Formal analysis of protocols based on TPM state registers

Stéphanie Delaune* and Steve Kremer* and Mark D. Ryan† and Graham Steel*
*LSV, ENS Cachan & CNRS & INRIA Saclay Île-de-France
†School of Computer Science, University of Birmingham, UK

Abstract—**We present a Horn-clause-based framework for analysing security protocols that use *platform configuration registers* (PCRs), which are registers for maintaining state inside the Trusted Platform Module (TPM). In our model, the PCR state space is unbounded, and our experience shows that a naïve analysis using ProVerif or SPASS does not terminate. To address this, we extract a set of instances of the Horn clauses of our model, for which ProVerif does terminate on our examples. We prove the soundness of this extraction process: no attacks are lost, that is, any query derivable in the more general set of clauses is also derivable from the extracted instances. The effectiveness of our framework is demonstrated in two case studies: a simplified version of Microsoft Bitlocker, and a digital envelope protocol that allows a user to choose whether to perform a decryption, or to verifiably renounce the ability to perform the decryption.**

I. Introduction

The Trusted Platform Module (TPM) is a security chip with a tamper resistant memory included in most modern laptops and many desktops and servers. The purpose of the TPM is to enable applications to achieve higher levels of security than can be ensured by software alone. To this end the TPM offers an application program interface (API) providing operations related to:

- *secure key management and storage*: the TPM can generate new keys, and impose restrictions on their use.
- *platform configuration registers (PCRs)*: the TPM contains several PCRs in its shielded memory. The only operation for changing the value u of a PCR is to *extend* it by a value v, resulting in the PCR value $\mathrm{h}(u, v)$. Some PCRs can also be reset to their initial values, while others can only be reset by rebooting the machine. Keys can be *locked* to a particular value of a PCR, *i.e.* key usage commands can only be executed if the PCR's current value is the one specified by the key.

These operations allow the TPM to provide a root of trust for a variety of applications involving secure storage, platform authentication, and platform measurement and reporting. For example, Microsoft's hard drive encryption system 'BitLocker' relies on the secure storage and platform measurement capabilities of the TPM to ensure that the drive is decrypted only if the machine boots in a particular state, guaranteeing that the expected OS is going to be loaded.

However, despite this rich functionality, and the fact that there are now over 300 million TPMs deployed, the take-up remains rather low. There are at least two reasons for this: one is that the specification of the API, which is an ISO/IEC standard [1] coordinated by the Trusted Computing Group [2], is very large and complex. Another is that several security vulnerabilities have been discovered in the API, (see *e.g.*, [3], [4], [5]). Both reasons make a strong case for a rigorous formal analysis of the TPM API, in order to check it for vulnerabilities, and to better understand how to build secure applications with it.

Related work: Much previous formal work on the TPM treats the API in an abstract way, giving a logical characterisation of its security properties [6], a logic for reasoning about secure systems built on TPMs [7], or a compiler for turning programs annotated with information flow labels to distributed code to run on TPM enabled machines [8]. There have also been several previous attempts to formally analyze the TPM API itself. Lin described an analysis of various fragments of the TPM API using the theorem prover Otter and the model finder Alloy [9]. He modelled several subsets of the API commands in a model which omits low level details. His results included a possible attack on the delegation model of the TPM, though experiments with a real TPM have shown that the attack is not possible [10]. Lin does discuss modelling PCR state but was unable to construct a satisfactory model for Otter [9, p. 82]. Gürgens *et al.* [3] describe an analysis of the TPM API using finite state automata, but details of their model are difficult to infer from the paper. In particular, the model fragments given do not seem to include PCR state. Coker *et al.* [11] discuss TPM API analysis work, but the details of the model have not been published. We have also recently analyzed a fragment of the TPM [12], using the applied pi calculus as a modelling language and using the ProVerif tool to automate our verification, but ignoring PCRs. In short, no previous analysis covers the verification of protocols which rely on the PCRs.

Developing a model which accounts for PCRs and related commands is challenging: one must model the state of the TPM, which can be updated, and can influence the execution of future commands. In terms of protocol analysis, one can think of the PCRs as a global state which can be read and updated by different sessions. This notion of state was already identified by Herzog [13] as a major barrier to the application of security protocol analysis tools to the verification of APIs. In a different context, Mödersheim [14] developed a protocol analysis tool which takes global state into account. However, his notion of state considers an unbounded number of stores which take a value chosen from a predetermined finite domain. In the case of PCRs we have a bounded number of stores which may take values from an unbounded domain and his techniques do not immediately apply. To be able to analyze optimistic fair exchange protocols, Guttman extended the strand space model with a notion of state [15]. However, this extended model does currently not have tool support. In parallel with the work described here, Arapinis, Ritter and Ryan [16] have extended the process language of ProVerif to allow one to model global state. Their work would allow us to describe the TPM in a process language, and derive clauses automatically. However, ProVerif will not terminate on the clauses they produce, for reasons that we identify and solve in this paper.

Our approach and contributions: We model a fragment of the TPM including key management and key usage commands, taking into account operations for setting and reading PCRs, in first-order logic. Our modelling and verification techniques follow previous work by Weidenbach using SPASS [17] and in particular Blanchet using the tool ProVerif [18]. In this approach one generally considers a unary predicate $\mathsf{att}(m)$ for modelling that the adversary has knowledge of message m. To allow ourselves to model a PCR, we consider a binary predicate att; the fact $\mathsf{att}(u, m)$ means that the attacker can reach a state where the PCR has value u and where the attacker knows message m. Unfortunately, the resolution algorithms of SPASS and ProVerif quickly encounter non-termination problems when we run them on a model of the TPM using such binary predicates. We therefore prove that for a class of *k-stable* clauses, we can safely bound the number of times a PCR may be extended between two resets: we show that if there exists an attack then there exists also an attack which only considers such "small" PCR values. This allows us to specialise the clauses of our model in a way such that ProVerif terminates.

We also give syntactic conditions which are sufficient to show that the clauses in the two case studies we consider are k-stable. Our first case study is a simplified version of the *BitLocker protocol* [19], focusing on the usage of the PCR to build a *chain of trust*. The second protocol is a secure *envelope protocol* [20]. Both protocols crucially rely on the use of the PCR and we are able to prove their correctness using ProVerif.

Outline of the paper: In Section II, we give the formal grammar, and some intuitions about how we use it to represent protocols involving the platform configuration registers (PCRs) of the TPM. We also develop a simple running example. Section III is devoted to our theoretical result, which says that ProVerif need only to consider some instances of the clauses we use to model the TPM. We prove that this is a sound abstraction, that is, that no attacks are lost. Section IV develops in more detail the rules we use to model the TPM (these are seen to satisfy the conditions for our theoretical result). Section V and VI present our two case studies. Conclusions are in Section VII.

II. Protocol representation

We use first-order Horn clauses to model the attacker and the functionality offered by the TPM, following the work of Weidenbach [17] and Blanchet [18]. To motivate and illustrate our model, we first introduce a simple PCR-based protocol. Then we formally define our model, using the simple protocol of Section II-A as a running example.

A. An introductory example

Assume that Alice has two secrets s_1 and s_2. The protocol should ensure that:

- Bob can learn one of the secrets, but not both.
- Alice commits to the secrets before knowing Bob's choice, *i.e.* Alice cannot change the secrets according to Bob's decision.
- Once Alice has committed to the secrets, Bob can open one of them without any interaction with or help from Alice.

Designing such a protocol using a TPM is rather easy. For the sake of simplicity, we assume that two key pairs $(k_1, \mathrm{pk}(k_1))$ and $(k_2, \mathrm{pk}(k_2))$ are already loaded in the TPM: one of them is locked to $\mathrm{h}(u_0, a_1)$, *i.e.* the initial PCR value u_0 which has been extended with the constant a_1, whereas the other key is locked to $\mathrm{h}(u_0, a_2)$.

By using a TPM command called $\mathsf{CertifyKey}$, Bob can obtain certificates for these keys and their lock values. When Alice receives these certificates, she uses the first public key $\mathrm{pk}(k_1)$ to encrypt s_1 and the second public key $\mathrm{pk}(k_2)$ to encrypt s_2 and sends both ciphertexts to Bob. If Bob decides to open the first secret, he extends the PCR with a_1 and uses a TPM command called Unbind to decrypt the first ciphertext. Similarly, if Bob decides to open the second ciphertext, he extends the PCR with a_2. Because an extension of a PCR cannot be undone, Bob is indeed unable to retrieve both secrets.

If Bob were able to reboot the TPM, he could obtain both secrets as follows: first he extends the PCR with a_1 and uses Unbind with k_1, then he reboots, and extends the PCR with a_2 allowing him to use Unbind with k_2. In this example, we suppose that Bob cannot reboot the TPM (perhaps because it is located on a server, out of his reach). In Section VI, we will see a more advanced protocol which allows rebooting and nevertheless avoids the problem of Bob being able to return to a given PCR value.

B. Terms

Terms represent messages that are exchanged. These terms are built inductively over a finite set of *variables* $\mathcal{X} = \{x, y, \ldots\}$, and two finite sets of function symbols $\Sigma_\mathsf{n} = \{\mathsf{a}, \mathsf{s}, \mathsf{nil}, \mathsf{u}_0, \mathsf{k}, \ldots\}$ and $\Sigma_\mathrm{f} = \{\mathrm{h}, \mathrm{aenc}, \mathrm{pk}, \ldots\}$. Variables can represent any term. Names are used to represent atomic values, such as keys and nonces. Following [18], we suppose that names are parametrized by terms: we model names as functions of messages previously received by the principal that generates the name, enriched with some additional parameters (such as some information on the current state of the principal). This is similar to the abstraction proposed in [18], and like that abstraction, it is weaker than generating a new name for each run of the protocol. Function symbols in Σ_f are used to model cryptographic primitives.

The terms are defined by the following grammar:

$M, N :=$	terms	
x	variables	$(x \in \mathcal{X})$
$\mathsf{a}[M_1, \ldots, M_k]$	name	$(\mathsf{a} \in \Sigma_\mathsf{n})$
$\mathrm{f}(M_1, \ldots, M_k)$	function application	$(\mathrm{f} \in \Sigma_\mathrm{f})$

A term is *ground* if it does not contain any variables.

We represent public key encryption by the binary function symbol aenc. The term $\mathrm{aenc}(\mathrm{pk}(k), m)$ models the encryption of message m with the public key $\mathrm{pk}(k)$. Additionally, we use the unary function pk to associate a public key to a secret key given as argument.

To extend a PCR the TPM applies a hash function to the concatenation of the current value and the new value which extends the register (see Section IV-A). We model this hash function by the binary symbol h: if u is the current value of the PCR then $\mathrm{h}(u, v)$ is the extension of the PCR with the value v.

A *substitution* is a function from variables to terms, which we extend homomorphically to terms as usual. We use postfix notation and write the application of the substitution σ to the term t as $t\sigma$. A substitution is *grounding* for t if $t\sigma$ is ground.

C. Facts

In order to represent *facts* about the messages, we consider a finite set of *predicate symbols* Σ_p.

$F :=$	facts	
$p(M_1, \ldots, M_n)$	predicate application	$(p \in \Sigma_p)$

In this paper, we consider two predicates, att and key. In most research building on a Horn clause representation of protocols, the attacker predicate $\mathsf{att}(v)$ simply models that the attacker knows the term v. In our work, we add another parameter to this predicate: informally, $\mathsf{att}(u, v)$ means that there is a reachable state in which the PCR has value u and the attacker knows v. The predicate key is used to model the content of the key table: the fact $\mathsf{key}(u, sk, pubk, v)$ means that there is a reachable state in which the PCR has value u, and the key table has an entry for secret key sk, with corresponding public key $pubk$, locked to the PCR value v.

For our introductory example, we assume that the following facts hold:

- $\mathsf{key}(\mathsf{u}_0[], \mathsf{k}_1[], \mathrm{pk}(\mathsf{k}_1[]), \mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[]))$ (F_1)
- $\mathsf{key}(\mathsf{u}_0[], \mathsf{k}_2[], \mathrm{pk}(\mathsf{k}_2[]), \mathrm{h}(\mathsf{u}_0[], \mathsf{a}_2[]))$ (F_2)
- $\mathsf{att}(\mathsf{u}_0[], \mathsf{a}_1[])$ (F_3)
- $\mathsf{att}(\mathsf{u}_0[], \mathsf{a}_2[])$ (F_4)

The two first facts model that the private keys $\mathsf{k}_1[]$ and $\mathsf{k}_2[]$ are stored on the device and locked respectively to the PCR value $\mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[])$ and $\mathrm{h}(\mathsf{u}_0[], \mathsf{a}_2[])$. The two last facts model that the values $\mathsf{a}_1[]$ and $\mathsf{a}_2[]$ are public values known by the attacker in the initial state $\mathsf{u}_0[]$.

D. Rules

Rules are used to model the attacker capabilities, the functionality offered by the TPM, and the protocol.

$\mathsf{R} :=$	rules
$F_1 \wedge \ldots \wedge F_n \rightarrow F$	implication

Intuitively, the implication $F_1 \wedge \ldots \wedge F_n \rightarrow F$ means that if all facts $F_1, \ldots, F_n$ are true, then F is also true. We sometimes write $H \rightarrow F$ where $H = \{F_1, \ldots, F_n\}$ is the set of hypotheses of the rules. A rule with no hypothesis $\rightarrow F$ is written F. A rule $\mathsf{R} = F_1 \wedge \ldots \wedge F_n \rightarrow C$ is a *tautology* if $C = F_i$ for some $i \in \{1, \ldots n\}$. We now give examples of rules for modelling attacker capabilities and TPM functionalities. The actual set of rules used in our analysis will be given in Section IV.

1) Attacker rules: As usual we assume that protocols are executed in the presence of an attacker that can intercept all messages, compute new messages from the messages it has received, and send any message it can build. Therefore we have a set of rules which reflect the attacker's capabilities of manipulating messages. For instance, for public key encryption, we have the following rules:

$$\mathsf{att}(x_p, x) \rightarrow \mathsf{att}(x_p, \mathrm{pk}(x)) \quad (\mathsf{R}_1)$$
$$\mathsf{att}(x_p, x) \wedge \mathsf{att}(x_p, y) \rightarrow \mathsf{att}(x_p, \mathrm{aenc}(x, y)) \quad (\mathsf{R}_2)$$
$$\mathsf{att}(x_p, \mathrm{aenc}(\mathrm{pk}(x), y)) \wedge \mathsf{att}(x_p, x) \rightarrow \mathsf{att}(x_p, y) \quad (\mathsf{R}_3)$$

2) TPM rules: We are interested in analyzing protocols which use the TPM, thus we need to model the actions performed by the TPM.

CertifyKey. This command allows one to obtain a certificate on a key that is stored in the device. In our example, we suppose that such a certificate is signed using a particular key $\mathsf{aik}[]$. This certificate contains the public part of the key pair (x_{sk}, x_{pk}) and also its lock value x_{pcr}.

$$\mathsf{key}(x_p, x_{sk}, x_{pk}, x_{pcr}) \rightarrow \mathsf{att}(x_p, \mathrm{certkey}(\mathsf{aik}[], x_{pk}, x_{pcr})) \quad (\mathsf{R}_4)$$

Unbind. This command allows one to retrieve the content of an encryption provided that the decryption key is stored in the key table of the TPM. Note that this command can only be executed if the PCR's current value is the one specified in the key table.

$$\begin{array}{l}\mathsf{att}(x_p, \mathrm{aenc}(x_{pk}, x_{data})) \\ \quad\wedge\ \mathsf{key}(x_p, x_{sk}, x_{pk}, x_p)\end{array} \rightarrow \mathsf{att}(x_p, x_{data}) \qquad (\mathsf{R}_5)$$

Extend. The TPM rule for extending the PCR is treated in a particular way. We have a dedicated set of *inheritance* rules for transferring the key table and the attacker knowledge when the PCR is extended.

$$\mathsf{att}(x_p, x_v) \wedge \mathsf{att}(x_p, x) \rightarrow \mathsf{att}(\mathrm{h}(x_p, x_v), x) \qquad (\mathsf{R}_6)$$

$$\mathsf{key}(x_p, x_{sk}, x_{pk}, x_{pcr}) \wedge \mathsf{att}(x_p, x_v) \rightarrow \mathsf{key}(\mathrm{h}(x_p, x_v), x_{sk}, x_{pk}, x_{pcr}) \qquad (\mathsf{R}_7)$$

The first rule can be intuitively explained as follows: whenever the attacker is able to extend the PCR with value x_v in state x_p ($\mathsf{att}(x_p, x_v)$) and the attacker knows some term in state x_p ($\mathsf{att}(x_p, x)$), then the attacker still knows that term after having extended the PCR with x_v ($\mathsf{att}(\mathrm{h}(x_p, x_v), x)$). The second rule expresses in a similar way that the key table is maintained when the PCR is extended.

3) Protocol rules: The protocol actions are modelled in a similar way to the TPM commands. Considering our running example, the role of Alice can be described by the following two rules:

$$\mathsf{att}(x_p, \mathrm{certkey}(\mathsf{aik}[], x_{pk}, \mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[]))) \rightarrow \mathsf{att}(x_p, \mathrm{aenc}(x_{pk}, \mathsf{s}_1[])) \qquad (\mathsf{R}_8)$$

$$\mathsf{att}(x_p, \mathrm{certkey}(\mathsf{aik}[], x_{pk}, \mathrm{h}(\mathsf{u}_0[], \mathsf{a}_2[]))) \rightarrow \mathsf{att}(x_p, \mathrm{aenc}(x_{pk}, \mathsf{s}_2[])) \qquad (\mathsf{R}_9)$$

The first rule can be read as: if the attacker can provide a certificate of a key bound to the PCR value $\mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[])$ then Alice will encrypt the first secret $\mathsf{s}_1[]$ using this key. The second rule is similar for the secret $\mathsf{s}_2[]$.

E. Query

Our goal is to analyse reachability properties such as secrecy: for instance, can the attacker learn $\mathsf{s}_1[]$ and $\mathsf{s}_2[]$, *i.e.* does there exist a term u such that the facts $\mathsf{att}(u, \mathsf{s}_1[])$ and $\mathsf{att}(u, \mathsf{s}_2[])$ can be derived from a given set of rules?

Definition 1 (valid derivation): Let $\mathcal{R}$ be a set of rules and $\delta = F_1, \ldots, F_n$ be a finite sequence of ground facts. We say that δ is a *valid derivation* w.r.t. $\mathcal{R}$ if for each $i \in \{1, \ldots, n\}$, we have that either $F_i = F_j$ for some $j < i$, or there exists a rule $(H \rightarrow C) \in \mathcal{R}$ and a substitution σ grounding for $H \rightarrow C$ such that

- $H\sigma \subseteq \{F_1, \ldots, F_{i-1}\}$, and
- $F_i = C\sigma$.

Definition 2 (query): A *query* $Q = \{F_1, \ldots, F_n\}$ is a set of facts. We say that Q is *satisfiable w.r.t.* $\mathcal{R}$ if there exists a substitution θ, and a valid derivation δ w.r.t. $\mathcal{R}$ such that $Q\theta \subseteq \delta$, *i.e.*, $F_1\theta, \ldots, F_n\theta$ occur in δ.

Example 1: Continuing our introductory example, we have that the query $Q_1 = \{\mathsf{att}(x, \mathsf{s}_1[])\}$ is satisfiable (with $\theta_1 = \{x \mapsto \mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[])\}$). To see this, consider the following derivation.

$$\begin{array}{ll}
\mathsf{key}(\mathsf{u}_0[], \mathsf{k}_1[], \mathrm{pk}(\mathsf{k}_1[]), \mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[])) & (F_1) \\
\mathsf{att}(\mathsf{u}_0[], \mathsf{a}_1[]) & (F_3) \\
\mathsf{att}(\mathsf{u}_0[], \mathrm{certkey}(\mathsf{aik}[], \mathrm{pk}(\mathsf{k}_1[]), \mathrm{h}(\mathsf{u}_0, \mathsf{a}_1[]))) & (\mathsf{R}_4) \\
\mathsf{att}(\mathsf{u}_0[], \mathrm{aenc}(\mathrm{pk}(\mathsf{k}_1[]), \mathsf{s}_1[])) & (\mathsf{R}_8) \\
\mathsf{key}(\mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[]), \mathsf{k}_1[], \mathrm{pk}(\mathsf{k}_1[]), \mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[])) & (\mathsf{R}_7) \\
\mathsf{att}(\mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[]), \mathrm{aenc}(\mathrm{pk}(\mathsf{k}_1[]), \mathsf{s}_1[])) & (\mathsf{R}_6) \\
\mathsf{att}(\mathrm{h}(\mathsf{u}_0[], \mathsf{a}_1[]), \mathsf{s}_1[]) & (\mathsf{R}_5)
\end{array}$$

This is actually a valid derivation w.r.t. the sets of rules F_i $(1 \leq i \leq 4)$ and R_j $(1 \leq j \leq 9)$. In the same way, we can show that the query $Q_2 = \{\mathsf{att}(x, \mathsf{s}_2[])\}$ is also satisfiable w.r.t. the same set of rules (with $\theta_2 = \{x \mapsto \mathrm{h}(\mathsf{u}_0[], \mathsf{a}_2[])\}$). However, the query

$$Q = \{\mathsf{att}(x, \mathsf{s}_1[]), \mathsf{att}(x, \mathsf{s}_2[])\}$$

is not satisfiable. Intuitively, this means that Bob can learn one of the secrets, but not both.

F. The ProVerif tool

In this paper we will rely on Blanchet's ProVerif tool [18] to automate our analysis. ProVerif takes protocols described in the applied pi calculus [21] as input and translates them into Horn clauses. ProVerif then applies a dedicated resolution algorithm to verify security properties. Alternatively, one can directly give the Horn clauses as input, as we do here. The reason for this is that the standard translation from the applied pi calculus would generate a unary attacker predicate, not encoding the PCR value. We may note that for some more sophisticated properties, *e.g.*, equivalence properties or when reasoning about successive global stages of a protocol, ProVerif considers more complicated predicates.

Unfortunately, even on simple examples, such as the protocol presented in Section II-A, the predicates we consider do not allow ProVerif to conclude. Manually changing resolution strategies resulted in non-termination and ProVerif's default strategy resulted in (false) hypotheses that the tool was unable to discharge. We therefore show in the following section how to restrict the state space, without losing any attacks and which allowed ProVerif to conclude on our examples.

III. Bounding the length of the PCR

In this section, we first introduce the notion of k-stable rules. We show that for this class of rules, when checking the satisfiability of a query, it is sound to restrict the search space by only considering PCR values of a bounded length, *i.e.*, having a bounded number of nested extends (Proposition 1). Note that the problem of deciding the satisfiability of a query is still undecidable in this setting. Next, we provide a syntactic criterion (Lemma 2) to ensure k-stability of a rule. This criterion allows us to conclude k-stability for all the rules we encountered in our case studies. Finally, we describe a transformation we apply to a set of k-stable rules in order to obtain an equivalent (for satisfiability of queries) set of rules on which the ProVerif tool manages to terminate on our case studies.

A. *k-stability*

Let u be a term (not necessarily ground). We define the *PCR length* of u as follows:

- $\mathsf{length_{pcr}}(\mathrm{h}(u_1, u_2)) = \mathsf{length_{pcr}}(u_1) + 1$, and
- $\mathsf{length_{pcr}}(u) = 0$ otherwise.

Let δ be a finite sequence of ground facts. We define the set $\mathsf{Pcr}(\delta)$ as follows:

$$\mathsf{Pcr}(\delta) = \{u \mid p(u, \dots) \in \delta \text{ with } p \in \Sigma_p\}.$$

Moreover, we say that a derivation δ is *k-bounded* if $\mathsf{length_{pcr}}(u) \leq k$ for any $u \in \mathsf{Pcr}(\delta)$.

Definition 3 (PCR value): A *PCR value* is either $\mathsf{u_0}[]$ or a ground term of the form $\mathrm{h}(u', v)$ where u' a PCR value.

Given a term t we denote by $t[u_1 \to u_2]$ the *replacement* of u_1 by u_2 in t, *i.e.*, the term obtained by replacing all the occurrences of u_1 by u_2 in t. This notion is extended as expected to facts, rules, derivations, and also to sets of facts and sets of rules.

Definition 4 (k-stable): Let k be an integer such that $k \geq 0$.

A fact F is *k-stable* if for any substitution θ grounding for F, for any PCR value $u = \mathrm{h}(u_1, u_2)$ such that $\mathsf{length_{pcr}}(u) > k$ we have that:

$$(F\theta)[\mathrm{h}(u_1, u_2) \to u_1] = F(\theta[\mathrm{h}(u_1, u_2) \to u_1]).$$

A rule R is *k-stable* if for any substitution θ grounding for R, for any PCR value $u = \mathrm{h}(u_1, u_2)$ such that $\mathsf{length_{pcr}}(u) > k$ we have that:

- either $(\mathsf{R}\theta)[\mathrm{h}(u_1, u_2) \to u_1] = \mathsf{R}(\theta[\mathrm{h}(u_1, u_2) \to u_1])$,
- or $(\mathsf{R}\theta)[\mathrm{h}(u_1, u_2) \to u_1]$ is a tautology.

This notion is extended as expected to sets of facts and sets of rules. It follows directly from the definition that whenever a fact or a rule is k-stable it is also $(k+1)$-stable.

Example 2: Consider the rule (R_8) of our introductory example. This rule is not 0-stable. Consider the substitution $\theta = \{x_p \mapsto \mathsf{u_0}[], x_{pk} \mapsto \mathrm{pk}(\mathsf{k_1}[])\}$, and the term $u = \mathrm{h}(\mathsf{u_0}[], \mathsf{a_1}[])$. First, the rule $(\mathsf{R}_8\theta)[\mathrm{h}(\mathsf{u_0}[], \mathsf{a_1}[]) \to \mathsf{u_0}[]]$, *i.e.*

$$\mathsf{att}(\mathsf{u_0}[], \mathrm{certkey}(\mathsf{aik}[], \mathrm{pk}(\mathsf{k_1}[]), \mathsf{u_0}[])) \to \mathsf{att}(\mathsf{u_0}[], \mathrm{aenc}(\mathrm{pk}(\mathsf{k_1}[]), \mathsf{s_1}[]))$$

is not a tautology. Moreover,

$$(\mathsf{R}_8\theta)[u \to \mathsf{u_0}[]] \neq \mathsf{R}_8(\theta[u \to \mathsf{u_0}[]])(= \mathsf{R}_8\theta).$$

The rule R_8 is however 1-stable. (As we will see, this will directly follow from Lemma 1.) As another example consider the inheritance rule

$$\mathsf{att}(x_p, x_v) \wedge \mathsf{att}(x_p, x) \to \mathsf{att}(\mathrm{h}(x_p, x_v), x)$$

This rule is 1-stable (as a direct consequence of Lemma 2) and hence, k-stable for any $k \geq 1$.

Proposition 1: Let $\mathcal{R}$ be a finite set of rules and Q be a query such that $\mathcal{R}$ and Q are k-stable. If Q is satisfiable then there exists a k-bounded derivation witnessing this fact.

Proof: Let δ be a derivation witnessing the fact that Q is satisfiable such that the multiset

$$S(\delta) = \{\mathsf{length_{pcr}}(u) \mid u \in \mathsf{Pcr}(\delta)\}$$

is minimal w.r.t. the multiset inclusion. If $s \leq k$ for any $s \in S(\delta)$, then we easily conclude. Otherwise, let $u_{max} \in \mathsf{Pcr}(\delta)$ be such that $\mathsf{length_{pcr}}(u_{max})$ is maximal. Note that $\mathsf{length_{pcr}}(u_{max}) > k$ and thus there exist u_1, u_2 such that $u_{max} = \mathrm{h}(u_1, u_2)$.

We show that $\delta' = \delta[\mathrm{h}(u_1, u_2) \to u_1]$ is a valid derivation witnessing the fact that Q is satisfiable. Moreover, we have that $S(\delta')$ is smaller than $S(\delta)$ which contradicts the minimality of $S(\delta)$.

First, we show that δ' is a valid derivation. We show this result by induction on the length ℓ of the derivation δ.
Base case: $\ell = 0$. In such a case, the result trivially holds.
Induction step: $\ell \geq 1$. In such a case, we have that:

- $\delta = \delta_0; F_\ell$, and
- $\delta' = \delta_0[u_{max} \to u_1]; F_\ell[u_{max} \to u_1]$.

By induction hypothesis, we know that

$$\delta_0' = \delta_0[u_{max} \to u_1]$$

is a valid derivation. Moreover, we know that there exist $H \to C \in \mathcal{R}$, and a substitution θ such that $H\theta \subseteq \delta_0$ and $F_\ell = C\theta$. Let $\rho = [u_{max} \to u_1]$ and $\theta' = \theta\rho$. Since $\mathcal{R}$ is k-stable, we are in one of the following cases:

1) $(R\theta)\rho = R(\theta\rho)$;
2) $(R\theta)\rho$ is a tautology, *i.e.* $(C\theta)\rho \in (H\theta)\rho$.

In Case 1, we have that $H\theta' = (H\theta)\rho \subseteq \delta_0'$ and $C\theta' = F_\ell\rho$. This allows us to conclude that $\delta' = \delta_0'; F_\ell\rho$ is a valid derivation. In Case 2, we have that

$$F_\ell\rho = (C\theta)\rho \in (H\theta)\rho \subseteq \delta_0'.$$

Thus, in both cases, we conclude.

Second, we show that δ' satisfies the query Q. Let $Q = \{F_1, \dots, F_n\}$ where $F_1, \dots, F_n$ are facts. Since δ is a derivation witnessing the fact that Q is satisfiable, there exists a substitution θ grounding for Q such that $Q\theta \subseteq \delta$. Let $\rho = [u_{max} \to u_1]$ and $\theta' = \theta\rho$. Since Q is k-stable, we have that $(Q\theta)\rho = Q\theta'$. Hence, we easily deduce that δ' is a derivation witnessing the fact that the query Q is satisfiable. ■

B. Syntactic criteria

In this section, we give syntactic criteria that allow us to conclude the k-stability of all of the rules we encountered during our case studies. The first criterion (see Lemma 1) allows us to show that a fact (or a rule) is k-stable and conclude also that most of our clauses are k-stable. However, this simple criterion is not satisfied by the inheritance rules. Hence we develop further criteria (see Lemma 2) which allow us to accommodate these rules. Proofs of these two lemmas are available in Appendix.

We denote by $st(t)$ the set of *subterms* of a term t. This notation is extended as expected to a fact.

Lemma 1: Let F be a fact and $k \geq 0$ be an integer such that for any subterm $v = \mathrm{h}(v_1, v_2) \in st(F)$, we have that $\mathsf{length}_{\mathsf{pcr}}(v) \leq k$ and $v_1 \notin \mathcal{X}$, *i.e.*, v_1 is not a variable. Then the fact F is k-stable.

However, this simple criterion is not satisfied by the inheritance rules. Hence we develop a further criteria which allow us to accommodate these rules.

Lemma 2: Let $k \geq 0$ be an integer and $\mathsf{R} = H \to C$ be a rule such that:

1) for all $\mathrm{h}(v_1, v_2) \in st(\mathsf{R})$, $\mathsf{length}_{\mathsf{pcr}}(v_1, v_2) \leq k$;
2) for all $\mathrm{h}(v_1, v_2) \in st(H)$, we have that $v_1 \notin \mathcal{X}$;
3) for all $\mathrm{h}(v_1, v_2) \in st(C)$ such that $v_1 \in \mathcal{X}$, we have that $C[\mathrm{h}(v_1, v_2) \to v_1] \in H$.

Then, we have that the rule R is k-stable.

Example 3: Thanks to Lemma 2, we can easily deduce that the facts F_i $(1 \leq i \leq 4)$, the rules R_j $(1 \leq j \leq 9)$, and the query $Q = \{\mathsf{att}(x, \mathsf{s_1}[]), \mathsf{att}(x, \mathsf{s_2})\}$ are 1-stable. Hence, when checking the satisfiability of the query Q, it is sufficient to consider 1-bounded derivations.

C. Transformations

We explain how a set of k-stable rules can be transformed into another equivalent set of rules that is more suitable for analysis with a tool such as ProVerif. We first introduce the notion of a k-complete set of rules. Intuitively, each rule can be replaced by a k-complete set of rules, while allowing the same set of k-bounded derivations.

Definition 5 (k-complete set of instances): Let $\mathcal{R}'$ be a finite set of rules. We say that $\mathcal{R}'$ is a *k-complete set of instances* of a rule R if each rule in $\mathcal{R}'$ is an instance of R, and for any substitution θ grounding for R such that

$$\mathsf{Pcr}(\mathsf{R}\theta) \subseteq \{u \mid u \text{ is a PCR value and } \mathsf{length}_{\mathsf{pcr}}(u) \leq k\}$$

we have that there exists $\mathsf{R}' \in \mathcal{R}'$ and a substitution θ' grounding for R' such that $\mathsf{R}'\theta' = \mathsf{R}\theta$.

We extend this definition in a natural way to queries and finite sets of rules.

Example 4: Going back to our introductory example, we will replace the inheritance rules by the following instances of them:

$$\mathsf{att}(\mathsf{u_0}[], x_v) \wedge \mathsf{att}(\mathsf{u_0}[], x) \to \mathsf{att}(\mathrm{h}(\mathsf{u_0}[], x_v), x) \quad (\mathsf{R}'_6)$$

$$\mathsf{key}(\mathsf{u_0}[], x_{sk}, x_{pk}, x_{pcr}) \wedge \mathsf{att}(\mathsf{u_0}[], x_v) \to \mathsf{key}(\mathrm{h}(\mathsf{u_0}[], x_v), x_{sk}, x_{pk}, x_{pcr}) \quad (\mathsf{R}'_7)$$

Theorem 1: Let $\mathcal{R}$ be a finite set of rules and Q be a query that are both k-stable and such that in any valid derivation δ w.r.t. $\mathcal{R}$, for any $u \in \mathsf{Pcr}(\delta)$, we have that u is a PCR value. Let $\tilde{Q}$ (resp. $\tilde{\mathcal{R}}$) be a finite and k-complete set of instances of Q (resp. $\mathcal{R}$).

We have that Q is satisfiable w.r.t. $\mathcal{R}$ if, and only if, there exists $Q' \in \tilde{Q}$ such that Q' is satisfiable w.r.t. $\tilde{\mathcal{R}}$.

Proof: We show the two directions separately.
($\Rightarrow$) By hypothesis, we know that Q is satisfiable w.r.t. $\mathcal{R}$. Moreover, Q and $\mathcal{R}$ are both k-stable. Thanks to Proposition 1, we know that there exists a valid k-bounded derivation witnessing this fact. Let δ be such a derivation. Moreover, by hypothesis, we know that $\mathsf{Pcr}(\delta) \subseteq \{u \mid u \text{ is a PCR value}\}$. Hence, we have that

$$\mathsf{Pcr}(\delta) \subseteq \{u \mid u \text{ is a PCR value and } \mathsf{length}_{\mathsf{pcr}}(u) \leq k\}$$

Hence, we know that any instance $\mathsf{R}\theta$ of a rule in $\mathcal{R}$ that is involved in this derivation is such that

$$\mathsf{Pcr}(\mathsf{R}\theta) \subseteq \{u \mid u \text{ is a PCR value and } \mathsf{length}_{\mathsf{pcr}}(u) \leq k\}$$

Since $\tilde{\mathcal{R}}$ is a k-complete set of instances of $\mathcal{R}$, we easily deduce that there exists $\mathsf{R}' \in \tilde{\mathcal{R}}$ and a substitution θ' grounding for R' such that $\mathsf{R}\theta = \mathsf{R}'\theta'$ and thus this allows us to mimic this step. The same reasoning can be applied to deal with the query. Hence, the result.

($\Leftarrow$) Assume that there exists $Q' \in \tilde{Q}$ such that Q' is satisfiable w.r.t. $\tilde{\mathcal{R}}$. Let δ be a derivation witnessing this fact. Since all the rules in $\tilde{\mathcal{R}}$ are instances of the rules in $\mathcal{R}$, we deduce that δ is also a valid derivation w.r.t. $\mathcal{R}$. Lastly, by hypothesis, we have that there exists θ' grounding for Q' such that $Q'\theta' \subseteq \delta$. Since by definition of an instance, we have that $Q' = Q\tau$ for some substitution τ, we easily deduce that $\theta = \theta' \circ \tau$ is a grounding substitution for Q such that $Q\theta \subseteq \delta$. This allows us to conclude. ∎

Example 5: Continuing our example, consider the substitutions

- $\theta_0 = \{x_p \mapsto \mathsf{u_0}[]\}$, and
- $\theta_1 = \{x_p \mapsto \mathrm{h}(\mathsf{u_0}[], x_1)\}$.

Let $\mathcal{R}$ be the set of rules consisting of F_i for $1 \leq i \leq 4$, and R_j for $1 \leq j \leq 9$. Let $\mathcal{R}'$ be the set of rules consisting of:

- the facts F_i for $1 \leq i \leq 4$;
- the rules R_6', and R_7';
- the rules $\mathsf{R}_j\theta_0$ for $j \in \{1,2,3,4,5,8,9\}$; and
- the rules $\mathsf{R}_j\theta_1$ for $j \in \{1,2,3,4,5,8,9\}$.

It is easy to check that $\mathcal{R}'$ only contains instances of rules in $\mathcal{R}$. Moreover, we have that $\mathcal{R}'$ is a 1-complete set of instances of $\mathcal{R}$.

As we will see in the following sections, in our case studies it is clear from inspection of the rules that for any valid derivation δ, the set $\mathsf{Pcr}(\delta)$ only contains PCR values. Hence we apply the following specific transformation: we replace each rule $\mathsf{R} = H \rightarrow C$ by a set of rules defined as

$$\{\mathsf{R}[x \mapsto u] \mid x \in \mathcal{X}, p(x, t_1, \ldots, t_\ell) \in H \cup \{C\}, u \in U_k\}$$

where $U_k = \{\mathsf{u_0}[],$
$\quad \mathrm{h}(\mathsf{u_0}[], x_1),$
$\quad \ldots,$
$\quad \mathrm{h}(\ldots\mathrm{h}(\mathsf{u_0}[], x_1), \ldots, x_k)\}.$

This transformation effectively bounds the PCR length of possible PCR values that may appear as the first argument of a predicate. It follows from Theorem 1 that if the initial set of rules is k-stable then the initial and transformed set of rules are equivalent w.r.t. satisfiability of queries.

This theorem allows us to run ProVerif on the rules $\tilde{\mathcal{R}}$ instead of $\mathcal{R}$ and restrict the search space to k-bounded derivations. ProVerif is indeed capable of terminating on $\tilde{\mathcal{R}}$ in our case studies, while it does not succeed on $\mathcal{R}$. To further help ProVerif terminate, we use a selection function obtained by adding the instruction "nounif att$(*u, x)$." This corresponds to the usual selection function specified by "nounif att(x)" which ProVerif uses to avoid resolving on hypotheses of the form att(x) for variable x. (Note that ProVerif is sound for any "nounif", so we are free to use any one that helps termination.)

IV. Modelling the TPM

A. Overview of the TPM

The TPM provides a command-based API. A software process can call commands of the TPM to create and use keys, and perform other tasks related to secure reporting of the platform configuration and platform authentication. A TPM has 24 160-bit registers called *platform configuration registers* that are used to record the state of the platform. On boot, the PCRs are set to an initial value (all zeros, or all ones, depending on the PCR). PCRs are updated with the Extend command, which takes as arguments a PCR name and a value. The effect of $\mathsf{Extend}(p, x)$ is to effect the assignment $p := \mathrm{SHA1}(p||x)$, that is, the old value of the PCR is concatenated with the supplied value, and the SHA1 hash of the result is assigned as the new value of the PCR.

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, rooted in a permanently loaded key called the *Storage Root Key* (SRK). A user process can call CreateWrapKey to create a child key of any existing key. Once a key has been created, it may be loaded using LoadKey2, and then can be used in an operation requiring a key (*e.g.* Seal command). To each TPM key is associated some data that specifies the circumstances in which the key can be used:

- *Authdata* is a kind of password that authorises use of the key. In order to use a key, a user process must prove knowledge of the relevant authdata by means of an HMAC when it calls a command. The authdata is set when the key is created.
- *PCR values* constrain the state of the TPM. The TPM will use a key only if certain PCRs currently have certain values. The set of affected PCRs and values are stipulated at the time the key is created.

We illustrate these mechanisms by explaining a few TPM commands. The CreateWrapKey command takes arguments that include the parent key of the key to be created, new encrypted authdata and a set of PCRs and values to be associated with the key to be created, and other information such as the key type (sealing, binding, signature, *etc.*). It returns a blob consisting of the public part of the new key and an encrypted package; the package is encrypted with the parent key and contains the private part, the authdata of the new key, and the PCR names and associated values. Thus, the command creates the key but does not store it; it simply returns it to the user process (protected by an encryption). The newly created key is not yet available to the TPM for use.

The TPM supports several types of asymmetric key pairs. The ones we use in this paper are bind keys, storage keys, and attestation identity keys (AIKs). A bind key allows data to be encrypted outside of the TPM using the public part of the key. The private decryption part of the key can be used only by the TPM. A storage key is more restricted: even the encryption must be done inside the TPM. An AIK is a key used for signing.

To use a TPM key, it must be loaded. LoadKey2 takes as argument the key blob, 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 LoadKey2 involves a decryption by the parent key, it requires the parent key to be loaded and it requires an authorisation HMAC that proves knowledge of the parent key authdata. It also requires the PCRs to match the state stipulated at the time the parent key was created. SRK is permanently loaded and has a well-known handle value, and therefore never needs to be loaded. SRK has no PCR constraints (and therefore can be used in all states).

Once the key is loaded, an encryption command such as Seal can be used. It takes arguments including the handle of the encrypting key, the data to be encrypted, information about PCRs to which the seal should be bound, and encrypted authdata for the sealed blob. It returns a sealed blob. Unseal works the other way; it requires arguments including the handle and the sealed blob, and it returns the original data. It requires HMACs that prove knowledge of the relevant authdata, and it requires the current PCRs to match the constraints that were stipulated when the sealed blob was created, and when the sealing key was created.

B. Simplifications and abstractions

In our model, we make three main simplifications of the TPM. The first one is that we do not consider authdata in this paper. Authdata is used to prove authorisation of commands, and to authenticate the responses as coming from the TPM. In a previous paper [12], we gave a model for a fragment of the TPM API focusing on authdata issues, using the applied-pi calculus, and we mechanised the analysis using ProVerif. Here we omit authdata completely, because (informally) it is orthogonal to the state-based properties we wish to prove. This simplification may be considered as an abstraction: allowing commands to proceed without authdata is equivalent to giving all the authdata to the attacker. In other words, the security properties that we consider in this paper do not rely on the secrecy of authdata.

The second simplification concerns attestation identity keys that may be used to sign PCR values by the TPM. To make this work securely, the TPM user must create an attestation identity key (AIK), obtain a certificate on it using a trusted party known as a Privacy CA (or using the DAA protocol), and then load it. The commands MakeIdentity and ActivateIdentity are provided for this purpose. If the TPM is rebooted, AIKs that are still needed must be reloaded. We make the simplification that the relevant AIK is initially and permanently loaded in the TPM, and the user already has a certificate for it. This simplification may also be seen as an abstraction: we give the intruder more power, since the key is always available to him.

The third simplification is that we consider only one PCR, instead of 24. We believe all of the methods we propose will work for any number of PCRs, but the clauses will become greater in size and more unwieldy. The real Bitlocker protocol uses several PCRs, and we have simplified it to use only one of them. The envelope protocol only uses one PCR.

C. Modelling with Horn clauses

We now describe our modelling of the TPM commands.

1) TPM rules: As an example consider the CreateWrapKey command. This command creates a new key pair and outputs a wrap of this key under a given parent key. Moreover, the new key can be *locked* to a given PCR value, *i.e.* the key can only be used when the current PCR value is equal to this value. We model this command as follows:

$$\begin{aligned}&\mathsf{att}(x_p, x_{pcr}) \wedge \mathsf{key}(x_p, x_{sk}, x_{pk}, x_p) \rightarrow \\ &\qquad \mathsf{att}(x_p, \langle \mathrm{pk}(\mathsf{bindk}[x_{pcr}]), \\ &\qquad\qquad \mathrm{wrap}(x_{pk}, \mathsf{bindk}[x_{pcr}], \mathsf{tpmpf}[], x_{pcr})\rangle)\end{aligned}$$

The secret key being created is $\mathsf{bindk}[x_{pcr}]$, which in this example is a TPM "bind" key. The CreateWrapKey command returns a pair consisting of the public part (in the clear), and the private part along with the $\mathsf{tpmpf}[]$ secret, used to identify keys created by this TPM, and PCR value to which the key is locked, all wrapped in an encrypted blob. We use the function $\mathrm{wrap}()$ for such key blobs. An analogous clause for creating seal storage keys ($\mathsf{sealk}[x_{pcr}]$) is included in Figure 1.

In the rule for CreateWrapKey above, x_p and x_{pcr} denote the PCR value to which the old and new keys (respectively) are locked. The requirement that the parent key is loaded is modelled by the fact $\mathsf{key}(x_p, x_{sk}, x_{pk}, x_p)$. Note that the first and the last argument of the key predicate are required to be identical: the parent key may only be used if its lock value corresponds to the current PCR value. We also model a variant of this rule where the last argument of key is $\mathsf{nil}[]$ modelling that the key is not locked to any value. The conclusion of the rule models that the attacker learns a pair $\langle \mathrm{pk}(\mathsf{bindk}[x_{pcr}]), \mathrm{wrap}(x_{pk}, \mathsf{bindk}[x_{pcr}], \mathsf{tpmpf}[], x_{pcr})\rangle$. The first component of this pair is the public part of the freshly generated key. The second component is the wrap containing the freshly generated secret key, the secret constant $\mathsf{tpmpf}[]$ and the PCR value to which this key is locked.

We use the functions $\mathrm{wrap}()$ and $\mathrm{seal}()$ for the encrypted blobs produced by CreateWrapKey and Seal, respectively; and we use $\mathrm{aenc}()$ for arbitrary encryptions done in software. In reality, they are all encryptions under public keys. Our use of different function names corresponds to the fact that the first two have constant-value tags inserted with the plain text, in order that the TPM can check that they were formed in the correct way (and refuse to return the decrypted value if they were not). Note that if the attacker has the relevant secret keys, he can decrypt them all (see Section IV-C4).

Read:
$$\mathsf{att}(x_p, x) \quad \rightarrow \quad \mathsf{att}(x_p, x_p)$$

Quote:
$$\mathsf{att}(x_p, x) \quad \rightarrow \quad \mathsf{att}(x_p, \mathrm{certpcr}(\mathsf{aik}[], x_p, x))$$

CreateWrapKey:
$$\begin{array}{lcl}
\mathsf{att}(x_p, x_{pcr}) \wedge \mathsf{key}(x_p, x_{sk}, x_{pk}, \mathsf{nil}[]) & \rightarrow & \mathsf{att}(x_p, \langle \mathrm{pk}(\mathsf{bindk}[x_{pcr}]), \mathrm{wrap}(x_{pk}, \mathsf{bindk}[x_{pcr}], \mathsf{tpmpf}[], x_{pcr})\rangle) \\
\mathsf{att}(x_p, x_{pcr}) \wedge \mathsf{key}(x_p, x_{sk}, x_{pk}, x_p) & \rightarrow & \mathsf{att}(x_p, \langle \mathrm{pk}(\mathsf{bindk}[x_{pcr}]), \mathrm{wrap}(x_{pk}, \mathsf{bindk}[x_{pcr}], \mathsf{tpmpf}[], x_{pcr})\rangle) \\
\\
\mathsf{att}(x_p, x_{pcr}) \wedge \mathsf{key}(x_p, x_{sk}, x_{pk}, \mathsf{nil}[]) & \rightarrow & \mathsf{att}(x_p, \langle \mathrm{pk}(\mathsf{sealk}[x_{pcr}]), \mathrm{wrap}(x_{pk}, \mathsf{sealk}[x_{pcr}], \mathsf{tpmpf}[], x_{pcr})\rangle) \\
\mathsf{att}(x_p, x_{pcr}) \wedge \mathsf{key}(x_p, x_{sk}, x_{pk}, x_p) & \rightarrow & \mathsf{att}(x_p, \langle \mathrm{pk}(\mathsf{sealk}[x_{pcr}]), \mathrm{wrap}(x_{pk}, \mathsf{sealk}[x_{pcr}], \mathsf{tpmpf}[], x_{pcr})\rangle)
\end{array}$$

LoadKey2:
$$\begin{array}{lcl}
\mathsf{att}(x_p, \mathrm{pk}(x_{key})) \wedge \mathsf{att}(x_p, \mathrm{wrap}(x_{pk}, x_{key}, \mathsf{tpmpf}[], x_{pcr})) \wedge \mathsf{key}(x_p, x_{sk}, x_{pk}, \mathsf{nil}[]) & \rightarrow & \mathsf{key}(x_p, x_{key}, \mathrm{pk}(x_{key}), x_{pcr}) \\
\mathsf{att}(x_p, \mathrm{pk}(x_{key})) \wedge \mathsf{att}(x_p, \mathrm{wrap}(x_{pk}, x_{key}, \mathsf{tpmpf}[], x_{pcr})) \wedge \mathsf{key}(x_p, x_{sk}, x_{pk}, x_p) & \rightarrow & \mathsf{key}(x_p, x_{key}, \mathrm{pk}(x_{key}), x_{pcr})
\end{array}$$

CertifyKey:
$$\mathsf{key}(x_p, x_{sk}, x_{pk}, y) \quad \rightarrow \quad \mathsf{att}(x_p, \mathrm{certkey}(\mathsf{aik}[], x_{pk}, y))$$

UnBind:
$$\begin{array}{lcl}
\mathsf{att}(x_p, \mathrm{aenc}(x_{pk}, x_{data})) \wedge \mathsf{key}(x_p, x_{sk}, x_{pk}, \mathsf{nil}[]) & \rightarrow & \mathsf{att}(x_p, x_{data}) \\
\mathsf{att}(x_p, \mathrm{aenc}(x_{pk}, x_{data})) \wedge \mathsf{key}(x_p, x_{sk}, x_{pk}, x_p) & \rightarrow & \mathsf{att}(x_p, x_{data})
\end{array}$$

Seal:
$$\begin{array}{lcl}
\mathsf{att}(x_p, x_{data}) \wedge \mathsf{att}(x_p, x_{pcr}) \wedge \mathsf{key}(x_p, \mathsf{sealk}[x], \mathrm{pk}(\mathsf{sealk}[x]), \mathsf{nil}[]) & \rightarrow & \mathsf{att}(x_p, \mathrm{seal}(\mathrm{pk}(\mathsf{sealk}[x]), x_{data}, \mathsf{tpmpf}[], x_{pcr})) \\
\mathsf{att}(x_p, x_{data}) \wedge \mathsf{att}(x_p, x_{pcr}) \wedge \mathsf{key}(x_p, \mathsf{sealk}[x], \mathrm{pk}(\mathsf{sealk}[x]), x_p) & \rightarrow & \mathsf{att}(x_p, \mathrm{seal}(\mathrm{pk}(\mathsf{sealk}[x]), x_{data}, \mathsf{tpmpf}[], x_{pcr}))
\end{array}$$

Unseal:
$$\begin{array}{lcl}
\mathsf{att}(x_p, \mathrm{seal}(\mathrm{pk}(\mathsf{sealk}[x]), x_{data}, \mathsf{tpmpf}[], \mathsf{nil}[])) \wedge \mathsf{key}(x_p, \mathsf{sealk}[x], \mathrm{pk}(\mathsf{sealk}[x]), \mathsf{nil}[]) & \rightarrow & \mathsf{att}(x_p, x_{data}) \\
\mathsf{att}(x_p, \mathrm{seal}(\mathrm{pk}(\mathsf{sealk}[x]), x_{data}, \mathsf{tpmpf}[], \mathsf{nil}[])) \wedge \mathsf{key}(x_p, \mathsf{sealk}[x], \mathrm{pk}(\mathsf{sealk}[x]), x_p) & \rightarrow & \mathsf{att}(x_p, x_{data}) \\
\mathsf{att}(x_p, \mathrm{seal}(\mathrm{pk}(\mathsf{sealk}[x]), x_{data}, \mathsf{tpmpf}[], x_p)) \wedge \mathsf{key}(x_p, \mathsf{sealk}[x], \mathrm{pk}(\mathsf{sealk}[x]), \mathsf{nil}[]) & \rightarrow & \mathsf{att}(x_p, x_{data}) \\
\mathsf{att}(x_p, \mathrm{seal}(\mathrm{pk}(\mathsf{sealk}[x]), x_{data}, \mathsf{tpmpf}[], x_p)) \wedge \mathsf{key}(x_p, \mathsf{sealk}[x], \mathrm{pk}(\mathsf{sealk}[x]), x_p) & \rightarrow & \mathsf{att}(x_p, x_{data})
\end{array}$$

Figure 1. Rules for modelling TPM commands

Some more rules used to model the TPM are in Figure 1. The full set of rules can be found in the code provided online to support this paper (see Section VII).

2) PCR extension and reboot: As already explained in Section II, we have a dedicated set of *inheritance* rules for transferring the key table and the attacker knowledge when the PCR is extended.

$$\mathsf{att}(x_p, x_v) \wedge \mathsf{att}(x_p, x) \rightarrow \mathsf{att}(\mathrm{h}(x_p, x_v), x)$$

$$\mathsf{key}(x_p, x_{sk}, x_{pk}, x_{pcr}) \wedge \mathsf{att}(x_p, x_v) \rightarrow \mathsf{key}(\mathrm{h}(x_p, x_v), x_{sk}, x_{pk}, x_{pcr})$$

It is also possible to reboot the TPM, thereby reverting to the initial PCR state $\mathsf{u_0}[]$. In that case the attacker knowledge is preserved. We therefore have the rule:

$$\mathsf{att}(x_p, x) \rightarrow \mathsf{att}(\mathsf{u_0}[], x)$$

3) Initial state: A set of rules provides the initial state of the attacker's knowledge, taking the form $\mathsf{att}(\mathsf{u_0}[], u)$ for a variety of terms u including $\mathrm{pk}(\mathsf{srk}[])$, $\mathsf{u_0}[]$, and a range of other constant values. There are also the following rules initialising the table of loaded keys:

$$\mathsf{key}(\mathsf{u_0}[], \mathsf{srk}[], \mathrm{pk}(\mathsf{srk}[]), \mathsf{nil}[])$$
$$\mathsf{key}(\mathsf{u_0}[], \mathsf{aik}[], \mathrm{pk}(\mathsf{aik}[]), \mathsf{nil}[])$$

The $\mathsf{nil}[]$ value in the key table indicates that these keys can be used no matter what the current PCR value is.

4) Equational theory: The usual Horn clauses exist for modelling the attacker's ability to apply constructors and destructors, for example:

$$\mathsf{att}(x_p, x_1) \wedge \mathsf{att}(x_p, x_2) \wedge \mathsf{att}(x_p, x_3) \wedge \mathsf{att}(x_p, x_4) \rightarrow \mathsf{att}(x_p, \mathrm{seal}(x_1, x_2, x_3, x_4))$$

$$\begin{array}{l}
\mathsf{att}(x_p, x_1) \wedge \mathsf{att}(x_p, \mathrm{seal}(\mathrm{pk}(x_1), x_2, x_3, x_4)) \rightarrow \mathsf{att}(x_p, x_2) \\
\mathsf{att}(x_p, x_1) \wedge \mathsf{att}(x_p, \mathrm{seal}(\mathrm{pk}(x_1), x_2, x_3, x_4)) \rightarrow \mathsf{att}(x_p, x_3) \\
\mathsf{att}(x_p, x_1) \wedge \mathsf{att}(x_p, \mathrm{seal}(\mathrm{pk}(x_1), x_2, x_3, x_4)) \rightarrow \mathsf{att}(x_p, x_4)
\end{array}$$

V. The Bitlocker protocol

In this section we present the details of our first example, which is a 'trusted boot' protocol for use with full disk encryption. It is based Microsoft Bitlocker, though we make several simplifications to the protocol as we explain below.

After describing the protocol, we present our model, and then the results we obtained using ProVerif and the method of Section III.

A. Description

Microsoft's Bitlocker [19] can operate in various modes involving the use of passwords, USB tokens and the TPM. We will model the mode that just uses the TPM. The hard drive of our machine is assumed to be encrypted under a volume encryption key (VEK), which is in turn encrypted with the volume master key (VMK). At boot time, an immutable section of firmware called the pre-BIOS takes control. It measures (*i.e.*, takes a hash of) the BIOS, and extends the hash value into a PCR. The BIOS similarly measures and extends into the PCR the values of other components before passing control to them. Those components repeat the process, resulting in a "trust chain". The TPM manages access to the VMK by sealing it against PCR values corresponding to the correct boot state of the machine. If the correct boot state is achieved, the active component can retrieve the VMK by unsealing it. To prevent future unauthorised retrievals, it extends a further "deny" value into the PCR. An attacker can replace components such as the BIOS and the bootloader, but in doing so he in theory destroys the ability to achieve the correct boot state, and the VMK remains inaccessible.

We will model the correct boot state as being determined by the combination of the BIOS and the bootloader, and we will assume a single PCR is used, though the real Bitlocker measures several other components of the OS and uses multiple PCRs. There are some further details we ignore in our modelling. There are protocols for key management and recovery that we don't model. Finally we simplify the setup process to a single command executed by our honest user Alice, who seals the VMK while the machine is in a "trusted state", *i.e.*, with the correct BIOS and bootloader.

B. Modelling

The attacker may tamper with the BIOS or the bootloader, but we assume that the TPM will correctly measure a tampered BIOS and extend the PCR as appropriate before executing it - there is no way for the attacker to tamper between measurement and execution. Similarly, the attacker can tamper with the bootloader, but if the correct BIOS has been loaded, the BIOS will measure the bootloader before executing it. If the correct BIOS and bootloader are loaded, the attacker cannot interfere until they have stopped executing. Our model contains constants $\mathsf{bios}[]$ and $\mathsf{loader}[]$ for the correct BIOS and bootloader and $\mathsf{bios\_rogue}[]$ and $\mathsf{loader\_rogue}[]$ representing code controlled by the attacker.

We then model the attackers access to the boot process by the first three rules described in Figure 2. The constant $\mathsf{u_0}[]$ is assumed to be the reset value of the PCR. One can read, for example, the second rule as stating that if the attacker know the term x in any PCR state x_p, then he can reboot the machine with the correct BIOS but with a rogue bootloader, and so obtain the same term x in a state where the PCR contains the hash of the correct BIOS extended with the rogue bootloader. Note that we do not allow him to use the reboot rule:

$$\mathsf{att}(x_p, x) \rightarrow \mathsf{att}(\mathsf{u_0}[], x),$$

since this corresponds to rebooting the machine without running the BIOS, which is not possible. The additional rule described in Figure 2 allows us to model Alice setting up the drive encryption in a trusted state.

In addition to these rules, the attacker is able to use the commands of the TPM given in Figure 1, and the extension rules (see Section IV-C2).

Our security property is posed as the reachability of a state in which the attacker knows the volume master key VMK (the PCR may have any value):

$$\mathsf{att}(x_p, \mathsf{vmk}[x])$$

C. Results of our analysis

Using the syntactic criteria from Section III-B we observe that all rules and queries are 3-stable. Hence, by Theorem 1 it is sound to replace the rules by a set of 3-complete rules which only yield 3-bounded derivations. We illustrate our transformation on one rule (see Figure 2). On the resulting set of rules ProVerif quickly concludes that the protocol is safe, giving that the VMK remains secret for unbounded reboots and PCR extends. If we change the model to additionally allow the attacker to reboot into a 'clean' PCR state by adding the rule

$$\mathsf{att}(x_p, x) \rightarrow \mathsf{att}(\mathsf{u_0}[], x)$$

we obtain an attack as would be expected.

VI. The envelope protocol

A. Description

We analyse the *envelope protocol* from [20] which allows Alice to provide digital data to Bob in such a way that Bob has only one of two possible actions available to him:

- He can access the data without any further action from Alice.
- Alternatively, he can revoke his right to access the data, and in this case he is able to prove to Alice that he did not and can no longer obtain access to the data.

To achieve this, Bob is required to have a TPM. Alice gives Bob the data, encrypted with a key on Bob's TPM. The TPM allows Bob to decrypt the data, or to prove to Alice that he did not and can no longer do so.

The protocol works as follows.

PCR reboot rules:

$$\mathsf{att}(x_p, x) \rightarrow \mathsf{att}(\mathrm{h}(\mathrm{h}(\mathrm{h}(\mathsf{u_0}[], \mathsf{bios}[]), \mathsf{loader}[]), \mathsf{deny}[]), x)$$
$$\mathsf{att}(x_p, x) \rightarrow \mathsf{att}(\mathrm{h}(\mathrm{h}(\mathsf{u_0}[], \mathsf{bios}[]), \mathsf{loader\_rogue}[]), x)$$
$$\mathsf{att}(x_p, x) \rightarrow \mathsf{att}(\mathrm{h}(\mathsf{u_0}[], \mathsf{bios\_rogue}[]), x)$$

Alice's role setting up the drive encryption in a trusted state:

$$\mathsf{key}(x_p, \mathsf{sealk}[x], \mathrm{pk}(\mathsf{sealk}[x]), x_{pcr}) \rightarrow \mathsf{att}(x_p, \mathsf{seal}(\mathrm{pk}(\mathsf{sealk}[x]), \mathsf{vmk}[x_{pcr}], \mathsf{tpmpf}, \mathrm{h}(\mathrm{h}(\mathsf{u_0}[], \mathsf{bios}[]), \mathsf{loader}[])))$$

A 3-complete set for the $\mathsf{att}(x_p, x) \rightarrow \mathsf{att}(\mathrm{h}(\mathsf{u_0}[], \mathsf{bios\_rogue}[]), x)$

$$\mathsf{att}(\mathsf{u_0}[], x) \rightarrow \mathsf{att}(\mathrm{h}(\mathsf{u_0}[], \mathsf{bios\_rogue}[]), x)$$
$$\mathsf{att}(\mathrm{h}(\mathsf{u_0}[], x_1), x) \rightarrow \mathsf{att}(\mathrm{h}(\mathsf{u_0}[], \mathsf{bios\_rogue}[]), x)$$
$$\mathsf{att}(\mathrm{h}(\mathrm{h}(\mathsf{u_0}[], x_1), x_2), x) \rightarrow \mathsf{att}(\mathrm{h}(\mathsf{u_0}[], \mathsf{bios\_rogue}[]), x)$$
$$\mathsf{att}(\mathrm{h}(\mathrm{h}(\mathrm{h}(\mathsf{u_0}[], x_1), x_2), x_3), x) \rightarrow \mathsf{att}(\mathrm{h}(\mathsf{u_0}[], \mathsf{bios\_rogue}[]), x)$$

Figure 2. Rules for modelling the Bitlocker protocol

1) Sealing the envelope: Alice asks Bob to reboot his platform, so that the PCR is in the initial state. Next, Alice creates an encrypted transport session with Bob's TPM and uses it to extend that PCR with a random nonce n that she has created. She keeps the value of n secret. The transport session is then closed.

Bob reads the value of the given PCR, finding it to be $\mathrm{h}(\mathsf{u_0}[], n)$ (although he does not know the value n). Then, Bob creates a bind key $(sk, \mathrm{pk}(sk))$, locked to the PCR value $\mathrm{h}(\mathrm{h}(\mathsf{u_0}, n), \mathsf{obtain}[])$, as well as a certificate attesting that this key is locked to the expected PCR value. The public key and certificate are sent to Alice. The chosen lock value ensures that the key can be used only if the PCR is extended by the value $\mathsf{obtain}[]$. Alice encrypts her data with $\mathrm{pk}(sk)$, and sends it to Bob. This protocol is illustrated in Figure 3.

2) Opening the envelope: Bob can use Extend to extend $\mathsf{obtain}[]$ into the relevant PCR. He can then use UnBind to decrypt the ciphertext sent to him by Alice, in order to obtain the data.

3) Returning the envelope: Alternatively, Bob can demonstrate that he has given up that possibility. To do that, he extends an agreed value, called $\mathsf{deny}[]$, into the TPM and requests a PCR quote, *i.e.*, a signature of the TPM attesting that the current value of the PCR is $\mathrm{h}(\mathrm{h}(\mathsf{u_0}, n), \mathsf{deny}[])$. Bob can use this quote as a proof that he can never use the key sk to decrypt the ciphertext.

B. Model

The model is built around the predicates att and key, as before. The attacker is able to use the commands of the TPM.

The security of the envelope protocol relies on the attacker's inability to achieve in subsequent reboots certain PCR values achieved in the boot in which Alice establishes her secret. Specifically, the nonce n is kept secret, and a fresh nonce n is chosen in each session of Alice's protocol. Therefore the attacker can't obtain a PCR value that involves the same n in future boots.

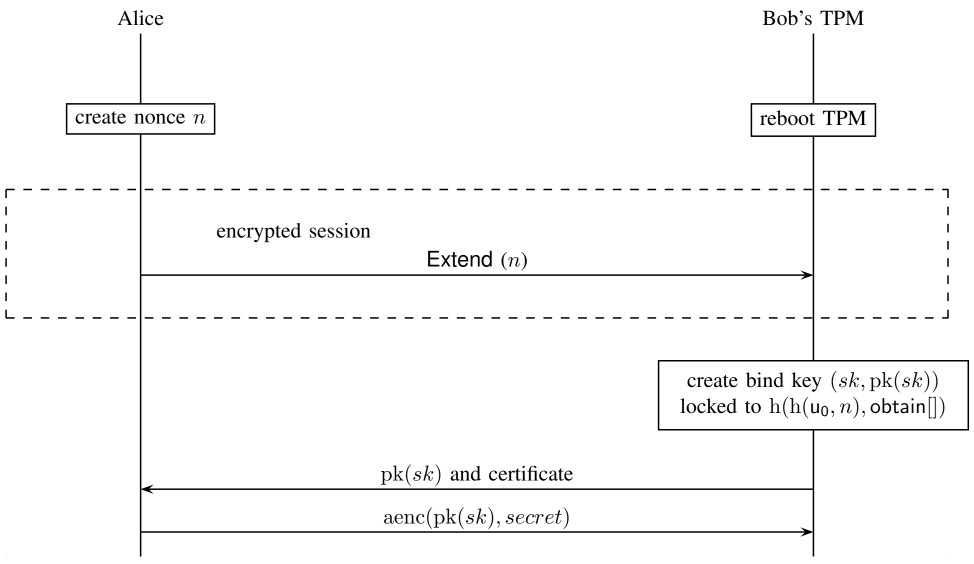
However, simply using clauses as presented before will result in a false attack. The attacker can reboot the TPM and reach the PCR value $\mathsf{u_0}[]$. With the current nonce abstraction (where nonces are parameterized) executing Alice's protocol several times in the state $\mathsf{u_0}[]$ will yield the same nonce n instead of different ones allowing the attacker to trivially both open and return the envelope. This is a well-known source of false attacks in models based on Horn clauses. To avoid this kind of false attacks we add an additional *boot parameter* to the att and key predicates. The sole role of this boot parameter is to *add freshness* to the nonce n which can be parameterized with the boot parameter. For example, the clause for CertifyKey becomes

$$\mathsf{key}(x_b, x_p, x_{sk}, x_{pk}, y) \rightarrow \mathsf{att}(x_b, x_p, \mathrm{certkey}(\mathsf{aik}[], x_{pk}, y))$$

If the TPM is rebooted, the PCR value is reset, and the boot parameter is updated. Attacker knowledge is preserved, as are the loaded keys $\mathsf{srk}[]$ and $\mathsf{aik}[]$. Therefore, we consider the clauses given in Figure 4 for rebooting the TPM. Alice's role is represented by three clauses (see Figure 4). The first two clauses allow the protocol to be started in any boot b, by creating a secret nonce $\mathsf{n}[x_b]$ and extending it into the PCR. Attacker knowledge and keys are preserved through this process. Note that with this modelling, where n is parameterized by x_b, we will have different nonces for each session of the protocol and avoid the above discussed false attack. In the third clause, Alice verifies a TPM key certificate (including information about the PCR values the key is bound to) before using the key to encrypt her secret.

C. Results of our analysis

We formulate the desired security property as a query containing the two following facts:



Alice
Bob's TPM
create nonce n
reboot TPM
encrypted session
Extend (n)
create bind key $(sk, \mathrm{pk}(sk))$
locked to $\mathrm{h}(\mathrm{h}(\mathsf{u}_0, n), \mathsf{obtain}[])$
$\mathrm{pk}(sk)$ and certificate
$\mathrm{aenc}(\mathrm{pk}(sk), secret)$

Figure 3. Sealing the envelope

PCR reboot rules:

$$\begin{aligned}
&\mathsf{att}(x_b, x_p, x) \rightarrow \mathsf{att}(\mathrm{boot}(x_b, x_p), \mathsf{u}_0[], x)\\
&\mathsf{key}(x_b, x_p, \mathsf{srk}[], \mathrm{pk}(\mathsf{srk}[]), \mathsf{nil}[]) \rightarrow \mathsf{key}(\mathrm{boot}(x_b, x_p), \mathsf{u}_0[], \mathsf{srk}[], \mathrm{pk}(\mathsf{srk}[]), \mathsf{nil}[])\\
&\mathsf{key}(x_b, x_p, \mathsf{aik}[], \mathrm{pk}(\mathsf{aik}[]), \mathsf{nil}[]) \rightarrow \mathsf{key}(\mathrm{boot}(x_b, x_p), \mathsf{u}_0[], \mathsf{aik}[], \mathrm{pk}(\mathsf{aik}[]), \mathsf{nil}[])
\end{aligned}$$

Alice's role:

$$\begin{aligned}
&\mathsf{att}(x_b, \mathsf{u}_0[], x) \rightarrow \mathsf{att}(x_b, \mathrm{h}(\mathsf{u}_0[], \mathsf{n}[x_b]), x)\\
&\mathsf{key}(x_b, \mathsf{u}_0[], x_{sk}, x_{pk}, x_{pcr}) \rightarrow \mathsf{key}(x_b, \mathrm{h}(\mathsf{u}_0[], \mathsf{n}[x_b]), x_{sk}, x_{pk}, x_{pcr})\\
&\mathsf{att}(x_b, \mathrm{h}(\mathsf{u}_0[], \mathsf{n}[x_b]), \mathrm{certkey}(\mathsf{aik}[], \mathrm{pk}(\mathsf{bindk}[y_b, y_p]), \mathrm{h}(\mathrm{h}(\mathsf{u}_0[], \mathsf{n}[x_b]), \mathsf{obtain}[]))) \rightarrow\\
&\qquad\qquad \mathsf{att}(x_b, \mathrm{h}(\mathsf{u}_0[], \mathsf{n}[x_b]), \mathrm{aenc}(\mathrm{pk}(\mathsf{bindk}[y_b, y_p]), \mathsf{secret}[x_b]))
\end{aligned}$$

Figure 4. fig:Rules for modelling the envelope protocol

- $\mathsf{att}(x_b, x_p, \mathsf{secret}[y])$, and
- $\mathsf{att}(x_b, x_p, \mathrm{certpcr}(\mathsf{aik}[], \mathrm{h}(\mathrm{h}(\mathsf{u}_0[], \mathsf{n}[y]), \mathsf{deny}[]), x))$.

It asks if Bob can obtain *both* Alice's secret *and* a certificate showing that he denied himself the secret by extending deny[] into the PCR. As for the Bitlocker protocol, we may use the results of Section III-B to show that the envelope protocol is 2-stable and apply our transformation. However, due to the additional, unbounded boot parameter, ProVerif encounters similar termination problems as for the PCR parameter in the untransformed model. Unfortunately, we are yet unable to show the soundness of a bound on the number of reboots. We nevertheless bound the number of reboots in a similar way than the depth of the PCR parameter, but the security guarantees we obtain only hold for a bounded number of reboots. We consider multiple sequential sessions of the protocol, but at most one per boot. (In our model, the TPM must be rebooted between sessions.) When we restrict the number of boot values to 3, ProVerif confirms that the above query is unsatisfiable. In the security property, the secret corresponding to boot y is paired with a certificate showing that deny[] was extended in the session involving the nonce from boot y. For larger number of possible boot values, ProVerif is unable to finish in a reasonable amount of time.

As an additional sanity check, we also considered a version where the nonce n is known to Bob. In that case, ProVerif finds the attack in which Bob first obtains the secret, then resets the PCR value, extends it with n and then deny[], and obtains the desired certificate.

VII. Conclusions

We have given a formal Horn-clause-based framework for modelling protocols of the TPM that use *platform configuration registers* (PCRs), giving a sound (*i.e.* attack preserving) transformation on the clause set that helps to address non-termination issues when using ProVerif on such models. We have presented two successful case studies: a simplified version of Microsoft Bitlocker, and the envelope protocol. In both cases, we submit a variety of queries in a variety of different configurations of the protocol, to show that it accords with intuition. The ProVerif files corresponding to our experiments are available at the url

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

As future work, we intend to generalise our method to analyse other stateful aspects of the TPM, such as the monotonic counters, and perhaps also saved contexts. We will also experiment with our method on other stateful APIs.

Acknowledgments. We gratefully acknowledge financial support from Microsoft Corporation, and from EPSRC via the projects *Verifying Interoperability Requirements in Pervasive Systems* (EP/F033540/1) and *Analysing Security and Privacy Properties* (EP/H005501/1). This work has been partly supported by project PFC ("plateforme de confiance"), pôle de compétitivité System@tic Paris-région Île-de-France.

References

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

[2] Trusted Computing Group, "TPM Specification version 1.2. Parts 1–3, revision 103," http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2007.

[3] 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)*, ser. LNCS, vol. 4734. Springer, 2007, pp. 438–453.

[4] L. Chen and M. D. Ryan, "Offline dictionary attack on TCG TPM weak authorisation data, and solution," in *Future of Trust in Computing*, D. Grawrock, H. Reimer, A. Sadeghi, and C. Vishik, Eds. Vieweg & Teubner, 2008.

[5] 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)*, 2009, pp. 201–216.

[6] M. Abadi and T. Wobber, "A logical account of NGSCB," in *24th IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, ser. Lecture Notes in Computer Science, vol. 3235. Springer, 2004, pp. 1–12.

[7] 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)*, May 2009, pp. 221–236.

[8] C. Fournet and J. Planul, "Compiling information-flow security to minimal trusted computing bases," in *Proc. 20th European Symposium on Programming (ESOP'11).*

[9] A. H. Lin, "Automated Analysis of Security APIs," Master's thesis, MIT, 2005, http://sdg.csail.mit.edu/pubs/theses/amerson-masters.pdf.

[10] 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.

[11] 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*, vol. To Appear, 2010.

[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] J. Herzog, "Applying protocol analysis to security device interfaces," *IEEE Security & Privacy Magazine*, vol. 4, no. 4, pp. 84–87, July-Aug 2006.

[14] S. Mödersheim, "Abstraction by set-membership: verifying security protocols and web services with databases," in *Proc. 17th ACM Conference on Computer and Communications Security (CCS'10).* ACM, 2010, pp. 351–360.

[15] J. D. Guttman, "Fair exchange in strand spaces," *Journal of Automated Reasoning*, 2011, to appear.

[16] M. Arapinis, E. Ritter, and M. Ryan, "Verification of stateful processes in ProVerif," in *Proc. of the 24th IEEE Computer Security Foundations Symposium (CSF'11).* IEEE Computer Society Press, 2011.

[17] C. Weidenbach, "Towards an automatic analysis of security protocols in first-order logic," in *Proc. 16th International Conference on Automated Deduction (CADE'99)*, ser. LNCS, vol. 1632. Springer, 1999, pp. 314–328.

[18] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proc. of the 14th IEEE Computer Security Foundations Workshop (CSFW'01).* Cape Breton, Nova Scotia, Canada: IEEE Computer Society Press, Jun. 2001, pp. 82–96.

[19] "Bitlocker FAQ," http://technet.microsoft.com/en-us/library/ee449438%28WS.10%29.aspx, 2011.

[20] K. Ables and M. Ryan, "Escrowed data and the digital envelope," in *Trust and Trustworthy Computing (TRUST 2010)*, ser. LNCS, vol. 6101. Springer, 2010, pp. 246–256.

[21] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Proc. 28th Symposium on Principles of Programming Languages (POPL'01)*, H. R. Nielson, Ed. London, UK: ACM Press, 2001, pp. 104–115.

Appendix

We say that a term t is *headed with* f if $t = \mathrm{f}(t_1, \ldots, t_n)$ for some terms $t_1, \ldots, t_n$.

Lemma 1: Let F be a fact and $k \geq 0$ be an integer such that for any subterm $v = \mathrm{h}(v_1, v_2) \in st(F)$, we have that $\mathsf{length}_{\mathsf{pcr}}(v) \leq k$ and $v_1 \notin \mathcal{X}$, *i.e.*, v_1 is not a variable. Then the fact F is k-stable.

Proof: First, we show that if $t = \mathrm{h}(t_1, t_2)$ is a term such that for any subterm $\mathrm{h}(v_1, v_2) \in st(t)$ we have that $v_1 \notin \mathcal{X}$, then for any substitution τ, we have that $\mathsf{length}_{\mathsf{pcr}}(t\tau) = \mathsf{length}_{\mathsf{pcr}}(t)$. We show this claim by induction on $\mathsf{length}_{\mathsf{pcr}}(t)$.

Base case: $\mathsf{length}_{\mathsf{pcr}}(t) = 1$. Since $\mathsf{length}_{\mathsf{pcr}}(t) = 1$, we have that t_1 is not headed with h. As $t_1 \notin \mathcal{X}$ we also have that $t_1\tau$ is not headed with h. Hence, we have that $\mathsf{length}_{\mathsf{pcr}}(t\tau) = \mathsf{length}_{\mathsf{pcr}}(t) = 1$.

Induction step: $\mathsf{length}_{\mathsf{pcr}}(t) > 1$. In this case we have that $t = \mathrm{h}(\mathrm{h}(t_{11}, t_{12}), t_2)$ with $\mathsf{length}_{\mathsf{pcr}}(\mathrm{h}(t_{11}, t_{12})) = \mathsf{length}_{\mathsf{pcr}}(t) - 1$. By induction hypothesis, we have that $\mathsf{length}_{\mathsf{pcr}}(t_1\tau) = \mathsf{length}_{\mathsf{pcr}}(t_1)$. Hence, we easily deduce that $\mathsf{length}_{\mathsf{pcr}}(t) = \mathsf{length}_{\mathsf{pcr}}(t\tau)$.

Now, let F be a fact and $k \geq 0$ be an integer such that for any subterm $v = \mathrm{h}(v_1, v_2) \in st(F)$, we have that $\mathsf{length}_{\mathsf{pcr}}(v) \leq k$ and $v_1 \notin \mathcal{X}$, *i.e.* v_1 is not a variable. Let θ be a substitution grounding for F and $u = \mathrm{h}(u_1, u_2)$ be a PCR value such that $\mathsf{length}_{\mathsf{pcr}}(u) > k$. Thanks to the above result, we know that $\mathsf{length}_{\mathsf{pcr}}(v\theta) = \mathsf{length}_{\mathsf{pcr}}(v) \leq k$ for any $v = \mathrm{h}(v_1, v_2) \in st(F)$. Since $\mathsf{length}_{\mathsf{pcr}}(u) > k$, we easily deduce that $u \neq v\theta$ for any $v = \mathrm{h}(v_1, v_2) \in st(F)$. Hence, we have that:

$$(F\theta)[\mathrm{h}(u_1, u_2) \to u_1] = F(\theta[\mathrm{h}(u_1, u_2) \to u_1]).$$

This allows us to conclude. ■

Lemma 2: Let $k \geq 0$ be an integer and $\mathsf{R} = H \to C$ be a rule such that:

1) for all $\mathrm{h}(v_1, v_2) \in st(\mathsf{R})$, $\mathsf{length}_{\mathsf{pcr}}(v_1, v_2) \leq k$;
2) for all $\mathrm{h}(v_1, v_2) \in st(H)$, we have that $v_1 \notin \mathcal{X}$;
3) for all $\mathrm{h}(v_1, v_2) \in st(C)$ such that $v_1 \in \mathcal{X}$, we have that $C[\mathrm{h}(v_1, v_2) \to v_1] \in H$.

Then, we have that the rule R is k-stable.

Before we prove this result, we need to introduce the notion of *positions of a term.* A *position* is a finite sequence of positive integers. The empty sequence is denoted ϵ. The set of positions $Pos(t)$ of a term t is defined inductively as

- $Pos(u) = \{\epsilon\}$ for $u \in \mathcal{X}$,
- $Pos(\mathrm{f}(t_1, \ldots, t_n)) = \{\epsilon\} \cup \bigcup_{1 \leq i \leq n} i \cdot Pos(t_i)$ for $\mathrm{f} \in \Sigma_{\mathrm{f}}$,
- $Pos(\mathsf{a}[t_1, \ldots, t_n]) = \{\epsilon\} \cup \bigcup_{1 \leq i \leq n} i \cdot Pos(t_i)$ for $\mathsf{a} \in \Sigma_{\mathsf{n}}$.

If p is a position of t then the expression $t|_p$ denotes the subterm of t at the position p, *i.e.*,

- $t|_\epsilon = t$,
- $\mathrm{f}(t_1, \ldots, t_n)|_{i \cdot p} = t_i|_p$, and
- $\mathsf{a}[t_1, \ldots, t_n]|_{i \cdot p} = t_i|_p$.

We denote by $Pos^*(t)$ the positions of t that do not correspond to a variable, *i.e.*,

$$Pos^*(t) = \{p \mid p \in Pos(t) \text{ and } t|_p \notin \mathcal{X}\}.$$

Proof: Let θ be a substitution, $u = \mathrm{h}(u_1, u_2)$ be a PCR value such that $\mathsf{length}_{\mathsf{pcr}}(u) > k$, and $\rho = [\mathrm{h}(u_1, u_2) \to u_1]$ be a replacement. It directly follows from Lemma 1 that the facts in H are k-stable, *i.e.*, $(F\theta)\rho = F(\theta\rho)$ for any fact $F \in H$.

Let $P = \{p \in Pos(C\theta) \mid (C\theta)|_p = u\}$. We distinguish two cases. Either for all $p \in P$, we have that $p \notin Pos^*(C)$. In such a case, we have that $(C\theta)\rho = C(\theta\rho)$, and thus we conclude that $(\mathsf{R}\theta)\rho = \mathsf{R}(\theta\rho)$, *i.e.*, R is k-stable. Otherwise, let $p_1, \ldots, p_n$ be the positions in $Pos^*(C) \cap P$ (note there exists at least one such position). For any $i \in \{1, \ldots, n\}$, we have that $C|_{p_i} = \mathrm{h}(v_i, v_i')$ for some terms v_i, v_i'.

First, we show that $v_1, \ldots, v_n$ are variables and $\mathrm{h}(v_1, v_1') = \ldots = \mathrm{h}(v_n, v_n')$.

Claim: Let $w = \mathrm{h}(w_1, w_2)$ be a term such that $\mathsf{length}_{\mathsf{pcr}}(w) = \ell$ and τ be a substitution grounding for w such that $\mathsf{length}_{\mathsf{pcr}}(w\tau) > \ell$. We have that either w_1 is a variable; or there exists $\mathrm{h}(\mathrm{h}(t_1, t_2), t_3) \in st(w)$ such that $t_1 \in \mathcal{X}$.

We show this result by induction on ℓ.

Base case: $\ell = 1$. In such a case, we necessarily have that $w_1 \in \mathcal{X}$.

Induction case: $\ell \geq 2$. In such a case, we have that $w = \mathrm{h}(\mathrm{h}(w_{11}, w_{12}), w_2)$. Let $w' = \mathrm{h}(w_{11}, w_{12})$. We have that $\mathsf{length}_{\mathsf{pcr}}(w') = \ell - 1$ and $\mathsf{length}_{\mathsf{pcr}}(w'\tau) > \ell - 1$. We can apply our induction hypothesis and we conclude that either $w_{11} \in \mathcal{X}$ or there exists $\mathrm{h}(\mathrm{h}(t_1'.t_2'), t_3') \in st(w')$ such that $t_1' \in \mathcal{X}$. In both case, we easily conclude that there exists $\mathrm{h}(\mathrm{h}(t_1, t_2), t_3) \in st(w)$ such that $t_1 \in \mathcal{X}$.

Then applying this claim when $w = \mathrm{h}(v_i, v_i')$ for any $i \in \{1, \ldots, n\}$, and $\tau = \theta$ allows us to conclude that v_i is either a variable or there exists $\mathrm{h}(\mathrm{h}(t_1, t_2), t_3) \in st(w)$ such that $t_1 \in \mathcal{X}$. The second case is however impossible: by Hypothesis 3 we would have that $\mathrm{h}(t_1, t_2) \in st(H)$ and $t_1 \in \mathcal{X}$ contradicting Hypothesis 2. Hence, $v_1, \ldots, v_n$ are variables. Moreover, we have that $C|_{p_i} = C|_{p_j}$ for any $i, j \in \{1, \ldots, n\}$. Indeed, otherwise, we again contradict Hypothesis 2. Hence, $\rho' = [\mathrm{h}(v_i, v_i') \to v_i]$ is uniquely defined.

Claim: We have that $(C\theta)\rho = (C\rho')(\theta\rho)$.

Let t be a term in $st(C)$. We show that $(t\theta)\rho = (t\rho')(\theta\rho)$ by structural induction on t. From this, it is then easy to conclude that $(C\theta)\rho = (C\rho')(\theta\rho)$.

Base case: t is a variable, say x. In such a case, we have that $(x\theta)\rho = x(\theta\rho)$ and $x\rho' = x$. This allows us to conclude.

Induction step: $t = \mathrm{f}(t_1, \ldots, t_m)$ for some function symbol $\mathrm{f} \in \Sigma_{\mathrm{f}}$. (The case where $t = \mathsf{a}[t_1, \ldots, t_m]$ for some symbol $\mathsf{a} \in \Sigma_{\mathsf{n}}$ can be done in a similar way but *Case 2* will be impossible in such a situation). We have to distinguish two cases:

- *Case 1: $t\theta \neq \mathrm{h}(u_1, u_2)$.* In such a case, by relying on our induction hypothesis, we have that

$$\begin{array}{rcl} (t\theta)\rho & = & (\mathrm{f}(t_1, \ldots, t_m)\theta)\rho \\ & = & \mathrm{f}(t_1\theta, \ldots, t_m\theta)\rho \\ & = & \mathrm{f}((t_1\theta)\rho, \ldots, (t_m\theta)\rho) \\ & = & \mathrm{f}((t_1\rho')(\theta\rho), \ldots, (t_m\rho')(\theta\rho)) \\ & = & \mathrm{f}(t_1\rho', \ldots, t_m\rho')(\theta\rho) \\ & = & (t\rho')(\theta\rho). \end{array}$$

  The last equality comes from the fact that $t \neq \mathrm{h}(v_i, v_i')$. Indeed, otherwise we would have that $t\theta = \mathrm{h}(u_1, u_2)$. This contradicts our hypothesis.
- *Case 2: $t\theta = \mathrm{h}(u_1, u_2)$.* First, it is easy to see that $(t\theta)\rho = u_1$. Let $p' \in Pos^*(C)$ such that $C|_{p'} = t$. We have that $t = \mathrm{h}(v_i, v_i')$, and thus $t\rho' = v_i$. Moreover, we have seen that v_i is a variable, and thus $v_i(\theta\rho) = (v_i\theta)\rho = u_1\rho = u_1$. This allows us to conclude.

Since v_i is a variable, thanks to Hypothesis 3, we know that $C\rho' = F$ for some $F \in H$. Hence, we easily deduce that $(C\rho')(\theta\rho) = F(\theta\rho) = (F\theta)\rho$. The last equality comes from the fact that F is k-stable. Lastly, we have also that $(C\theta)\rho = (C\rho')(\theta\rho)$. Hence, altogether, we deduce that $(C\theta)\rho \in (H\theta)\rho$, *i.e.* $(\mathsf{R}\theta)\rho$ is a tautology. ■