# A Formal Analysis of Authentication in the TPM

Stéphanie Delaune[1], Steve Kremer[1], Mark D. Ryan[2], and Graham Steel[1]

[1] LSV, ENS Cachan & CNRS & INRIA Saclay Île-de-France, France
[2] School of Computer Science, University of Birmingham, UK

**Abstract.** The Trusted Platform Module (TPM) is a hardware chip designed to enable computers to achieve a greater level of security than is possible in software alone. To this end, the TPM provides a way to store cryptographic keys and other sensitive data in its shielded memory. Through its API, one can use those keys to achieve some security goals. The TPM is a complex security component, whose specification consists of more than 700 pages.
We model a collection of four TPM commands, and we identify and formalise their security properties. Using the tool ProVerif, we rediscover some known attacks and some new variations on them. We propose modifications to the API and verify our properties for the modified API.

## 1 Introduction

The Trusted Platform Module (TPM) is a hardware chip designed to enable commodity computers to achieve greater levels of security than is possible in software alone. There are 300 million TPMs currently in existence, mostly in high-end laptops, but now increasingly in desktops and servers. Application software such as Microsoft's BitLocker and HP's HP ProtectTools use the TPM in order to guarantee security properties. The TPM specification is an industry standard [16] and an ISO/IEC standard [13] coordinated by the Trusted Computing Group.

In the last few years, several vulnerabilities in the TPM API have been discovered, particularly in relation to secrecy and authentication properties (we detail a few of them in Section 5). These attacks highlight the necessity of formal analysis of the API specification. We perform such an analysis in this paper, focussing on the mechanisms for authentication and authorisation.

Formal analysis of security APIs involves many of the tools and techniques used in the analysis of security protocols, but the long-term state information held by the TPM presents an additional challenge. Tools such as ProVerif are not optimised to reason with stateful protocols. There is no easy solution to this problem, although we show how to alleviate it in this case with suitable abstractions.

In this paper, we model a collection of four TPM commands, concentrating on the authentication mechanisms. We identify security properties which we will argue are central to correct and coherent design of the API. We formalise these properties for our fragment, and using ProVerif, we rediscover some known attacks on the API and some new variations on them. We discuss some fixes to the API, and prove our security properties for the modified API.

## 2 An overview of the TPM

The TPM stores cryptographic keys and other sensitive data in its shielded memory, and provides ways for platform software to use those keys to achieve security goals. The TPM offers three kinds of functionality:

- *Secure storage.* User processes can store content that is encrypted by keys only available to the TPM.
- *Platform authentication.* A platform can obtain keys by which it can authenticate itself reliably.
- *Platform measurement and reporting.* A platform can create reports of its integrity and configuration state that can be relied on by a remote verifier.

To store data using a TPM, one creates TPM keys and uses them to encrypt the data. TPM keys are arranged in a tree structure, with the *storage root key* (SRK) at its root. To each TPM key is associated a 160-bit string called *authdata*, which is analogous to a password that authorises use of the key. A user can use a key loaded in the TPM through the interface provided by the device. This interface is actually a collection of commands that (for example) allow one to load a new key inside the device, or to certify a key by another one. All the commands have as an argument an authorisation HMAC that requires the user to provide a proof of knowledge of the relevant authdata. Each command has to be called inside an *authorisation session*. We first describe this mechanism before we explain some commands in more detail.

### 2.1 Sessions

The TPM provides two kinds of authorisation sessions:

- *Object Independent Authorisation Protocol* (OIAP), which creates a session that can manipulate any object, but works only for certain commands.
- *Object Specific Authorisation Protocol* (OSAP), which creates a session that manipulates a specific object specified when the session is set up.

An authorisation session begins when the command TPM_OIAP or TPM_OSAP is successfully executed.

*OIAP authorisation session.* To set up an OIAP session, the user process sends the command TPM_OIAP to the TPM. Then, the user receives back a *session handle*, together with a nonce. Each command within the session sends the session handle as part of its arguments, and also a new nonce. Nonces from the user process are called *odd nonces*, and nonces from the TPM are called *even nonces*. This system of rotating nonces guarantees freshness of the commands and responses. All authorisation HMACs include the most recent odd and even nonce. In an OIAP session, the authorisation HMACs required to execute a command are keyed on the authdata for the resource (*e.g.* key) requiring authorisation.

*OSAP authorisation session.* For an OSAP session, the user process sends the command TPM_OSAP to the TPM, together with the name of the object (*e.g.* key handle), and an *OSAP odd nonce* $No^{\mathsf{OSAP}}$. The response includes a session handle, an even nonce for the rolling nonces, and an *OSAP even nonce* $Ne^{\mathsf{OSAP}}$. Then, the user process and the TPM both compute the *OSAP shared secret* $\mathsf{hmac}(auth, \langle Ne^{\mathsf{OSAP}}, No^{\mathsf{OSAP}}\rangle)$, *i.e.* an HMAC of the odd OSAP nonce and the even OSAP nonce, keyed on the object's authdata. Now commands within such a session may be executed. In an OSAP session, the authorisation HMACs are keyed on the OSAP shared secret. The purpose of this arrangement is to permit the user process to cache the session key for a possibly extended session duration, without compromising the security of the authdata on which it is based.

### 2.2 Commands

The TPM provides a collection of commands that allow one to create some new keys and to manipulate them. These commands are described in [16]. Here, we explain only a few of them in order to illustrate how the TPM works.

To store data using a TPM, one creates a TPM key and uses it to encrypt the data. TPM keys are arranged in a tree structure. The *Storage Root Key* (SRK) is created by a command called TPM_TakeOwnership. At any time afterwards, a user process can call TPM_CreateWrapKey to create a child key of an existing key. Once a key has been created, it may be loaded using TPM_LoadKey2, and then can be used in an operation requiring a key (*e.g.* TPM_CertifyKey, TPM_UnBind).

TPM_CreateWrapKey. The command creates the key but does not store it; it simply returns it to the user process (protected by an encryption). Assume that the parent key pair $(sk, \mathsf{pk}(sk))$ under which we want to create a new key is already loaded inside the device with the authdata $auth$. Assume also that an OSAP session has been established on this key, with $ss = \mathsf{hmac}(auth, \langle ne^{\mathsf{OSAP}}, no^{\mathsf{OSAP}}\rangle)$ as the OSAP shared secret and assume the current even rolling nonce associated to this session is $ne$. The command can be informally described as follows:

$$\begin{array}{rcl} \mathsf{handle}(auth, sk), no, \mathrm{cipher}, & \multirow{2}{*}{$\rightarrow$} & \mathsf{new}\ SK, \mathsf{new}\ Ne \cdot Ne, \mathsf{pk}(SK), \mathrm{wrp}, \\ \mathsf{hmac}(ss, \langle \mathsf{cwk}, \mathrm{cipher}, ne, no\rangle) & & \mathsf{hmac}(ss, \langle \mathsf{cwk}, \mathrm{wrp}, \mathsf{pk}(SK), Ne, no\rangle) \end{array}$$

where:

- $\mathrm{cipher} = \mathsf{senc}(NewAuth, \mathsf{hash}(ss, ne))$, and
- $\mathrm{wrp} = \mathsf{wrap}(\mathsf{pk}(sk), \langle SK, NewAuth, \mathsf{tpmproof}\rangle)$.

The intuitive meaning of such a rule is: if an agent (possibly the attacker) has the data items on the left, then, by means of the command, he can obtain the data items on the right. The new keyword indicates that data, *e.g.* nonces or keys, is freshly generated.

The TPM_CreateWrapKey command takes arguments that include the handle for the parent key of the key to be created, an odd rolling nonce that is *a priori* freshly generated by the user, the new encrypted authdata of the key to be created, and the authorisation HMAC based on the authdata of the parent key. It returns the public part

$\mathsf{pk}(SK)$ of the new key and an encrypted package; the package contains the private part and the authdata of the new key, as well as the constant tpmproof (a private constant only known by the TPM). This package is encrypted with the parent key $\mathsf{pk}(sk)$. An authentication HMAC is also constructed by the TPM to accompany the response. The newly created key is not yet available to the TPM for use.

Note that this command introduced a new authdata that is encrypted with the OSAP secret. Because this arrangement could expose the OSAP shared secret to cryptanalytic attacks if used multiple times, an OSAP session that is used to introduce new authdata is subsequently terminated by the TPM. Commands that want to continue to manipulate the object have to create a new session.

TPM_LoadKey2. To use a TPM key, it must be loaded. TPM_LoadKey2 takes as argument a wrap created by the command TPM_CreateWrapKey, and returns a handle, that is, a pointer to the key stored in the TPM memory. Commands that use the loaded key refer to it by this handle. Since TPM_LoadKey2 involves a decryption by the parent key, it requires the parent key to be loaded and it requires an authorisation HMAC based on the parent key authdata. The root key SRK is permanently loaded and has a well-known handle value, and therefore never needs to be loaded.

Once loaded, a key can be used, for example to encrypt or decrypt data, or to sign data. As an illustrative example, we describe the command TPM_CertifyKey in more detail.

TPM_CertifyKey. This command requires two key handle arguments, representing the certifying key and the key to be certified, and two corresponding authorisation HMACs. It returns the certificate.

Assume that two OIAP sessions are already established with their current even rolling nonce $ne_1$ and $ne_2$ respectively, and that two keys $\mathsf{handle}(auth_1, sk_1)$ and $\mathsf{handle}(auth_2, sk_2)$ are already loaded inside the TPM. The TPM_CertifyKey command can be informally described as follows:

$$\begin{array}{rcl} n, no_1, no_2, & & \mathsf{new}\ Ne_1, \mathsf{new}\ Ne_2 \cdot \\ \mathsf{hmac}(auth_1, \langle \mathsf{cfk}, n, ne_1, no_1 \rangle) & & Ne_1, Ne_2, \mathrm{certif} \\ \mathsf{hmac}(auth_2, \langle \mathsf{cfk}, n, ne_2, no_2 \rangle) & \rightarrow & \mathsf{hmac}(auth_1, \langle \mathsf{cfk}, n, Ne_1, no_1, \mathrm{certif} \rangle) \\ \mathsf{handle}(auth_1, sk_1), \mathsf{handle}(auth_2, sk_2) & & \mathsf{hmac}(auth_2, \langle \mathsf{cfk}, n, Ne_2, no_2, \mathrm{certif} \rangle) \end{array}$$

where $\mathrm{certif} = \mathsf{cert}(sk_1, \mathsf{pk}(sk_2))$.

The user requests to certify the key $\mathsf{pk}(sk_2)$ with the key $sk_1$. For this, he generates a nonce $n$ and two odd rolling nonces $no_1$ and $no_2$ that he gives to the TPM together with the two authorisation HMACs. The TPM returns the certificate $\mathrm{certif}$. Two authentication HMACs are also constructed by the TPM to accompany the response. The TPM also generates two new even rolling nonces to continue the session.

## 3 Modelling the TPM

In this section, we first give a short introduction to the tool ProVerif that we used to perform our analysis. We chose ProVerif after first experimenting with the AVISPA toolsuite [3], which provides support for mutable global state. However, of the AVISPA backends that support state, OFMC and CL-AtSe require concrete bounds on the number of command invocations and fresh nonces to be given. It is possible to avoid this restriction using SATMC [10], but SATMC performed poorly in our experiments due to the relatively large message length, a known weakness of SATMC. We therefore opted for ProVerif using abstractions to model state, as we explain below. We will explain how the TPM commands as well as its security properties can be formalized in our framework.

### 3.1 The tool ProVerif

ProVerif takes as input a process written in a syntax close to the one used in the applied pi calculus [1], a modelling language for security protocols. In particular, it allows processes to send first-order terms built over a signature, names and variables. These terms model the messages that are exchanged during a protocol.

*Example 1.* Consider for example the signature $\Sigma = \{\mathsf{senc}, \mathsf{sdec}, \mathsf{hmac}\}$ which contains three binary function symbols. The signature is equipped with an equational theory and we interpret equality up to this theory. For instance the theory $\mathsf{sdec}(\mathsf{senc}(x, y), y) = x$ models that decryption and encryption cancel out whenever the same key is used. We do not need to consider any equations for modelling an HMAC function.

Processes $P, Q, R, \ldots$ are constructed as follows. The process $0$ is the empty process which does nothing. $\mathsf{new}\ a.P$ restricts the name $a$ in $P$ and can for instance be used to model that $a$ is a fresh random number. $\mathsf{in}(c, x).P$ models the input of a term on a channel $c$, which is then substituted for $x$ in process $P$. $\mathsf{out}(c, t)$ outputs a term $t$ on a channel $c$. $P \mid Q$ models the parallel composition of processes $P$ and $Q$. In particular, an input and an output on the same channel in parallel can synchronize and perform a communication. The conditional $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$ behaves as $P$ when $M$ and $N$ are equal modulo the equational theory and behaves as $Q$ otherwise. $!P$ is the replication of $P$, modelling an unbounded number of copies of the process $P$. Moreover we annotate processes with events $\mathsf{ev}(t_1, \ldots, t_n)$ which are useful for modelling correspondence properties, discussed below. ProVerif can automatically check security properties assuming that an arbitrary adversary process is run in parallel.

*Example 2.* Consider the process $\mathsf{new}\ k.(!P \mid !Q)$ where

$$
\begin{array}{ll}
P \mathrel{\hat{=}} \mathsf{new}\ n_P.\mathsf{begin1}(n_P). & Q \mathrel{\hat{=}} \mathsf{in}(c, y); \\
\quad \mathsf{out}(c, \mathsf{senc}(n_P, k)).\mathsf{in}(c, x). & \quad \mathsf{if}\ y = \mathsf{senc}(\mathsf{sdec}(y, k), k)\ \mathsf{then} \\
\quad \mathsf{let}\ (x_P, x_Q) = \mathsf{sdec}(x, k)\ \mathsf{in} & \quad \mathsf{let}\ y_P = \mathsf{sdec}(y, k)\ \mathsf{in}\ \mathsf{end1}(y_P).\mathsf{new}\ n_Q. \\
\quad \mathsf{if}\ x_P = n_P\ \mathsf{then}\ \mathsf{end2}(n_P, x_Q) & \quad \mathsf{begin2}(y_P, n_Q); \mathsf{out}(c, \mathsf{senc}((y_P, n_Q), k))
\end{array}
$$

The processes $P$ and $Q$ share a long term symmetric key $k$ which they use to perform a handshake protocol. Note that we use some syntactic sugar of the ProVerif syntax for readability: final $0$ and $\mathsf{else}\ 0$ are omitted, we use a $\mathsf{let}$ construct to introduce local variables and use a variadic tuple operator $(t_1, \ldots, t_n)$ for concatenating terms. For the moment we ignore the $\mathsf{begin}$ and $\mathsf{end}$ events. The process $P$ first generates a nonce $n_P$

which is encrypted with the key $k$. This ciphertext is sent to $Q$ over a channel $c$. In $Q$, the test $y = \mathsf{senc}(\mathsf{sdec}(y, k), k)$ is used to check whether decryption succeeds. If it does, then the process $Q$ generates a fresh nonce $n_Q$ and sends the encryption of both nonces back to $P$. Lastly, the process $P$ checks that the received nonce matches the previously generated nonce $n_P$.

We now discuss correspondence properties, which can be automatically verified by ProVerif [4]. A correspondence property $\mathsf{ev2}(x_1, \ldots, x_n) \Rightarrow \mathsf{ev1}(y_1, \ldots, y_n)$ holds if on every execution trace each occurrence of $\mathsf{ev2}(x_1, \ldots, x_n)\sigma$ is preceded by an occurrence of $\mathsf{ev1}(y_1, \ldots, y_n)\sigma$ where $\sigma$ is a substitution mapping the $x_i$s and $y_i$s to terms. An *injective* correspondence property holds if each occurrence of $\mathsf{ev2}(x_1, \ldots, x_n)\sigma$ is preceded by a different occurrence of $\mathsf{ev1}(y_1, \ldots, y_n)\sigma$. Intuitively, injective correspondence avoids replay attacks.

*Example 3.* Coming back to Example 2, the property $\mathsf{end1}(x) \Rightarrow \mathsf{begin1}(x)$ models that whenever the process $Q$ receives a correctly encrypted message with key $k$ ($\mathsf{end1}(x)$ occurs) it must have originated from $P$ ($\mathsf{begin1}(x)$ occurs). Moreover, the processes $P$ and $Q$ agree on the value of $x$. This correspondence property indeed holds. However, the stronger injective version does not hold, as the first message of $P$ can be replayed to $Q$. The second correspondence property $\mathsf{end2}(x_1, x_2) \Rightarrow \mathsf{begin1}(x_1, x_2)$ models that whenever the process $P$ finishes a session successfully, it must have interacted with the process $Q$. This property holds injectively.

### 3.2 Modelling commands of the TPM

One of the difficulties in reasoning about security APIs such as that of the TPM is *non-monotonic state*. If the TPM is in a certain state $s$, and then a command is successfully executed, then typically the TPM ends up in a state $s' \neq s$. Commands that require it to be in the previous state $s$ will no longer work. Some of the over-approximations made by tools such as ProVerif do not work well with non-monotonic state. For example, although private channels could be used to represent the state changes, the abstraction of private channels that ProVerif makes prevents it from being able to verify correctness of the resulting specification. Moreover, ProVerif does not model a state transition system, but rather a set of derivable facts representing attacker knowledge, together with the assumption that the attacker never forgets any fact.

We address these restrictions by introducing the assumption that only one command is executed in each OIAP or OSAP session. This assumption appears to be quite reasonable. Indeed, the TPM imposes the assumption itself whenever a command introduces new authdata. Moreover, tools like TPM/J [15] that provide software-level APIs also implement this assumption. Again to avoid non-monotonicity, we do not allow keys to be deleted from the memory of the TPM; instead, we allow an unbounded number of keys to be loaded.

An important aspect of the TPM is its key table that allows one to store cryptographic keys and other sensitive data in its shielded memory. Our aim is to allow the key table to contain dishonest keys, *i.e.* keys for which the attacker knows the authdata, as well as honest keys. Some of these keys may also share the same authdata. Indeed,

$$
\begin{array}{l}
\mathsf{User\_CertifyKey} \mathrel{\hat{=}} \\
\quad \mathsf{in}(c, h_1).\mathsf{in}(c, h_2).\mathsf{in}(c, ne_1).\mathsf{in}(c, ne_2). \\
\quad \mathsf{new}\ N.\mathsf{new}\ No_1.\mathsf{new}\ No_2. \\
\quad \mathsf{let}\ (auth_1, pk_1) = (\mathsf{getAuth}(h_1), \mathsf{getPK}(h_1))\ \mathsf{in} \\
\quad \mathsf{let}\ (auth_2, pk_2) = (\mathsf{getAuth}(h_2), \mathsf{getPK}(h_2))\ \mathsf{in} \\
\quad \mathsf{let}\ hmac_1 = \mathsf{hmac}(auth_1, (\mathsf{cfk}, N, ne_1, No_1))\ \mathsf{in} \\
\quad \mathsf{let}\ hmac_2 = \mathsf{hmac}(auth_2, (\mathsf{cfk}, N, ne_2, No_2))\ \mathsf{in} \\
\quad \mathsf{UserRequestsC}(auth_1, pk_1, auth_2, pk_2). \\
\quad \mathsf{out}(c, (N, No_1, hmac_1, No_2, hmac_2)). \\
\quad \mathsf{in}(c, (xcert, ne'_1, ne'_2, hm_1, hm_2)). \\
\quad \mathsf{if}\ hm_1 = \mathsf{hmac}(auth_1, (\mathsf{cfk}, N, ne'_1, No_1, xcert))\ \mathsf{then} \\
\quad \mathsf{if}\ hm_2 = \mathsf{hmac}(auth_2, (\mathsf{cfk}, N, ne'_2, No_2, xcert))\ \mathsf{then} \\
\quad \mathsf{UserConsidersC}(auth_1, pk_1, auth_2, pk_2, xcert).
\end{array}
$$

**Fig. 1.** Process User_CertifyKey

it would be incorrect to suppose that all keys have distinct authdata, as the authdata may be derived from user chosen passwords. Our first idea was to use a binary function symbol $\mathsf{handle}(auth, sk)$ to model a handle to the secret key $sk$ with authdata $auth$. We use private functions, *i.e.* functions which may not be applied by the attacker, to allow the TPM process to extract the authdata and the secret key from a handle. This models a lookup in the key table where each handle can indeed be associated to its authdata and private key. Unfortunately, with this encoding ProVerif does not succeed in proving some expected properties. The tool outputs a false attack based on the hypothesis that the attacker knows two handles $\mathsf{handle}(auth_1, sk)$ and $\mathsf{handle}(auth_2, sk)$ which are built over two distinct authdata but the same secret key (which is impossible). We therefore use a slightly more involved encoding where the handle depends on the authdata and a *seed*; the secret key is now obtained by applying a binary private function symbol (denoted hsk hereafter) to both the authdata and the seed. Hence, $\mathsf{handle}(auth_1, s)$ and $\mathsf{handle}(auth_2, s)$ will now point to two different private keys, namely $\mathsf{hsk}(auth_1, s)$ and $\mathsf{hsk}(auth_2, s)$. This modelling avoids false attacks.

In our modelling we have two processes for each command: a user process and a TPM process. The user process (*e.g.* User_CertifyKey) models an honest user who makes a call to the TPM while the TPM process (*e.g.* TPM_CertifyKey) models the TPM itself. The user process first takes parameters, such as the key handles used for the given command, and can be invoked by the adversary. This allows the adversary to schedule honest user actions in an arbitrary way without knowing himself the authdata corresponding to the keys used in these commands.

Our model assumes that the attacker can intercept, inject and modify commands sent by applications to the TPM, and the responses sent by the TPM. While this might not be the case in all situations, it seems to be what the TPM designers had in mind; otherwise, neither the authentication HMACs keyed on existing authdata, nor the encryption of new authdata described in section 2.2 would be necessary.

$$
\begin{array}{l}
\mathsf{TPM\_CertifyKey} \mathrel{\hat{=}} \\
\quad \mathsf{new}\ Ne_1.\mathsf{new}\ Ne_2.\mathsf{out}(c, Ne_1).\mathsf{out}(c, Ne_2). \\
\quad \mathsf{in}(c, n).\mathsf{in}(c, (h_1, no_1, hm_1)).\mathsf{in}(c, (h_2, no_2, hm_2)). \\
\quad \mathsf{let}\ (auth_1, sk_1, pk_1) = (\mathsf{getAuth}(h_1), \mathsf{getSK}(h_1), \mathsf{getPK}(h_1))\ \mathsf{in} \\
\quad \mathsf{let}\ (auth_2, sk_2, pk_2) = (\mathsf{getAuth}(h_2), \mathsf{getSK}(h_2), \mathsf{getPK}(h_2))\ \mathsf{in} \\
\quad \mathsf{if}\ hm_1 = \mathsf{hmac}(auth_1, (\mathsf{cfk}, n, Ne_1, no_1))\ \mathsf{then} \\
\quad \mathsf{if}\ hm_2 = \mathsf{hmac}(auth_2, (\mathsf{cfk}, n, Ne_2, no_2))\ \mathsf{then} \\
\quad \mathsf{let}\ certif = \mathsf{cert}(sk_1, pk_2)\mathsf{in} \\
\quad \mathsf{out}(c, certif). \\
\quad \mathsf{TpmC}(auth_1, pk_1, auth_2, pk_2, certif). \\
\quad \mathsf{new}\ Ne'_1.\mathsf{new}\ Ne'_2. \\
\quad \mathsf{let}\ hmac_1 = \mathsf{hmac}(auth_1, (\mathsf{cfk}, n, Ne'_1, no_1, certif))\ \mathsf{in} \\
\quad \mathsf{let}\ hmac_2 = \mathsf{hmac}(auth_2, (\mathsf{cfk}, n, Ne'_2, no_2, certif))\ \mathsf{in} \\
\quad \mathsf{out}(c, (Ne'_1, Ne'_2, hmac_1, hmac_2)).
\end{array}
$$

**Fig. 2.** Process TPM_CertifyKey

We now illustrate our modelling on the TPM_CertifyKey command in an OIAP session. The process User_CertifyKey is detailed in Figure 1. This process starts by inputing two handles $h_1, h_2$ which are provided by the attacker. In this way the attacker can schedule with which keys this command is executed. The user also inputs two even nonces which are supposed to come from the TPM. Then the user constructs the two authorisation HMACs for the corresponding nonces using the authdata and public keys extracted out of the handles and outputs these HMACs together with the nonces. The event UserRequestsC is used to declare that the user requested the command with the given parameters. When receiving the reply the user checks the received HMACs. If all checks go through the user triggers the event UserConsidersC. The process TPM_CertifyKey is detailed in Figure 2 and does the complementary actions to the user process. We may note that when the attacker knows the authdata corresponding to a handle he can directly interact with this process without the user process.

### 3.3 Security Properties of the TPM

The TPM specification does not detail explicitly which security properties are intended to be guaranteed, although it provides some hints. The specification [16, Part I, p.60] states that: "*The design criterion of the protocols is to allow for ownership authentication, command and parameter authentication and prevent replay and man in the middle attacks.*" We will formalise these security properties as *correspondence properties*:

1. If the TPM has executed a certain command, then a user in possession of the relevant authdata has previously requested the command.
2. If a user considers that the TPM has executed a certain command, then either the TPM really has executed the command, or an attacker is in possession of the relevant authdata.

The first property expresses authentication of user commands, and is achieved by the authorisation HMACs that accompany the commands. The second one expresses authentication of the TPM, and is achieved by the HMACs provided by the TPM with its answer. We argue that the TPM certainly aims at achieving these properties, as otherwise there would be no need for the HMAC mechanism. Going back to the example of the TPM_CertifyKey command (Figures 1 and 2) the above mentioned properties can be expressed by the injective correspondence properties:

1. $\mathsf{TpmC}(x_1, x_2, x_3, x_4, x_5) \Rightarrow \mathsf{UserRequestsC}(x_1, x_2, x_3, x_4)$, and
2. $\mathsf{UserConsidersC}(x_1, x_2, x_3, x_4, x_5) \Rightarrow \mathsf{TpmC}(x_1, x_2, x_3, x_4, x_5)$.

These properties, however, cannot hold if we provide the attacker with a dishonest key, *i.e.* a handle for which he knows the corresponding authdata. Indeed, the attacker can simply execute the command without the user process. Hence, if we provide the attacker with a handle $\mathsf{handle}(auth_i, seed_i)$ and the authdata $auth_i$, we weaken the property to avoid the trivial failure of the property. We cannot expect the property to hold when $x_1$ and $x_3$ are both instantiated with $auth_i$. However, as soon as we give the attacker the possibility to create new keys (using the TPM_CreateWrapKey command), the attacker can create new keys and again make the property trivially fail. Hence, when we consider a scenario in which new keys can be loaded, we consider the following formalization:

1. $\mathsf{TpmC}(x_1, x_2, x_3, x_4, x_5) \Rightarrow \mathsf{UserRequestsC}(x_1, x_2, x_3, x_4, x_5) \vee (\mathsf{I}(x_1) \wedge \mathsf{I}(x_3))$
2. $\mathsf{UserConsidersC}(x_1, x_2, x_3, x_4, x_5) \Rightarrow \mathsf{TpmC}(x_1, x_2, x_3, x_4, x_5) \vee (\mathsf{I}(x_1) \wedge \mathsf{I}(x_3))$.

where I is the attacker knowledge predicate. Hence, we allow the property to fail if the commands are executed with keys for which the attacker knows the authdata.

## 4 Analysing the TPM with ProVerif

All the files for our experiments described below are available on line at:

`http://www.lsv.ens-cachan.fr/~delaune/TPM/`

In the figures describing attacks, for the sake of clarity, we sometimes omit some parts of the messages, especially session handles that we do not consider in our model and key handles when they are clear from the other messages.

Our methodology was to first study some core key management commands in isolation to analyse the weakness of each command. This leads us to propose some fixes for these commands. Then, we carried out an experiment where we consider the commands TPM_CertifyKey, TPM_CreateWrapKey, TPM_LoadKey2, and TPM_UnBind together. We consider the fixed version of each of these commands and we show in Experiment 10 that the security properties are satisfied for a scenario that allows:

- an attacker to load his own keys inside the TPM, and
- an honest user to use the same authdata for different keys.

In our first six experiments, we model the command TPM_CertifyKey in isolation. Then, in Experiments 7-9, we model the command TPM_CreateWrapKey only. Lastly, in Experiment 10, we consider a model where the commands TPM_CertifyKey, TPM_CreateWrapKey, TPM_LoadKey2, and TPM_UnBind are taken into account. In all experiments, the security properties under test are the correspondence properties explained above.

**Experiment 1.** In our first two experiments, we consider a configuration with two keys loaded inside the TPM. The attacker knows the two handles $\mathsf{handle}(auth_1, sk_1)$ and $\mathsf{handle}(auth_2, sk_2)$. From the handle $\mathsf{handle}(auth_1, sk_1)$, the attacker can obtain the corresponding public key $\mathsf{pk}(sk_1)$. However, he can obtain neither the private key $sk_1$, nor the authdata $auth_1$ required to manipulate the key through the device. For the moment, we assume that the attacker does not have his own key loaded onto the device.

ProVerif immediately discovers an attack, described in Figure 3, that comes from the fact that the command involved two keys. The attacker can easily swap the role of these two keys: he swaps the two HMACs, the two key handles, and the rolling nonces provided in input of the command. Hence, the TPM will output the certificate $\mathsf{cert}(sk_2, \mathsf{pk}(sk_1))$ whereas the user asked for obtaining the certificate $\mathsf{cert}(sk_1, \mathsf{pk}(sk_2))$.

By performing also the swap on the answer provided by the TPM, the attacker can provide two valid HMACs to the user who will accept the wrong certificate. Hence, the second correspondence property is not satisfied. Note that if the user chooses to verify the certificate he received with the checkcert algorithm, then this attack is not valid anymore and ProVerif is able to verify that this second correspondence property holds.

**Initial knowledge of Charlie:** $\mathsf{handle}(auth_1, sk_1)$, $\mathsf{handle}(auth_2, sk_2)$.

**Trace:** Charlie swaps the two authorisation HMACs, and swaps the two response HMACs.

USER $\rightarrow$ TPM : request to open two OIAP sessions
TPM $\rightarrow$ USER : $ne_1, ne_2$

USER requests key certification to obtain $\mathsf{cert}(sk_1, \mathsf{pk}(sk_2))$

USER $\rightarrow$ Charlie : $n, no_1, no_2,$
$\mathsf{hmac}(auth_1, \langle \mathsf{cfk}, n, ne_1, no_1\rangle)$,$\mathsf{hmac}(auth_2, \langle \mathsf{cfk}, n, ne_2, no_2\rangle)$
Charlie $\rightarrow$ TPM : $n, no_2, no_1,$
$\mathsf{hmac}(auth_2, \langle \mathsf{cfk}, n, ne_2, no_2\rangle)$,$\mathsf{hmac}(auth_1, \langle \mathsf{cfk}, n, ne_1, no_1\rangle)$

TPM $\rightarrow$ Charlie : $ne'_1, ne'_2, \mathsf{cert}(sk_2, \mathsf{pk}(sk_1)),$
$\mathsf{hmac}(auth_2, \langle \mathsf{cfk}, n, ne'_1, no_2\rangle)$, $\mathsf{hmac}(auth_1, \langle \mathsf{cfk}, n, ne'_2, no_1\rangle)$
Charlie $\rightarrow$ USER : $ne'_2, ne'_1, \mathsf{cert}(sk_2, \mathsf{pk}(sk_1)),$
$\mathsf{hmac}(auth_1, \langle \mathsf{cfk}, n, ne'_2, no_1\rangle)$, $\mathsf{hmac}(auth_2, \langle \mathsf{cfk}, n, ne'_1, no_2\rangle)$

USER checks the HMACs and accepts the certificate $\mathsf{cert}(sk_2, \mathsf{pk}(sk_1))$.

**Fig. 3.** Attack trace for Experiment 1.

**Experiment 2.** We patch the command TPM_CertifyKey by considering two different tags for the two different HMACs. More precisely, we replace the constant cfk with $\mathsf{cfk}_1$ (resp. $\mathsf{cfk}_2$) in the first (resp. second) HMAC provided by the user and also the one provided by the TPM. The attacks reported in our first experiment are prevented. ProVerif is now able to verify the two correspondence properties.

**Experiment 3.** We add in the initial configuration another key for Alice and we assume that this new key $sk'_2$ has the same authdata as a previous key of Alice already loaded onto the TPM. Hence, in our model, we have that $\mathsf{handle}(auth_1, sk_1)$, $\mathsf{handle}(auth_2, sk_2)$, and $\mathsf{handle}(auth_2, sk'_2)$ are terms known by the attacker Charlie.

ProVerif immediately discovers another attack, described in Figure 4. The attacker can exchange the key handle $\mathsf{handle}(auth_2, sk_2)$ provided by the honest user in entry of the command with another handle having the same authdata, i.e. $\mathsf{handle}(auth_2, sk'_2)$. The TPM will answer by sending the certificate $\mathsf{cert}(sk_1, \mathsf{pk}(sk'_2))$ together with the two HMACs. After verifying the HMACs, the user will accept this certificate which is not the right one. Indeed, the user was expecting to receive $\mathsf{cert}(sk_1, \mathsf{pk}(sk_2))$. The trace described in Figure 4 shows that none of the two correspondence properties holds.

**Initial knowledge of Charlie:** $\mathsf{handle}(auth_1, sk_1)$, $\mathsf{handle}(auth_2, sk_2)$, $\mathsf{handle}(auth_2, sk'_2)$.

**Trace:** Charlie swaps a key handle for another one that has the same authdata.

USER $\rightarrow$ TPM : request to open two OIAP sessions
TPM $\rightarrow$ USER : $ne_1, ne_2$

USER requests key certification to obtain $\mathsf{cert}(sk_1, \mathsf{pk}(sk_2))$

USER $\rightarrow$ Charlie : $n, no_1, \mathsf{handle}(auth_1, sk_1), \mathsf{hmac}(auth_1, \langle \mathsf{cfk}, n, ne_1, no_1\rangle),$
$no_2, \mathsf{handle}(auth_2, sk_2), \mathsf{hmac}(auth_2, \langle \mathsf{cfk}, n, ne_2, no_2\rangle)$
Charlie $\rightarrow$ TPM : $n, no_1, \mathsf{handle}(auth_1, sk_1), \mathsf{hmac}(auth_1, \langle \mathsf{cfk}, n, ne_1, no_1\rangle),$
$no_2, \mathsf{handle}(auth_2, sk'_2), \mathsf{hmac}(auth_2, \langle \mathsf{cfk}, n, ne_2, no_2\rangle)$

TPM $\rightarrow$ USER : $ne'_1, ne'_2, \mathsf{cert}(sk_1, \mathsf{pk}(sk'_2)),$
$\mathsf{hmac}(auth_1, \langle \mathsf{cfk}, n, ne'_1, no_1\rangle), \mathsf{hmac}(auth_2, \langle \mathsf{cfk}, n, ne'_2, no_2\rangle)$

USER checks the HMACs and accepts the certificate $\mathsf{cert}(sk_1, \mathsf{pk}(sk'_2))$.

**Fig. 4.** Attack trace for Experiment 3.

**Experiment 4.** The attack described in the previous experiment comes from the fact that the HMAC is only linked to the key via the authdata. Thus, as soon as two keys share the same authdata, this leads to some confusion. A way to fix this would be to add the key handle inside the HMAC, but the TPM designers chose not to do this because they wanted to allow middleware to unload and reload keys (and therefore possibly change key handles) without the knowledge of application software that produces the HMACs. A more satisfactory solution that has been proposed for future versions of the TPM is to add (the digest of) the public key inside the HMAC. Hence, for instance, the HMAC built by the user is now of the form $\mathsf{hmac}(auth, \langle \mathsf{cfk1}, \mathsf{pk}(sk), n, ne_1, no_1\rangle)$. The same transformation is done on all the HMACs.

The previous attacks do not exist anymore. ProVerif is able to verify that the two correspondence properties hold.

**Experiment 5.** We now assume that the attacker has his own key loaded onto the device. This means that he knows a key handle $\mathsf{handle}(auth_i, sk_i)$ and the authdata $auth_i$ that allows him to manipulate $sk_i$ through the device. He has also access to the public key $\mathsf{pk}(sk_i)$. However, he does not know $sk_i$ that is stored inside the TPM.

We immediately rediscover the attack of [12], showing that the attacker can manipulate the messages exchanged between the USER and the TPM in such a way that the TPM will provide the certificate $\mathsf{cert}(sk_1, \mathsf{pk}(sk_i))$ to a user that has requested the certificate $\mathsf{cert}(sk_1, \mathsf{pk}(sk_2))$.

**Initial knowledge of Charlie:** $\mathsf{handle}(auth_1, sk_1)$, $\mathsf{handle}(auth_2, sk_2)$, $\mathsf{handle}(auth_i, sk_i')$, $auth_i$.

**Trace:** Charlie replaces they key to be certified his own key.

USER $\rightarrow$ TPM : request to open two OIAP sessions
TPM $\rightarrow$ USER : $ne_1, ne_2$

USER requests key certification to obtain $\mathsf{cert}(sk_1, \mathsf{pk}(sk_2))$

USER $\rightarrow$ Charlie : $n, no_1, \mathsf{hmac}(auth_1, \langle \mathsf{cfk}, \mathsf{pk}(sk_1), n, ne_1, no_1\rangle),$
$no_2, \mathsf{hmac}(auth_2, \langle \mathsf{cfk}, \mathsf{pk}(sk_2), n, ne_2, no_2\rangle)$
Charlie $\rightarrow$ TPM : $n, no_1, \mathsf{hmac}(auth_1, \langle \mathsf{cfk}, \mathsf{pk}(sk_1), n, ne_1, no_1\rangle),$
$no_2, \mathsf{hmac}(auth_i, \langle \mathsf{cfk}, \mathsf{pk}(sk_i), n, ne_2, no_2\rangle)$

TPM $\rightarrow$ Charlie : $ne_1', ne_2', \mathsf{cert}(sk_1, \mathsf{pk}(sk_i)), \ldots$

**Fig. 5.** Attack trace for Experiment 5

ProVerif succeeds in proving the other correspondence property. Note that in the trace described in Figure 5, Charlie is not able to build the HMACs expected by the user in order to accept the wrong certificate.

**Experiment 6.** The attack of [12] comes from the fact that the attacker can replace the user's HMAC with one of his own (pertaining to his own key). The TPM will not detect this change since the only link between the two HMACs is the nonce $n$ known by the attacker. To fix this, it seems important that each HMAC contains something that depends on the two keys involved in the certificate. So, we add a digest of each public key inside each HMAC. For instance, the first HMAC built by the user will be now of the form:

$$\mathsf{hmac}(auth_1, \langle \mathsf{cfk1}, \mathsf{pk}(sk_1), \mathsf{pk}(sk_2), n, ne_1, no_1\rangle).$$

The attack described in Experiment 5 is not possible anymore. The TPM will only accept two HMACs that refer to the same pair of public keys.

ProVerif is now able to verify that the two correspondence properties hold. However, it does not succeed in proving injectivity for the property expressing authentication of the user. This is due to a limitation of the tool and does not correspond to a real attack.

**Experiment 7.** We now study the command TPM_CreateWrapKey in isolation. For this command, we need an OSAP session. We consider a configuration with 2 keys pairs $(sk_1, \mathsf{pk}(sk_1))$ and $(sk_2, \mathsf{pk}(sk_2))$ loaded inside the TPM. The attacker knows the handles $\mathsf{handle}(auth_1, sk_1)$ and $\mathsf{handle}(auth_2, sk_2)$. He has access to the public part of these two keys but he does not know the private part of the keys and the authdata associated to these keys that allows one to manipulate the keys through the device. For the moment, the intruder does not have his own key loaded onto the device.

For this simple configuration, ProVerif is able to verify that the two correspondence properties hold. Note that this command involves only one key, thus the kind of confusion that exists for the TPM_CertifyKey command is not possible on the command TPM_CreateWrapKey.

**Experiment 8.** We add another key for Alice in the initial configuration and assume that this new key $sk_2'$ has the same authdata as a previous key of Alice already loaded in the TPM.

We discover an attack of the same type as the one presented in Experiment 3. When the user opens the OSAP session on $\mathsf{handle}(auth_2, sk_2)$, the attacker can replace this handle by $\mathsf{handle}(auth_2, sk_2')$. Similarly, when the user requests to create a wrap with $\mathsf{handle}(auth_2, sk_2)$ the attacker replaces this handle by $\mathsf{handle}(auth_2, sk_2')$. Hence, at the end, the user will obtain a wrap that is not the expected one. Note that the user cannot detect that the wrap he received has been performed with $\mathsf{pk}(sk_2')$ instead of $\mathsf{pk}(sk_2)$ since he does not have access to the private part of the key. Hence, a trace similar to the one presented in Figure 4 allows one to falsify both correspondence security properties.

**Experiment 9.** As in the case of the TPM_CertifyKey command, a way to fix this, is to add $\mathsf{pk}(sk)$ inside the HMAC. Then ProVerif is able to verify the two correspondence properties even if we load a 'dishonest' key inside the TPM, *i.e.* a key for which the attacker knows the authdata.

**Experiment 10.** We now consider a much richer scenario. We consider the commands:

- TPM_CertifyKey (the version described in Experiment 6 in order to avoid the previous attacks),
- TPM_CreateWrapKey, TPM_LoadKey2, and TPM_UnBind for which we add the public key inside the HMAC (again to avoid the kind of attacks that we described in Experiment 3 and Experiment 8).

We consider a scenario where an honest key and a dishonest key are already loaded inside the TPM. Note that by using TPM_CreateWrapKey and TPM_LoadKey2, the honest user and the attacker will be able to create and load new keys into the device. Hence, having only two keys loaded in the TPM in the initial configuration is not a restriction. An honest user is allowed to use the same authdata for different keys.

ProVerif is able to establish the 8 correspondence security properties. However, in one case, as in Experiment 6, it is not able to verify the injective version of the property.

## 5 Related Work

Several papers have appeared describing systems that leverage the TPM to create secure applications, but most of these assume that the TPM API correctly functions and provides the high-level security properties required [9, 11]. Lower level analyses of the

TPM API are more rare. Coker *et al.* discuss such work, but the details of the model remain classified [8]. Lin described an analysis of various fragments of the TPM API using Otter and Alloy. He modelled several subsets of the API commands in a model which omits details such as sessions, HMACs and authdata, but does include (monotonic) state. His results included a possible attack on the delegation model of the TPM [14]. However, experiments with a real TPM have shown that the attack is not possible [2]. Gürgens *et al.* [12] describe an analysis of the TPM API using finite state automata. Details of their model are difficult to infer from the paper, but is seems to include a finite number of fresh nonces and handles, and HMACs. They also formalise security as secrecy of certain terms in the model, giving examples of concrete scenarios where these secrets must be protected. They show how an attacker can in some circumstances illegitimately obtain a certificate on a TPM key of his choice (see Experiment 5).

In our work, we have proposed correspondence properties as a more general security goal for the API, and shown how attacks such as the attack in Experiment 5 are in fact an instance of a violation of these goals. We have also verified these properties on a patched API for unbounded numbers of command invocations, fresh nonces, keys and handles. We have only treated one small subset of commands so far, but runtimes with ProVerif are just a few seconds, and we are optimistic about extending our approach to cover more of the API.

Other attacks on the TPM found without the aid of formal methods include offline dictionary attacks on the passwords or authdata used to secure access to keys [7], and attacks exploiting the fact that the same authdata can be shared between users [6]. Both of these can be detected in our formal model by small adjustments. However, for the moment we deliberately omit these, since fixes have already been proposed, and we are interested in analysis of the underlying API. There is a further known attack whereby an attacker intercepts a message, aiming to cause the legitimate user to issue another one, and then causes both to be received, resulting in the message being processed twice [5]. This is not a violation of our correspondence property, hence our formal model does not consider it as an attack.

## 6 Conclusion and Future Work

We presented a detailed modelling of a fragment of the TPM in the applied pi calculus. We model core security properties as correspondence properties and use the ProVerif tool to automate our security analysis. We were able to rediscover several known attacks and some new variants of these attacks.

As future work, we foresee extending our model with more commands such as those involved in key migration. 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. PCRs are crucial when using the TPM for checking the integrity of a system. Modelling the PCRs and the commands for manipulating these registers for automated verification seems to be a challenging task, because of non-monotonicity of state.

*Acknowledgments.* Mark Ryan gratefully thanks Microsoft and Hewlett-Packard for interesting discussions and financial support that contributed to this research.

## References


1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. 28th Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115. ACM Press, 2001.
2. K. Ables. An attack on key delegation in the Trusted Platform Module (first semester mini-project in computer security). Master's thesis, School of Computer Science, University of Birmingham, 2009.
3. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proc. 17th International Conference on Computer Aided Verification (CAV'05)*, pages 281–285, 2005.
4. B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.
5. D. Bruschi, L. Cavallaro, A. Lanzi, and M. Monga. Replay attack in TCG specification and solution. In *Proc. 21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 127–137. IEEE Computer Society, 2005.
6. L. Chen and M. Ryan. Attack, solution and verification for shared authorisation data in TCG TPM. In *Proc. 6th International Workshop on Formal Aspects in Security and Trust (FAST'09)*, pages 201–216, 2009.
7. L. Chen and M. D. Ryan. Offline dictionary attack on TCG TPM weak authorisation data, and solution. In *Future of Trust in Computing*. Vieweg & Teubner, 2008.
8. G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of remote attestation. *International Journal of Information Security*, To Appear, 2010.
9. A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Proc. 30th IEEE Symposium on Security and Privacy (S&P'09)*, pages 221–236, 2009.
10. S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *Proc. Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, volume 5511 of *LNCS*, pages 92–106, York, UK, 2009. Springer.
11. Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, and N. Asokan. Beyond secure channels. In *Scalable Trusted Computing (STC'07)*, pages 30–40, November 2007.
12. S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga. Security evaluation of scenarios based on the TCG's TPM specification. In *Proc. 12th European Symposium On Research In Computer Security (ESORICS'07)*, volume 4734 of *LNCS*, pages 438–453. Springer, 2007.
13. ISO/IEC PAS DIS 11889: Information technology – Security techniques – Trusted Platform Module.
14. A. H. Lin. Automated Analysis of Security APIs. Master's thesis, MIT, 2005. `http://sdg.csail.mit.edu/pubs/theses/amerson-masters.pdf`.
15. L. Sarmenta. TPM/J developer's guide. Massachussetts Institute of Technology.
16. Trusted Computing Group. TPM Specification version 1.2. Parts 1–3, revision 103. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, 2007.