Automated Proof for Authorization Protocols of TPM 2.0 in Computational Model (full version)

Weijin Wang, Yu Qin, Dengguo Feng, Xiaobo Chu

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

Abstract. We present the first automated proof of the authorization protocols in TPM 2.0 in the computational model. The Trusted Platform Module(TPM) is a chip that enables trust in computing platforms and achieves more security than software alone. The TPM interacts with a caller via a predefined set of commands. Many commands reference TPM-resident structures, and use of them may require authorization. The TPM will provide an acknowledgement once receiving an authorization. This interact ensure the authentication of TPM and the caller. In this paper, we present a computationally sound mechanized proof for authorization protocols in the TPM 2.0. We model the authorization protocols using a probabilistic polynomial-time calculus and prove authentication between the TPM and the caller with the aid of the tool CryptoVerif, which works in the computational model. In addition, the prover gives the upper bounds to break the authentication between them.

Key words: TPM, Trusted Computing, formal methods, computational model, authorization

1 Introduction

The Trusted Platform Module(TPM) is a chip that enables trust in computing platforms and achieves higher levels of security than software alone. Starting in 2006, many new laptop computers have been sold with a Trusted Platform Module chip built-in. Currently TPM is used by nearly all PC and notebook manufacturers and Microsoft has announced that all computers will have to be equipped with a TPM 2.0 module since January 1, 2015 in order to pass the Windows 8.1 hardware certification. Moreover, the TPM specification is an industry standard [20] and an ISO/IEC standard [14] coordinated by the Trusted Computing Group.

Many commands to the TPM reference TPM-resident structures, and use of them may require authorization. When an authorization is provided to a TPM, the TPM will provide an acknowledgement. As we know, several vulnerabilities of the authorization in the TPM 1.2 have been discovered [9, 10, 16, 12]. Most of them are found by the formal analysis of the secrecy and authentication properties. These attacks highlight the necessity of formal analysis of the authorization

in the TPM 2.0. But as far as we know, there is not yet any analysis of authorization protocols in the TPM 2.0. Hence, we perform such an analysis in this paper.

There are two main approaches to the verification of cryptographic protocol. One approach, known as the computational model, is based on probability and complexity theory. Messages are bitstring and the adversary is a probability polynomial-time Turing machine. Security properties proved in this model give strong security guarantees. Another approach, known as the symbolic or Dalev-Yao model, can be viewed as an idealization of the former approach formulated using an algebra of terms. Messages are abstracted as terms and the adversary is restricted to use only the primitives. For the purpose of achieving stronger security guarantees, we provide the security proof of the authorization protocols in the computational model. Up to now, the work in the literatures are almost based on the symbolic model, our work is the first trial to formally analyze the authorization in the computation model.

Related work and Contributes. Regarding previous work on analyzing the API or protocols of TPM, most of them are based on the TPM 1.2 specifications and analyses of the authorization are rare. Lin [16] described an analysis of various fragments of the TPM API using the theorem prover Ptter and the model finder Alloy. He modeled the OSPA and DSAP in a model which omits low level details. His results in the authorization included a key-handle switching attack in the OSAP and DSAP. Bruschi *et al.* [9] proved that OIAP is exposed to replay attack, which could be used for compromising the correct behavior of a Trusted Computing Platform. Chen *et.al* found that the attacks about authorization include offline dictionary attacks on the passwords or *authdata* used to secure access to keys [10], and attacks exploiting the fact that the same *authdata* can be shared between users [11]. Nevertheless, they did not get the aid of formal methods. Delaune *et.al.* [12] have analyzed a fragment of the TPM authentication using the ProVerif tool, yet ignoring PCRs and they subsequently analyzed the authorization protocols which rely on the PCRs [13]. Recently, Shao [18] *et.al.* have modeled the Protect Storage part of the TPM 2.0 specification and proved their security using type system.

In our work, we first present the automated proof of authorization protocols in the TPM 2.0 at the computational leval. To be specific, we model the authorization protocols in the TPM 2.0 using a probabilistic polynomial-time calculus inspired by pi calculus. Also, we propose correspondence properties as a more general security goal for the authorization protocols. Then we apply the tool CryptoVerif [4–6] proposed by Blanchet, which works in the computational model, to prove the correspondence properties of the authorization protocols in the TPM 2.0 automatically. As a result, we show that authorization protocols in the TPM 2.0 guarantee the authentication of the caller and the TPM and give the upper bounds to break the authentication.

Outline. We review the TPM 2.0 and the authorization sessions in the next section. Section 3 describes our authorization model and the definition of security properties, Section 4 illustrates its results using the prover CryptoVerif. We conclude in Section 5.

2 An overview of the TPM authorization

When a protected object is in the TPM, it is in a shielded location because the only access to the context of the object is with a Protected Capability (a TPM command). Each command has to be called inside an authorization session. To provide flexibility in how the authorizations are given to the TPM, the TPM 2.0 specification defines three authorization types:

1. Password-based authorization;
2. HMAC-based authorization;
3. Policy-based authorization.

We focus on the HMAC-based authorization. The commands that requires the caller to provide a proof of knowledge of the relevant $authValue$ via the HMAC-based authorization sessions have an authorization HMAC as an argument.

2.1 Session

A session is a collection of TPM state that changes after each use of that session. There are three uses of a session:

1. *Authorization* – A session associated with a handle is used to authorize use of an object associated with a handle.
2. *Audit* – An audit session collects a digest of command/response parameters to provide proof that a certain sequence of events occurred.
3. *Encryption* – A session that is not used for authorization or audit may be present for the purpose of encrypting command or response parameters.

We pay attention to the authorization sessions. Both HMAC-based authorization sessions and Policy-based authorization sessions are initiated using the command **TPM2_StartAuthSession**. The structures of this command can be found in TPM 2.0 specification, Part 3 [20]. The parameters of this command may be chosen to produce different sessions. As mentioned before, we just consider the HMAC-based authorization sessions and set the $sessionType$ =HMAC. The TPM 2.0 provides four kinds of HMAC sessions according to various combination of the parameters *tpmkey* and *bind*:

1. **Unbound and Unsalted Session.** In the version of session, *tpmkey* and *bind* are both null.
2. **Bound Session.** In this session type, *tpmkey* is null but *bind* is present and references some TPM entity with $authValue$.

3. **Salted Session.** For this type of session, *bind* is null but *tpmkey* is present, indicating a key used to encrypt the *salt* value.
4. **Salted and Bound Session.** In this session, both *bind* and *tpmkey* is present. The *bind* is used to provide an $authValue$, *tpmkey* encrypts the *salt* value and the *sessionkey* is computed using both of them.

A more detailed description of the sessions is given in [20].

2.2 Authorization protocols

We start with modelling the authorization protocols constructed from an example command, named **TPM2_Example**, within some authorization sessions. **TPM2_Example** is a more generic command framework other than a specific command which can be found in TPM 2.0 specification, Part 1 [20]. This command has two handles ($handleA$ and $handleB$) and use of the entity associated with $handleA$ required authorization while $handleB$ does not. Therefore, $handleB$ does not necessarily appear in our protocol models.

We take the authorization protocol based on Salted and Bound Session as an example. The other three protocols will be presented in the Appendix A.

Protocol based on Salted and Bound Session. We omit some size parameters that will not be involved in computation, such as $commandSize$, $authorizationSize$ and $nonceCallerSize$. The specification of the protocol is given in the Figure 1. For the protocol based on Salted and Bound Session, the **Caller** sends the command **TPM2_StartAuthSession** to the **TPM**, together with a handle of the bound entity, an encrypted salt value, a hash algorithm to use for the session and a nonce $nonCallerStart$ which is not only a parameter for computing session key but also a initial nonce setting nonce size for the session. The response includes a session handle and a nonce $nonceTPM$ for rolling nonce. Then the **Caller** and **TPM** both compute the session key as

$$sessionKey = \mathbf{KDFa}(sessionAlg, bind.authValue||salt, \mathrm{ATH}, \\ nonceTPM, nonceCallerStart, bits)$$

and save $nonceTPM$ as $lastnonceTPM$, where **KDFa()** is a key derivation function (a hash-based function to generate keys for multiple purposes). After that, **TPM2_Example** within such a session will be executed. If the session is used to authorize use of the bound entity, i.e. $bind.Handle = key.Handle$, then

$$comAuth = \mathbf{HMAC}_{sessionAlg}(sessionKey, \\ (cpHash||nonceCaller||lastnonceTPM||sessionAttributes)),$$

where $cpHash = H_{sessionAlg}(commandCode||key.name||comParam)$. Next the **TPM** will generate a new $nonceTPM$ named $nextnonceTPM$ for next

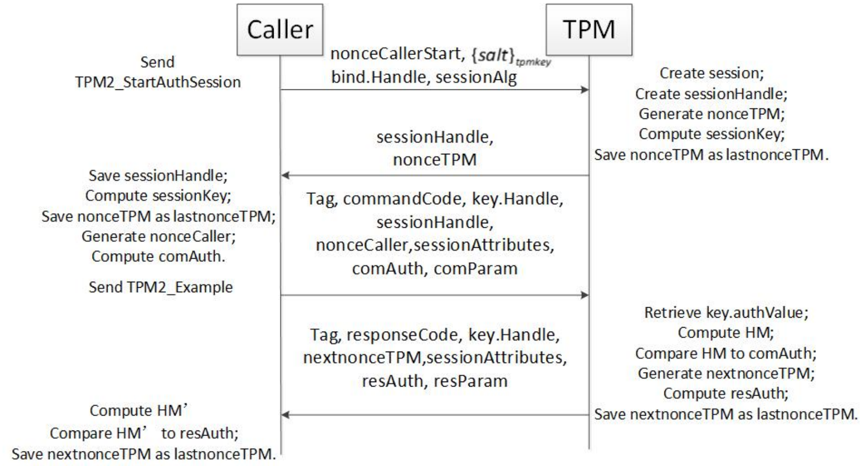

Fig. 1. Protocol based on Unbound and Unsalted Session

rolling and send back an acknowledgment

$$resAuth = \mathbf{HMAC}_{sessionAlg}(sessionKey, \\ (rpHash||nextnonceTPM||nonceCaller||sessionAttributes)),$$

where $rpHash = H_{sessionAlg}(responseCode||commandCode||resParam)$.

Else if the session is used to access a different entity, i.e. $bind.Handle \neq key.Handle$, then

$$comAuth = \mathbf{HMAC}_{sessionAlg}(sessionKey||key.authVaule, \\ (rpHash||nonceCaller||lastnonceTPM||sessionAttributes)),$$

and

$$resAuth = \mathbf{HMAC}_{sessionAlg}(sessionKey||key.authValue, \\ (rpHash||nextnonceTPM||nonceCaller||sessionAttributes)).$$

When finalizing current session, Both **Caller** and **TPM** save $nextnonceTPM$ as $lastnonceTPM$.

3 Authorization Model and Security Properties

In the beginning of this section, we model the authorization protocols of TPM 2.0. Our model uses a probabilistic polynomial-time process calculus, which is inspired by the pi calculus and the calculi introduced in [17] and [15], to formalize the cryptographic protocols. In this calculus, messages are bitstrings and cryptographic primitives are functions operating on bitstrings. Then we define the security properties of the participants in our model.

$Q_C = !^{i_C \leqslant N} c_4[i_C]();$
$\quad$ **new** $N_C_Start : nonce$; `new` $salt : nonce$; **new** $r_1 : seed$;
$\quad \overline{c_5[i_C]}(handle_{bind}, N_C_Start, \mathbf{enc}(salt, tpmkey, r_1));$
$\quad c_8[i_C](n_T : nonce);$
$\quad$ **let** $skseed = \mathbf{hash}_1(\mathbf{concat}_6(salt, \mathbf{getAuth}(handle_{bind}, auth_{bind})),$
$\quad\quad \mathbf{concat}_5(ATH, n_T, N_C_Start, bits))$ **in**
$\quad$ **let** $sk_C = \mathbf{mkgen}(skseed)$ **in**
$\quad$ **new** $N_C : nonce$;
$\quad$ **let** $cpHash = \mathbf{hash}(hk, \mathbf{concat}_3(comCode, \mathbf{getName}(handle_{bind}),$
$\quad\quad comParam))$ **in**
$\quad$ **let** $comAuth = \mathbf{mac}(\mathbf{concat}_1(cpHash, N_C, n_T, sAtt), sk_C)$ **in**
$\quad$ **even** $CallerRequest(N_C, n_T, sAtt);$
$\quad \overline{c_9[i_C]}(comCode, handle_{bind}, N_C, sAtt, comAuth, comParam);$
$\quad c_{12}[i_C](= resCode, = handle_{bind}, n_T_next : nonce, = sAtt,$
$\quad\quad resHM : macres, = resParam);$
$\quad$ **let** $rpHa = \mathbf{hash}(hk, \mathbf{concat}_4(comCode, resCode, resParam)$ **in**
$\quad$ **if** $\mathbf{check}(\mathbf{concat}_2(rpHa, n_T_next, N_C, sAtt), sk_C, resHM)$ **then**
$\quad$ **event** $CallerAccept(N_C, n_T_next, sAtt).$

Fig. 2. Formalization of Caller's actions

3.1 Modelling the Authorization Protocols.

To be more general, we present the **Caller**'s actions base on Salted and Bounded Session used to access the bound entity in the process calculus as an example. (The formalizations of the other sessions will be present in Appendix C.)

We defined a process Q_C to show **Caller**'s actions, detailed in Figure 2. The replicated process $!^{i_C \leq N} P$ represents N copies of P, available simultaneously, where N is assumed to be polynomial in the security parameter η. Each copy starts with an input $c_4[i_C]$, means that the adversary gives the control to the process. Then the process chooses a random nonce N_C_Start, a *salt* value for establishing a session key, and a random seed r_1 for encryption. The process then sends a message $handle_{bind}, NCStart, enc(salt, tpmkey, r_1)$ on the channel $c_5[i_C]$. The $handle_{bind}$ is the key handle of the bound entity. This message will be received by the adversary, and the adversary can do whatever he wants with it.

After sending this message, the control is returned to the adversary and the process waits for the message on the channel $c_8[i_C]$. The expected type of this message is *nonce*. Once receiving the message, the process will compute a session key sk_C and an authorization *comAuth*. The function $\mathbf{concat}_i (1 \leq i \leq 6)$ are concatenations of some types of bitstrings. We also use the functions

$$
\begin{aligned}
Q_T = &!^{i_T \leqslant N} c_2[i_T](bdhandle : keyHandle, cCode : code, rCode : code,\\
&\qquad\qquad\qquad cParam : parameter, rParam : parameter);\\
&\overline{c_3[i_T]}();\\
&c_6[i_T](= bdhandle, n_C_Start : nonce, e : ciphertext);\\
&\mathbf{new}\ N_T : nonce;\\
&\mathbf{let}\ injbot(salt_T) = dec(e, tpmkey)\ \mathbf{in}\\
&\mathbf{let}\ skseed = \mathbf{hash}_1(\mathbf{concat}_6(salt_T, \mathbf{getAuth}(bdhandle, auth_{bind})),\\
&\qquad\qquad\qquad \mathbf{concat}_5(ATH, N_T, n_C_Start, bits))\ \mathbf{in}\\
&\mathbf{let}\ sk_T = \mathbf{mkgen}(skseed)\ \mathbf{in}\\
&\overline{c_7[i_T]}(N_T);\\
&c_{10}[i_T](= cCode, = bdhandle, n_C : nonce, sAttRec : flags,\\
&\qquad comHM : macres, = cParam);\\
&\mathbf{if}\ \mathbf{getContinue}(sAttRec) = \mathbf{true}\ \mathbf{then}\\
&\mathbf{let}\ cpHa = \mathbf{hash}(hk, \mathbf{concat}_3(cCode, \mathbf{getName}(bdhandle), cParam))\ \mathbf{in}\\
&\mathbf{if}\ \mathbf{check}(\mathbf{concat}_1(cpHa, n_C, N_T, sAttRec), sk_T, comHM)\ \mathbf{then}\\
&\mathbf{even}\ TPMAccept(n_C, N_T, sAttRec);\\
&\mathbf{new}\ N_T_next : nonce;\\
&\mathbf{let}\ rpHash = \mathbf{hash}(hk, \mathbf{concat}_4(cCode, rCode, rParam)\ \mathbf{in}\\
&\mathbf{let}\ resAuth = \mathbf{mac}(\mathbf{concat}_2(rpHash, N_T_next, n_C, sAttRes), sk_T)\ \mathbf{in}\\
&\mathbf{event}\ TPMAcknowledgment(n_C, N_T_next, sAttRec);\\
&\overline{c_{11}[i_T]}(rCode, bdhandle, N_T_next, sAttRec, recAuth, rParam).
\end{aligned}
$$

Fig. 3. Formalization of TPM's actions

getAuth and **getName** to model the actions getting the authorization value and key name of the enitity from the key handle. $comCode$, $resCode$, $comParam$ and $resParam$ represent the command code, respond code, remaining command parameters and the response parameters respectively. $sAtt$ stands for the session attributes, which is a octet used to identify the session type. Since our analysis uses the same session type, we model it a fixed octet here.

When finalizing the computation, the process will execute the event **Caller-Request**$(N_C, n_T, sAtt)$ and send the authorization $comAuth$, together with $comCode$,$handle_{bind}$,N_C,$sAtt$ and $comParam$ on the channel $c_9[i_C]$. Then the process waits for the second message from the environment on the channel $c_{12}[i_C]$. The expected message is $resCode$, $handle_{bind}$, n_T_next, $sAtt$, $resHM$ and $resParam$. The process checks the first component of this message is $resCode$ by using the pattern $= resCode$, so do the $handle_{bind}$, $sAtt$ and $resParam$; the two other parts are stored in variables. The process will verify the received acknowledgment $resHM$. If the check succeeds, Q_C executes the event **CallerAccept**$(N_C, n_T_next, sAtt)$.

In this calculus, executing these events does not affect the execution of the protocol, it just records that a certain program point is reached with certain values of the variables. Events are used for specifying authentication properties, as explained next session. We show the **TPM**'s action in the Figure 3, corresponding to the **Caller**'s action.

3.2 Security Properties

Definition of Authentication. The formal definitions can be found in [6]. The calculus use the correspondence properties to prove the authentication of the participants in the protocols. The correspondence properties are properties of the form if some event has been executed, then some other events also have been executed, with overwhelming probability. It distinguishes two kinds of correspondences, we employ the description from [8] below.

1. A process Q *satisfies the non-injective correspondence* $\mathbf{event}(e(M_1, ..., M_m)) \Rightarrow \bigwedge_{i=1}^{k} \mathbf{event}(e_i(M_{i1}, ..., M_{im_i}))$ if and only if, with overwhelming probability, for all values of the variables in $M_1, ..., M_m$, if the event $e(M_1, ..., M_m)$ has been executed, then the events $e_i(M_{i1}, ..., M_{im_i})$ for $i \leq k$ have also been executed for some values of the variables of $M_{ij} (i \leq k, j \leq m_i)$ not in $M_1, ..., M_m$.
2. A process Q *satisfies the injective correspondence* $\textbf{inj-event}(e(M_1, ..., M_m)) \Rightarrow \bigwedge_{i=1}^{k} \textbf{inj-event}(e_i(M_{i1}, ..., M_{im_i}))$ if and only if, with overwhelming probability, for all values of the variables in $M_1, ..., M_m$, for each execution of the event $e(M_1, ..., M_m)$, there exist distinct corresponding executions of the events $e_i(M_{i1}, ..., M_{im_i})$ for $i \leq k$ for some values of the variables of $M_{ij} (i \leq k, j \leq m_i)$ not in $M_1, ..., M_m$.

Security Properties of the authorization. One of the design criterion of the authorization protocol is to allow for ownership authentication. We will formalize these security properties as correspondence properties. Firstly, we give the informal description of the security properties.

1. When the TPM receives a request to use some entity requiring authorization and the HMAC verification has succeeded, then a caller in possession of the relevant $authValue$ has really requested it before.
2. When a caller accepts the acknowledgment and believes that the TPM has executed the command he sent previously, then the TPM has exactly finished this command and sent an acknowledgment.

The first property expresses the authentication of the **Caller** and the second one expresses the authentication of the **TPM**. We can formalize the properties above as injective correspondence properties:

$$\text{inj} : \mathbf{TPMAccept}(x, y, z) \Rightarrow \text{inj} : \mathbf{CallerRequest}(x, y, z). \quad (1)$$

$$\text{inj} : \mathbf{CallerAccept}(x, y, z) \Rightarrow \text{inj} : \mathbf{TPMAcknowledgment}(x, y, z). \quad (2)$$

4 Authentication Results with CryptoVerif

In this section, we will take a brief introduction of CryptoVerif and the assumption used in our model. Then we present security properties directly proven by CryptoVerif under the assumptions in the computational model.

4.1 CryptoVerif

There are two main approaches to the verification of cryptographic protocols. One approach is known as the computational model and another approach, is known as the symbolic or Dalev-Yao model. The CryptoVerif, proposed by Blanchet[4–7], can directly prove security properties of cryptographic protocols in the computational model. This tool is available on line at:

`http://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/`

CryptoVerif builts proofs by sequences of games [19, 3]. It starts from the initial game given as input, which represents the protocol to prove in interaction with an adversary (real mode). Then, it transforms this game step by step using a set of predefined game transformations, such that each game is indistinguishable from the previous one. More formally, we call two consecutive games Q and Q' are observationally equivalent when they are computationally indistinguishable for the adversary. CryptoVerif transforms one game into another by applying the security definition of a cryptographic primitive or by applying syntactic transformations. In the last game of a proof sequence the desired security properties should be obvious (ideal mode).

Given a security parameter η, CryptoVerif proofs are valid for a number of protocol sessions polynomial in η, in the presence of an active adversary. CryptoVerif is sound: whatever indications the user gives, when the prover shows a security property of the protocol, the property indeed holds assuming the given hypotheses on the cryptographic primitives.

4.2 Assumptions

We introduce the basic assumptions and cryptographic assumptions adopted by our model and the CryptoVerif as follow.

Basic Assumptions. One of the difficulties in reasoning about authorization 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 $s' \neq s$. Suppose two commands use the same session, the latter must use the nonce generated by the former called $nextnonceTPM$ as the $lastnonceTPM$ when computing the authorization $comAuth$. In other words, the $lastnonceTPM$ in the latter is equal to the $nextnonceTPM$ in the former. CryptoVerif does not model such a state transition system.

We address these restrictions by introducing the assumption described by the S. Delaune *et.al* [12], such that only one command is executed in each session.

Cryptographic Assumptions. In the analysis of the authorization protocols, the Message Authentication Code (MAC) scheme is assumed to be unforgeable under chosen message attacks (UF-CMA). Symmetric encryption is assumed to be indistinguishable under chosen plaintext attacks (IND-CPA) and to satisfy ciphertext integrity (INT-CTXT). These properties guarantee indistinguishability under adaptive chosen ciphertext attacks (IND-CCA2), as shown in [2].

We assume that the key derivation function is a pseudo-random function and use it to derive, from a key seed, a key for the message authentication code. The key seed is generated from a keying hash function. The keying hash function is assumed to be a message authentication code, weakly unforgeable under chosen message attacks, which is in accordance with [1]. To be specific, we compute the *sessionkey* in a more flexible way, the result of the keying hash function is a *keyseed* and the *sessionkey* is generated from this *keyseed* using a pseudo-random function.

4.3 Experiment Results

Here we present authentication results directly proven in the computational model by CryptoVerif 1.16 under assumptions mentioned above.

Experiment 1: Case of Unbound and Unsalted Session. In this case, we consider a protocol without session key. The attacker can obtain the key handle but cannot get the corresponding $authValue$. The **Caller** and **TPM** will compute the HMAC keyed by $authValue$ directly.

But unfortunately, we cannot achieve the injective correspondences between the event **CallerAccept** and **TPMAcknowledgment** in (2) by CryptoVerif directly because of limitations of the prover: it crashes when proving this property. However, it succeeds in the non-injective case, hence we complete this proof by hand.

Lemma 1. *In the protocol based on Unbound and Unsalted Session, if the property:*

$$\boldsymbol{CallerAccept}(NC, nextnT, sAtt) \Rightarrow \\ \boldsymbol{TPMAcknowledgment}(nC, nextNT, sAttRec)$$

holds, then we have

$$\boldsymbol{inj{:}CallerAccept}(NC, nextnT, sAtt) \Rightarrow \\ \boldsymbol{inj{:}TPMAcknowledgment}(nC, nextNT, sAttRec).$$

Proof. Since the non-injective property succeeds, we can find $i_C \leq N$ such that $N_C[i_C] = n_C[u[i_C]]$, $n_T_next[i_C] = N_T_next[u[i_C]]$, $sAtt[i_C] = sAttRec[u[i_C]]$ and $i_T \leq N$ such that $u[i_C] = i_T$.

Suppose that there exists another i'_C and i'_T satisfy the property above, and $u[i'_C] = i'_T$. In order to prove injectivity, It remains to show that the

probability of $\{i_T = i'_T, i_C \neq i'_C\}$ is negligible. The equality $i_T = i'_T$, i.e. $u[i_C] = u[i'_C]$, combined with $N_C[i_C] = n_C[u[i_C]]$ and $N_C[i'_C] = n_C[u[i'_C]]$ implies that $N_C[i_C] = n_C[u[i_C]] = n_C[u[i'_C]] = N_C[i'_C]$. Since N_C is defined by restrictions of the large type nonce, $N_C[i_C] = N_C[i'_C]$ implies $i_C = i'_C$ with overwhelming probability, by eliminating collisions. This implies that the probability of $\{i_T = i'_T, i_C \neq i'_C\}$ is negligible. □

With Lemma 1, we prove the injective correspondence properties in (1) and (2) under assumptions of UF-MAC in the MAC scheme and collision resistant in hash function using CryptoVerif.

Experiment 2: Case of Bound Session. We must compute the *sessionkey* bound to an entity in this protocol. According to the authorization entity, there are two kinds of protocols in this experiment. Firstly we consider the session is used to authorize use of the bound entity with an authorization value $authVaule_{bind}$, the HMAC is keyed by $sessionKey$. In another situation, we employ the bound session to access a different entity with an authorization value $authVaule_{entity}$. The $sessionKey$ is still bound to the entity with authorization value $authValue_{bind}$ while the HMAC will take the concatenation of *sessionkey* and the $authVaule_{entity}$ as a key.

Providing the MAC scheme assumed to be UF-MAC and hash function in the random oracle model, we prove the injective correspondence properties of two kinds of protocols mentioned above using CryptoVerif.

Experiment 3: Case of Salted Session. This session can be treated as the enhanced version of unbound and unsalted session. Salting provides a mechanism to allow use of low entropy $authValue$ and still maintain confidentiality for the $authVaule$. If the $authValue$ used in an unsalted session has low entropy, the attacker will perform an off-line attack, which is detailed in the TPM 2.0 specification, Part 1 [20].

The *salt* value may be symmetrically or asymmetrically encrypted. In our analysis, We assume an IND-CPA and INT-CTXT probabilistic symmetric encryption scheme is adopted by the participants. We show that this protocol satisfies the injective correspondence properties in (1) and (2) under the assumption of IND-CPA, INT-CTXT and UF-MAC.

Experiment 4: Case of Salted and Bound Session. If the bound entity has a low entropy, it will still be under threat of the off-line attack. This session looks like the enhanced version of bound session. Unlike the bound session only using the authorization value of bound entity to compute the $sessionKey$, this session employs both the $authValue_{bind}$ and the *salt* value. The remaining computation is the same as the bound session and the session also exist two kinds of the protocols.

Nevertheless, we can still prove the injective correspondence properties of two kinds of protocols using CryptoVerif under the assumption of IND-CPA, INT-CTXT and UF-MAC.

As a result, We formalize the experiment results mentioned above as the following theorems. The authentication of **TPM** can be represented as the theorem 1.

Theorem 1. *In the all kinds of authorization protocols, if there is an instance of:*

1. *The TPM received a Caller's command with a request for authorization of some sensitive data,*
2. *The TPM executed this command and the HMAC check in this command has succeeded.*

Then with overwhelming probability, there exists a distinct corresponding instance of:

1. *The Caller is exactly in possession of the $authValue$ of this sensitive data.*
2. *The Caller has exactly send this command with a request for authorization of this sensitive data.*

We formalize the authentication of **TPM** as the following theorem.

Theorem 2. *In the all kinds of authorization protocols, if there is an instance of:*

1. *The Caller received an acknowledgment from the TPM,*
2. *The HMAC check in the response has succeeded and the Caller accepted the acknowledgment.*

Then with overwhelming probability, there exists a distinct corresponding instance of:

1. *The TPM has precisely received the callers request and executed this command,*
2. *The TPM has really send an acknowledgment to the Caller.*

The proofs for the Theorem 1 and Theorem 2 in the case of **Salted and Bound Session** used to access the bound entity and the corresponding upper bounds to break the authentication between the **Caller** and **TPM** can be found in Appendix B. The other cases can be proved in a similar way, so we omit the details because of length constrains.

Note that in the case of **Unbound and Unsalted Session**, CryptoVerif is only able to prove the non-injective correspondence property between the even **CallerAccept** and **TPMAcknowledgment**, but thanks to Lemma 1, we can obtain the results of Theorem 2.

5 CONCLUSIONS

We have proved the security of authorization protocols in the TPM 2.0 using the tool CryptoVerif working in the computational model. Specifically, we presented a detailed modelling of the protocols in the probabilistic calculus inspired by the pi calculus. Additionally, we model security properties as correspondence properties. Then we have formalized and mechanically proved these security properties of authorization protocols in the TPM 2.0 using Cryional model.

As future work, we will find out the reason why the prover crashes when proving the injective correspondences between the event **CallerAccept** and event **TPMAcknowledgment** in the protocols based on the **Unbound and Unsalted Sessions** and try to improve the prover to fix it. We will extend our mode with the asymmetric case encrypting the *salt* value. Also, we argue that our model can be adapted to prove the confidentiality using CryptoVerif and it will be our future work.

Acknowledgments. The research presented in this paper is supported by the National Natural Science Foundation of China under Grant Nos. 91118006, 61202414 and the National Grand Fundamental Research 973 Program of China under Grant No. 2013CB338003. We also thank the anonymous reviewers for their comments.

References

1. M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In CRYPTO'96, volume 1109 of LNCS. Springer, 1996
2. M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In ASIACRYPT 2000, volume 1976 of LNCS Springer, December 2000
3. M. Bellare and P. Rogaway, The security of triple encryption and a framework for code-based game-playing proofs, in EUROCRYPT 2006, ser. LNCS, vol. 4004. Springer, 2006, pp. 409-426, 2006
4. B. Blanchet. B. Blanchet, A computationally sound mechanized prover for security protocols, IEEE Transactions on Dependable and Secure Computing, vol. 5, no. 4, pp. 193-207, October-December 2008
5. B. Blanchet. A Computationally Sound Mechanized Prover for Security Protocols. In IEEE Symposium on Security and Privacy, May 2006
6. B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In CSF 2007, July 2007
7. B. Blanchet and D. Pointcheval. Automated Security Proofs with Sequences of Games. In CRYPTO 2006, volume 4117 of LNCS. Springer, Aug. 2006
8. B. Blanchet, A. D. Jaggard, A. Scedrov and J.-K. Tsay. Computationally Sound Mechanized Proofs for Basic and Public-Key Kerberos. In Proceedings of the 2008 ACM symposium on Information, computer and communications security. ACM Tokyo, Japan, pp. 87-99, 2008
9. 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

10. L. Chen and M. D. Ryan. Offine dictionary attack on TCG TPM weak authorisation data, and solution. In Future of Trust in Computing. Vieweg & Teubner, 2009
11. L. Chen and M. D. 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
12. S. Delaune, S. Kremer, M. D. Ryan, and G. Steel, A formal analysis of authentication in the TPM. in Proc. 7th International Workshop on Formal Aspects in Security and Trust(FAST'10), Pisa, Italy, 2010
13. S. Delaune, S. Kremer, M. D. Ryan and G. Steel. Formal Analysis of Protocols Based on TPM State Registers. In Proc. 24th IEEE Computer Security Foundations Symposium (CSF'11), pp. 66-80, 2011
14. ISO/IEC PAS DIS 11889: Information technology C security techniques C Trusted Platform Modules
15. P. Laud. Secrecy Types for a Simulatable Cryptographic Library. In CCS 2005, May 2005
16. A. H. Lin. Automated Analysis of Security APIs. Masters thesis, MIT, 2005.`http://groups.csail.mit.edu/cis/theses/amerson-masters.pdf`
17. J. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A Probabilistic Polynomial-Time Process Calculus for the Analysis of Cryptographic Protocols. Theoretical Computer Science, 353(1-3), 2006
18. J. Shao, D. Feng, and Y. Qin. Type-based analysis of protected storage in the tpm. In Proc. 15th International Conference on Information and Communications Security (ICICS'13), pages 135-150. Springer International Publishing, November 2013
19. V. Shoup, Sequences of games: a tool for taming complexity in security proofs, Cryptology ePrint Archive, Report2004/332, 2004, available at http://eprint.iacr.org/2004/332
20. Trusted Computing Group. TPM Specification version 2.0. Parts 1-4, revision 00.99, 2013. `http://www.trustedcomputinggroup.org/resources/tpm_library_specification`

A Authorization Protocols

We describe another three authorization protocols in here roughly.

Protocol based on Unbound and Unsalted Session. Showed in the Figure 4, where

$$comAuth = HMAC_{sessionAlg}(key.authValue, \\ (cpHash||nonceCaller||lastnonceTPM||sessionAttributes)),$$

and

$$resAuth = HMAC_{sessionAlg}(key.authValue, \\ (rpHash||nextnonceTPM||nonceCaller||sessionAttributes)),$$

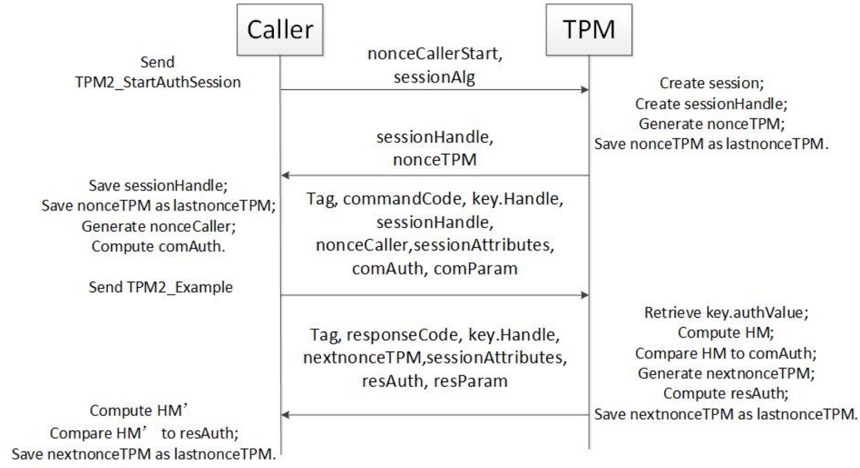

Fig. 4. Protocol based on Unbound and Unsalted Session

Protocol based on Bound Session. Showed in the Figure 5, where

$$sessionKey = KDFa(sessionAlg, bind.authValue, 'ATH', nonceTPM, nonceCallerStart, bits),$$

and if the session is used to authorize use of the bound entity, that is $key.Handle = bind.Handle$, then

$$comAuth = HMAC_{sessionAlg}(sessionKey, (cpHash||nonceCaller||lastnonceTPM||sessionAttributes)).$$

$$resAuth = HMAC_{sessionAlg}(sessionKey, (rpHash||nextnonceTPM||nonceCaller||sessionAttributes)),$$

else

$$comAuth = HMAC_{sessionAlg}(sessionKey||key.authVaule, (rpHash||nonceCaller||lastnonceTPM||sessionAttributes)),$$

$$resAuth = HMAC_{sessionAlg}(sessionKey||key.authValue, (rpHash||nextnonceTPM||nonceCaller||sessionAttributes)),$$

Protocol based on Salted Session. Showed in Figure 6, where

$$sessionKey = KDFa(sessionAlg, salt, 'ATH', nonceTPM, nonceCallerStart, bits),$$

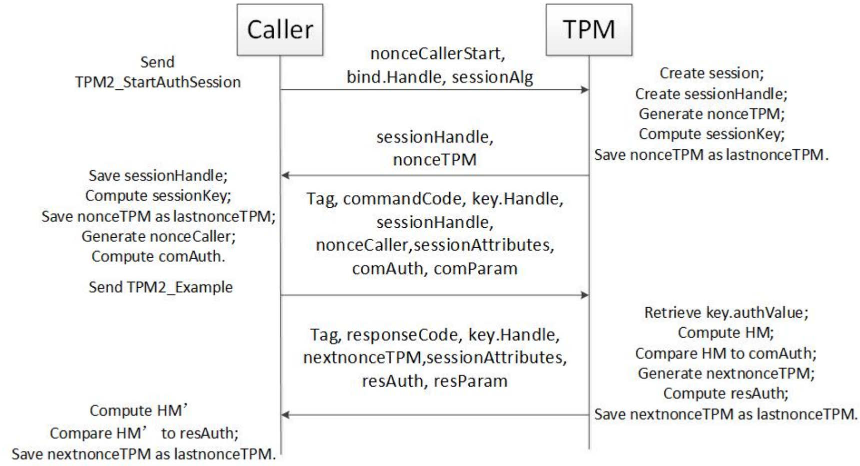

Fig. 5. Protocol based on Bound Session

$$\begin{aligned}comAuth =& HMAC_{sessionAlg}(sessionKey,\\ &(cpHash||nonceCaller||lastnonceTPM||sessionAttributes)),\end{aligned}$$

and

$$\begin{aligned}resAuth =& HMAC_{sessionAlg}(sessionKey,\\ &(rpHash||nextnonceTPM||nonceCaller||sessionAttributes)).\end{aligned}$$

B Proof of Theorem 1 and Theorem 2

B.1 Proof of Theorem 1

Proof. Case of Salted and Bound Session used to access the bound entity: When the **TPM** successes to check the HMAC it receives, it executes an event **TPMAccept**$(n_C, N_T, sAttRec)$ that contains the nonce n_C from the **Caller**, the nonce N_T it generates and the session attributes $sAttRec$ it receives, where the $sAttRec$ is a octet to indicate how the session is to be applied. When the Caller completes his computations of HMAC, he executes an event **CallerRequest**$(N_C, n_T, sAtt)$ that contains the nonce N_C generated himself and a nonce n_T he receives and the session attributes $sAtt$ he sets. CryptoVerif can then automatically prove the query:

$$\text{inj} : \textbf{TPMAccept}(x, y, z) \Rightarrow \text{inj} : \textbf{CallerRequest}(x, y, z).$$

The proof done by the CryptoVerif consists essentially in applying the security assumptions on symmetric key encryptions, the MAC and the random hash

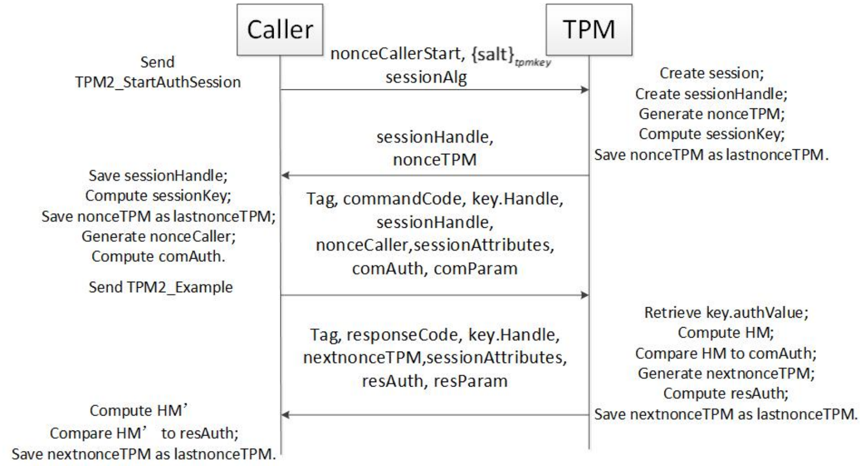

Fig. 6. Protocol based on salted Session

oracle, after some simplifications. In more detail, the CryptoVerif performs the following transformations:

– Firstly it simplifies the key of $\mathbf{hash}_1$:

$$\mathbf{concat}_6(salt, \mathbf{getAuth}(handle_{bind}, auth_{bind})),$$

We already know that the role of salt value is to improve the entropy so that the sessions can resist the off-line attack. In the CryptoVerif, the type nonce is defined as large, which means that the type nonce is large enough so that all collisions with random elements of nonce can be eliminated, and the random number chosen in the type nonce has a high entropy. Therefore, it will not lose the security if we treat the new bitstring $salt||auth_{bind}$ as a nonce in the CryptoVerif. Without loss of generality, the CryptoVerif treat it as salt (the salt value is type nonce) when process the game transformation.

After these simplifications, the Caller's session key will be

$$\mathbf{hash}_1(salt, \mathbf{concat}_5(ATH, n_T, N_C_Start, bits))$$

and the TPM's session key will be

$$\mathbf{hash}_1(salt_T, \mathbf{concat}_5(ATH, N_T, n_C_Start, bits)).$$

However, CryptoVerif still cannot apply the security assumption of the $\mathbf{hash}_1$ in that it cannot insure that the $salt_T$ is the same as $salt$ unless it has applied the security assumption of the symmetric encryption.

– CryptoVerif then apply the security assumption of hash with the key hk. This security property is represented in CryptoVerif by the equivalence shown in

$$
\begin{array}{l}
!^{ih \leq nh} \ \mathbf{new} \ k : T_k; \ !^{i \leq n}(x : bitstring) \rightarrow \mathbf{h}(k, x) \\
\approx_0 \\
!^{ih \leq nh} \ !^{i \leq n}(x : bitstring) \rightarrow \\
\quad \mathbf{find} \ u \leq n \ \mathbf{suchthat} \ \mathbf{defined}(x[u], r[u]) \wedge x = x[u] \\
\quad \mathbf{then} \ r[u] \\
\quad \mathbf{else} \ \mathbf{new} \ r : T; r
\end{array}
$$

Fig. 7. Definition of Hash Function in the Random Oracle Model

Figure 7, where h stands for function **hash**. In this equivalence, the left-hand side chooses a hash key hk and provides an oracle returning the hash of its argument x. The right-hand side provides a corresponding oracle. This oracle, however, looks for x in the array x that contains all the hash inputs that have accessed to the hash oracle. When x is found in this array, that is, there exists u such that $x = x[u]$, the oracle returns the corresponding hash $r[u]$. When no such x is found, the oracle randomly chooses a new value r and returns it. This definition is related to the fact that a random oracle is unimplementable: otherwise, the adversary could implement it without being explicitly given access to it.

Using this equivalence, CryptoVerif can transform a game by replacing the left-hand side with the right-hand side. Then each argument of a call to hash is first stored in an intermediate variable, x_{259} for **concat**$_3$ (*comCode*,**getName**(*handle*$_{bind}$),*comParam*) and x_{257} for **concat**$_4$ (*comCode, resCode,resParam*), and each occurrence of a call to hash is replaced with a lookup in the two arrays that contain arguments of calls to hash, x_{259} and x_{257}. When the argument of hash is found in one of these arrays, the returned result is the same as the result previously returned by hash. Otherwise, we pick a fresh random number and return it.

– After each cryptographic transformation , the game is simplified. CryptoVerif uses essentially equational reasoning to replace terms with simpler terms and tries to determine the result of tests, thus removes branches that cannot be executed.

Then the CryptoVerif removes the assignments on *tpmkey*, that is, it replaces *tpmkey* with its value $kgen(r)$.

CryptoVerif applies the INT-CTXT property, represented by the equivalence of Figure 8, on the key $tpmkey = kgen(r)$. The left-hand side chooses a random seed r and provides two oracles: one for encryption and another for decryption. The right-hand side provides two correspondence and indistinguishable oracles. The first one still uses for encryption, but additionally stores the ciphertext in the variable z, which is implicitly an array indexed by the number of the call to the encryption oracle. The second one, instead

of decrypting its argument y, looks for y in the array z that contains all computed ciphertexts. When y is found in this array, the oracle returns the corresponding plaintext $x[u]$, otherwise, the oracle returns $\perp$, meaning that decryption failed. This equivalence means that, with overwhelming probability, the attacker is unable to produce a valid ciphertext without calling the encryption oracle, so this equivalence represents the INT-CTXT property.

CryptoVerif can apply equivalence **int_ctxt(enc)** to transform a game as follows: it replaces occurrences of $\mathbf{enc}(salt, \mathbf{kgen}(r), r_1)$ with $\mathbf{let}\ x_{309} = salt\ \mathbf{in}\ \mathbf{let}\ z_{308} = enc(x_{309}, kgen'(r), r_1)\ \mathbf{in}\ z_{308}$, and $\mathbf{dec}(e, \mathbf{kgen}(r))$ with a lookup that looks for e in the array z and return the x_{309} in case of success and $\perp$ in case of failure.

– After the cryptographic transformation, the game is simplified again. In particular, it removes assignments on $salt_T$ and replace with $salt$.

  Then CryptoVerif applies the IND-CPA property, as shown in Figure 8. This equivalence expresses that the oracle that encrypts x is indistinguishable from a oracle that encrypts $Z(x)$, where $Z(x)$ represents a bitstring of zeros, of the same length as x.

  CryptoVerif then replace $enc(salt, kgen'(r), r_1)$ with $enc'(Z(salt), kgen'(r), r_1)$.

– After applying this transformation, the game is simplified. In particular, terms of the form $Z(x)$ are simplified to constants when the length of x is constant, which removes the dependency on x.

  Now CryptoVerif can apply the security assumption of the $\mathbf{hash}_1$ since $salt_T$ has been guaranteed to be the same as $salt$. The security property is also represented by the equivalence shown in the Figure 7, whereas h stands for $\mathbf{hash}_1$. Using this equivalence, CryptoVerif can transform a game by replacing the left-hand side with the right-hand side, just like the case of **hash**, hence, we omit the details here.

– After replacing terms with simpler terms and removing assignments that are useless and branches that cannot be executed, CryptoVerif achieves a simplified game. In this game, The sk_C appears in two branches, $sk_C = mkgen(r_{318})$ and $sk_C = mkgen(r_{316})$, while sk_T appears in just one branch, $sk_T = mkgen(r_{318})$. Then CryptoVerif renames variable sk_C into $sk_{C_{333}}$,$sk_{C_{332}}$ such that $sk_{C_{332}} = mkgen(r_{318})$ and $sk_{C_{333}} = mkgen(r_{316})$.

  CryptoVerif removes assignments on $sk_{C_{333}} = mkgen(r_{316})$ and applies equivalence of **uf_cma(mac)**, shown in Figure 9, with r_{316} and simplifies the game. However, provided key seed r_{316}, there is no corresponding sk_T in the process of **TPM**, hence, this attempt of CryptoVerif is meaningless except for some knowledge for adversary.

– After that, CryptoVerif removes assignments on $sk_{C_{332}}$ and sk_T, that is , replaces $sk_{C_{332}}$ and sk_T with $\mathbf{mkgen}(r_{318})$. Then it applies equivalence

$$
\begin{array}{l}
!^{i_k \leq n_k} \ \mathbf{new}\ r : \mathit{keyseed}; (\\
\quad !^{i_e \leq n_e} \ \mathbf{new}\ r' : \mathit{seed}; (x : \mathit{cleartex}) \rightarrow \mathbf{enc}(x, \mathbf{kgen}(r), r'), \\
\quad !^{i_d \leq n_d} (y : \mathit{ciphertext}) \rightarrow \mathbf{dec}(y, \mathbf{kgen}(r))) \\
\approx \\
!^{i_k \leq n_k} \ \mathbf{new}\ r : \mathit{keyseed}; (\\
\quad !^{i_e \leq n_e} \ \mathbf{new}\ r' : \mathit{seed}; (x : \mathit{cleartex}) \rightarrow \\
\qquad \mathbf{let}\ z : \mathit{ciphertext} = \mathbf{enc}(x, \mathbf{kgen}'(r), r')\ \mathbf{in}\ z, \\
\quad !^{i_d \leq n_d} (y : \mathit{ciphertext}) \rightarrow \\
\qquad \mathbf{find}\ u \leq n_e\ \mathbf{suchthat}\ \mathbf{defined}(x[u], r'[u], z[u]) \wedge z[u] = y \\
\qquad \mathbf{then}\ \mathbf{injbot}(x[u])\ \mathbf{else}\ \ \bot
\end{array}
\qquad \text{(INT-CTXT)}
$$

$$
\begin{array}{l}
!^{i_k \leq n_k} \ \mathbf{new}\ r : \mathit{keyseed}; (\\
\quad !^{i_e \leq n_e} (x : \mathit{cleartex}) \rightarrow \mathbf{new}\ r' : \mathit{seed}; \mathbf{enc}(x, \mathbf{kgen}(r), r') \\
\approx \\
!^{i_k \leq n_k} \mathbf{new}\ r : \mathit{keyseed}; (\\
\quad !^{i_e \leq n_e} (x : \mathit{cleartex}) \rightarrow \mathbf{new} r' : \mathit{seed}; \mathbf{enc}'(\mathbf{Z}(x), \mathbf{kgen}'(r), r')
\end{array}
\qquad \text{(IND-CPA)}
$$

Fig. 8. Definition of IND-CPA and INT-CTXT Encryption

of **uf_cma(mac)** again, but with r_{318}. The equivalence is represented in Figure 9. In this equivalence, the left-hand side chooses a random seed r and provides two oracles: the first one computes the MAC of x under key $\mathbf{mkgen}(r)$ and the second one checks whether the ma is the MAC of m. The right-hand side provides two corresponding oracles: the first one still compute the MAC of x, we use $\mathbf{mac}'$ and $\mathbf{mkgen}'$ just to prevent a repeated application of the transformation induced by this equivalence. The second one replace a MAC checks $\mathbf{check}(m, \mathbf{mkgen}(r), ma)$ with a lookup in the array x of messages whose mac has been computed with key $\mathbf{mkgen}(r)$: if m is found in the array of x and $\mathbf{check}(m, \mathbf{mkgen}(r), ma)$ successes, it returns true, otherwise, it turn false since the check fails (up to negligible probability). In other words. If the check succeeded with m not in the array x, the adversary would have forged a MAC.

Specifically, provided r_{318} is a key seed used only in terms of form **mac** $(x,\mathbf{mkgen}(r_{318}))$ and $\mathbf{check}(m, \ \mathbf{mkgen}(r_{318}), ma))$. It replace all occurrences of $\mathbf{mac}(x, \mathbf{mkgen}(r_{318}))$ with $\mathbf{mac}'(x, \mathbf{mkgen}'(r_{318}))$ and $\mathbf{check}(m, \mathbf{mkgen}(r_{318}), ma))$ with a lookup.

$!^{i_k \leq n_k}$**new** $r : mkeyseed$; (
$!^{i_m \leq n_m}(x : macinput) := \mathbf{mac}(x, \mathbf{mkgen}(r))$,
$!^{i_c \leq n_c}(m : macinput, ma : macres) := \mathbf{check}(m, \mathbf{mkgen}(r), ma))$
$\approx$
$!^{i_k \leq n_k}$**new** $r : mkeyseed$; (
$!^{i_m \leq n_m}(x : macinput) := \mathbf{mac}'(x, \mathbf{mkgen}'(r))$,
$!^{i_c \leq n_c}(m : macinput, ma : macres)$
find $u \leq n$ **suchthat** **defined**$(x[u]) \wedge (m = x[u]) \wedge \mathbf{check}'(m, \mathbf{mkgen}'(r), ma)$
then ture **else** false)

Fig. 9. Definition of UF-CMA MAC

CryptoVerif then simplifies the game and succeeds proving the desired correspondence.

The probability $P(t)$ that an attacker running in time t breaks the correspondence

$$\textbf{inj-event} : \mathbf{TPMAccept}(x, y, z) \Rightarrow \textbf{inj-event} : \mathbf{CallerRequest}(x, y, z)$$

is bounded by CryptoVerif by $P(t) \leq \frac{42.5 \times N^2}{|nonce|} + N \times Pmac(t_{G_{31}} + (N^2 + 2N - 3)t_{check} + (N^2 + 8N - 9)t_{mac} + (N - 1)t_{mkgen}, N + 9, N + 3, l) + N \times Pmac(t_{G_{24}} + t + (9N - 9)t_{check} + (3N - 3)t_{mac} + (N - 1)t_{mkgen}, 3, 9, l) + Penc(t_{G_{14}} + t, N) + Pencctxt(t_{G_{11}} + t, N, N))$ where N is the maximum number of sessions of the protocol participants, $|nonce|$ is the cardinal of the set of nonces, $Pmac(t, N, N', l)$ is the probability of breaking the UF-CMA property in time t for one key, N MAC queries, N' verification queries for massages of length at most l, $Penc(t, N)$ is the probability of breaking the IND-CPA property in time t and N encryption queries, $Pencctxt(t, N, N')$ is the probability of breaking the INT-CTXT property in time t, N encryption queries, and N' decryption queries, and $t_{G_{11}}$,$t_{G_{14}}$,$t_{G_{24}}$,$t_{G_{31}}$ are bounds on the running time of the part of the transformed games not included in the UF-CMA or INT-CTXT or IND-CPA equivalence, which are therefore considered as part of the attacker against the UF-CMA or INT-CTXT or IND-CPA equivalence, and t_{check}, t_{mac} and t_{mkgen} are the maximal runtime of one call to functions, correspondingly, **check**, **mac** and **mkgen**. The first terms of $P(t)$ comes from elimination of collisions between nonces, while the other terms come from cryptographic transformations. □

B.2 Proof of Theorem 2

Proof. Similar to the proof of theorem 1, and The probability $P(t)$ that an attacker running in time t breaks the correspondence

$$\textbf{inj} : \mathbf{CallerAccept}(x, y, z) \Rightarrow \textbf{inj} : \mathbf{TPMAcknowledgment}(x, y, z)$$

is bounded by CryptoVerif by $P(t) \leq \frac{6.5\times N^2}{|nonce|} + N \times Pmac(t_{G_{31}} + (N^2+2N-3)t_{check} + (N^2+8N-9)t_{mac} + (N-1)t_{mkgen}, N+9, N+3, l) + N \times Pmac(t_{G_{24}} + t + (9N-9)t_{check} + (3N-3)t_{mac} + (N-1)t_{mkgen}, 3, 9, l) + Penc(t_{G_{14}} + t, N) + Pencctxt(t_{G_{11}} + t, N, N))$. □

C CryptoVerif formalizations

In this section, we will show all the CryptoVerif formalizations of both **Caller** and **TPM**.

C.1 Unbound and Unsalted Session

We formalize the Caller's action as follow.

$Q_C = !^{i_C \leqslant N} c_4[i_C]();$
new $N_C : nonce;$
let $cpHash = \mathbf{hash}(hk, \mathbf{concat}_3(comCode, \mathbf{getName}(handle_{entity}), comParam))$ **in**
let $comAuth = \mathbf{mac}(\mathbf{concat}_1(cpHash, N_C, n_T, sAtt), \mathbf{getAuth}(handle_{entity}, r))$ **in**
even $CallerRequest(N_C, n_T, sAtt);$
$\overline{c_5[i_C]}(comCode, handle_{entity}, N_C, sAtt, comAuth, comParam);$
$c_8[i_C](= resCode, = handle_{entity}, n_T_next : nonce, = sAtt, resHM : macres, = resParam);$
let $rpHa = \mathbf{hash}(hk, \mathbf{concat}_4(comCode, resCode, resParam)$ **in**
if $\mathbf{check}(\mathbf{concat}_2(rpHa, n_T_next, N_C, sAtt), \mathbf{getAuth}(handle_{entity}, r), resHM)$ **then**
event $CallerAccept(N_C, n_T_next, sAtt).$

We formalize the TPM's action as follow.

$Q_T = !^{i_T \leqslant N} c_2[i_T](enhandle : keyHandle, cCode : code, rCode : code, cParam : parameter, rParam : parameter);$
new $N_T : nonce;$
$\overline{c_3[i_T]}(N_T);$
$c_6[i_T](= cCode, = enhandle, n_C : nonce, sAttRec : flags, comHM : macres, = cParam);$
if $\mathbf{getContinue}(sAttRec) = \mathbf{true}$ **then**

$\textbf{let } cpHa = \textbf{hash}(hk, \textbf{concat}_3(cCode, \textbf{getName}(enhandle),$
$cParam)) \textbf{ in}$
$\textbf{if check}(\textbf{concat}_1(cpHa, n_C, N_T, sAttRec), \textbf{getAuth}(enhandle, r),$
$comHM) \textbf{ then}$
$\textbf{even } TPMAccept(n_C, N_T, sAttRec);$
$\textbf{new } N_T_next : nonce;$
$\textbf{let } rpHash = \textbf{hash}(hk, \textbf{concat}_4(cCode, rCode, rParam) \textbf{ in}$
$\textbf{let } resAuth = \textbf{mac}(\textbf{concat}_2(rpHash, N_T_next, n_C, sAttRes),$
$\textbf{getAuth}(enhandle, r)) \textbf{ in}$
$\textbf{event } TPMAcknowledgment(n_C, N_T_next, sAttRec);$
$\overline{c_7[i_T]}(rCode, enhandle, N_T_next, sAttRec, recAuth, rParam).$

C.2 Bound Session used to access bound entity

We formalize the Caller's action as follow.

$Q_C =!^{i_C \leqslant N} c_4[i_C]();$
$\textbf{new } N_C_Start : nonce;$
$\overline{c_5[i_C]}(handle_{bind}, N_C_Start);$
$c_8[i_C](n_T : nonce);$
$\textbf{let } skseed = \textbf{hash}_1(\textbf{getAuth}(handle_{bind}, auth_{bind}),$
$\textbf{concat}_5(ATH, n_T, N_C_Start, bits)) \textbf{ in}$
$\textbf{let } sk_C = \textbf{mkgen}(skseed) \textbf{ in}$
$\textbf{new } N_C : nonce;$
$\textbf{let } cpHash = \textbf{hash}(hk, \textbf{concat}_3(comCode, \textbf{getName}(handle_{bind}),$
$comParam)) \textbf{ in}$
$\textbf{let } comAuth = \textbf{mac}(\textbf{concat}_1(cpHash, N_C, n_T, sAtt), sk_C) \textbf{ in}$
$\textbf{even } CallerRequest(N_C, n_T, sAtt);$
$\overline{c_9[i_C]}(comCode, handle_{bind}, N_C, sAtt, comAuth, comParam);$
$c_{12}[i_C](= resCode, = handle_{bind}, n_T_next : nonce, = sAtt,$
$resHM : macres, = resParam);$
$\textbf{let } rpHa = \textbf{hash}(hk, \textbf{concat}_4(comCode, resCode, resParam) \textbf{ in}$
$\textbf{if check}(\textbf{concat}_2(rpHa, n_T_next, N_C, sAtt), sk_C, resHM) \textbf{ then}$
$\textbf{event } CallerAccept(N_C, n_T_next, sAtt).$

We formalize the TPM's action as follow.

$$
\begin{aligned}
Q_T = &!^{i_T \leqslant N} c_2[i_T](bdhandle : keyHandle, cCode : code, rCode : code,\\
&\qquad\qquad cParam : parameter, rParam : parameter);\\
&\overline{c_3[i_T]}();\\
&c_6[i_T](= bdhandle, n_C_Start : nonce);\\
&\textbf{new } N_T : nonce;\\
&\textbf{let } skseed = \textbf{hash}_1(\textbf{getAuth}(bdhandle, auth_{bind}),\\
&\qquad\qquad \textbf{concat}_5(ATH, N_T, n_C_Start, bits)) \textbf{ in}\\
&\textbf{let } sk_T = \textbf{mkgen}(skseed) \textbf{ in}\\
&\overline{c_7[i_T]}(N_T);\\
&c_{10}[i_T](= cCode, = bdhandle, n_C : nonce, sAttRec : flags,\\
&\qquad comHM : macres, = cParam);\\
&\textbf{if getContinue}(sAttRec) = \textbf{true then}\\
&\textbf{let } cpHa = \textbf{hash}(hk, \textbf{concat}_3(cCode, \textbf{getName}(bdhandle),\\
&\qquad\qquad cParam)) \textbf{ in}\\
&\textbf{if check}(\textbf{concat}_1(cpHa, n_C, N_T, sAttRec), sk_T, comHM) \textbf{ then}\\
&\textbf{even } TPMAccept(n_C, N_T, sAttRec);\\
&\textbf{new } N_T_next : nonce;\\
&\textbf{let } rpHash = \textbf{hash}(hk, \textbf{concat}_4(cCode, rCode, rParam) \textbf{ in}\\
&\textbf{let } resAuth = \textbf{mac}(\textbf{concat}_2(rpHash, N_T_next, n_C, sAttRes), sk_T) \textbf{ in}\\
&\textbf{event } TPMAcknowledgment(n_C, N_T_next, sAttRec);\\
&\overline{c_{11}[i_T]}(rCode, bdhandle, N_T_next, sAttRec, recAuth, rParam).
\end{aligned}
$$

C.3 Bound Session used to access a different entity

We formalize the Caller's action as follow.

$$
\begin{aligned}
Q_C = &!^{i_C \leqslant N} c_4[i_C]();\\
&\textbf{new } N_C_Start : nonce;\\
&\overline{c_5[i_C]}(handle_{bind}, N_C_Start);\\
&c_8[i_C](n_T : nonce);\\
&\textbf{let } skseed = \textbf{hash}_1(\textbf{getAuth}(handle_{bind}, auth_{bind}),\\
&\qquad\qquad \textbf{concat}_5(ATH, n_T, N_C_Start, bits)) \textbf{ in}\\
&\textbf{let } sk_C = \textbf{mkgen}(skseed) \textbf{ in}\\
&\textbf{new } N_C : nonce;\\
&\textbf{let } cpHash = \textbf{hash}(hk, \textbf{concat}_3(comCode, \textbf{getName}(handle_{entity}),\\
&\qquad\qquad comParam)) \textbf{ in}
\end{aligned}
$$

$$\begin{aligned}
&\textbf{let } comAuth = \textbf{mac}(\textbf{concat}_1(cpHash, N_C, n_T, sAtt),\\
&\qquad\qquad\qquad \textbf{concat}_6(sk_C, \textbf{getAuth}(handle_{entity}, auth_{entity})))\ \textbf{in}\\
&\textbf{even } CallerRequest(N_C, n_T, sAtt);\\
&\overline{c_9[i_C]}(comCode, handle_{entity}, N_C, sAtt, comAuth, comParam);\\
&c_{12}[i_C](= resCode, = handle_{entity}, n_T_next : nonce, = sAtt,\\
&\qquad resHM : macres, = resParam);\\
&\textbf{let } rpHa = \textbf{hash}(hk, \textbf{concat}_4(comCode, resCode, resParam)\ \textbf{in}\\
&\textbf{if check}(\textbf{concat}_2(rpHa, n_T_next, N_C, sAtt),\\
&\qquad \textbf{concat}_6(sk_C, \textbf{getAuth}(handle_{entity}, auth_{entity})), resHM)\ \textbf{then}\\
&\textbf{event } CallerAccept(N_C, n_T_next, sAtt).
\end{aligned}$$

We formalize the TPM's action as follow.

$$\begin{aligned}
Q_T = !^{i_T \leqslant N} & c_2[i_T](bdhandle : keyHandle, enhandle : keyHandle, cCode : code,\\
&\qquad rCode : code, cParam : parameter, rParam : parameter);\\
&\overline{c_3[i_T]}();\\
&c_6[i_T](= bdhandle, n_C_Start : nonce);\\
&\textbf{new } N_T : nonce;\\
&\textbf{let } skseed = \textbf{hash}_1(\textbf{getAuth}(bdhandle, auth_{bind}),\\
&\qquad\qquad \textbf{concat}_5(ATH, N_T, n_C_Start, bits))\ \textbf{in}\\
&\textbf{let } sk_T = \textbf{mkgen}(skseed)\ \textbf{in}\\
&\overline{c_7[i_T]}(N_T);\\
&c_{10}[i_T](= cCode, = enhandle, n_C : nonce, sAttRec : flags,\\
&\qquad comHM : macres, = cParam);\\
&\textbf{if getContinue}(sAttRec) = \textbf{true then}\\
&\textbf{let } cpHa = \textbf{hash}(hk, \textbf{concat}_3(cCode, \textbf{getName}(enhandle),\\
&\qquad\qquad cParam))\ \textbf{in}\\
&\textbf{if check}(\textbf{concat}_1(cpHa, n_C, N_T, sAttRec),\\
&\qquad \textbf{concat}_6(sk_T, \textbf{getAuth}(enhandle, auth_{entity})), comHM)\ \textbf{then}\\
&\textbf{even } TPMAccept(n_C, N_T, sAttRec);\\
&\textbf{new } N_T_next : nonce;\\
&\textbf{let } rpHash = \textbf{hash}(hk, \textbf{concat}_4(cCode, rCode, rParam)\ \textbf{in}\\
&\textbf{let } resAuth = \textbf{mac}(\textbf{concat}_2(rpHash, N_T_next, n_C, sAttRes),\\
&\qquad\qquad \textbf{concat}_6(sk_T, \textbf{getAuth}(enhandle, auth_{entity})))\ \textbf{in}\\
&\textbf{event } TPMAcknowledgment(n_C, N_T_next, sAttRec);\\
&\overline{c_{11}[i_T]}(rCode, enhandle, N_T_next, sAttRec, recAuth, rParam).
\end{aligned}$$

C.4 Salted Session

We formalize the Caller's action as follow.

$Q_C =!^{i_C \leqslant N} c_4[i_C]();$

new $N_C_Start : nonce$; `new` $salt : nonce$; **new** $r_1 : seed$;

$\overline{c_5[i_C]}(handle_{bind}, N_C_Start, \mathbf{enc}(salt, tpmkey, r_1));$

$c_8[i_C](n_T : nonce);$

let $skseed = \mathbf{hash}_1(salt, \mathbf{concat}_5(ATH, n_T, N_C_Start, bits))$ **in**

let $sk_C = \mathbf{mkgen}(skseed)$ **in**

new $N_C : nonce$;

let $cpHash = \mathbf{hash}(hk, \mathbf{concat}_3(comCode, \mathbf{getName}(handle_{entity}),$
$comParam))$ **in**

let $comAuth = \mathbf{mac}(\mathbf{concat}_1(cpHash, N_C, n_T, sAtt),$
$\mathbf{concat}_6(sk_C, \mathbf{getAuth}(handle_{entity}, auth_{entity})))$ **in**

even $CallerRequest(N_C, n_T, sAtt);$

$\overline{c_9[i_C]}(comCode, handle_{entity}, N_C, sAtt, comAuth, comParam);$

$c_{12}[i_C](= resCode, = handle_{entity}, n_T_next : nonce, = sAtt,$
$resHM : macres, = resParam);$

let $rpHa = \mathbf{hash}(hk, \mathbf{concat}_4(comCode, resCode, resParam)$ **in**

if $\mathbf{check}(\mathbf{concat}_2(rpHa, n_T_next, N_C, sAtt),$
$\mathbf{concat}_6(sk_C, \mathbf{getAuth}(handle_{entity}, auth_{entity})), resHM)$ **then**

event $CallerAccept(N_C, n_T_next, sAtt).$

We formalize the TPM's action as follow.

$Q_T =!^{i_T \leqslant N} c_2[i_T](enhandle : keyHandle, cCode : code, rCode : code,$
$cParam : parameter, rParam : parameter);$

$\overline{c_3[i_T]}();$

$c_6[i_T](n_C_Start : nonce, e : ciphertext);$

new $N_T : nonce$;

let $injbot(salt_T) = dec(e, tpmkey)$ **in**

let $skseed = \mathbf{hash}_1(salt_T, \mathbf{concat}_5(ATH, N_T, n_C_Start, bits))$ **in**

let $sk_T = \mathbf{mkgen}(skseed)$ **in**

$\overline{c_7[i_T]}(N_T);$

$c_{10}[i_T](= cCode, = enhandle, n_C : nonce, sAttRec : flags,$
$comHM : macres, = cParam);$

if getContinue$(sAttRec) =$ **true then**
let $cpHa =$ **hash**$(hk,$ **concat**$_3(cCode,$ **getName**$(enhandle),$
$cParam))$ **in**
if check$($**concat**$_1(cpHa, n_C, N_T, sAttRec),$
concat$_6(sk_T,$ **getAuth**$(enhandle, auth_{entity})), comHM)$ **then**
even $TPMAccept(n_C, N_T, sAttRec);$
new $N_T_next : nonce;$
let $rpHash =$ **hash**$(hk,$ **concat**$_4(cCode, rCode, rParam)$ **in**
let $resAuth =$ **mac**$($**concat**$_2(rpHash, N_T_next, n_C, sAttRes),$
concat$_6(sk_T,$ **getAuth**$(enhandle, auth_{entity})))$ **in**
event $TPMAcknowledgment(n_C, N_T_next, sAttRec);$
$\overline{c_{11}[i_T]}(rCode, enhandle, N_T_next, sAttRec, recAuth, rParam).$

C.5 Salted and Bound Session used to access a different entity

We formalize the Caller's action as follow.

$Q_C =!^{i_C \leqslant N} c_4[i_C]();$
new $N_C_Start : nonce;$ `new` $salt : nonce;$ **new** $r_1 : seed;$
$\overline{c_5[i_C]}(handle_{bind}, N_C_Start,$ **enc**$(salt, tpmkey, r_1));$
$c_8[i_C](n_T : nonce);$
let $skseed =$ **hash**$_1($**concat**$_6(salt,$ **getAuth**$(handle_{bind}, auth_{bind})),$
concat$_5(ATH, n_T, N_C_Start, bits))$ **in**
let $sk_C =$ **mkgen**$(skseed)$ **in**
new $N_C : nonce;$
let $cpHash =$ **hash**$(hk,$ **concat**$_3(comCode,$ **getName**$(handle_{entity}),$
$comParam))$ **in**
let $comAuth =$ **mac**$($**concat**$_1(cpHash, N_C, n_T, sAtt),$
concat$_7(sk_C,$ **getAuth**$(handle_{entity}, auth_{entity})))$ **in**
even $CallerRequest(N_C, n_T, sAtt);$
$\overline{c_9[i_C]}(comCode, handle_{entity}, N_C, sAtt, comAuth, comParam);$
$c_{12}[i_C](= resCode, = handle_{entity}, n_T_next : nonce, = sAtt,$
$resHM : macres, = resParam);$
let $rpHa =$ **hash**$(hk,$ **concat**$_4(comCode, resCode, resParam)$ **in**
if check$($**concat**$_2(rpHa, n_T_next, N_C, sAtt),$
concat$_7(sk_C,$ **getAuth**$(handle_{entity}, auth_{entity})), resHM)$ **then**
event $CallerAccept(N_C, n_T_next, sAtt).$

We formalize the TPM's action as follow.

$$
\begin{aligned}
Q_T =& !^{i_T \leqslant N} c_2[i_T](bdhandle : keyHandle, enhandle : keyHandle, cCode : code,\\
&\qquad rCode : code, cParam : parameter, rParam : parameter);\\
&\overline{c_3[i_T]}();\\
&c_6[i_T](= bdhandle, n_C_Start : nonce, e : ciphertext);\\
&\textbf{new } N_T : nonce;\\
&\textbf{let } injbot(salt_T) = dec(e, tpmkey) \textbf{ in}\\
&\textbf{let } skseed = \textbf{hash}_1(\textbf{concat}_6(salt_T, \textbf{getAuth}(bdhandle, auth_{bind})),\\
&\qquad \textbf{concat}_5(ATH, N_T, n_C_Start, bits)) \textbf{ in}\\
&\textbf{let } sk_T = \textbf{mkgen}(skseed) \textbf{ in}\\
&\overline{c_7[i_T]}(N_T);\\
&c_{10}[i_T](= cCode, = enhandle, n_C : nonce, sAttRec : flags,\\
&\qquad comHM : macres, = cParam);\\
&\textbf{if getContinue}(sAttRec) = \textbf{true then}\\
&\textbf{let } cpHa = \textbf{hash}(hk, \textbf{concat}_3(cCode, \textbf{getName}(enhandle), cParam)) \textbf{ in}\\
&\textbf{if check}(\textbf{concat}_1(cpHa, n_C, N_T, sAttRec),\\
&\qquad \textbf{concat}_7(sk_T, \textbf{getAuth}(enhandle, auth_{entity})), comHM) \textbf{ then}\\
&\textbf{even } TPMAccept(n_C, N_T, sAttRec);\\
&\textbf{new } N_T_next : nonce;\\
&\textbf{let } rpHash = \textbf{hash}(hk, \textbf{concat}_4(cCode, rCode, rParam) \textbf{ in}\\
&\textbf{let } resAuth = \textbf{mac}(\textbf{concat}_2(rpHash, N_T_next, n_C, sAttRes),\\
&\qquad \textbf{concat}_7(sk_T, \textbf{getAuth}(enhandle, auth_{entity}))) \textbf{ in}\\
&\textbf{event } TPMAcknowledgment(n_C, N_T_next, sAttRec);\\
&\overline{c_{11}[i_T]}(rCode, enhandle, N_T_next, sAttRec, recAuth, rParam).
\end{aligned}
$$