in tmp, F \notin tmp}{\vdash tmp : Dec^{LL}}, \frac{E, S \in tmp, F \notin tmp}{\vdash tmp : Sym^{LL}}.$$

The type $\phi K^{\sigma}[\tilde{T}]$ describes the key values at security level σ which are used to perform cryptographic operations on payloads of type $\tilde{T}$. For the sake of readability, we let $\tilde{T}$ denote a sequence $T_1, \cdots, T_n$ of types and use $\tilde{x} : \tilde{T}$ to type a sequence $x_1, \cdots, x_n$ of variables. Label ϕ specifies the usage of the key values. Intuitively, $Seed$ value is stored as v_s in a Storage Key Object to be used for the derivation of HMAC key and symmetric key which are used for the protection of the other objects; $Wrap$ and $Unwrap$ are a pair of asymmetric keys stored as v_k in a Storage Key Object used in the process of duplication; Enc and Dec are similar but stored in a Decryption Key Object; Sym is used in symmetric encryption and decryption; $hmac$ is used in the computation of HMAC for the protection of integrity.

Based on security level of types, we have *subtyping* relations. Formally, $\leq$ is defined as the least preorder such that:

(1) $\sigma_1 \leq \sigma_2$ iff $\sigma_1 \sqsubseteq \sigma_2$;
(2) $LL \leq \phi K^{LL}[LL, \ldots, LL], LL \leq \rho^{LL}, LL \leq SeedK^{LL}[]$;
(3) $\phi K^{\sigma}\left[\tilde{T}\right] \leq \sigma, SeedK^{\sigma}[] \leq \sigma, \rho^{\sigma} \leq \sigma$;
(4) $\rho K^{\sigma}\left[\tilde{T}\right] \leq AnyK^{\sigma}\left[\tilde{T}\right], \rho^{\sigma} \leq Any^{\sigma}$.

It is obvious that subtyping relationship does not compromise the security, since $T \leq T'$ implies $\mathcal{L}(T) \sqsubseteq \mathcal{L}(T')$.

Typing Expressions. After the definition of types, we introduce a typing environment $\Gamma : x \mapsto T$, namely a map from variables to their respective types. Type judgement for expressions is written as $\Gamma \vdash e : T$ meaning that expression e is of type T under Γ. The typing rules for expressions are described as follows.

$$[\text{var}]\ \frac{\Gamma(x)=T}{\Gamma\vdash x:T},\ [sub]\ \frac{\Gamma\vdash e:T', T'\leq T}{\Gamma\vdash e:T},\ [tuple]\ \frac{\Gamma\vdash \tilde{x}_1:\tilde{T}_1, \Gamma\vdash x_2:T_2}{\Gamma\vdash(\tilde{x}_1,x_2):(\tilde{T}_1,T_2)},$$

$$[kdfSH]\ \frac{\Gamma\vdash x:SeedK^{HH}[], usg=STORAGE}{\Gamma\vdash kdf(usg,x):SymK^{HH}[\tilde{T}]},\ [kdfSL]\ \frac{\Gamma\vdash x:SeedK^{LL}[], usg=STORAGE}{\Gamma\vdash kdf(usg,x):SymK^{LL}[LL,\cdots,LL]},$$

$$[kdfIH]\ \frac{\Gamma\vdash x:SeedK^{HH}[], usg=INTEGRITY}{\Gamma\vdash kdf(usg,x):hmacK^{HH}[\tilde{T}]},\ [kdfIL]\ \frac{\Gamma\vdash x:SeedK^{LL}[], usg=INTEGRITY}{\Gamma\vdash kdf(usg,x):hmacK^{LL}[LL,\cdots,LL]},$$

$$[wrapK]\ \frac{\Gamma\vdash x:UnwrapK^{\sigma_C\sigma_I}[\tilde{T}]}{\Gamma\vdash ek(x):WrapK^{L\sigma_I}[\tilde{T}]},\ [encK]\ \frac{\Gamma\vdash x:DecK^{\sigma_C\sigma_I}[\tilde{T}]}{\Gamma\vdash ek(x):EncK^{L\sigma_I}[\tilde{T}]},$$

$$[Sym]\ \frac{\Gamma\vdash x:SymK^{\sigma_C\sigma_I}[\tilde{T}], \Gamma\vdash\tilde{y}:\tilde{T}}{\Gamma\vdash senc(x,\tilde{y}):L\sigma_I},\ [hmac]\ \frac{\Gamma\vdash x:hmacK^{\sigma_C\sigma_I}[\tilde{T}], \Gamma\vdash\tilde{y}:\tilde{T}, \sigma'_I=\sigma_I\cup_{T\in\tilde{T}}L_I(T)}{\Gamma\vdash HMAC(x,\tilde{y}):L\sigma_I},$$

$$[Wrap]\ \frac{\Gamma\vdash x:WrapK^{\sigma_C\sigma_I}[\tilde{T}], \Gamma\vdash\tilde{y}:\tilde{T}}{\Gamma\vdash aenc(x,\tilde{y}):L\sigma_I},\ [Enc]\ \frac{\Gamma\vdash x:EncK^{\sigma_C\sigma_I}[\tilde{T}], \Gamma\vdash\tilde{y}:\tilde{T}}{\Gamma\vdash aenc(x,\tilde{y}):L\sigma_I}.$$

Rules $[var]$, $[sub]$, and $[tuple]$ are standard to derive types directly from Γ or via subtyping relationship. Rules $[kdfSH]$, $[kdfSL]$, $[kdfIH]$, and $[kdfIL]$ states that given a seed and its usage, we may derive a new key of the security level inherited from the seed. The security level of the seed value can only be HH (Trusted) or LL (Untrusted). Rules $[wrapK]$ and $[encK]$ says that if an asymmetric decryption key k_x is of type $\rho K^{\sigma_C\sigma_I}[\tilde{T}]$ where ρ ranges over $\{Unwrap, Dec\}$, then the corresponding encryption key $ek(k_x)$ is of type $\rho K^{L\sigma_I}[\tilde{T}]$. Notice that the confidentiality level is L(Low), since public keys are allowed to be known to the attacker, while the integrity level is the same with its decryption key. Rules $[Sym]$, $[Wrap]$, and $[Enc]$ state the encryption of data. The type of the operand e is required to be compatible with that of the payload which is specified by the type of the key. The integrity level of the ciphertext should be the same with that of the key. Rules $[hmac]$ requires that the integrity level of the HMAC should be $\sigma_I \bigsqcup_{T\in\tilde{T}} \mathcal{L}_I(T)$, which represents the lowest integrity level of σ_I and each level of $\mathcal{L}_I(T)$ while $T \in \tilde{T}$. The reason for it is the fact that if the attacker may generate either the HMAC key or the plaintext, he could modify the computation of HMAC. Ciphertexts and the HMAC can be returned to the caller and consequently their confidentiality level is L.

Typing API Commands. Type judgement for API commands is denoted as $\Gamma \vdash p$ meaning that p is well-typed under the typing environment Γ. For simplicity, we write $\Gamma(\tilde{x}) = \tilde{T}$ or $\tilde{x} \mapsto \tilde{T}$ for the binding of variables $\tilde{x} = (x_1, \cdots, x_n)$ respectively to their types $\tilde{T} = (T_1, \cdots, T_n)$. The judgement for API commands is formalized as follows.

$$[API]\ \frac{\forall c \in \mathrm{A}\quad \Gamma \vdash c}{\Gamma \vdash \mathrm{A}},\ [assign]\ \frac{\Gamma \vdash e{:}T\quad \Gamma, x \mapsto T \vdash p}{\Gamma \vdash x := e; p},\ [seq]\ \frac{\Gamma \vdash p_1\quad \Gamma \vdash p_2}{\Gamma \vdash p_1; p_2},$$

$$[checktmp]\ \frac{\Gamma \vdash y_h{:}LL\quad \forall \tilde{T} \in PTS(tmp,\wp) \Rightarrow \Gamma, \tilde{x} \mapsto \tilde{T} \vdash p}{\Gamma \vdash \tilde{x} := checkTemplate(y_h, tmp); p},\ [sdec]\ \frac{\Gamma \vdash y_k{:}SymK^{\sigma}[\tilde{T}]\quad \Gamma, \tilde{x} \mapsto \tilde{T} \vdash p}{\Gamma \vdash \tilde{x} := sdec(y_k, y_c); p},$$

$$[genKey-H]\ \frac{\Gamma \vdash y_t{:}Any^{LH}\quad \Gamma, x_k \mapsto AnyK^{HH}[\tilde{T}] \vdash p}{\Gamma \vdash x_k := genKey(y_t); p},\ [genKey-L]\ \frac{\Gamma \vdash y_t{:}LL\quad \Gamma, x_k \mapsto LL \vdash p}{\Gamma \vdash x_k := genKey(y_t); p},$$

$$[genSeed-H]\ \frac{\Gamma \vdash y_t{:}Any^{LH}\quad \Gamma, x_s \mapsto SeedK^{HH}[] \vdash p}{\Gamma \vdash x_s := genSeed(y_t); p},\ [genSeed-L]\ \frac{\Gamma \vdash y_t{:}LL\quad \Gamma, x_s \mapsto LL \vdash p}{\Gamma \vdash x_s := genSeed(y_t); p},$$

$$[ObjLoad-H]\ \frac{\Gamma \vdash y_s{:}SeedK^{HH}[]\quad \Gamma \vdash y_k{:}\rho K^{HH}[\tilde{T}]\quad \Gamma \vdash y_t{:}\rho^{LH}\quad \Gamma, x_h \mapsto LL \vdash p}{\Gamma \vdash x_h := ObjectLoad(y_s, y_k, y_t); p},$$

$$[ObjLoad-L]\ \frac{\Gamma \vdash y_s{:}LL\quad \Gamma \vdash y_k{:}LL\quad \Gamma \vdash y_t{:}LL\quad \Gamma, x_h \mapsto LL \vdash p}{\Gamma \vdash x_h := ObjectLoad(y_s, y_k, y_t); p},$$

$$[Dec]\ \frac{\Gamma \vdash y_k{:}DecK^{\sigma}[\tilde{T}]\quad \Gamma \vdash y_c{:}T\quad \Gamma, \tilde{x} \mapsto \tilde{T} \vdash p\quad L_I(T) = L \Rightarrow \Gamma, \tilde{x} \mapsto (LL, \cdots, LL) \vdash p}{\Gamma \vdash \tilde{x} := adec(y_k, y_c); p},$$

$$[Unwrap]\ \frac{\Gamma \vdash y_k{:}UnwrapK^{\sigma}[\tilde{T}]\quad \Gamma \vdash y_c{:}T\quad \Gamma, \tilde{x} \mapsto \tilde{T} \vdash p\quad L_I(T) = L \Rightarrow \Gamma, \tilde{x} \mapsto (LL, \cdots, LL) \vdash p}{\Gamma \vdash \tilde{x} := adec(y_k, y_c); p},$$

$$[PAV-H]\ \frac{\Gamma \vdash (y_{pA}, y_{inA}){:}\left(Unwrap^{LH}, LL\right)\quad \Gamma, x_{inA} \mapsto Any^{L\sigma_I} \vdash p}{\Gamma \vdash x_{inA} := PAV(y_{pA}, y_{inA}); p},$$

$$[PAV-L]\ \frac{\Gamma \vdash (y_{pA}, y_{inA}){:}(LL, LL)\quad \Gamma, x_{inA} \mapsto LL \vdash p}{\Gamma \vdash x_{inA} := PAV(y_{pA}, y_{inA}); p},$$

$$[chkHMAC]\ \frac{\Gamma \vdash y_k{:}hmacK^{\sigma}[\tilde{T}]\quad \Gamma, \tilde{x} \mapsto \tilde{T} \vdash p}{\Gamma \vdash \tilde{x} := checkHMAC(y_k, y_{hmac}, \tilde{y}_v); p},$$

$$[return]\ \frac{\Gamma \vdash \tilde{x}{:}(LL, \cdots, LL)}{\Gamma \vdash return\ \tilde{x}},\ [command]\ \frac{\Gamma \vdash x_1{:}LL\quad \cdots\quad \Gamma \vdash x_k{:}LL\quad \Gamma \vdash p}{\Gamma \vdash \lambda x_1, \cdots, x_k.p}$$

Most of the above rules are standard. We just explain [*checktmp*] and [PAV]. The details of the others are in full version [16]. Rule [*checktmp*] is adapted form the same rule in [4]. We have to type-check all the permitted templates tmp' in $\wp$ matching the checked template tmp, such that $tmp \subseteq tmp'$. The Permitted Templates Set is denoted as

$$PTS(tmp, \wp) = \{(\rho^{L\sigma_I}, SeedK^{\sigma_I\sigma_I}[], \rho K^{\sigma_I\sigma_I}[\tilde{T}]) | \\ \exists \mathrm{tmp}' \in \wp, \mathrm{tmp} \subseteq \mathrm{tmp}' \wedge \vdash \mathrm{tmp}' : \rho^{\mathrm{L}\sigma_\mathrm{I}}\}.$$

For example, if $tmp = \{W\}$, the permitted templates matching with tmp are $\{W, A\}$, $\{W, A, N\}$, and $\{W, A, F\}$. The corresponding types are (ρ^{LL}, $SeedK^{LL}[]$, $\rho K^{LL}[\tilde{T}]$) and (ρ^{LH}, $SeedK^{HH}[]$, $\rho K^{HH}[\tilde{T}]$), where $\rho = Unwrap$. We need to type-check the following clauses under the assumption that $\tilde{x}$ may have all the types in PTS. Meanwhile, $PTS(\{W, F\}, \wp) = (Unwrap^{LH}$, $SeedK^{HH}[]$, $Unwrap^{HH}[\tilde{T}])$. The rules $[PAV-H]$ and $[PAV-L]$ are used for public hierarchy attributes validation. The purpose for `PAV` is to check the consistency of hierarchy attributes between the parent object and the child. The former rule says that if the parent object has the attribute *FixedTPM* (F), then any allowable combination of the hierarchy attributes would be fine for the child. The latter rule states that if the template of the parent object does not include the attribute F, then F cannot be in the template of the child.

4.2 Properties of the Type System

In this section, some properties of our Type System are introduced, including the main result, well-typed APIs are secure. The proof of the main theorem can be found in the full version [16]. Centenaro, et al.[4] have proposed the notion of

value well-formedness in their type system in order to track the value integrity at run-time. Their judgement was based on a mapping Θ from atomic values to types. We follow this method but lay more restriction on the foundation of this typing environment for values to obtain more valuable properties. Rules for typing values are given in Table. They are close to those for typing expressions.

$$[empty]\ \phi \vdash \emptyset,$$

$$[Env]\ \frac{\Theta\vdash\emptyset, v\notin dom(\Theta), T=\varphi K^{\sigma}[\tilde{T}], SeedK^{\sigma}[]\Rightarrow(\varphi\in\{Sym,Dec,Unwrap,hmac\}\wedge\sigma=HH)}{\Theta\cup\{val\mapsto T\}\vdash\emptyset},$$

$$[atom]\ \frac{\Theta(val)=T}{\Theta\vdash val:T}, [sub]\ \frac{\Theta\vdash v:T', T'\leq T}{\Theta\vdash v:T}, [tuple]\ \frac{\Theta\vdash\tilde{v}:\tilde{T}, \Theta\vdash v':T'}{\Theta\vdash(\tilde{v},v'):(\tilde{T},T')},$$

$$[kdfSH]\ \frac{\Theta\vdash v:SeedK^{HH}[], usg=STORAGE}{\Theta\vdash kdf(usg,v):SymK^{HH}[\tilde{T}]}, [kdfSL]\ \frac{\Theta\vdash v:SeedK^{LL}[], usg=STORAGE}{\Theta\vdash kdf(usg,v):SymK^{LL}[LL,\cdots,LL]},$$

$$[kdfIH]\ \frac{\Theta\vdash v:SeedK^{HH}[], usg=INTEGRITY}{\Theta\vdash kdf(usg,v):hmacK^{HH}[\tilde{T}]}, [kdfIL]\ \frac{\Theta\vdash v:SeedK^{LL}[], usg=INTEGRITY}{\Theta\vdash kdf(usg,v):hmacK^{LL}[LL,\cdots,LL]},$$

$$[wrapK]\ \frac{\Theta\vdash v:UnwrapK^{\sigma}C^{\sigma}{}_{I}[\tilde{T}]}{\Theta\vdash ek(v):WrapK^{L\sigma}{}_{I}[\tilde{T}]}, [encK]\ \frac{\Theta\vdash v:DecK^{\sigma}C^{\sigma}{}_{I}[\tilde{T}]}{\Theta\vdash ek(v):EncK^{L\sigma}{}_{I}[\tilde{T}]},$$

$$[Sym]\ \frac{\Theta\vdash v':SymK^{\sigma}C^{\sigma}{}_{I}[\tilde{T}], \Theta\vdash\tilde{v}:\tilde{T}}{\Theta\vdash senc(v',\tilde{v}):L\sigma_{I}}, [HMAC]\ \frac{\Theta\vdash v':hmacK^{\sigma}C^{\sigma}{}_{I}[\tilde{T}], \Theta\vdash\tilde{v}:\tilde{T}, \sigma'{}_{I}=\sigma_{I}\cup_{T\in\tilde{T}}L_{I}(T)}{\Theta\vdash HMAC(v',\tilde{v}):L\sigma_{I}},$$

$$[Wrap]\ \frac{\Theta\vdash v':WrapK^{\sigma}C^{\sigma}{}_{I}[\tilde{T}], \Theta\vdash\tilde{v}:\tilde{T}}{\Theta\vdash aenc(v',\tilde{v}):L\sigma_{I}}, [Enc]\ \frac{\Theta\vdash v':EncK^{\sigma}C^{\sigma}{}_{I}[\tilde{T}], \Theta\vdash\tilde{v}:\tilde{T}}{\Theta\vdash aenc(v',\tilde{v}):L\sigma_{I}}$$

However, two additional rules $[empty]$ and $[env]$ are set to define the well-formedness of our typing environment Θ. The rule $[env]$ requires that Θ does not contain multiple bindings for the same value. Moreover, only atomic fresh keys at a security level of HH are allowable. It is sound because in operation semantics for commands in section 2.2, atomic fresh keys can only be generated by `genKey` and `genSeed`, which are internal functions that cannot be touched by the attackers. On the basis of these rules, some properties for the types of key values can be obtained by easy induction on the derivation of $\Theta \vdash v : \phi K^{\sigma}[\tilde{T}]$.

Proposition 1 (Private Keys). If $\Theta \vdash \emptyset$, $\Theta \vdash v : \phi K^{\sigma}[\tilde{T}]$, and $\phi \in \{Seed, Sym, Dec, Unwrap, hmac\}$, then $\sigma \in \{HH, LL\}$.

Proposition 2 (Low Keys). If $\Theta \vdash \emptyset$, then $\Theta \vdash v : \phi K^{LL}[\tilde{T}]$ implies $\tilde{T} = LL, \cdots, LL$.

Proposition 3 (Public Keys). If $\Theta \vdash \emptyset$, $\Theta \vdash v : \phi K^{\sigma}[\tilde{T}]$, and $\phi \in \{Wrap, Enc\}$, then $\sigma \in \{LH, LL\}$.

The next proposition says the type of private key is unique, if it has a security level of HH.

Proposition 4 (Uniqueness of Key Types). Let $\Theta \vdash \emptyset$. If $\Theta \vdash k : \phi K^{\sigma}[\tilde{T}]$ and $\Theta \vdash k : \phi' K^{\sigma'}[\tilde{T'}]$ with $\phi, \phi' \in \{Seed, hamc, Sym, Unwrap, Dec\}$, then $\sigma = \sigma'$. If $\sigma = \sigma' = HH$, we also have $\phi = \phi'$.

The notion of well-formedness for memory environment follows the definition 3 in [4] except that we add item (1), which requires Θ is well formed. With this requirement, we may apply proposition 1 to 4.

Definition 2 (Well-formedness). The judgement of well-formedness for memory environment and key handle-map is denoted as $\Gamma, \Theta \vdash \mathbf{M}, \mathbf{H}$ if
(1) $\Theta \vdash \emptyset$, ie., the typing environment Θ is well formed by the typing rules $[empty]$ and $[Env]$;

(2) $\Gamma, \Theta \vdash \mathbf{M}$, ie., $\mathbf{M}(x) = v, \Gamma(x) = T$ implies $\Theta \vdash v : T$;
(3) $\Theta \vdash \mathbf{H}$. Let $\mathbf{H}(h) = (tmp, v_s, v_k)$. $\vdash tmp : \rho^{LH}$ implies $\Theta \vdash v_s : SeedK^{HH}[]$, $\Theta \vdash v_k : \rho K^{HH}[\tilde{T}]$; $\vdash tmp : LL$ implies $\Theta \vdash v_s : LL$, $\Theta \vdash v_k : LL$.

As we have mentioned above, the security level σ restrict the capability of attackers such that they can read from LL, LH and modify LL, HL. Due to $\rho K^{LH}[\tilde{T}] \leq LH \leq LL$ and the subtyping rule, we may assume the knowledge of attackers has a security level of LL. Proposition 5 proves that if we only give the attacker atomic values of type LL, all the values that can be derived from his capability are of a security level LL. In the proof of this proposition, we may use proposition 2 (Low Keys) in some cases.

Proposition 5 (Attacker typability). Let $\Theta \vdash \emptyset$, $\Theta \vdash \mathbf{H}$ and V be a set of atomic values. Suppose $\forall v \in V, \Theta(v) = LL$. Then, $v' \in \mathcal{K}(V)$ implies $\Theta \vdash v' : LL$ if v' is an atomic values, and $\Theta \vdash v' : (LL, \cdots, LL)$ if v' is a tuple.

Lemma 1 states that in a well-formed memory, each expression has a type matched with its evaluation. Lemma 2 states that well-typed commands remain well-typed at run-time and preserve well-formedness of typing environment.

Lemma 1. If $\Theta \vdash \emptyset$, $\Gamma, \Theta \vdash \mathbf{M}$, $\Gamma \vdash e : T$, and $e \downarrow^{\mathbf{M}} v$, then $\Theta \vdash v : T$.

Lemma 2. Let $\Gamma, \Theta \vdash \mathbf{M}, \mathbf{H}$ and $\Gamma \vdash p$. If $\langle \mathbf{M}, \mathbf{H}, p \rangle \rightarrow \langle \mathbf{M}', \mathbf{H}', p' \rangle$ then we have
(1) if $p' \neq \varepsilon$ then $\Gamma \vdash p'$;
(2) $\exists \Theta' \supseteq \Theta$ such that $\Gamma, \Theta' \vdash \mathbf{M}', \mathbf{H}'$.

With Lemma 1 and 2 above, we finally prove our main result that well-typed API commands are secure.

Theorem 1. If $\Gamma \vdash \mathcal{A}$, then $\mathcal{A}$ is secure.

5 Type-Based Analysis of TPM 2.0 Specification Commands

In this section, we show that the TPM 2.0 Specification commands such as `TPM2_Duplicate`, `TPM2_Import`, `TPM2_Create` and `TPM2_Load` are secure in the framework of our model (It is expected to include more commands). We will prove that these commands guarantee the secrecy of the key object with its *FixedTPM* SET, even in case of the worst scenario in which the attacker may access all loaded key objects via API commands to perform operations corresponding to the protected storage hierarchies rooted in the TPM.

The API is defined in Trusted Platform Module Library (TPML) Family 2.0, Part 3: Commands [15], which specifies the input parameters, the response, and the detailed actions of each command. We may translate the detailed actions to our language introduced in section 2.2. The commands that need to be formalized include Object Commands in Chapter 14 and Duplication Commands in Chapter 15 of TPML 2.0, Part 3. As we have discussed in section 2.1, we focus on these commands since they decide how an object might be connected to the protected storage hierarchy rooted in the TPM.

The detailed actions in these commands contain internal functions specified in section 7.6 of TPML 2.0, Part 4: Supporting Routines. These internal functions should be called by Protected Capabilities. We have transferred these functions to our language. Now we give an example of `AreAttributesForParent()`, which decides whether the input handle refers to a parent object. It can be implemented by (ObjTemplate, ObjSeed, ObjSensitive):= `checkTemplate`(ObjHandle, {W}); In a similar way, we could formalize a set of internal functions in section 7.6 of Part 4.

After this formalization, we could translate to our language the protected storage API commands such as `TPM2_Create()`, `TPM2_Load()`, `TPM2_Duplicate()`, and `TPM2_Import()` in Part 3. We give an example of `TPM2_Load()`. The detailed translation is in the full version.

Command `TPM2_Load` takes as input the handle of the parent object (parentH), the public area of the loaded object (inAttributes), an HMAC to check the integrity (inHMAC), and the encrypted sensitive area of the loaded object (LoadPrivate). The execution of the command depends on whether the loaded object has *FixedTPM* SET in its template ($F \in$ inAttributes) since it decides whether *FixedTPM* is needed in the parent object. In the detailed actions of Part 3, it is expressed by a standard if/else statement. For the former case, F is needed to be included in the template of the parent object. The latter is not. Thus we have different requirements for the first `checkTemplate`. There are no differences in the following clauses. Then, the public attributes of the loaded object should be checked to be consistent with the parent's by `PAV`. If passed, a symmetric key (symKeyP) for secure storage and an HMAC key (HMACkeyP) for integrity are derived from the secret seed (parentSeed) in the parent object. After checking the integrity of the public area (inAttributesC) and the encrypted sensitive area (LoadPrivate), the command will decrypt the sensitive area by `sdec`. At last, new object are loaded and its handle (ObjH) is returned.

In a similar way, we have translated the Object Commands and Duplication Commands in Trusted Platform Module Library (TPML) Family 2.0, Part 3: Commands. In the following, we need to type-check these API commands by our type system in section 3.1 to enforce the security of API commands. We will give an example of the command `TPM2_Load`. The detailed specification is in the full version [16].

Since the command `TPM2_Load` requires a branch, we need to devise two typing environment Γ respectively to type these two cases. For both cases, it is required that all the input parameters have type LL (line 00 and line 10). For the former case, when `checkTemplate` requires a handle for a parent key object with W, F SET. Then the type returned is $(Unwrap^{LH}, SeedK^{HH}[], UnwrapK^{HH}[\tilde{T}])$ according to section 3.1. Then by the rule $[PAV - H]$, we get the input attributes after check should have type Any^{LH} because $F \in$ inAttributes. By `kdf`, we get two keys derived from the seed value in the parent sensitive area with types $SymK^{HH}[SeedK^{HH}[], AnyK^{HH}]$ and $hmacK^{HH}[LH, Any^{LH}]$. The payload type is decided by the usage of the parent key object. Then after checking the HMAC and symmetric decryption, the returned sensitive area types are

$(SeedK^{HH}[], AnyK^{HH}[\tilde{T}])$. With appropriate types of sensitive area and public area, `ObjectLoad` could load the object into the TPM. Then the type of the returned handle value is LL, which could be returned as the response. For the latter case, `checkTemplate` requires a handle for a parent key with just W SET and the returned type is in $PTS(\{W\}, \wp)$. There are two types in this set, (LL, LL, LL) and $(Unwrap^{LH}, SeedK^{HH}[], UnwrapK^{HH}[\tilde{T}])$. We have to type-check the continuation clauses twice, under these two assumptions. The two typing derivations are the same for `PAV` since $F \notin$ inAttributes. The input template (inAttributes: LL) after `PAV` has type LL. For `kdf`, since the types of payloads are decided by the usage of the parent key object, they both have type LL, LL for the payloads. Thus these two cases are the same for checking HMAC, decryption and loading the object. We finally type-check `return` ObjH by $[return]$.

We have shown that the command `TPM2_Load` is well-typed. By Theorem 1, we know that `TPM2_Load` is secure. In a similar way, we could type-check the other commands that have been formalized in our model and enforce the security of protected storage APIs of the TPM 2.0 specification. We have Theorem 2 to state the security of the TPM 2.0 API commands concentrating on Protected Storage part.

Theorem 2. For the protected storage API $\mathcal{A} = \{$`TPM2_Create()`, `TPM2_Load()`, `TPM2_Duplicate()`, `TPM2_Import()`$\}$ defined by TPM 2.0 specification, $\mathcal{A}$ is secure.

6 Conclusion

We have prososed a type system to statically enforce the security of storage part of the TPM 2.0 API commands. Our type system consumes type-checks for asymmetric cryptographic primitives. A formal proof has been proposed that the commands can guarantee the secret of key values in security devices under the worst scenario where the attackers in Delov-Yao style may gain access to all keys loaded on the device and the API commands can be called by any sequence with any parameters. This has not been proved before.

As future work, we foresee extending our model with more commands such as those involved in Credential Management. We also plan to model the TPM's platform configuration registers (PCRs) which allow one to condition some commands on the current value of a register. Moreover, more security properties such as *integrity* and *noninterference* will be the subject of future work.

Acknowledgments. The research presented in this paper is supported by the National Basic Research Program of China (No. 2013CB338003) and National Natural Science Foundation of China (No. 91118006, No.61202414).

References

1. Abadi, M., Blanchet, B.: Secrecy types for asymmetric communication. Theoretical Computer Science 298(3), 387–415 (2003); In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 25–41. Springer, Heidelberg (2001)
2. Backes, M., Maffei, M., Unruh, D.: Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In: IEEE Symposium on Security and Privacy 2008, pp. 202–215 (2008)
3. Bruschi, D., Cavallaro, L., Lanzi, A., Monga, M.: Replay attack in TCG specification and solution. In: Proceedings of ACSAC 2005, Tucson, AZ (USA), vol. 10, pp. 127–137. ACSA, IEEE Computer Society (December 2005)
4. Centenaro, M., Focardi, R., Luccio, F.L.: Type-based analysis of PKCS#11 key management. In: Degano, P., Guttman, J.D. (eds.) Principles of Security and Trust. LNCS, vol. 7215, pp. 349–368. Springer, Heidelberg (2012)
5. Chen, L., Ryan, M.: Offline dictionary attack on TCG TPM weak authorisation data, and solution. In: Gawrock, D., Reimer, H., Sadeghi, A.-R., Vishik, C. (eds.) Future of Trust in Computing, pp. 193–196. Vieweg Teubner (2009)
6. Chen, L., Ryan, M.: Attack, solution and verification for shared authorisation data in TCG TPM. In: Degano, P., Guttman, J.D. (eds.) FAST 2009. LNCS, vol. 5983, pp. 201–216. Springer, Heidelberg (2010)
7. Delaune, S., Kremer, S., Ryan, M.D., Steel, G.: A formal analysis of authentication in the TPM. In: Degano, P., Etalle, S., Guttman, J. (eds.) FAST 2010. LNCS, vol. 6561, pp. 111–125. Springer, Heidelberg (2011)
8. Focardi, R., Maffei, M.: Types for Security Protocols. In: Formal Models and Techniques for Analyzing Security Protocol, vol. 5, ch. 7, pp. 143–181. IOS Press (2010)
9. Gürgens, S., Rudolph, C., Scheuermann, D., Atts, M., Plaga, R.: Security evaluation of scenarios based on the TCG's TPM specification. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 438–453. Springer, Heidelberg (2007)
10. Lin, A.H., Rivest, R.L., Lin, A.H.: Automated analysis of security APIs. Technical report, MIT (2005)
11. ISO/IEC PAS DIS 11889: Information technology –Security techniques – Trusted Platform Module
12. Trusted Computing Group. TPM Specification version 1.2. Parts 1–3, revision, `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`
13. Keighren, G., Aspinall, D., Steel, G.: Towards a Type System for Security APIs. In: Degano, P., Viganò, L. (eds.) ARSPA-WITS 2009. LNCS, vol. 5511, pp. 173–192. Springer, Heidelberg (2009)
14. Centenaro, M., Focardi, R., Luccio, F.L., Steel, G.: Type-based analysis of PIN processing APIs. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 53–68. Springer, Heidelberg (2009)
15. Trusted Computing Group. TPM Specification version 2.0. Parts 1–4, revision, `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`
16. Shao, J., Feng, D., Qin, Y.: Type-Based Analysis of Protected Storage in the TPM (full version). Cryptology ePrint Archive (2013), `http://eprint.iacr.org/2013/501`