Formal Analysis of Enhanced Authorization in the TPM 2.0

Jianxiong Shao, Yu Qin, Dengguo Feng and Weijin Wang
Trusted Computing and Information Assurance Laboratory,
Institute of Software, Chinese Academy of Sciences, Beijing, China
{shaojianxiong, qin_yu, feng, wangweijin}@tca.iscas.ac.cn

ABSTRACT

The Trusted Platform Module (TPM) is a system component that provides a hardware-based approach to establish trust in a platform by providing protected storage, robust platform integrity measurement, secure platform attestation and other secure functionalities. The access to TPM commands and TPM-resident key objects are protected via an authorization mechanism. Enhanced Authorization (EA) is a new mechanism introduced by the TPM 2.0 to provide a rich authorization model for specifying flexible access control policies for TPM-resident objects.

In our paper, we conduct a formal verification of the EA mechanism. Firstly, we propose a model of the TPM 2.0 EA mechanism in a variant of the applied pi calculus. Secondly, we identify and formalize the security properties of the EA mechanism (Prop.1 and 2) in its design. We also give out a misuse problem that is easily to be neglected (Lemma 7). Thirdly, using the SAPIC tool and the tamarin prover, we have verified both the two security properties. Meanwhile, we have found 3 misuse cases and one of them leads to an attack on the application in [12].

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods

General Terms

Security

Keywords

TPM; Formal Verification; Enhanced Authorization; Trusted Computing

1. INTRODUCTION

The Trusted Platform Module (TPM) is an important system component separated from the system on which it reports (the host system). The only interaction is through the interface, i.e. application programming interface (API), predefined in the TPM specification. Through API commands, the TPM offers facilities to enable trustworthy computation and communication over open networks. The TPM specification [1] is an industry standard proposed by a computer industry consortium called the Trusted Computing Group (TCG). TPM 1.2 specification has become the ISO standard [2]. Hundreds of millions of TPMs are deployed in a wide range of devices ranging from servers and personal computers to mobile devices.

However, several papers have indicated the vulnerabilities in the TPM 1.2 API designs [3, 4, 5, 6, 7], particularly in relation to the secrecy and authentication properties. The attacks on the TPM include the replay attack on TPM authorization protocols [3], the impersonation attack on shared authdata [4], the offline dictionary attack on weak authdata [5], and the illegitimate acquirement of certificates on selected TPM keys [7]. These attacks highlight the importance of formal analysis and verification of the TPM API commands.

To enhance the security and functionalities of the TPM, TCG continues to revise the TPM specifications. In 2013, TCG has published the TPM 2.0 specifications [8]. New version fixes the known flaws in the TPM 1.2 and makes several changes and enhancements from the previous versions especially on the authorization mechanisms. The TPM 2.0 API defines 3 authorization types: password, HMAC, and policy. The last one is denoted by the enhanced authorization (EA), which allows object owners and administrators to require specific policy assertions to be satisfied before access to a protected object is allowed. The policy relies on the object's *authPolicy* setting. The EA mechanism extends the original platform configuration register (PCR) binding operation in TPM 1.2 with more authorization assertions. Moreover, the EA mechanism has a top priority since the other two kinds can be disabled by the specific settings of attributes.

We provide, to the best of our knowledge, the first formal analysis and verification of the TPM 2.0 EA model in our paper. Firstly, we propose a formal model for the API analysis of the simplified TPM 2.0 EA mechanism in the stateful applied pi calculus [9, 10]. Our model could cover almost all aspects of the EA functionalities except for the *CounterTimer* assertion. Secondly, we define and formalize the security properties (Prop.1 and 2) of the EA mechanism in its design. We have met two challenges in this part. The first is that there is more than one expression for a given policy. We solve it by transforming the policy into a dis-

ASIA CCS'15, April 14–17, 2015, Singapore, Singapore.

http://dx.doi.org/10.1145/2714576.2714610.

junctive normal form and providing a generic way to compute the authPolicy setting for it. The second challenge is that the universal quantifier over the policy setting makes it difficult to verify Prop.1. We solve it by dividing the whole proof into a chain of pieces (Lemma 1 and 2) which can be verified and composed by induction. We also identify a misuse problem (Lemma 7). At last, using the SAPIC tool [9] and the tamarin prover [11], we have verified both the two security properties of the EA model. In our analysis, three misuse cases are discovered. The first one is for the *NV* assertion. It leads to an attack on the application in [12], where an NV counter is used to restrict the number of PIN attempts. We find that a malicious user can double this number by using one more session. The other two focus on a theoretical analysis of the race conditions when the TPM APIs are implemented in a concurrency-based model.

Related Work. A few researches have been done for the formal analysis and verification of the TPM 2.0 specifications on direct anonymous attestation [13, 14], protected storage [15], and the traditional HMAC-based authorization [16]. However, there are no results about the formal verification of TPM 2.0 EA mechanism as far as we know. In [12], the first accessible description of the TPM 2.0 EA model is provided and being used to build an Electronic Identification (eID) architecture. They limit the scope of security analysis only to the eID requirements without considering the security properties of EA mechanism, although they acknowledge its formal verification is a worthy pursuit. However, we find that the NV assertion is misused in their application, which might lead to an attack. Some researches have been done for the formal verification of the PCR binding operations in the TPM 1.2 API commands. In [17], a logic for reasoning about the properties of secure systems based on the PCRs is proposed. However, their model assumes that the TPM functions well and does not provide an API analysis as in our model. Another technique presented in [18] extends the logic used in the tool ProVerif [19] with a binary predicate to introduce a non-monotonic global state. Their model focuses on the usage of the PCR to build a chain of trust. This method is generalized to a tool StatVerif [10] but it could only handle a finite number of memory cells. Thus it cannot be used to analyze the TPM API with unbounded number of objects, which has been done by our model.

The paper is organized as follows. In Section 2 we give a brief introduction to the stateful applied pi calculus and the TPM 2.0 EA mechanism. In Section 3 we model the EA mechanism in TPM API commands and its security properties. In Section 4 we present the results of the formal verification of EA and 3 misuse cases. We conclude in Section 5.

2. PRELIMINARIES

2.1 Stateful applied pi calculus

Stateful applied pi calculus is a variant of the applied pi calculus first proposed in [10]. In addition to the usual operators for concurrency, replication, name restriction, communication, and condition, it offers the constructs for manipulation of an explicit global state. In this subsection, we will give a brief introduction to the syntax and semantics of the stateful applied pi calculus, and the trace formulas for the formalization of security properties. The details can be found in [9, 10].

The syntax of terms assumes two disjoint, infinite sets $\mathcal{N}$ and $\mathcal{V}$ of *names* and *variables*. A *signature* Σ consists of a finite set of function symbols, each with an arity. A function symbol with arity 0 is a constant. *Terms*, ranged over by M, N, are built up from names and variables by function applications, which is described in **Table 1**. We denote by $\sigma = \{^{t_1}/_{x_1}, \cdots, ^{t_n}/_{x_n}\}$ the substitution which maps x_i to t_i. Terms are equipped with an equational theory $=_E$ to capture the cryptographic primitives used by protocols. We give an example of the digital signature scheme below.

Example 1. Digital signature scheme can be modeled by a signature $\Sigma = \{\text{Sig}/2, \text{PK}/1, \text{CkSig}/2\}$ equipped with an equational theory defined by $\text{CkSig}(\text{Sig}(m, k), \text{PK}(k)) =_E m$. It allows to check whether the message m is correctly signed by the key k.

The syntax of *processes* is described in **Table 1**. 0 denotes the terminal process. $P|Q$ is the parallel composition of processes P and Q and $!P$ the replication of P, allowing an unbounded number of sessions in protocol executions. The construct $\nu n; P$ binds the name n in P and models the generation of a fresh, random value. Processes out $([M,]N); P$ and in $([M,]N); P$ represent the output, respectively input, of message N on channel M. Note that if the channel M is unique and public, it can be omitted from the expressions. The process if $M = N$ then P [else Q] will execute P if $M =_E N$ and Q otherwise. The branch else Q can be trailed when Q is 0. The event construct F is merely used for annotating processes and will be used in the definition of trace formulas. The construct insert M, N binds the value N to a handle M. Successive inserts allow to change this binding. The delete M operation simply flushes the mapping for the handle M. The lookup M as x in P [else Q] allows to retrieve the value associated to M, binding it to the variable x in P. If the mapping is undefined for M, the process behaves as Q. The process lock $M; P$ locks the resource handle M for the subsequent process P. When the resource M is locked, another parallel process that intends to access M has to wait until a primitive unlock $M; P'$. This is essential for formalizing API where the parallel commands cannot read and update a common memory.

We give an informal description of the semantics of processes and the trace. The formal definition could be found in [9] (Section 3.2). The semantics is defined by a set of the labelled transition relations between process configuration to model the single-step execution (transition) of processes. The transitions are labelled by facts. $\mathcal{F}$, the set of facts, includes two subsets, one for the event construct F, the other for the attacker's deduction of new message M, denoted by $\text{K}(M)$. The trace of a process P, denoted by $tr^P = [F_1, \cdots, F_n]$ where $F_i \in \mathcal{F}$, is defined by a finite sequence of facts, which have been executed. It is the possible executions that the process P admits. Given a ground process P, the set of traces of P is $Tr(P) = \{[F_1, \cdots, F_n] | \exists tr^P = [F_1, \cdots, F_n]\}$. For a sequence tr, $idx(tr)$ is the set of positions in tr, and the i-th label is denoted by tr_i for $i \in idx(tr)$.

In the following, we give out the definitions of trace formula and its validity, which are the same to the definitions 8, 9, 10 in [9]. These definitions could help us formalize the security properties of the TPM 2.0 EA mechanism. Trace formula is described in a two-sorted first-order logic. The sort *temp* is used for timepoints. A valuation θ is a function from variables to ground terms and timepoints that respects sorts.

Table 1: Syntax of processes

$\langle M, N\rangle ::=$	terms
$a, b, c, m, n, h, ...$	names
$x, y, z, ...$	variable
$f(M_1, ..., M_n), f \in \Sigma$ of arity n	function
$\langle P, Q\rangle ::=$	processes
0	termination
$P\|Q$	concurrency
$!P$	replication
$\nu n; P$	restriction
out($[M,]N$); P	output
in($[M,]N$); P	input
if $M = N$ then P [else Q]	conditional
event $F; P$	event
insert $M, N; P$	state
delete $M, N; P$	delete
lookup M as x in P [else Q]	read
lock $M; P$	lock state
unlock $M; P$	unlock state

Definition 1(Trace formulas). A trace atom is either false $\perp$, a term equality $M_1 \approx M_2$, a timepoint ordering $i < j$, a timepoint equality $i = j$, or an action $F@i$ for a fact $F \in \mathcal{F}$ and a timepoint i. A trace formula is a first-order formula over trace atoms.

Definition 2(Satisfaction relation). The satisfaction relation $(tr, \theta) \models \phi$ between trace tr, valuation θ and trace formula ϕ is defined as follows:

$(tr, \theta) \models \perp$	never
$(tr, \theta) \models F@i$	iff $\theta(i) \in idx(tr)$ and $F\theta =_E tr_{\theta(i)}$
$(tr, \theta) \models i < j$	iff $\theta(i) < \theta(j)$
$(tr, \theta) \models i = j$	iff $\theta(i) = \theta(j)$
$(tr, \theta) \models M_1 \approx M_2$	iff $M_1\theta =_E M_2\theta$
$(tr, \theta) \models \neg\phi$	iff not $(tr, \theta) \models \phi$
$(tr, \theta) \models \phi_1 \wedge \phi_2$	iff $(tr, \theta) \models \phi_1$ and $(tr, \theta) \models \phi_2$
$(tr, \theta) \models \exists x.\phi$	iff there is u such that $(tr, \theta[x \mapsto u]) \models \phi$

Definition 3(Validity). Given a ground process P, a trace formula ϕ is said to be valid for the set of the process P's traces $Tr(P)$, written $Tr(P) \models^{\forall} \phi$, if for any trace $tr^P \in Tr(P)$ and any valuation θ we have that $(tr^P, \theta) \models \phi$.

Example 2. The following trace formula expresses the correspondence property. It holds if each occurrence of event Ev_2 is preceded by an event Ev_1.

$$\phi := \forall x_1, \cdots, x_m, t_2.\ Ev_2(x_1, \cdots, x_m)@t_2 \Rightarrow \exists t_1.\ Ev_1(M_1, \cdots, M_n)@t_1 \wedge t_1 < t_2.$$

2.2 The enhanced authorization in the TPM

In the TPM 2.0 specification, the TPM stores key objects in its shielded memory, and access to these internal objects require authorizations. By using the enhanced authorization mechanism, the creator of the object can require specific tests or actions to be performed before it is operated by some specific TPM commands. In this case, the authorization is a proof that the TPM has completed the required actions. It should be provided by the user of these commands.

The specific requirements, defined as policy, are encapsulated in a value called *authPolicy* within the data structure of the object. Once the creator of the object sets the authPolicy value, it cannot be modified. In order to use the object, the user should first start a policy session located in the TPM's shielded memory and then invoke a sequence of policy assertion commands to complete the tests required by the policy. These commands will check some conditions and modify the session's *policyDigest* and other context values. After executing all the assertion commands required by the policy, the TPM keeps a policy session. This session accumulates all policy information needed to make the authorization decision. Finally, the user references the object accompanied with the session in the command, which is to be authorized. If the session's policyDigest matches the object's authPolicy (and some other optional conditions are valid), then the command is authorized to operate the object.

In this subsection, we will give a brief overview of the EA mechanism in the TPM 2.0, including the classification of the policy assertion commands and the way they update the policy session's context. We also give an example of how the EA mechanism works. Details of EA could be found in the specification [8] (Section 19.7, Part I).

2.2.1 Policy assertions

In a policy for authorization, an assertion is a statement of something that must be true before the policy is satisfied. For example, an assertion may be that the selected platform configuration register (PCR) in the TPM should have the desired value to allow an object to be authorized for use in a command. The policy assertion commands are defined in [8] (Section 19.7.6, Part I). In part 3, the policy commands are with the names of the form `TPM2_PolicyXXX()` where `"XXX"` is a Label of the type of policy assertions. In this subsection, we will give a classification of the policy assertion commands. The assertion commands are listed in **Table 2**. The table also includes the actions that the TPM will take in each command, the description of each assertion, and the classification.

Two or more assertions can be combined by logical connectives AND and OR. First, we describe an AND way. There is no explicit AND command. An AND policy may be expressed in an equation as a sequence of assertions that must all be satisfied. To verify it, the user should sequentially invoke all the assertion commands in the equation with the same policy session. These commands will validate the assertions and update the session's policyDigest (optionally with other context values). The final digest value indicates the evaluated policy. In each assertion command, the policyDigest value is computed as

$$polD_{new} := \text{HASH}\,(polD_{old} || polLabel_{new} || Param_{new}) \quad (1)$$

where $polLabel_{new}$ identifies the new policy assertion and $Param_{new}$ is the parameters associated to the assertion. For example in `TPM2_PolicyPCR`, $polLabel_{new}$ equals 'PCR' and $Param_{new}$ is the desired PCR value. The way of computation is like a chain to record the sequential evaluation of the assertions in an AND policy.

The TPM 2.0 has classified the AND policy assertion commands into 3 different types: immediate, deferred and combined. For an immediate command, the TPM validates the assertion of the input values and then updates the ses-

Table 2: Policy assertion commands

Label	Actions	Description of assertion	classification
NV	validate selected NV (Non-Volatile) update polD (policyDigest)	NV Index has the desired relationship with the input value	Immediate
PCR	validate selected pcr update polD, pcrUpdateCounter	Selected pcr has the desired value	Combined
CounterTimer	validate internal clock update polD	Internal clock has the desired relationship with the input value	Immediate
CpHash	update polD, cpHash	Auth for commands with a specified cpHash	Deferred
NameHash	update polD, nameHash	Auth for commands with a specified nameHash	Deferred
CommandCode	update polD, commandCode	Auth for a specified command	Deferred
DuplicationSelect	update polD, cpHash, commandCode	Auth for `TPM2_Duplicate` with a specified cpHash	Deferred
AuthValue	update polD, isAuthValueNeeded	an HMAC keyed on the authValue required	Deferred
Password	update polD, isPasswordNeeded	a password required	Deferred
Locality	update polD, locality	Auth for commands executed at specified locality	Deferred
PhysicalPresence	update polD, isPPRequired	physical presence required	Deferred
NvWritten	update polD, checkNvWritten, NvWrittenState	NV Index has the desired attribute for written	Deferred
Signed	validate signature of param update polD, timeout	Auth bound with session (used once), specified polRef, expiration for auth	Combined
Secret	validate HMAC of param update polD, timeout	Auth bound with session (used once), specified polRef, expiration for auth	Combined
Ticket	validate tickets for specific authorization update polD, timeout	specified cpHash, expiration for auth	Combined
Authorize	validate and update polD	polD has been signed by a specified key	Authorize
OR	validate and update polD	polD is in the list of digest	OR

sion's policyDigest if and only if the assertion is valid. For a deferred command, the TPM conducts no validation but only updates the policyDigest based on the input values and records some parameters in the session's context. These parameters will be checked when the session is used for authorization. For a combined assertion, the TPM both validates some conditions of the input and records some parameters in the session's context.

Among the AND policy assertion commands, there is a special assertion, `TPM2_PolicyAuthorize`, which allows a dynamical permission of new policies. It is an immediate assertion, but we consider it independently. When this assertion is included in a policy, it allows a designated entity (with a signing key) to authorize a policyDigest to be included in the policy by signing it. During the evaluation of this assertion, a signature of the session's current policyDigest value should be provided and verified by the TPM. If the signature is verified, the session's current policyDigest value is updated by the public part of the signing key as

$$polD_{new} := \text{HASH}\,(0\cdots0||\text{'Authorize'}||\,(pk, polR)) \quad (2)$$

For OR connection, it allows a certain list of branches for the OR policy. Each branch corresponds to one digest value. During the evaluation of the assertion command `TPM2_PolicyOR`, the TPM compares the session's current policyDigest value with a list of digest values provided by the caller. If the current one is in the list, the TPM will set the current policyDigest to zero, concatenate all the digests in the list, and update the new policyDigest as

$$polD_{new} := \text{HASH}\,(0\cdots0||\text{'OR'}||digestLists) \quad (3)$$

2.2.2 An example for the EA mechanism

We will give a simple example for how the EA mechanism works in the TPM API commands. The administrator needs to create a signing key on the platform with a TPM. He may require that Alice can sign with this key, but only Bob can certify it, and further,

- Alice (with a signing key sk_A) can only sign with this key five times (as measured by the NV counter);
- Bob (with a signing key sk_B) can only certify the key when the platform is in a certain state (as measured by the PCR).

First, the administrator should set an authPolicy value for the policy. It is set by the following computation.

$$\begin{aligned} aP &:= \text{HASH}(0\cdots0||\text{'OR'}||Z_1||Z_2)\\ Z_1 &:= \text{HASH}(Z_N||\text{'Signed'}||PK(sk_A)||\text{'Alice'})\\ Z_2 &:= \text{HASH}(Z_P||\text{'Signed'}||PK(sk_B)||\text{'Bob'})\\ Z_N &:= \text{HASH}(Z_{C1}||\text{'NV'}||NVC1||LT||5)\\ Z_P &:= \text{HASH}(Z_{C2}||\text{'PCR'}||PCR1||v)\\ Z_{C1} &:= \text{HASH}(0\cdots0||\text{'CmdCode'}||\text{CC\_Sign})\\ Z_{C2} &:= \text{HASH}(0\cdots0||\text{'CmdCode'}||\text{CC\_Certify}) \end{aligned}$$

By calling the commands `TPM2_Create`$(aP) \rightarrow kB$ and `TPM2_Load`(kB) $\rightarrow hK$, the administrator creates a signing key in the TPM. He also sets the NV counter $NVC1$ to 0 and requires that each use of signing with this key will cause the counter value plus 1[1].

Suppose Alice wants to sign with this key. Before she uses it, Alice should do the following steps to get the authorization session. If any step fails, Alice cannot use the key.

1. By calling `TPM2_StartAuthSession`() $\rightarrow (hS, nt)$, Alice starts a new policy session with the policyDigest initialized to $0\cdots0$.

2. She calls the assertion command `TPM2_CommandCode`(hS, CC_Sign) to update the session's commandCode to CC_Sign. The policyDigest is extended to Z_{C1}.

[1]This can be done by a more complex setting of Z_1, which is described in [12](Section 4.2.2). For simplicity, we just suppose it can be achieved here.

3. She triggers the command `TPM2_PolicyNV`(hS, $NVC1$, LT, 5) to validate that the counter $NVC1$'s value is less than 5. If valid, the TPM updates the policyDigest to Z_N, which accumulates the name of the NV Index, the comparison operation LT, and the operand '5'.

4. By using Alice's signing key, she can obtain a signature $\sigma_A = \text{Sig}(\langle nt, \text{'Alice'}\rangle, sk_A)$ where nt is the session's nonce, the text 'Alice' is called the policy reference ($polR$), which is an opaque value determined by the authorizing entity (Alice). Then she calls the command `TPM2_PolicySigned`(hS, σ_A, 'Alice', $PK(sk_A)$) to validate the signature. If valid, the TPM updates the policyDigest to Z_1, which encapsulates her public key and the signed label.

5. She calls the command `TPM2_PolicyOR`(hS, $\langle Z_1, Z_2\rangle$) to check that the session's current policyDigest is on the list $\langle Z_1, Z_2\rangle$. If valid, the TPM updates the session's policyDigest to a value equals aP.

6. Alice triggers the command `TPM2_Sign`(hK, hS, msg). The TPM will check the authorization according to the input handles. Since the session's policyDigest matches the key's authPolicy value and the session's commandCode value equals the authorized command's code CC_Sign, the TPM executes the command to sign msg. The TPM also increments $NVC1$ by 1.

It is similar for Bob to get the authorization for the command `TPM2_Certify`(), except he should use the command code CC_Certify in `TPM2_CommandCode`() and the assertion command `TPM2_PolicyPCR`() instead of `TPM2_PolicyNV`().

3. MODEL OF EA IN TPM 2.0 API

3.1 An overview of the model

In this section, we will propose a model of security API analysis in a stateful process calculus introduced in Section 2.1. This model, as a framework, is applied to the API analysis of EA mechanism in TPM 2.0. The main process is as follows.

$$\begin{aligned} P_{main} &:= P_{Init}; !(P_1|P_2|\cdots) \\ P_{Init} &:= \nu\tilde{k}_I; \bar{P}_{Init} \\ P_i &:= \text{in}(\langle {}'Label'_i, \tilde{x}_i\rangle); \nu\tilde{k}_i; [\text{lock } \tilde{h}_i;] \bar{P}_i; \\ &\quad \text{out}(\langle \tilde{M}_i\rangle)[; \text{unlock } \tilde{h}_i] \end{aligned}$$

This is a general model. The instantiation is in Section 3.2. P_{main} is the main process of the API, in which P_{Init} and P_i respectively model the program of initialization and the specific commands. Initialization of the TPM is executed before all the other commands. The process $\nu\tilde{k}_I$ models the generation of global values. $\bar{P}_{Init}$ is the concrete initialization process of the TPM internal states. All the commands P_i could be executed concurrently and repeatedly. Each command has one input and one output. The process $\nu\tilde{k}_i$ models the generation of fresh local values. The lock $\tilde{h}_i$ process locks all the resource handles $\tilde{h}_i$ operated by the following concrete execution processes in $\bar{P}_i$ and decides exclusive access to them until a process unlock $\tilde{h}_i$. The (un)lock processes in square brackets might be omitted according to the two execution modes of the TPM API. There are no communication and (un)lock processes in the concrete execution process $\bar{P}_i$. All the unbound variables in $\bar{P}_i$ should be bound in the input variables $\tilde{x}_i$. In fact, $\bar{P}_i$ deals with the handle maps, which are usually modeled as global states.

We will give an informal description for the threat model. We assume the administrator creates the object and sets its authPolicy value. The malicious user, as the adversary, needs to gain authorization for the uses of the object and tries to violate the policy assertions encapsulated in its authPolicy value. The adversary model is formalized in a classic Dolev-Yao style. The malicious user, as the adversary, could only receive and send messages on the public channel as well as deduce terms. He could call all the API commands with any parameters in his knowledge. However, the adversary cannot touch the internal states except through the API provided by the TPM.

By injecting the processes lock $\tilde{h}_i$ and unlock $\tilde{h}_i$ into the command processes P_i, we establish two modes for the execution model of TPM API commands.

Mode 1, with (un)lock processes, specifies that all the commands are called sequentially and cannot be executed in parallel. Each command gains exclusive access to its resources. This execution mode is suitable for the TPM API implemented by a dedicated chip.

Mode 2, without (un)lock processes, models a concurrency based TPM API. Although TPM is originally designed to be implemented as a dedicated chip and does not allow concurrent execution of commands, the TPM 2.0 specification [8] (Section 9.3, Part I) mentions another reasonable implementation. This version of a TPM is implemented as the code run on the host processor in a hardware-based Trusted Execution Environments (TEEs). They provide secure, integrity-protected processing environments, consisting of processing, memory and storage capabilities, that are isolated from the regular processing environment. This mode may allow concurrent execution of commands. There are several different schemes for achieving this mode including System Management Mode, Trust Zone™, and processor virtualization. In mode 2, the security API should be analyzed more carefully due to race conditions.

3.2 Modeling the TPM commands

In this section, we instantiate the above model to formalize the TPM API commands described in Section 2.2. We do not limit the number of sessions started by the adversary. However, there are 109 commands in part 3 of the TPM 2.0 specifications [8] in total. EA mechanism can be applied to Most of them. One of the most difficult problems we have faced is the high complexity of the analysis. Thus we need to do 3 simplifications in our model.

First, we need to simplify the object management commands. We find that the policy set by the internal object's authPolicy is not type-specific. Thus we use a generic type instead of any specific one. We also give out a fictional framework of commands for the management of this type, including 2 commands: `TPM2_NewObj` and `TPM2_UseObj` for the creation and use of such an object. In the first command, a new authPolicy value is set for the created object. In fact, it completes the sequential executions of the real TPM 2.0 API commands `TPM2_Create` and `TPM2_Load`. In the second command, a session should be provided for the authorization and an authorization check is done. It simulates the real TPM function `CheckPolicyAuthSession()` which is performed before any commands that need a policy based authorization. We also introduce the assumption that each session could be used for authorization only once. This

assumption appears to be quite reasonable since the session is initialized after each use in the practical case.

Second, we cut off several branches for the authorization checking logic. In the practical case, all of the session's context values should be checked before the authorized command is executed. In our model, we just check the authPolicy value in the command `TPM2_UseObj`. Instead we set the events CmdXXX(ap, ctv) to denote the check of the context value ctv but with no block if failed. Actually, we do not care whether such a check is successful or failed in our model. We do care the fact that in the authorized command, the session's context values ctv should be consistent with the policy encapsulated in the authPolicy value ap. If it is verified, ctv could be used in a check in the practical case. More details about this security property is described in Section 3.3. In this way, we cut off 2 checking logic branches for each context value[2].

Third, we do not model an internal clock of the TPM. The internal clock mechanism could be modeled as a monotonic increasing counter which is incremented every time a command is called. However, adding such a counter to our model will increase the complexity of analyzing. Thus we omit the time-related policy in our model, which includes the CounterTimer assertion and the input parameter *timeout* in the combined assertions. It does no harm to the soundness of our model since the caller could disable time-related policy in the practical case. For the other internal states, we take only one NV Index and one PCR. All the results in our paper could be generalized to a model with a finite number of NV Indexes and PCRs without any extra efforts.

The signature used in our model is $\Sigma := \{$H/1, zero/0, nvh/0, $pcrh$/0, PK/1, Sig/2, CkSig/2$\}$ with the equation CkSig(Sig(m, k), PK(k)) = m. Besides the digital signature functions PK/1, Sig/2, CkSig/2 described in Example 1, the function H/1 specifies the digest function used in PCR extending and policy digest updating. zero/0 is the initial value for some session's context values. We use nvh/0 for the handle of the NV Index and $pcrh$/0 for the handle of the PCR.

Now we are able to model the main process of the EA mechanism in the TPM API according to our model in Section 3.1. For simplicity, we divide the commands into 3 groups of parallel processes: management of objects and sessions (P_O with 3 commands), management of the internal states (P_S with 2 commands), and policy assertion commands (P_A with 14 commands). Thus, the main process is $P_{TPM} = P_I; !(P_O|P_S|P_A)$. Note that our model includes 2 modes which respectively specify the TPM API implemented in hardware and software. For simplicity, we only give out Mode 1 with (un)lock processes. Mode 2 can be obtained by omitting them.

The process of initialization P_I is described in **Table 3**. P_I initializes the value of NV and PCR with zero/0. The context of the PCR includes a monotonic counter *pcrUpdateCounter* (denoted by label 'pcrUpdC') which is incremented by 'one' every time the PCR value is changed. We model this counter as a multiset only consisting of the symbols 'zero' and 'one'. The union operator is denoted with a plus sign (+).

P_O includes 3 commands: $P_O := P_{NewObj} | P_{UseObj} | P_{StartAS}$. The details are described in **Table 4**. P_{NewObj} for the command `TPM2_NewObj` takes as input the *authPolicy* value ap, binds it to a fresh handle, and outputs the handle. P_{UseObj} for the command `TPM2_UseObj` conducts an authorization check for the object $hObj$ through the policy session with the input handle hS. Note that we omit several events CmdXXX(ap, ctv) here since they are similar to CmdCPH. $P_{StartAS}$ for the command `TPM2_StartAuthSession` is invoked to generate a new policy session and initialize the session's context values.

Table 3: The process of initialization

$P_I :=$ insert $\langle$'NV', $nvh\rangle$, zero;
insert $\langle$'PCRV', $pcrh\rangle$, zero;
insert $\langle$'pcrUpdC', $pcrh\rangle$, 'zero';

P_S includes 2 commands: $P_S := P_{NVWrite} | P_{PCRExtend}$. For the NV Index, the internal state is modified by $P_{NVWrite}$ and for PCR by $P_{PCRExtend}$. Note that when the 'PCRV' value is extended, the old counter 'pcrUpdC' plus 'one' makes the new counter value. The details are given in **Table 4**.

P_A contains 14 policy assertion commands, which includes the assertions in **Table 2** except for the CounterTimer assertion, Secret assertion, and Ticket assertion. As in **Table 2**, the policy assertion commands can be divided into 5 classifications: immediate, deferred, combined, assertion OR, and assertion Authorize. For the limited space, we only present 6 commands. We present the typical one for each classification except for the combined assertions, for which we present two assertions. The full model including the rest assertions are listed in the full version of this paper. Note that all the assertion commands take as input a policy session's handle h and update the session's policyDigest (labeled by 'polD') according to the equations (1) to (3) in Section 2.2.1. We denote it by a hash digest of a 3-tuple. The first element is for the old policyDigest, the second is for the label of the assertion command, and the third is a vector of some input parameters used in the assertion.

- The immediate assertion group includes two assertions: NV and CounterTimer. As we have clarified before, we omit the CounterTimer assertion. The command P_{NV}, for NV assertion, takes as input a comparison value v and validates that it matches the current NV's context value nv. If matched, TPM will update the 'polD' of the input session according to the equation (1). Note that the immediate assertion might be misused in a deferred manner, which is described in Section 4.

- The deferred assertion group includes 9 commands as in **Table 2**. These commands just need two operations: updating the policyDigest value and modifying some specific context values of the session. In convenience we present P_{cpHash} for the assertion CpHash as an example. This command takes as input a value cph that should be used to modify the session's context 'cpHash'. This modification will be checked in the authorized command P_{UseObj} by the event CmdCPH(ap, cph), which is described in Section 3.3. P_{cpHash} also updates the input session's context 'polD' according to the equation (1).

- The combined assertions are composed of 4 assertions: Signed, Secret, Ticket, and PCR. The Signed and Se-

[2]One is for default setting and one for failed validation

Table 4: The processes of API commands

P_{NewObj} :=
in(⟨'NewObj', ap⟩);
ν h;
lock h;
insert ⟨'AuthP', h⟩, ap;
out(h);
unlock h

$P_{StartAS}$:=
in(⟨'StartAS'⟩); ν h; ν nt;
lock h;
event SessionBind(nt, h)
insert ⟨'PolD', h⟩, zero;
insert ⟨'cpHash', h⟩, zero;
insert ⟨'nameHash', h⟩, zero;
insert ⟨'commandCode', h⟩, zero;
insert ⟨'commandLocality', h⟩, zero;
insert ⟨'isPPRequired', h⟩, 'CLEAR';
insert ⟨'isAuthValueNeeded', h⟩, 'CLEAR';
insert ⟨'isPasswordNeeded', h⟩, 'CLEAR';
insert ⟨'pcrUC', h⟩, 'zero';
insert ⟨'NonceTPM', h⟩, nt;
insert ⟨'checkNVWritten', h⟩, 'CLEAR';
insert ⟨'nvWrittenState', h⟩, zero;
out(⟨h, nt⟩);
unlock h

$P_{PCRExtend}$:=
in(⟨'PCRExtend', v⟩);
lock $pcrh$;
lookup ⟨'PCRV', $pcrh$⟩ as $pcrv$ in
lookup ⟨'pcrUpdC', $pcrh$⟩ as pc in
event PCRExtend(pc);
insert ⟨'pcrUpdC', $pcrh$⟩, pc + 'one';
insert ⟨'PCRV', $pcrh$⟩, H(⟨$pcrv, v$⟩);
unlock $pcrh$

P_{PCR} :=
in(⟨'PolicyPCR', h, v⟩);
lock h; lock $pcrh$;
lookup ⟨'PCRV', $pcrh$⟩ as $pcrv$ in
lookup ⟨'PolD', h⟩ as pd in
if $v = pcrv$ then
  event Pol('PCR', h, pd, v);
  insert ⟨'PolD', h⟩, H(⟨pd, 'PCR', v⟩);
  lookup ⟨'pcrUpdC', $pcrh$⟩ as pc in
  insert ⟨'pcrUC', h⟩, pc;
  unlock $pcrh$; unlock h
else unlock $pcrh$; unlock h

P_{cpHash} :=
in(⟨'PolicyCpHash', h, cph⟩);
lock h;
lookup ⟨'PolD', h⟩ as pd in
event Pol('CpHash', h, pd, cph);
insert ⟨'PolD', h⟩, H(⟨pd, 'CpHash', cph⟩);
insert ⟨'cpHash', h⟩, cph
unlock h;

$P_{Authorize}$
in(⟨'PolicyAuthorize', h, sig, pk, $polR$⟩);
lock h;
lookup ⟨'PolD', h⟩ as pd in
if CkSig(sig, pk) = ⟨$pd, polR$⟩ then
  event Pol('Authorize', h, zero, ⟨$pk, polR$⟩);
  insert ⟨'PolD', h⟩, H(⟨zero, 'Authorize',⟨$pk, polR$⟩⟩);
  unlock h
else unlock h

P_{UseObj} :=
in(⟨'UseObj', $hObj$, hS⟩);
lock $hObj$; lock hS; lock $pcrh$; lock nvh;
lookup ⟨'AuthP', $hObj$⟩ as ap in
lookup ⟨'cpHash', hS⟩ as cph in
lookup ⟨'nameHash', hS⟩ as nmh in
lookup ⟨'commandCode', hS⟩ as cc in
lookup ⟨'commandLocality', hS⟩ as Loc in
lookup ⟨'isPPRequired', hS⟩ as PP in
lookup ⟨'isAuthValueNeeded', hS⟩ as AV in
lookup ⟨'isPasswordNeeded', hS⟩ as PW in
lookup ⟨'checkNVWritten', hS⟩ as NW in
lookup ⟨'nvWrittenState', hS⟩ as WS in
lookup ⟨'NV', nvh⟩ as nv in
lookup ⟨'PCRV', $pcrh$⟩ as $pcrv$ in
lookup ⟨'pcrUpdC', $pcrh$⟩ as pc in
lookup ⟨'pcrUC', hS⟩ as pcS in
lookup ⟨'PolD', hS⟩ as pd in
unlock nvh; unlock $pcrh$; unlock hS; unlock $hObj$;
if $pd = ap$ then
  event UseObj(ap, hS);
  event CmdCPH(ap, cph);
  event CmdNV(ap, nv);
  event CmdPCRV(ap, $pcrv$, pc, pcS);
  out('Success')

$P_{NVWrite}$:=
in(⟨'NVWrite', v⟩);
lock nvh;
insert ⟨'NV', nvh⟩, v;
unlock nvh

P_{NV} :=
in(⟨'PolicyNV', h, v⟩);
lock nvh; lock h;
lookup ⟨'NV', nvh⟩ as nv in
lookup ⟨'PolD', h⟩ as pd in
if $v = nv$ then
  event Pol('NV', h, pd, v);
  insert ⟨'PolD', h⟩, H(⟨pd, 'NV', v⟩);
  unlock h; unlock nvh

P_{Signed} :=
in(⟨'PolicySigned', h, sig, pk, $polR$⟩);
lock h;
lookup ⟨'PolD', h⟩ as pd in
lookup ⟨'NonceTPM', h⟩ as nt in
if CkSig(sig, pk) = ⟨$nt, polR$⟩ then
  event Pol('Signed', h, pd, ⟨$pk, polR$⟩);
  insert ⟨'PolD', h⟩, H(⟨pd, 'Signed', ⟨$pk, polR$⟩⟩);
  unlock h
else unlock h

P_{OR} :=
in(⟨'PolicyOR', h, ⟨ld, rd⟩⟩);
lock h;
lookup ⟨'PolD', h⟩ as pd in
if $pd = ld$ then
  event Pol('OR', h, pd, ⟨ld, rd⟩);
  insert ⟨'PolD', h⟩, H(⟨zero, 'OR', ⟨ld, rd⟩⟩);
  unlock h
  else if $pd = rd$ then
    event Pol('OR', h, pd, ⟨ld, rd⟩);
    insert ⟨'PolD', h⟩, H(⟨zero, 'OR', ⟨ld, rd⟩⟩);
    unlock h
    else unlock h

cret assertions convey an authorization by signing a set of parameters that indicate the nature of the authorization. In the former assertion command, the signature is signed by an asymmetric signing key and in the latter one, the signature is signed by an HMAC key. The signed parameters include nonceTPM (nt) for binding the authorization to the specified session, and policy reference ($polR$) for an opaque value relating to the authorizing entity (An example could be found in Section 2.2.2). These two commands both can output an HMAC (called a ticket) which can be used in the Ticket assertion, which functions in the same way. Since the 3 assertion commands have similar operations, we only present one typical command P_{Signed} for the Signed assertion. P_{Signed} takes as input a signature sig and its verification key pk (instead of a handle for pk, in the practical case). The TPM validates the signature and updates the session's context 'polD' according to the equation (1).

- We present the command P_{PCR} for combined assertion PCR. This command takes as input a digest value v and validates that it matches the current PCR's context 'PCRV' value $pcrv$. If matched, TPM will update the 'polD' of the input session according to the equation (1). Note that in P_{PCR}, the session's context 'pcrUC' is updated by the PCR's context 'pcrUpdC' for a future match in the authorized command P_{UseObj}. This mechanism is set to defend against time-of-check time-of-use (TOCTOU) attack.

- The assertion command `TPM2_PolicyAuthorize` is modeled as $P_{Authorize}$. By using the input public key pk, the TPM validates that the input value sig is a signature of the session's current policyDigest pd and the policy reference $polR$. If valid, the TPM updates the session's context 'polD' according to the equation (2). In the real TPM, the signature is verified by another command. Successful validation produces a TPM-specific ticket (HMAC). We merge the validation into the assertion command since the ticket is signed by a TPM hierarchy proof value and is unforgeable (Section 11.4.5.3 and 14.4, Part I, specification [8]).

- The OR assertion command `TPM2_PolicyOR` takes as input a digest list of at most 8 digests. In our model we cut this number to 2 in P_{OR}, since it can be expanded by using cascading OR assertions. The command P_{OR} takes as input a pair of digest values $\langle ld, rd\rangle$ and checks whether the session's policyDigest pd matches ld or rd, or neither. If matched, the TPM updates the session's context 'polD' according to the equation (3).

In our model, the main process of the EA mechanism in TPM API is as follows. We present the full model transcripts of the two modes in [20].

$$
\begin{aligned}
P_{TPM} &:= P_I; !(P_O|P_S|P_A)\\
P_O &:= P_{NewObj}|P_{UseObj}|P_{StartAS}\\
P_S &:= P_{NVWrite}|P_{PCRExtend}\\
P_A &:= P_{PCR}|P_{NV}|P_{Signed}|P_{Authorize}|P_{OR}|P_{cpHash}|\\
&\quad P_{nameHash}|P_{commandCode}|P_{locality}|P_{PhysicalPresence}|\\
&\quad P_{AuthValue}|P_{Password}|P_{DuplicationSelect}|P_{NvWritten}.
\end{aligned}
$$

3.3 Modeling the security properties

The security properties of EA mechanism are not explicitly explained in the TPM 2.0 specifications, although some hints are provided. The specification [8] (Section 19.7, Part I) states that "Enhanced authorization is a TPM capability that allows entity-creators or administrators to require **specific tests** or **actions** to be performed before an action can be completed. The specific policy is encapsulated in a value called an authPolicy that is associated with an entity". We give out an informal description of the security properties of EA mechanism.

1. (For required **specific actions**) If the TPM has executed the authorized command, it MUST have executed a sequence of policy assertion commands corresponding to the policy encapsulated in the object's *authPolicy*;

2. (For required **specific tests**) If the TPM has executed the authorized command, its execution environment MUST have been tested to be consistent with the policy encapsulated in the object's *authPolicy*.

3.3.1 Formalization of Property 1

The adversary model has been described in Section 3.1. In the formalization of the first property, we have met two challenges. The first is that there is more than one expression of the authPolicy setting for a given policy. We solve it by transforming a given policy into a disjunctive normal form and providing a generic way to compute the authPolicy setting for it.

As stated in Section 2.2.1, the policy can be expressed by the OR and AND combinations of policy assertions. We denote by $Assert(polLabel, Param)$ all the assertions listed in **Table 2** except for the OR assertion (which we call AND assertions). $polLabel$ is for the Label of the assertion and $Param$ for the parameters in equation (1). Although for the AND assertions the order of execution matters, we could list all the possible permutations and combine them by using the OR connective. Thus a policy can be expressed in a disjunctive normal form (DNF):

$$\text{policy} = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} Assert(L_{i,j}, pm_{i,j}). \quad (4)$$

Given the policy, we provide a generic way to set the corresponding authPolicy value ap.

For $n = 1$, $(1 \le j \le m_1)$

$$
\begin{aligned}
ap &:= \mathrm{H}(\langle ap_{1,1}, L_{1,1}, pm_{1,1}\rangle)\\
ap_{1,j} &:= \mathrm{H}(\langle ap_{1,j+1}, L_{1,j}, pm_{1,j}\rangle)\\
ap_{1,m_1+1} &:= \text{zero}
\end{aligned}
$$

As we have stated in Section 2.2.1, ap encodes the unique branch for AND assertions by linking $ap_{1,j}$ to $ap_{1,j+1}$. Note that in each branch, the test order is from $j = m_1$ to 1.

For $n \ge 2$, $(0 \le k \le n-2,\ 1 \le i \le n,\ 1 \le j \le m_i)$

$$
\begin{aligned}
ap &:= rd_0\\
rd_k &:= \mathrm{H}(\langle \text{zero}, \text{'OR'}, \langle ld_{k+1}, rd_{k+1}\rangle\rangle)\\
rd_{n-1} &:= ld_n\\
ld_i &:= \mathrm{H}(\langle ap_{i,1}, L_{i,1}, pm_{i,1}\rangle)\\
ap_{i,j} &:= \mathrm{H}(\langle ap_{i,j+1}, L_{i,j}, pm_{i,j}\rangle)\\
ap_{i,m_i+1} &:= \text{zero}
\end{aligned}
$$

Each ld_i encodes the i-th branch by linking $ap_{i,j}$ to $ap_{i,j+1}$. Each rd_k encodes the remained disjunctive formula from $i = k+1$ to n. The OR assertion connects ld_i with rd_i.

With the expression of the specific authPolicy values for any given policies in (4), we can formalize Property 1. Each assertion in the policy should be preceded by one execution of the policy assertion command, which is denoted by the event Pol.

Property 1(*SpecAct*). For the authPolicy value ap corresponding to any given policies in a disjunctive normal form (4), it should have $Tr(P_{TPM}) \models^{\forall} \phi$, where $t_{i,0} = t_0$ for $1 \le i \le n$ and

$$\phi := \forall h_S, t_0.\text{UseObj}(ap, h_S)@t_0 \Rightarrow \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} (\exists t_{i,j}.\text{Pol}(L_{i,j}, h_S, ap_{i,j+1}, pm_{i,j})@t_{i,j} \wedge t_{i,j} < t_{i,j-1}).$$

Example 3. In Section 2.2.2, we give a simple example of the EA mechanism. Its policy in disjunctive normal form is as follows. Due to the simplification in our model, we omit some parameters in the 'NV' and 'PCR' assertions

$$\begin{pmatrix} Assert(\text{'Signed'}, \langle \text{PK}(sk_A), \text{'Alice'}\rangle) \wedge \\ Assert(\text{'NV'}, 5) \wedge Assert(\text{'CmdCode'}, \text{CC\_Sign}) \end{pmatrix} \vee \begin{pmatrix} Assert(\text{'Signed'}, \langle \text{PK}(sk_B), \text{'Bob'}\rangle) \wedge \\ Assert(\text{'PCR'}, v) \wedge Assert(\text{'CmdCode'}, \text{CC\_Certify}) \end{pmatrix}$$

The trace formula to be verified is as follows.

$$\phi := \forall h_S, t_0.\text{UseObj}(aP, h_S)@t_0 \Rightarrow \begin{pmatrix} \exists t_{11}, t_{12}, t_{13}.t_{13} < t_{12} \wedge t_{12} < t_{11} \wedge t_{11} < t_0 \wedge \\ \text{Pol}(\text{'Signed'}, h_S, Z_N, \langle \text{PK}(sk_A), \text{'Alice'}\rangle)@t_{11} \wedge \\ \text{Pol}(\text{'NV'}, h_S, Z_{C1}, 5)@t_{12} \wedge \\ \text{Pol}(\text{'CmdCode'}, h_S, \text{zero}, \text{CC\_Sign})@t_{13} \end{pmatrix} \vee \begin{pmatrix} \exists t_{21}, t_{22}, t_{23}.t_{23} < t_{22} \wedge t_{22} < t_{21} \wedge t_{21} < t_0 \wedge \\ \text{Pol}(\text{'Signed'}, h_S, Z_P, \langle \text{PK}(sk_B), \text{'Bob'}\rangle)@t_{21} \wedge \\ \text{Pol}(\text{'PCR'}, h_S, Z_{C2}, v)@t_{22} \wedge \\ \text{Pol}(\text{'CmdCode'}, h_S, \text{zero}, \text{CC\_Certify})@t_{23} \end{pmatrix}$$

The second challenge we have met is that the universal quantifier over the authPolicy value makes it difficult to verify Property 1. In fact, the number of branches n and the length of digest chain m_i is unbounded. We solve this problem by using the linking of digest chain in the event and divides the whole proof into a chain of pieces. If we can prove the fact that every piece of the chain is true and can be reused, then we could give out a whole proof for Property 1. We achieve this goal by providing 2 lemmas.

Lemma 1(*CorUsePol*). $Tr(P_{TPM}) \models^{\forall} \phi_1$, where

$$\phi_1 := \forall h_S, pd, L, pm, t_0. \text{UseObj}(\text{H}(\langle pd, L, pm\rangle), h_S)@t_0 \Rightarrow (\exists t_1.L \neq \text{'OR'} \wedge \text{Pol}(L, h_S, pd, pm)@t_1 \wedge t_1 < t_0) \vee \begin{pmatrix} \exists ld, rd, t_2.L = \text{'OR'} \wedge pd = \text{zero} \wedge pm = \langle ld, rd\rangle \wedge \\ \text{Pol}(L, h_S, ld, pm)@t_2 \wedge t_2 < t_0 \end{pmatrix} \vee \begin{pmatrix} \exists ld, rd, t_3.L = \text{'OR'} \wedge pd = \text{zero} \wedge pm = \langle ld, rd\rangle \wedge \\ \text{Pol}(L, h_S, rd, pm)@t_3 \wedge t_3 < t_0 \end{pmatrix}.$$

Lemma 1 states that any execution of the event UseObj with ap = H($\langle pd, L, pm\rangle$) and h_S should be preceded by the execution of the event Pol with the same payload L, h_S, pd, and pm. There are 3 branches. For the case that $L \neq$'OR', it is an AND assertion with the old policyDigest in the event Pol equals pd. For the other two cases that $L =$'OR', the policyDigest pd in the event UseObj should be zero but the old policyDigest in the event Pol should be either ld or rd in the digest list pm.

Lemma 2(*CorPol1Pol2*). $Tr(P_{TPM}) \models^{\forall} \phi_2$, where

$$\phi_2 := \forall L_1, pm_1, L_2, pm_2, pd, h_S, t_0. \text{Pol}(L_1, h_S, \text{H}(\langle pd, L_2, pm_2\rangle), pm_1)@t_0 \Rightarrow (\exists t_1.L \neq \text{'OR'} \wedge \text{Pol}(L_2, h_S, pd, pm_2)@t_1 \wedge t_1 < t_0) \vee \begin{pmatrix} \exists ld, rd, t_2.L_2 = \text{'OR'} \wedge pd = \text{zero} \wedge pm_2 = \langle ld, rd\rangle \wedge \\ \text{Pol}(\text{'OR'}, h_S, ld, pm_2)@t_2 \wedge t_2 < t_0 \end{pmatrix} \vee \begin{pmatrix} \exists ld, rd, t_3.L_2 = \text{'OR'} \wedge pd = \text{zero} \wedge pm_2 = \langle ld, rd\rangle \wedge \\ \text{Pol}(\text{'OR'}, h_S, rd, pm_2)@t_3 \wedge t_3 < t_0 \end{pmatrix}.$$

Lemma 2 states that any execution of the event Pol with the policyDigest $polD$ = H($\langle pd, L_2, pm_2\rangle$) should be preceded by the execution of another event Pol with the payloads encapsulated in $polD$ and the same policy session handle h_S. As in the lemma *CorUsePol*, we should differentiate 3 branches of L_2.

(**Proof sketch of Property 1.**) If Lemma 1 and Lemma 2 are both verified, we can prove Property 1. In the following, we give a sketch. The formal proof could be found in the full version of this paper. Given an authPolicy value ap, by lemma 1, the event UseObj(ap, h_S) should be preceded by the execution of an event Pol according to the end part of the digest chain ap. There might be three cases. For case 1, the end part is an AND assertion. Then the number of branches n is one. By using the inductive method, it is trivial to give out the proof by lemma 2 (for only one branch). If the end part is an OR assertion, then we will go into the left part case, which is similar to case 1, and the right part case, which can be proved by lemma 2 through the induction method. The conclusion of ϕ in Property 1 can be achieved by omitting all the events Pol labeled with the OR assertion in the proof we have obtained.

3.3.2 Formalization of Property 2

The second property requires specific tests before the execution of the authorized command. The assertions are classified into 5 groups as in **Table 2**, which are managed separately. By TPM design, the tests for the immediate NV assertion and the OR assertion are performed in their respective policy assertion commands (in an immediate manner). We have stated that in Property 1, the use of object should be preceded by the execution of the corresponding policy assertion commands, which have performed the immediate assertion tests. Thus we do not need to consider these two assertions. For the deferred assertions, we use the CpHash assertion as a standard. The verification of the other deferred assertions such as NameHash and CommandCode are similar to that of the CpHash assertion. It is trivial to extend the analysis results. Thus we only focus on the CpHash assertion. In the following, we give out Lemma 3 to Lemma 6 respectively for the Authorize assertion, the combined Signed assertion, the deferred CpHash assertion, and the PCR assertion.

Lemma 3(*PolAuth*). $Tr(P_{TPM}) \models^{\forall} \phi_3$, where

$$\phi_3 := \forall h_S, sk, polR, t_0. \text{Pol}(\text{'Authorize'}, h_S, \text{zero}, \langle \text{PK}(sk), polR\rangle)@t_0 \Rightarrow (\exists pd, t_1.\text{K}(\text{Sig}(\langle pd, polR\rangle, sk))@t_1 \wedge t_1 < t_0)$$

Lemma 4(*PolSigned*). $Tr(P_{TPM}) \models^{\forall} \phi_4$, where

$$\phi_4 := \forall h_S, sk, pd, polR, t_0. \text{Pol}(\text{'Signed'}, h_S, pd, \langle \text{PK}(sk), polR\rangle)@t_0 \Rightarrow \begin{pmatrix} \exists nt, t_1, t_2.\text{K}(\text{Sig}(\langle nt, polR\rangle, sk))@t_1 \wedge t_1 < t_0 \wedge \\ \text{SessionBind}(nt, h_S)@t_2 \wedge t_2 < t_0 \end{pmatrix}$$

Lemma 3 and Lemma 4 respectively states that in order to successfully execute the Authorize and Signed assertion commands, the attacker should know the signature of some specific payloads (pd and $polR$ for 'Authorize'; nt and $polR$ for 'Signed'). Note that for the Signed assertion, the signed nt should be bound with the policy session h_S.

Lemma 5($PolCpHash$). $Tr(P_{TPM}) \models^{\forall} \phi_5$, where

$\phi_5 := \forall pd, ch_1, ch_2, t.$
$\text{CmdCPH}(\text{H}(\langle pd, \text{'CpHash'}, ch_1\rangle), ch_2)@t \Rightarrow (ch_1 = ch_2)$

Lemma 6($PolPCR$). $Tr(P_{TPM}) \models^{\forall} \phi_6$, where

$\phi_6 := \forall pd, pc, pcrv_1, pcrv_2, t.$
$\text{CmdPCRV}(\text{H}(\langle pd, \text{'PCR'}, pcrv_1\rangle), pcrv_2, pc, pc)@t$
$\Rightarrow (pcrv_1 = pcrv_2)$

Lemma 5 states that in the authorized command, the session's context value should be consistent with the policy set in the object's authPolicy. This lemma is for the CpHash assertion. As stated in Section 3.2, we use the event CmdCPH(ap, ch) to denote the check of the context value ch with no block if failed. Thus the verification of Lemma 5 means that the session's context ch_2 equals ch_1 set in ap and can be used in a check in the practical case. Lemma 6 is similar. It means that when TPM executes the authorized command, the PCR should have the desired value specified in the object's authPolicy. In this lemma, the session's context 'pcrUC' equals the PCR's current counter 'pcrUpdC' to force a match of the PCR value. This is equivalent to the case that the command P_{UseObj} only executes the event CmdPCRV after checking that 'pcrUC' and 'pcrUpdC' match. Note that the repetition of the same assertion commands can replace the previous policy settings. Thus in Lemma 5 and Lemma 6, we assume that the end part of ap is respectively the CpHash and PCR assertion.

With lemma 3 to 6, we could give out Property 2 for the TPM 2.0 EA mechanism.

Property 2($SpecTest$). $Tr(P_{TPM}) \models^{\forall} \phi_3 \wedge \phi_4 \wedge \phi_5 \wedge \phi_6$.

Property 1 and 2 are two security properties stated by the design of TPM 2.0 specifications. However, we find that the administrator is likely to misuse the immediate NV assertion in a deferred manner. Thus we propose Lemma 7 to analyze this case.

Lemma 7($CmdNV$). $Tr(P_{TPM}) \models^{\forall} \phi_7$, where

$\phi_7 := \forall pd, nv_1, nv_2, t.$
$\text{CmdNV}(\text{H}(\langle pd, \text{'NV'}, nv_1\rangle), nv_2)@t \Rightarrow (nv_1 = nv_2)$

It is similar to lemma 6. nv_1 is the desired NV value set in authPolicy. nv_2 is the NV Index value at the authorization decision moment. If lemma 7 is falsified, which means $nv_1 \neq nv_2$, then there exists an attack on the misuse case.

4. ANALYSIS RESULTS OF EA

In this section, we conduct an experiment to analyze the security properties of the EA mechanism in the TPM 2.0. Using the tool SAPIC [9], our model of the EA mechanism in Section 3.2 could be translated into the script of rewriting systems, that can be automatically analyzed by the tamarin prover [11]. The script contains 267 multiset rewriting rules and axioms for Mode 1 and 213 rules for Mode 2. We choose the tamarin prover as the analyzer since it supports an interactive mode with a GUI which allows to manually guide the tool in its proof. The analyzing results are summarized in **Table 5**. The running times on PC with Intel Core(TM)2 Quad 2.66GHz and 4GB RAM are less than 2 minutes. All the formal proof transcripts presented in this section can be found in [20]. They can be automatically verified by the tamarin-prover.

4.1 Results for Mode 1

For mode 1, the lemmas 1 to 6 have been verified. Among them, the lemmas $PolCpHash$ and $PolPCR$ are verified manually by using the interactive mode of the tamarin prover. By proving Property 1 and 2, we have achieved a formal verification of the EA mechanism in TPM 2.0 design. However, as we have mentioned in Section 3.3, the administrator of the TPM may misuse the NV assertion in a deferred manner. In this case, we have proved that the lemma $CmdNV$ is falsified. We have found a TOCTOU attack in this case.

We present a sketch for the interaction in the proof of the lemmas $PolPCR$ and $PolCpHash$. In the verification of the lemma $PolPCR$, we introduce an auxiliary formula ψ and verify the lemma $Counter$ that $Tr(P_{TPM}) \models^{\forall} \psi$ by using the inductive method. It states that the PCR counter 'pcrUpdC' is monotonically increasing, where

Lemma 8($Counter$). $Tr(P_{TPM}) \models^{\forall} \psi$, where

$\psi := \forall c_1, c_2, t_1, t_2.\text{PCRExtend}(c_1)@t_1 \wedge \text{PCRExtend}(c_2)@t_2$
$\Rightarrow ((t_1 < t_2 \wedge \exists z. c_1 + z = c_2) \vee (t_2 < t_1) \vee (t_1 = t_2)).$

With the lemma $Counter$, we can prove the lemma $PolPCR$ step by step. The tamarin prover transfers the formula $p \Rightarrow c$ to a constraint system $p \wedge \text{not(c)}$, thus we get the condition $pcrv_1 \neq pcrv_2$ and try to find a contradiction. $ap = \text{H}(\langle pd, \text{'PCR'}, pcrv_1\rangle)$ means that the event CmdPCRV should be preceded by a PCR assertion command, in which the TPM checks the 'PCRV' of pcrh as $pcrv_1$ and inserts the session's 'pcrUC' (by lock) with the value pc. This context of pcrh should be inserted by a command $P_{PCRExtend}$ or by initialization P_I. Since the current 'pcrUpdC' of pcrh has the same value pc but the current 'PCRV' of pcrh has a different value pcrv_2, this context of pcrh should be inserted by another $P_{PCRExtend}$ command. We could find a contradiction with the lemma $Counter$ by selecting these two $P_{PCRExtend}$ commands. For the lemma $CmdCPH$, it could be easily verified by selecting the two commands P_{cpHash} which respectively insert the session's 'polD' and 'cpHash' since $ch_1 \neq ch_2$.

TOCTOU attack. For the immediate NV assertion, the time of check is at the assertion command `TPM2_PolicyNV` but the time of authorization decision is at the authorized command `TPM2_UseObj`. It is possible for the adversary to cause the NV value to change between these two commands, which might lead to an attack if the administrator does not expect its happening.

We present an attack of the policy setting in [12] (Section 4.2.2). Their scheme binds the PIN value to multiple keys stored on personal device and uses the policy NV assertion to restrict the number of PIN attempts to 3. The policy for the key authorization is policy = $Assert(\text{'Secret'}, \text{PIN\_handle}) \wedge \bigvee_{n=0}^{2} \{Assert(\text{'NV'}, n+1) \wedge Assert(\text{'NV'}, n)\}$. It can be rewritten in DNF: policy = $\bigvee_{n=0}^{2} \{Assert(\text{'Secret'}, \text{PIN\_handle}) \wedge Assert(\text{'NV'}, n+1) \wedge Assert(\text{'NV'}, n)\}$. n is the current NV counter value and the initial value is 0. Since the order of verification of policy is reversed in each OR branch. To ver-

Table 5: Results of analyzing

Lemmas	Property	Mode 1		Mode 2	
		Results	Automated Run	Results	Automated Run
CorUsePol	Prop.1	verified	yes (257 steps)	verified	yes (257 steps)
CorPol1Pol2		verified	yes (615 steps)	verified	yes (549 steps)
PolAuth	Prop.2	verified	yes (4 steps)	verified	yes (4 steps)
PolSigned		verified	yes (6 steps)	verified	yes (6 steps)
PolCpHash		verified	no (200 selections)	falsified	no (36 selections)
PolPCR		verified	no (352 selections)	falsified	no (59 selections)
CmdNV	misuse	falsified	no (83 selections)	falsified	no (37 selections)

ify the policy setting, the user invokes the first NV assertion command to check its value matches n, increments the NV counter by one, and again invokes the second NV assertion command with $n+1$. Then the user attempts to enter and verify the PIN value by using the Secret assertion. In this assertion, the PIN_handle (associated with the PIN value as its authorization value) is extended into the session's policy digest only if an hmac keyed on the PIN value is inputted into the TPM by the user. At last, the user invokes the OR assertion command, which only accepts the successful verification of PIN value within 3 attempts. It is supposed to restrict the number of authorizations.

However, a malicious user can double this number by using one more policy session. Suppose that the personal device is stolen by a malicious user who does not know the PIN value. He attempts to guess the low-entropy PIN value by brute force. Suppose that the user starts 2 sessions S_1 and S_2, he can use S_1 to invoke the first NV assertion command with $n = 0$ and sequentially use S_2 still with $n = 0$. Then he increments the NV counter by one, and sequentially uses S_1 and S_2 to invoke the second NV assertion command with $n + 1 = 1$. Now he can use S_1 and S_2 to invoke the Secret assertion command to try 2 PIN candidates since the 2 sessions both have the same policy digest. Now the malicious user achieves 2 attempts but only increments the counter by 1. With $n = 1$ and $n = 2$, the user can double the number of attempts by the same way.

To avoid the misuse, the administrator should be cautious when using the immediate assertions. For the NV assertion, we suggest the current specifications change it to a deferred assertion just like the PCR assertion and add a *NVUpdateCounter* which acts a similar role as *pcrUpdateCounter*. For [12] based on current specification, we suggest the administrator uses the TPM 2.0 dictionary attack (DA) protection mechanism (Section 19.11, Part I, specification [8]) that provides protection against guessing or exhaustive searches of authorization values stored within the TPM.

4.2 Results for Mode 2

For mode 2, lemmas 1 to 4 are all verified. Lemma *CmdNV* is falsified due to the same attack in Mode 1. Lemmas *PolCpHash* and *PolPCR* are falsified due to the race conditions. The race conditions are only possible when the API commands can be executed in parallel. The two race conditions may lead to theoretical attacks in mode 2.

Race-condition-1. For the lemma *PolCpHash*, we find that the order of modifying session's context in the deferred assertion commands causes race conditions. Note that in all of these commands, the TPM modifies the session's *policyDigest* before it modifies the other context values. If the adversary can simultaneously invoke the deferred assertion command `TPM2_PolicyCpHash` and the authorized command `TPM2_UseObj` with the same session handle, then it is possible for the TPM to firstly update the session's policyDigest in the assertion command, and secondly conduct the authPolicy check in the authorized command before the assertion command modifies the session's context values. In this way, the adversary has a chance to bypass the deferred assertion checks since the context values are not modified.

Race-condition-2. For the lemma *PolPCR*, we find that the time difference in the command `TPM2_PolicyPCR` between the check of PCR value and the modification of the session's context pcrUC causes the race condition. Note that in this command, the TPM checks the PCR value before it copies the PCR's counter pcrUpdC into the session's context pcrUC for a future match. If the adversary can simultaneously invoke the commands `TPM2_PCRExtend` and `TPM_PolicyPCR`, then it is possible for the TPM to firstly check the PCR value, secondly change the PCR's value and counter in the extending command, and thirdly modify the session's context. In this way, the adversary has a chance to bypass the defence of the TOCTOU attack.

Although the probability of making a successful attack might be very small, we suggest to disable the execution mode 2. The TPM specification gives manufactures some leeway in its implementation. Our results of formal analysis show that the TPM resources should be locked when used to prevent the race conditions. It is easy to be achieved by using an I/O buffer and only allowing the execution of the next command when the last command returns a result.

5. CONCLUSION

In this paper, we have conducted a formal verification and analysis of the enhanced authorization (EA) mechanism in the TPM 2.0 API. We have proposed a model of security API analysis in process calculus with global states. We have identified and formalized the two security properties of EA mechanism in its design. Using SAPIC tool and tamarin prover, we have verified both the two security properties and discovered 3 misuse cases in the present EA mechanism.

As future work, we foresee extending our model with more TPM API commands such as those involved in attestation and platform integrity measurement. We also plan to improve and apply our formal verification results to the security analysis of more EA-based applications in Trusted Network Connection (TNC), digital rights management (DRM), Bring Your Own Device (BYOD), etc. We expect to do some improvements on our model when dealing with coun-

ters. Since at present it cannot be verified by the prover automatically.

Acknowledgments

We would like to thank all the anonymous reviewers who have reviewed our work and have provided us with valuable feedbacks. 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).

6. REFERENCES

[1] Trusted Computing Group. TPM Specification version 1.2. Parts 1–3, revision. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[2] ISO/IEC PAS DIS 11889: Information technology –Security techniques – Trusted Platform Module.

[3] D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Replay attack in TCG specification and solution. In *Proceedings of ACSAC 2005*, volume 10, pages 127–137, Tucson, AZ (USA), December 2005. ACSAC, IEEE Computer Society.

[4] L. Chen and M. Ryan. Attack, solution and verification for shared authorisation data in TCG TPM. In P. Degano and J. Guttman, editors, *Formal Aspects in Security and Trust*, volume 5983 of *Lecture Notes in Computer Science*, pages 201–216. Springer Berlin Heidelberg, 2010.

[5] L. Chen and M. Ryan. Offline dictionary attack on tcg tpm weak authorisation data, and solution. In D. Gawrock, H. Reimer, A.-R. Sadeghi, and C. Vishik, editors, *Future of Trust in Computing*, pages 193–196. Vieweg Teubner, 2009.

[6] S. Delaune, S. Kremer, M. Ryan, and G. Steel. A formal analysis of authentication in the TPM. In P. Degano, S. Etalle, and J. Guttman, editors, *Formal Aspects of Security and Trust*, volume 6561 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin Heidelberg, 2011.

[7] S. Gurgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga. Security evaluation of scenarios based on the TCG's TPM specification. In J. Biskup and J. Lĺőpez, editors, *Computer Security–ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*, pages 438–453. Springer Berlin Heidelberg, 2007.

[8] Trusted Computing Group. TPM Specification version 2.0. Parts 1–4, revision. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[9] S. Kremer and R. Künnemann. Automated analysis of security protocols with global state. IEEE Symposium on Security and Privacy, 2014. http://sapic.gforge.inria.fr/.

[10] M. Arapinis, E. Ritter, and M. Ryan. Statverif: Verification of stateful processes. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 33–47, June 2011.

[11] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 78–94, June 2012.

[12] T. Nyman, J.E. Ekberg, N. Asokan. Citizen Electronic Identities using TPM 2.0. *Trustworthy Embedded Devices 2014*. http://arxiv.org/abs/1409.1023.

[13] L. Chen and J. Li. Flexible and scalable digital signatures in tpm 2.0. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 37–48, New York, NY, USA, 2013. ACM.

[14] L. Xi. DAA-related APIs in TPM 2.0 Revisited. Cryptology ePrint Archive, 2014. https://eprint.iacr.org/2014/052.

[15] J. Shao, D. Feng, and Y. Qin. Type-based analysis of protected storage in the tpm. In S. Qing, J. Zhou, and D. Liu, editors, *Information and Communications Security*, volume 8233 of *Lecture Notes in Computer Science*, pages 135–150. Springer International Publishing, 2013.

[16] W. Wang, Y. Qin, and D. Feng. Automated proof for authorization protocols of tpm 2.0 in computational model. In X. Huang and J. Zhou, editors, *Information Security Practice and Experience*, volume 8434 of *Lecture Notes in Computer Science*, pages 144–158. Springer International Publishing, 2014.

[17] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 221–236, Washington, DC, USA, 2009. IEEE Computer Society.

[18] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel. Formal analysis of protocols based on tpm state registers. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 66–80, Washington, DC, USA, 2011. IEEE Computer Society.

[19] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, pages 82–, Washington, DC, USA, 2001. IEEE Computer Society.

[20] TPM2-EA-formal-verification. https://github.com/sjxzmc/TPM2-EA-formal-verification/