Automated Verification of Dynamic Root of Trust Protocols

Sergiu Bursuc[1], Christian Johansen[2(✉)], and Shiwei Xu[3]

[1] University of Bristol, Bristol, UK
[2] University of Oslo, Oslo, Norway
cristi@ifi.uio.no
[3] Wuhan Digital Engineering Institute, Wuhan, China

Abstract. Automated verification of security protocols based on dynamic root of trust, typically relying on protected hardware such as TPM, involves several challenges that we address in this paper. We model the semantics of trusted computing platforms (including CPU, TPM, OS, and other essential components) and of associated protocols in a classical process calculus accepted by ProVerif. As part of the formalization effort, we introduce new equational theories for representing TPM specific platform states and dynamically loaded programs.

Formal models for such an extensive set of features cannot be readily handled by ProVerif, due especially to the search space generated by unbounded extensions of TPM registers. In this context we introduce a transformation of the TPM process, that simplifies the structure of the search space for automated verification, while preserving the security properties of interest. This allows to run ProVerif on our proposed models, so we can derive automatically security guarantees for protocols running in a dynamic root of trust context.

1 Introduction

A hardware root of trust, including dynamic measurement of programs and their protected execution, is a promising concept for ensuring the integrity of a platform and the privacy of sensitive data, despite powerful software attackers [19]. This relies on the idea that hardware is more difficult to compromise than software, and therefore, it can play a crucial role in protocols for handling sensitive data. When a secure computing platform is needed, a special sequence of instructions allows for a trusted piece of hardware to attest the integrity of the software to be run and to give access to data in a protected environment.

However, turning this idea into a secure design and implementation is not easy, as various attacks have shown [13,29]. For more assurance, one could use models and tools that allow automated verification of desired properties against trusted computing protocols and implementations. One main challenge for automated verification is the size and number of components involved in running programs protected by a dynamic root of trust. Furthermore, messages of such protocols consist not only of data, but also of programs that are to be executed

on the platform, and that can be supplied by an attacker or by an honest participant. At the same time, modelling the platform configuration registers (PCR) of the trusted platform module (TPM) [20] poses problems, because PCRs can be extended an unbounded number of times. Even the most efficient symbolic methods struggle with the structure of the resulting search space [6,12].

Our contributions. We propose a formal model in the ProVerif process calculus [7] for the technology and for the security properties of a dynamic root of trust (as instantiated by Intel's Trusted Execution Technology or AMD's Secure Virtual Machine). Our model is more realistic than [12] and it covers aspects of trusted computing that [10] does not cover (Sect. 4). We show how a platform state can be naturally represented as a term in ProVerif (or applied pi-calculus [1,27]) and how operations on the platform state can be expressed as equations in a term algebra (Sects. 4.3 and 4.4). Furthermore, we show how to model the dynamic loading of protected programs. Our model is simple and does not require heavy encodings, being based on the classic idea of processes as data, with a twist to take protection into account (Sect. 4.2).

We propose a new abstraction to model the extension of PCR registers that allows automated verification for a larger class of protocols than in [12]. We show how to over-approximate the model of the TPM such that the structure of the search space is simplified, without losing possible attacks or introducing false attacks. The main idea is that we can let the attacker set the PCR to *any* value, as long as it is "big enough" (Sect. 5).

Putting the formalisation and the abstraction together, we obtain the first automated verification for a realistic model of a dynamic root of trust. As security properties, we prove code integrity (the PCR values correctly record the measurement of the platform) and secrecy of sealed data (only a designated program can access data that has been sealed for its use in a protected environment).

2   Related Work

A programming language and a logic for specifying trusted computing protocols and properties are proposed in [10]. The setting is quite expressive and it allows the analysis of protocols similar to the ones that we study in this paper. [10] does not consider the seal/unseal functions of the TPM, but their language could be extended to capture them. However, the formal analysis of [10] is manual, and considering the complexity of the proofs involved, the lack of automation can be a limitation. We also believe some of their axioms (like those linking the PCR values to a late launch action) could be decomposed into more atomic formulas, in closer relation to the computational platform. Their security properties include correctly reading PCR values and the ability of honest parties to launch roots of trust; our property of code integrity, modeled as a correspondence assertion, can be seen as an additional constraint for these two events.

The analysis of [12] is automated with ProVerif and is based on a Horn clause model. Microsoft's Bitlocker protocol is shown to preserve the secrecy of data sealed against a static sequence of PCR values. Their model considers a static

root of trust, and cannot handle dynamically loaded programs. Furthermore, there is no way to express a program that has access to data in a protected environment. Without a richer model of platform states, code integrity properties cannot be expressed either. To help with automation, [12] shows that, for a specific class of Horn clauses, it is sound to bound the number of extensions of PCR registers. Since our model is in applied pi-calculus and our security properties are different, we cannot directly rely on their result, and we propose a new way of handling the unbounded PCR extension problem.

Information-flow security and computational models. [14] presents a secure compiler for translating programs and policies into cryptographic implementations, distributed on several machines equipped with TPMs. A computational model capturing functionalities similar to ours, in conjunction with additional features such as authenticated key exchange, was recently proposed in [5]. Our models are more abstract, yet could be related to particular implementations - a closer connections between formal and computational models could be explored in future.

Unbounded search space. Several works tackle the problem of an unbounded search space for automated verification, but technically they are all based on principles that cannot be translated to PCR registers. In [25], it is shown that, for a class of Horn clauses, verification of protocols with unbounded lists can be reduced to verification of protocols with lists containing a single element. In [9], it is shown that to analyse routing protocols it is sufficient to consider topologies with at most four nodes. These are strong results, based on the fact that the elements of a list or the nodes in a route are handled uniformly by the protocol. Similar results, in a different context, are shown in [15,16]. Their reductions are based on the principle of data independence for memory stores. In [22] and respectively [2], it is shown how to handle an unbounded number of Diffie-Hellman exponentiations and respectively reencryptions in ProVerif. Surprisingly, the underlying associative-commutative properties of Diffie-Hellman help in [22], while [2] can rely on the fact that a re-encryption does not change the semantics of a ciphertext. Another case where an unbounded number of operations is problematic is file sharing [8]. In order to obtain an automated proof, [8] assumes a bound on the number of access revocations, without providing justifications for soundness. A sound abstraction for an unbounded number of revocations, in a more general setting, is proposed in [24]. Still, it is specialized to databases and it seems to rely on the same principle as several results mentioned above: it does not matter what the data is, it only matters to what set it belongs.

Tools and models for non-monotonic state. StatVerif [3] is aimed specifically for the verification of protocols relying on non-monotonic states, encoding the semantics of applied pi-calculus enriched with states into a set of Horn clauses for input to ProVerif. Tamarin [28] is based on multiset rewriting and inherently allows specification and automated reasoning for non-monotonic states, where the set of facts can both augment and decrease. SAPIC [21] takes as input

a stateful variant of applied pi-calculus and produces a multiset-based model, which is then analysed using Tamarin.

StatVerif [3], SAPIC [21], and Tamarin directly [23], have been used with success to verify security protocols that rely on non-monotonic states or trusted hardware: $PKCS\sharp11$ for key management [26], YubiKey for user authentication [32], and protocols for contract signing [17]. Our models, on the other hand, are tailored for direct input to ProVerif, while extending the scope of formal models for platform state operations and dynamic root of trust protocols based on a TPM [18–20]. It is one of our main interests for future work to see how the models of this paper can be analysed with tools like [3,21,28], in order to obtain a closer alignment with the state semantics of real systems.

3   Preliminaries

3.1   Trusted Computing

We first describe the required computing platform (hardware and software) and then describe the considered class of dynamic root of trust protocols.

A. Computing platform. We consider a general purpose computing platform equipped with a `CPU` and a `TPM` (both trusted), as well as a generic untrusted operating system.

Trusted hardware. Trusted computing relies on the `CPU` and the `TPM`[1] to perform certain operations whose integrity cannot be compromised by any software attacker. Regarding the `TPM`, two of its trusted features are fundamental for the applications that we consider in this paper: the ability to record a chain of values in its *platform configuration registers* (`PCR`) and the ability to *seal data* against specified values of the `PCR`.

The `TPM` allows the `PCR` to be *reset* only by the `CPU` or by a system reset. On the other hand, the `PCR` can be *extended* with any value by software. If a `PCR` records a value p and is extended with a value v, the new value of the `PCR` is $h((p,v))$, i.e. the result of applying a hash function to the concatenation of p and v. Crucially, these are the only two ways in which the values of a `PCR` can be modified. The role of the `PCR` for the protocols that we consider in this paper is to store the measurement of programs, recording a chain of loaded programs. When data d is *sealed* against some specified value v of the `PCR`, the `TPM` stores d internally and can release it in future only if the value recorded in its `PCR` matches the value v against which d was sealed.

For the purpose of formal verification, we are flexible about who exactly of the `CPU` or the `TPM` is doing a trusted operation, like measuring, sealing, etc. This depends on the implementation, e.g., the Intel SGX can do all the operations of a `TPM`. Changing the formalization from this paper to fit a particular implementation should be easy.

---

[1] See recent book [4] detailing the TPM version 2.0 specification and implementations.

Privileged software. When a system interrupt is triggered (e.g. by network communication or user interface action), all physical memory can be accessed by the system management interrupt (SMI) handler. This means that any memory protection mechanism, in particular the protocols that we consider in this paper, must either disable interrupts for their whole duration (not practical in general) or else rely on the fact that the SMI handler cannot be compromised. That is why the SMI handler is stored in a memory area called SMRAM, which enjoys special hardware protection. Still, as shown in [13,29], the security guarantees of trusted computing can be violated using the CPU caching mechanism to compromise the SMI handler. Roughly, these attacks work because the protection of the SMRAM is not carried on to its cached contents. A countermeasure against such attacks, that we also adopt in this paper at an abstract level, is a software transfer monitor (STM) [18]. It also resides in the SMRAM, but it cannot be cached while a dynamic root of trust is running (special registers of the CPU should ensure that), and its role is to protect some memory regions from the SMI handler.

B. Dynamic root of trust.

We consider the technology of dynamic measurement and protected execution, also called dynamic root of trust (DRT), as instantiated for example in Intel's Trusted Execution Technology (TXT) or AMD Secure Virtual Machine (SVM), and as illustrated in Fig. 1.

The goal of DRT is to establish a protected execution environment for a program, where private data can be accessed without being leaked to an attacker that controls the operating system. Assume a program, that we will call PP (called measured launch

Trust assumptions

Untrusted	Measured	Trusted
OS	INIT	CPU
SMI	PP	TPM
	STM	

Fig. 1. Execution flow in DRT

environment on Intel and secure kernel on AMD), needs to be loaded in a protected environment. The first entry point of the DRT protocol is a trusted instruction of the CPU (called GETSEC[SENTER] on Intel and SKINIT on AMD), that takes as input the program PP. To help with the establishment of a protected environment, the CPU also receives as input another program, that we will call INIT (called SINIT authenticated code module on Intel and secure loader on AMD). The DRT launch and execution sequence can then be summarized as follows:
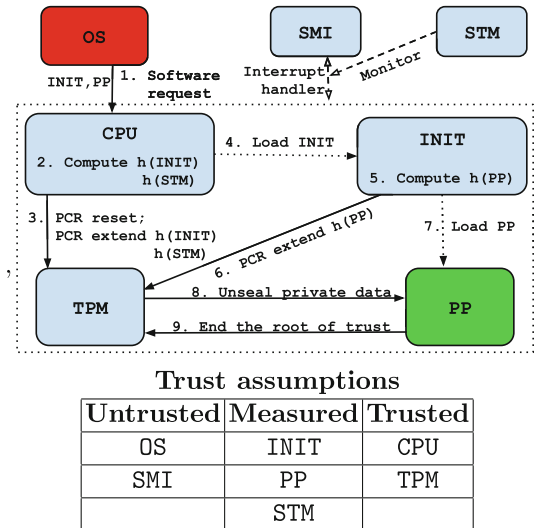
1. The CPU receives a request from the operating system containing the INIT code and the PP code. The system interrupts are disabled at this step, as an additional protection against untrusted interrupt handlers.

2–3. A software attacker that controls the operating system could compromise INIT and the STM, and that is why the CPU computes their measurement and extends the result into the TPM, to keep a trace of programs responsible for the DRT. Measuring a program means applying a hash function to its source code. This computation is performed on the CPU and is trusted, entailing that the resulting value is a correct measurement of INIT and STM. The CPU communicates with the TPM on a trusted channel and requests that the PCR is reset and extended with the resulting value (h(INIT),h(STM)).

4–7. The INIT program is loaded and it computes the measurement of the PP program, extending it into the PCR. The communication between INIT and the TPM is performed on a private channel established by the CPU. INIT also allocates protected memory for the execution of PP and loads it.

8. The PP program can re-enable interrupts once appropriate interrupt handlers are set. Furthermore, it can now request the TPM to unseal data that has been sealed against the current PCR value, and it can have access to that data in a protected environment. The communication between PP and the TPM is performed on a private channel established by the CPU.

9. Before ending its execution, the PP program extends the PCR with a dummy value, to record that the platform state is not to be trusted any more.

Since the OS is untrusted it can supply malicious programs INIT and PP. Therefore, INIT, PP and the STM are not trusted, but they are *measured*. If their measurement does not correspond to some expected *trusted* values, this will be recorded in the TPM and secret data will not be unsealed for this environment.

Security goals. Let us summarize the two main security goals of the DRT.

Code integrity: In any execution of the platform, if the measurements recorded in the PCR value of the TPM correspond to the sequence of programs $\mathcal{P}_{\text{INIT}}$, $\mathcal{P}_{\text{STM}}$, $\mathcal{P}_{\text{PP}}$, then the platform is indeed running a DRT for the protected execution of $\mathcal{P}_{\text{PP}}$ in the context of $\mathcal{P}_{\text{INIT}}$ and $\mathcal{P}_{\text{STM}}$. In particular, this means that the programs $\mathcal{P}_{\text{PP}}$, $\mathcal{P}_{\text{INIT}}$ and $\mathcal{P}_{\text{STM}}$ cannot be modified while a DRT is running.

Secrecy of sealed data: Any secret data that is sealed only against a PCR value recording the sequence of programs $\mathcal{P}_{\text{INIT}}$, $\mathcal{P}_{\text{STM}}$, $\mathcal{P}_{\text{PP}}$, is only available for the program $\mathcal{P}_{\text{PP}}$, in any execution of the platform.

3.2   ProVerif Process Calculus

We review ProVerif [6,7] and the special way in which we use (a restriction of) its input calculus in our modelling.

A. Terms, equational theories and deducibility. We consider an infinite set of *names*, $a, b, c, k, n \ldots$, an infinite set of *variables*, $x, y, z, \ldots$ and a possibly infinite set of *function symbols* $\mathcal{F}$. Names and variables are *terms*; new terms

are built by applying function symbols to names, variables and other terms. We split $\mathcal{F}$ into two disjoint sets of *public* functions $\mathcal{F}^{\text{pub}}$ and *private* functions $\mathcal{F}^{\text{priv}}$. Public functions can be applied by anyone to construct terms, including the attacker, whereas private functions can be applied only as specified by the protocol. When $\mathcal{F}^{\text{priv}}$ is not explicit, we assume that all functions are public.

A *substitution* σ is a partial function from variables to terms. The replacement of every variable x with $x\sigma$ in a term T is denoted by $T\sigma$. A *context* is a term $\mathcal{C}[\_]$ that contains a special symbol $\_$ in place of a subterm. For a context $\mathcal{C}[\_]$ and a term T, we denote by $\mathcal{C}[T]$ the term obtained by replacing $\_$ with T in $\mathcal{C}[\_]$. For any formal object $\mathcal{D}$, we denote by $\text{sig}(\mathcal{D})$ the set of function symbols appearing in $\mathcal{D}$, and by $\text{top}(T)$ the outer-most function symbol in term T.

En equational theory $\mathcal{E}$ is defined by a set of rewrite rules $U_1 \to V_1, \ldots, U_n \to V_n$, where $U_1, \ldots, U_n, V_1, \ldots, V_n$ are terms with variables. A term U rewrites to V in one step, denoted by $U \to V$, if there is a context $\mathcal{C}[\_]$, a substitution σ and an index $i \in \{1, \ldots, n\}$ such that $U = \mathcal{C}[U_i\sigma]$ and $V = C[V_i\sigma]$. Several rewrite steps from U to V are denoted by $U \to^* V$. We consider only convergent equational theories, i.e., for any term T there exists a unique non-reducible term $T\!\downarrow$ s.t. $T \to^* T\!\downarrow$. We write $U =_{\mathcal{E}} V$ iff $U\!\downarrow = V\!\downarrow$. ProVerif also allows operations on sequences: for all n, from any terms $T_1, \ldots, T_n$, one can derive the term $(T_1, \ldots, T_n)$, and conversely.

Deduction. Given an equational theory $\mathcal{E}$, a set of terms S and a term T, the ability of an attacker to obtain T from S is captured by the deduction relation $S \vdash_{\mathcal{E}} T$ (or simply $S \vdash T$ when $\mathcal{E}$ is understood) defined as being true iff:

- there exists a term $T' \in S$ such that $T' =_{\mathcal{E}} T$, or
- there are terms $T_1, \ldots, T_n$ such that $S \vdash_{\mathcal{E}} T_1, \ldots, S \vdash_{\mathcal{E}} T_n$ and a function symbol $f \in \mathcal{F}^{\text{pub}}$ such that $f(T_1, \ldots, T_n) =_{\mathcal{E}} T$.

B. Processes and operational semantics. *Processes* of the calculus are built according to Fig. 2. Replication spawns instances of a process: $!P$ is formally equivalent with $P \mid !P$. Names introduced by `new` are called *bound* or *private*; they represent the creation of fresh data. Names that are not bound are called *free*, or *public*. The term T in an input $\text{in}(U, T)$ allows to specify filters for messages received on U: a message M will be accepted only if there is a substitution σ such that $M = T\sigma$. A variable x is *free* in a process P if P neither contains x in any of its input patterns nor does it contain any term evaluation of the form $x = T$. Consecutive term evaluations can be written together as `let` $(x_1, \ldots, x_n) = (T_1, \ldots, T_n)$ `in` P. The notions of substitution, contexts and normal forms translate to processes as expected.

Operational semantics is defined as a transition system on configurations of the form $(\mathcal{N}, \mathcal{M}, \mathcal{P})$, where: $\mathcal{N}$ is a set of fresh names created during the execution of a process; $\mathcal{M}$ is the set of terms made available to the attacker; and $\mathcal{P}$ is the set of processes executing in parallel at a given point in time. We write $(\mathcal{N}, \mathcal{M}, \mathcal{P}) \to^* (\mathcal{N}', \mathcal{M}', \mathcal{P}')$ if the configuration $(\mathcal{N}', \mathcal{M}', \mathcal{P}')$ can be reached from $(\mathcal{N}, \mathcal{M}, \mathcal{P})$ in zero or more executions steps. Such a sequence of execution steps is called a trace of P.

$P, Q, R ::=$

0	null process	$\texttt{in}(U, T); P$	message input on U
$P \mid Q$	parallel composition	$\texttt{out}(U, T); P$	message output on U
$!P$	replication	if $U = V$ then P else Q	conditional
$\texttt{new}\, n; P$	name restriction	let $x = T$ in P	term evaluation

Fig. 2. Process algebra, with n a name, x a variable, and T, U, V terms.

C. Security properties. The ability of an attacker to learn a term T by interacting with a process P is denoted by $P \models \texttt{Att}(T)$, defined as true iff there exists a process Q, with $\texttt{sig}(Q) \cap \mathcal{F}^{\texttt{priv}} = \emptyset$, such that $(\mathcal{N}_{\texttt{init}}, \emptyset, \{P \mid Q\}) \rightarrow^*$ $(\mathcal{N}', \mathcal{M}', \mathcal{P}')$ and $\mathcal{M} \vdash_{\mathcal{E}} T$, for some configuration $(\mathcal{N}', \mathcal{M}', \mathcal{P}')$. Intuitively, Q represents any computation that can be performed by the attacker.

A (simplified) *correspondence assertion* [7] is a formula of the form

$$\texttt{Att}(T) \implies false \quad \text{or} \quad \texttt{Att}(T) \implies (U = V).$$

For a correspondence assertion $\texttt{Att}(T) \implies \varPhi$ as above, we have

$$P \models \texttt{Att}(T) \implies \varPhi \quad \text{iff} \quad \forall \sigma. \ [\ (P \models \texttt{Att}(T\sigma)) \implies \varPhi\sigma \ ]$$

Correspondence assertions of the first type model the *secrecy* of T, while those of second type enforce the constraint $U = V$ for deducible terms matching the pattern T (typically the terms U, V will share variables with T).

4   Formalisation

Our formal specification for the trusted computing platform and protocols described in Sect. 3.1 assumes an attacker that controls the operating system and can execute a DRT any number of times, with any INIT and PP programs. Moreover, using the CPU cache, the attacker can compromise the STM and SMI handler, and use them to access protected memory. The attacker has access to all TPM functions. However, we assume that the attacker cannot compromise the CPU nor the TPM, and that the platform state can only be modified according to the equations that we present in Sect. 4.4.

We model a system state as a term that can be updated by the CPU process, the TPM process and, once it has been output on a public channel, by the attacker. Multiple system states can be explored in parallel by the attacker, whose knowledge monotonically accumulates the set of all reachable states. This is an abstraction with respect to a real platform, where the CPU and the TPM have their own internal state, part of a global, non-monotonic system state. We also have a simplified model of TPM sealing: in reality, it relies on encryption with a TPM private key and refers to a specific system state; in our model, it is represented by the pair of public/private functions seal/unseal. For unsealing, the TPM process will require the input of a system state and check that the corresponding unseal request is valid for that state.

4.1   Cryptographic Primitives and Platform Constants

To model cryptographic primitives and various constants on the platform state, we consider the signature $\mathcal{F}_{\texttt{data}}$, where $\mathcal{F}_{\texttt{data}}^{\texttt{priv}} = \{\texttt{unseal}/2\}$ and

$$\mathcal{F}_{\texttt{data}}^{\texttt{pub}} = \{\texttt{p}_{\texttt{s}}/0, \texttt{p}_{\texttt{d}}/0, \texttt{true}/0, \texttt{false}/0, \texttt{h}/1, \texttt{senc}/2, \texttt{sdec}/2, \texttt{seal}/2\}.$$

We also consider the set of rewrite rules $\mathcal{E}_{\texttt{data}}$:

$$\texttt{sdec}(\texttt{senc}(x_{\texttt{val}}, x_{\texttt{key}}), x_{\texttt{key}}) \rightarrow x_{\texttt{val}}$$
$$\texttt{unseal}(\texttt{seal}(x_{\texttt{val}}, x_{\texttt{pcr}}), x_{\texttt{pcr}}) \rightarrow x_{\texttt{val}}$$

The constant $\texttt{p}_{\texttt{d}}$ (resp. $\texttt{p}_{\texttt{s}}$) represents the result of a dynamic (resp. static) PCR reset. A dynamic reset marks the start of a dynamic root of trust, and can only be performed by the CPU. The functions $\texttt{senc}$ and $\texttt{sdec}$, and the corresponding rewrite rule, model symmetric key encryption. The symbol $\texttt{h}$ represents a hash function. Anyone can seal a value, while the corresponding rewrite rule and the fact that $\texttt{unseal}$ is private ensure that a value can be unsealed only according to the specification of the TPM.

4.2   Dynamically Loaded Programs

To model the fact that arbitrary programs can be dynamically loaded on the platform state (e.g. for the roles of INIT and PP), we consider a new public function symbol $\texttt{prog}/1$ and an infinite signature of *private constants* $\mathcal{F}_{\mathcal{P}}$, containing a different constant n_P for every possible process P. Intuitively, the term $\texttt{prog}(n_P)$ is a public and unique identifier for the program P. In a computational model, such an identifier can for example be obtained by hashing the source code of P. The first action of a process that models a program will be to output the corresponding program identity $\texttt{prog}(n_P)$ on a public channel.

On the other hand, the constant n_P represents a *private entry point* for the program P. Specifically, we consider a private function $\texttt{get\_entry}$ and the rewrite rule $\texttt{get\_entry}(\texttt{prog}(x)) \rightarrow x$. The idea is that a trusted loader of programs (the CPU in our case) has access to the private function $\texttt{get\_entry}$ and, using this rewrite rule, it can gain access to the private entry point of any program. Now, n_P can play the role of a private channel between the trusted loader and the loaded program. Furthermore, we can store program identifiers in the platform state, to record what programs are loaded. Then, we can rely on n_P to model the ability of certain loaded programs to affect the platform state (shown in Sect. 4.4). We denote by $\mathcal{E}_{\texttt{prog}}$ the equational theory defined in this subsection: $\mathcal{F}_{\texttt{prog}} = \{\texttt{prog}/1\} \cup \mathcal{F}_{\mathcal{P}} , \quad \mathcal{E}_{\texttt{prog}} = \{\texttt{get\_entry}(\texttt{prog}(x)) \rightarrow x\}.$

4.3   Platform State

To model a platform state, we consider the signature:

$$\mathcal{F}_{\texttt{state}} = \{\texttt{state}/4, \texttt{tpm}/1, \texttt{cpu}/2, \texttt{smram}/2, \texttt{drt}/3\}$$

where all the symbols of $\mathcal{F}_{\texttt{state}}$ are private. This ensures that a platform state can be constructed or modified only according to the specification, relying on equations that we present in Subsect. 4.4. Intuitively, a term of the form

$$\texttt{state}(\texttt{tpm}(T_{\texttt{PCR}}), \texttt{cpu}(T_{\texttt{INT}}, T_{\texttt{CACHE}}), \texttt{smram}(T_{\texttt{STM}}, T_{\texttt{SMIH}}), \texttt{drt}(T_{\texttt{INIT}}, T_{\texttt{PP}}, T_{\texttt{LOCK}}))$$

represents a platform state where:

- $T_{\texttt{PCR}}$ is a term that represents the value of the PCR register of the TPM;
- $T_{\texttt{INT}}$ is the value of a register of the CPU showing if interrupts are enabled;
- $T_{\texttt{CACHE}}$ represents the contents of the CPU cache;
- $T_{\texttt{SMIH}}$ represents the program for the SMI handler and STM represents the STM program, which are located in SMRAM;
- $T_{\texttt{LOCK}}$ is showing if a dynamic root of trust is running;
- $T_{\texttt{INIT}}$ represents the INIT program;
- $T_{\texttt{PP}}$ represents the protected program PP.

4.4 Read and Write Access

The read access is universal: any agent who has access to a platform state

$$\texttt{state}(\texttt{tpm}(T_{\texttt{PCR}}), \texttt{cpu}(T_{\texttt{INT}}, T_{\texttt{CACHE}}), \texttt{smram}(T_{\texttt{STM}}, T_{\texttt{SMIH}}), \texttt{drt}(T_{\texttt{INIT}}, T_{\texttt{PP}}, T_{\texttt{LOCK}}))$$
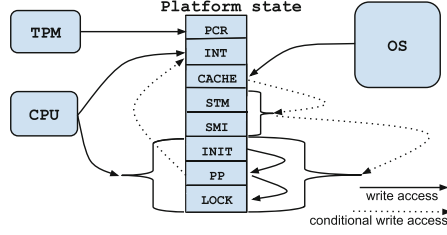
can read any of its components relying on the public unary function symbols $\mathcal{F}_{\texttt{read}} = \{\texttt{pcr}, \texttt{int}, \texttt{cache}, \texttt{stm}, \texttt{smi}, \texttt{init}, \texttt{pp}, \texttt{lock}\}$ and associated rewrite rules:

$$
\begin{aligned}
\texttt{pcr}(\texttt{state}(\texttt{tpm}(y), x_1, x_2, x_3)) &\rightarrow y \\
\texttt{int}(\texttt{state}(x_1, \texttt{cpu}(y_1, y_2), x_2, x_3)) &\rightarrow y_1 \\
\texttt{cache}(\texttt{state}(x_1, \texttt{cpu}(y_1, y_2), x_2, x_3)) &\rightarrow y_2 \\
\texttt{init}(\texttt{state}(x_1, x_2, \texttt{drt}(y_1, y_2, y_3), x_3)) &\rightarrow y_1 \\
\texttt{pp}(\texttt{state}(x_1, x_2, \texttt{drt}(y_1, y_2, y_3), x_3)) &\rightarrow y_2 \\
\texttt{lock}(\texttt{state}(x_1, x_2, \texttt{drt}(y_1, y_2, y_3), x_3)) &\rightarrow y_3 \\
\texttt{stm}(\texttt{state}(x_1, x_2, x_3, \texttt{smram}(y_1, y_2))) &\rightarrow y_1 \\
\texttt{smi}(\texttt{state}(x_1, x_2, x_3, \texttt{smram}(y_1, y_2))) &\rightarrow y_2
\end{aligned}
$$

The write access to the platform state is restricted by the equational theory described and illustrated in Fig. 3, where $\texttt{tpm\_acc}$ and $\texttt{cpu\_acc}$ are private constants and all other new symbols are public.

PCR. Only the TPM can reset, extend or set the value of the PCR. This capability of the TPM is modeled by the *private constant* $\texttt{tpm\_acc}$, which will be used only in the TPM process, described later in Fig. 4.

INT. The interrupts can be enabled or disabled by the CPU, whose capability is modeled by the *private constant* $\texttt{cpu\_acc}$. Additionally, if a DRT is running, then the corresponding protected program PP also has the ability to enable or disable interrupts. This is modeled in the second $\texttt{set\_int}$ equation, by relying on the fact that, if $\texttt{prog}(x)$ represents the public identity of a program (as explained in Sect. 4.2), then x represents a private entry point for that program. Therefore, we can use x to model the ability of $\texttt{prog}(x)$ to change certain elements of the platform state when it is loaded.

$$\text{reset}(\text{state}(\text{tpm}(y), x_1, x_2, x_3), \text{tpm\_acc}, \text{p}_\text{s}) \rightarrow \text{state}(\text{tpm}(\text{p}_\text{s}), x_1, x_2, x_3)$$
$$\text{reset}(\text{state}(\text{tpm}(y), x_1, x_2, x_3), \text{tpm\_acc}, \text{p}_\text{d}) \rightarrow \text{state}(\text{tpm}(\text{p}_\text{d}), x_1, x_2, x_3)$$
$$\text{extend}(\text{state}(\text{tpm}(y), x_1, x_2, x_3), \text{tpm\_acc}, v) \rightarrow \text{state}(\text{tpm}(\text{h}((y, v))), x_1, x_2, x_3)$$
$$\text{set\_pcr}(\text{state}(\text{tpm}(y), x_1, x_2, x_3), \text{tpm\_acc}, v) \rightarrow \text{state}(\text{tpm}(v), x_1, x_2, x_3)$$
$$\text{set\_int}(\text{state}(x_1, \text{cpu}(y_1, y_2), x_2, x_3), \text{cpu\_acc}, v) \rightarrow \text{state}(x_1, \text{cpu}(v, y_2), x_2, x_3)$$
$$\text{set\_int}(\text{state}(x_1, \text{cpu}(y, z), x_2, \text{drt}(z_1, \text{prog}(z_2), \text{true})), z_2, v)$$
$$\quad \rightarrow \text{state}(x_1, \text{cpu}(v, z), x_2, \text{drt}(z_1, \text{prog}(z_2), \text{true}))$$
$$\text{cache}(\text{state}(x_1, \text{cpu}(y_1, y_2), x_2, x_3), v) \rightarrow \text{state}(x_1, \text{cpu}(y_1, v), x_2, x_3)$$
$$\text{flush\_stm}(\text{state}(x_1, \text{cpu}(y_1, v), \text{smram}(z_1, z_2), \text{drt}(w_1, w_2, \text{false})))$$
$$\quad \rightarrow \text{state}(x_1, \text{cpu}(y_1, v), \text{smram}(v, z_2), \text{drt}(w_1, w_2, \text{false})))$$
$$\text{flush\_smi}(\text{state}(x_1, \text{cpu}(y_1, v), \text{smram}(z_1, z_2), x_2))$$
$$\quad \rightarrow \text{state}(x_1, \text{cpu}(y_1, v), \text{smram}(z_1, v), x_2)$$
$$\text{set\_init}(\text{state}(x_1, x_2, x_3, \text{drt}(y_1, y_2, y_3)), \text{cpu\_acc}, v)$$
$$\quad \rightarrow \text{state}(x_1, x_2, x_3, \text{drt}(v, y_2, y_3))$$
$$\text{set\_pp}(\text{state}(x_1, x_2, x_3, \text{drt}(y_1, y_2, y_3)), \text{cpu\_acc}, v)$$
$$\quad \rightarrow \text{state}(x_1, x_2, x_3, \text{drt}(y_1, v, y_3))$$
$$\text{set\_pp}(\text{state}(x_1, x_2, x_3, \text{drt}(\text{prog}(y_1), y_2, y_3)), y_1, v)$$
$$\quad \rightarrow \text{state}(x_1, x_2, x_3, \text{drt}(\text{prog}(y_1), v, y_3))$$
$$\text{set\_pp}(\text{state}(x, \text{cpu}(\text{true}, z), \text{smram}(\text{prog}(z_1), \text{prog}(z_2)), \text{drt}(y_1, y_2, y_3)), (z_1, z_2), v)$$
$$\quad \rightarrow \text{state}(x, \text{cpu}(\text{true}, z), \text{smram}(\text{prog}(z_1), \text{prog}(z_2)), \text{drt}(y_1, v, y_3))$$
$$\text{set\_lock}(\text{state}(x_1, x_2, x_3, \text{drt}(y_1, y_2, y_3)), \text{cpu\_acc}, v)$$
$$\quad \rightarrow \text{state}(x_1, x_2, x_3, \text{drt}(y_1, y_2, v))$$
$$\text{set\_lock}(\text{state}(x_1, x_2, x_3, \text{drt}(y_1, \text{prog}(y_2), y_3)), y_2, v)$$
$$\quad \rightarrow \text{state}(x_1, x_2, x_3, \text{drt}(y_1, \text{prog}(y_2), v))$$
$$\text{set\_lock}(\text{state}(x, \text{cpu}(\text{true}, z), \text{smram}(\text{prog}(z_1), \text{prog}(z_2)), \text{drt}(y_1, y_2, y_3)), (z_1, z_2), v)$$
$$\quad \rightarrow \text{state}(x, \text{cpu}(\text{true}, z), \text{smram}(\text{prog}(z_1), \text{prog}(z_2)), \text{drt}(y_1, y_2, v))$$

Fig. 3. Write access to the platform state.

CACHE. Any values can be cached. The cache values can then be copied into the contents of the SMI handler and, when a DRT is not running, into the STM component of the state.

INIT. Only the CPU has the ability to load an INIT program on the platform.

PP. The PP program can be loaded by the CPU (the first equation for set_pp) or by an INIT program, if the latter is already loaded on the platform (the second equation for set_pp). Furthermore, the SMI in conjunction with the STM can also modify the PP program, if the interrupts are enabled (the third equation for set_pp).

LOCK. Similarly, the DRT lock can be set/unset by the CPU, by the running PP, or by the SMI in conjunction with the STM, if the interrupts are enabled.

We denote by $\mathcal{E}_{\texttt{state}}$ the equational theory defined in this subsection.

4.5   Communication Channels

The public constant os models a communication channel for platform states and other messages that may be intercepted, modified or provided by the intruder as inputs to the CPU or the TPM. A private constant cpu_tpm models the secure channel between the CPU and the TPM. A private function tpm_ch models the ability of the CPU to establish a private channel between a loaded program and the TPM. Generally, these channels will be of the form $\texttt{tpm\_ch}(\texttt{prog}(t))$ and the CPU will send this term both to the program represented by $\texttt{prog}(t)$ (on channel t) and to the TPM (on channel cpu_tpm). We also use message tags that will be clear from the context.

4.6   The Trusted Platform Module

We model the TPM by the process in Fig. 4. A PCR reset request can come either from the CPU, and then the PCR is reset to the value $\texttt{p}_\texttt{d}$ marking a dynamic root of trust, or else from the operating system. A PCR extend request can come from the CPU, from the operating system or from a private channel that the CPU can establish between the TPM and some other process. To unseal a value, the TPM relies on the value of the PCR registers recorded in the platform state that is associated to an unseal

$$
\begin{aligned}
\texttt{TPM} \quad &= \ !\texttt{TPM}_{\texttt{RESET}} \mid !\texttt{TPM}_{\texttt{EXTEND}} \mid !\texttt{TPM}_{\texttt{UNSEAL}} \\
\texttt{TPM}_{\texttt{RESET}} &= \texttt{let } (ch, rv) = (\texttt{cpu\_tpm}, \texttt{p}_\texttt{d}) \texttt{ in } !\texttt{PCR}_{\texttt{RESET}} \mid \\
&\quad \texttt{let } (ch, rv) = (\texttt{os}, \texttt{p}_\texttt{s}) \texttt{ in } !\texttt{PCR}_{\texttt{RESET}} \\
\texttt{PCR}_{\texttt{RESET}} &= \texttt{in}(ch, (\texttt{reset\_req}, nonce, pf\_state)); \\
&\quad \texttt{let } new\_st = \texttt{reset}(pf\_state, \texttt{tpm\_acc}, rv) \texttt{ in} \\
&\quad \texttt{out}(ch, (\texttt{reset\_resp}, nonce, new\_st)) \\
\texttt{TPM}_{\texttt{EXTEND}} &= \texttt{let } ch = \texttt{cpu\_tpm in } !\texttt{PCR}_{\texttt{EXTEND}} \mid \\
&\quad \texttt{let } ch = \texttt{os in } !\texttt{PCR}_{\texttt{EXTEND}} \mid \\
&\quad ! \, (\texttt{in}(\texttt{cpu\_tpm}, (\texttt{ext\_channel}, ch)); !\texttt{PCR}_{\texttt{EXTEND}}) \\
\texttt{PCR}_{\texttt{EXTEND}} &= \texttt{in}(ch, (\texttt{extend\_req}, nonce, pf\_state, v)); \\
&\quad \texttt{let } new\_st = \texttt{extend}(pf\_state, \texttt{tpm\_acc}, v) \texttt{ in} \\
&\quad \texttt{out}(ch, (\texttt{extend\_resp}, nonce, new\_st)) \\
\texttt{TPM}_{\texttt{UNSEAL}} &= \texttt{in}(\texttt{os}, pf\_state); \\
&\quad \texttt{if } \texttt{lock}(pf\_state) = \texttt{true then} \\
&\quad\quad \texttt{let } ch = \texttt{tpm\_ch}(\texttt{pp}(pf\_state)) \texttt{ in UNSEAL} \\
&\quad \texttt{else let } ch = \texttt{os in UNSEAL} \\
\texttt{UNSEAL} \quad &= \texttt{in}(ch, (\texttt{tag\_unseal}, blob)); \\
&\quad \texttt{let } v = \texttt{unseal}(blob, \texttt{pcr}(pf\_state)) \texttt{ in} \\
&\quad \texttt{out}(ch, (\texttt{tag\_plain}, v)))
\end{aligned}
$$

Fig. 4. The TPM process

request. The corresponding equation for unseal ensures that this operation will succeed only if the PCR values from the state match the PCR values against which plain data was sealed. If a DRT is running, we perform the unseal for the protected program PP, on the private channel $\texttt{tpm\_ch}(pp(pf\_state))$; otherwise, the unsealed value is made public on channel os.

4.7 Dynamic Root of Trust: Launch

The procedure for launching a dynamic root of trust, i.e. steps 1–7 from Fig. 1, is modeled by the processes CPU and INIT, from Fig. 5. The CPU receives a request including the INIT and PP programs and the platform state where the DRT is to be launched. If a DRT is not already running in the corresponding platform state, then the CPU disables the interrupts and sets the DRT lock (step 1). Next, the CPU measures the INIT and STM programs and extends the result into the PCR (steps 2–3). In step 4a, the INIT program is loaded and we use the term tpm_ch(init) to model an established private channel between the TPM and the running INIT program. We use the program abstraction introduced in Sect. 4.2 to model the loading and the execution of INIT, relying on the private constant Tinit. In turn, the loaded INIT program measures the PP program, records the measurement into the TPM, and loads PP on the platform state (steps 4b–7a). After the INIT program has measured the PP program and loaded it into memory, the CPU gets back

```
CPU =   !     (*** The CPU process ***)
(* Step 1: receive a DRT request *)
 in(os, (drt_req, init, pp, pf_state))
 if lock(pf_state) = false then
 let s'_0 = set_int(pf_state, cpu_acc, false) in
 let s_0 = set_lock(s'_0, cpu_acc, true) in

(* Step 2: measure INIT and the STM *)
 let measure = (h(init), h(stm(pf_state))) in

(* Step 3: reset and extend the PCR *)
 new nonce; out(cpu_tpm, (reset_req, nonce, s_0));
 in(cpu_tpm, (reset_resp, nonce, s_1));
 out(cpu_tpm, (extend_req, nonce, s_1, measure));
 in(cpu_tpm, (extend_resp, nonce, s_2));
(*Step 4a: load INIT & grant TPM access*)
 let s_3 = set_init(s_2, cpu_acc, init) in
 let einit = get_entry(init) in
 out(einit, (nonce, s_3, tpm_ch(init), pp));
 out(cpu_tpm, (ext_channel, tpm_ch(init))));
(* Step 7b: establish TPM access for PP *)
 in(einit, (drt_resp, nonce, new_state));
 let epp = get_entry(pp(new_state)) in
 out(epp, (new_state, tpm_ch(prog(epp))));
 out(cpu_tpm, (ext_channel, tpm_ch(prog(epp))))

INIT =    (*** A trusted INIT program ***)
 out(os, prog(Tinit)); out(os, prog(Tstm));
(* Step 4b: receive PP and TPM channel *)
 in(Tinit, (nonce, pf_st, tpmc, pp));
(* Steps 5-6: extend h(PP) into PCR *)
 let measure = h(pp) in new nonce_1;
 out(tpmc, (extend_req, nonce_1, pf_st, measure));
 in(tpmc, (extend_resp, nonce_1, ext_st));
(* Step 7a: load PP on platform state *)
 let new_st = set_pp(ext_st, Tinit, pp) in
 out(exp_init, (drt_resp, nonce, new_st));
 out(os, new_st)
```

Fig. 5. DRT process for CPU and INIT

the new platform state and sets up the private channel for communication between the loaded PP and the TPM (step 7b).

4.8   Dynamic Root of Trust: Execution

We illustrate the execution of a trusted PP program with an example in Fig. 6, where step 8 is an example of some useful execution of PP, i.e., unsealing and decrypting, whereas the rest is behaviour we expect from any protected program. The private constant Tpp represents the private entry point of PP according to the model from Sect. 4.2.

In Fig. 7 we consider a fresh symmetric key k_{pp} and assume that this key has been sealed against the measurement of the trusted PP program, with identity prog(Tinit), of the trusted INIT program,

```
PP = (* Example of protected program *)
(* Step 7c: launch and get TPM access *)
out(os, prog(Tpp)));
in(Tpp, (pf_state₀, tpmc));
(* Re-enable interrupts *)
let pf_st = set_int(pf_state₀, Tpp, true)
 in out(os, pf_st);

(* Step 8: unseal and decrypt *)
in(os, x_seal); in(os, x_enc);
out(tpmc, (tag_unseal, x_seal));
in(tpmc, (tag_plain, x_k));
let mess = sdec(x_enc, x_k) in out(os, mess);

(* Step 9: Ending the execution *)
new rand; out(tpmc, (extend_req, rand, pf_st, ⊥));
in(tpmc, (extend_resp, rand, exts));
let ends = set_lock(exts, Tpp, false) in
out(os, ends)
```

Fig. 6. DRT execution

with identity prog(Tinit), and of the trusted STM program, with identity prog(Tstm). This is represented by the term sealed_key in the process DATA (see the code in the figure below), which we publish on the channel os. We also assume that some private message $\mathrm{hi}_{\mathrm{pp}}$ is encrypted with k_{pp} and senc($\mathrm{hi}_{\mathrm{pp}}, k_{\mathrm{pp}}$) is made publicly available on channel os.

In the context of a DRT, the program PP should be able to unseal the key k_{pp}, decrypt and publish $\mathrm{hi}_{\mathrm{pp}}$. Before the execution of PP ends, the DRT lock is set to false, and also the PCR is extended with a dummy value in order to leave the PCR in a state which is not to be trusted any more. We verify, in Sect. 4.9, that secret DATA sealed for this program remains secret.

The SETUP process ties everything together, i.e., it loads and publishes an initial state, and runs any DRT request from the operating system. We call EXEC, all the processes put together, whereas the TPM is the one providing the trusted functionalities of reset, extend, and unseal. We use DRT = (TPM | EXEC).

4.9   Security Properties in the Formal Model

Reachability. The reachability of a state in the platform can be expressed as a (non-)secrecy property: a state is reachable when a corresponding state term can be obtained by the attacker after interacting with the process DRT modulo the theory $\mathcal{E}_{\mathrm{drt}} = \mathcal{E}_{\mathrm{data}} \cup \mathcal{E}_{\mathrm{prog}} \cup \mathcal{E}_{\mathrm{state}}$, expressed as a formula of the form

$$\mathrm{DRT} \models_{\mathcal{E}_{\mathrm{drt}}} \mathrm{Att}(\mathrm{state}(T_{\mathrm{tpm}}, T_{\mathrm{cpu}}, T_{\mathrm{smram}}, T_{\mathrm{drt}})).$$

```
DATA =   (* Seal and encrypt private data *)
new k_pp; new hi_pp; out(os, senc(hi_pp, k_pp));
let sealed_key = seal(k_pp, hchain) in out(os, sealed_key);
(* where hchain = h(h(p_d, (h(prog(Tinit)), h(prog(Tstm)))), h(prog(Tpp))) *)

SETUP =   (* Launching the system *)
(* Load the initial state *)
in(os, x_stm); in(os, x_smi);
out(os, state(tpm(p_s), cpu(true, ⊥), smram(x_stm, x_smi), drt(⊥, ⊥, false)));
(* Run a DRT with any loaded programs *)
in(os, init); in(os, pp); in(os, pf_state); out(os, (drt_req, init, pp, pf_state));
(* The main processes put together *)
EXEC = ( CPU | ! INIT | SETUP | DATA | ! PP )        DRT =   ( TPM | EXEC )
```

Fig. 7. DRT setup and full process.

The property that the DRT = (TPM | EXEC) process can reach an expected state where some trusted programs INIT and PP have been correctly measured and loaded on the platform can be expressed as follows:

$$\text{DRT} \models_{\mathcal{E}_{\text{drt}}} \text{Att(state(} \qquad\qquad\qquad\qquad\qquad where$$
$$\text{tpm(h((h((p_d, } v_1)), v_2))), \text{cpu(true, } x), \qquad v_1 = (\text{h(prog(Tinit))},$$
$$\text{smram(prog(Tstm), prog(}y)) \qquad\qquad\qquad \text{h(prog(Tstm))}$$
$$\text{drt(prog(Tinit), prog(Tpp), true)))} \qquad v_2 = \text{h(prog(Tpp))}.$$

An additional reachability property of interest is whether the program PP has succeeded to unseal the key k_{pp}, decrypt the private message hi_{pp} and output it on the public channel os. This is captured by the following (non-)secrecy formula:

$$\text{DRT} \models_{\mathcal{E}_{\text{drt}}} \text{Att(hi}_{\text{pp}}).$$

Code integrity. We say that the trusted platform ensures code integrity if the measurement contained in the PCR value correctly reflects the state of the platform. Specifically, we require that whenever a dynamic root of trust is active with a PCR value of p_d extended with the expected measurements v_1 and v_2, then only the corresponding PP, INIT and STM are running on the platform, and they cannot be modified. This can be expressed by the following correspondence assertion, which we will *denote by* Φ_{int} in the rest of the paper:

$$\text{DRT} \models_{\mathcal{E}_{\text{drt}}} \text{Att(state(tpm(h((h((p_d, } v_1)), v_2))), \text{cpu(}x, y), \text{smram(}x_{\text{stm}}, x_{\text{smi}}),$$
$$\text{drt(}x_{\text{init}}, x_{\text{pp}}, \text{true)))} \implies (x_{\text{init}}, x_{\text{pp}}, x_{\text{stm}}) = (p_1, p_2, p_3)$$

where $p_1 = \text{prog(Tinit)}$, $p_2 = \text{prog(Tpp)}$, $p_3 = \text{prog(Tstm)}$.

Note that we ensure the property only for trusted programs. Indeed, if any of PP, INIT or STM are malicious, they could use their privileges to reach a platform state that does not reflect the PCR values. This is fine, because the PCR values will correctly record the identity of running programs in the chain of trust.

In particular, our property shows that untrusted `DRT` programs cannot make the `PCR` values record the measurement of trusted programs.

Secrecy of sealed data. We also verify that data sealed for `PP`, i.e. the key $k_{\mathtt{pp}}$, remains secret (we *denote this formula by* $\Phi_{\mathtt{sec}}$):

$$(\Phi_{\mathtt{sec}}) \quad \mathtt{DRT} \models_{\mathcal{E}_{\mathtt{drt}}} \mathtt{Att}(k_{\mathtt{pp}}) \implies false.$$

5 Process Transformation for Automated Verification

ProVerif does not terminate for the `DRT` process and the equational theory $\mathcal{E}_{\mathtt{drt}}$. The main reason is the rewrite rule from $\mathcal{E}_{state}$ that allows an unbounded number of `PCR` extensions, reflecting a problem first noticed in [12]. In this section, we propose a general transformation of processes that allows a more efficient exploration of the search space by ProVerif. The transformation is based on a general observation formalised in Proposition 1: we can replace a process P with a process Q as input for ProVerif, as long as Q and P are equivalent with respect to the security properties of interest. Concretely, we will replace the process `DRT` with a process $\mathtt{DRT}^b$ that bounds the number of `PCR` extensions, while allowing a direct way for the attacker to set the `PCR` to any value that is bigger than the considered bound.

For a process P, let $\mathtt{Att}(P) = \{T \mid P \models \mathtt{Att}(T)\}$ be the set of terms that can be obtained by the attacker when interacting with P. For a set of terms $\mathcal{M}$, we let $\mathtt{Att}(\mathcal{M}) = \{T \mid \mathcal{M} \vdash T\}$. We notice the following.

Proposition 1. *Let P, Q be processes and $\mathit{Att}(T) \implies \Phi$ be a correspondence assertion such that, for any substitution σ,*

$$T\sigma \in \mathit{Att}(P) \smallsetminus \mathit{Att}(Q) \implies \Phi\sigma \ \ and \ \ T\sigma \in \mathit{Att}(Q) \smallsetminus \mathit{Att}(P) \implies \Phi\sigma.$$

Then we have: $P \models \mathit{Att}(T) \implies \Phi$ if and only if $Q \models \mathit{Att}(T) \implies \Phi$.

The proof of Proposition 1 follows immediately from definitions, yet this result is crucial to make our models amenable for ProVerif. We are thus allowed to transform the process `DRT` into a process $\mathtt{DRT}^b$, that is equivalent to `DRT` with respect to code integrity and secrecy properties $\Phi_{\mathtt{int}}$ and $\Phi_{\mathtt{sec}}$, and whose search space can be handled by ProVerif. It will be easier to express $\mathtt{DRT}^b$ using some additional rewrite rules. In conjunction with Proposition 1, we will then rely on the following result for soundness and completeness:

Proposition 2. *Let $\mathcal{P}$ be a process, $\mathcal{E}$ be an equational theory and $\mathit{Att}(T) \implies \Phi$ be a correspondence assertion. Assume $\mathcal{E}^b$ is a set of rewrite rules such that $\forall U \rightarrow V \in \mathcal{E}^b : \mathit{top}(U) \in \mathcal{F}_{priv}$, i.e., is a private symbol. Then we have:*

$$P \models_{\mathcal{E}} \mathit{Att}(T) \implies \Phi \ if \ and \ only \ if \ P \models_{\mathcal{E} \cup \mathcal{E}^b} \mathit{Att}(T) \implies \Phi.$$

Notation. We denoted a term of the form $h((\ldots h((T_0, T_1))), \ldots, T_n))$ by $\mathtt{chain}(T_0, \ldots, T_n)$, using $\mathtt{chain}(T_0)$ for T_0. We define $\mathtt{length}(\mathtt{chain}(T_0, \ldots, T_n)) = n$, representing the number of extensions of a $\mathtt{PCR}$.

Problematic rewrite rule. We recall the rewrite rule that poses non-termination problems for ProVerif:

$$\mathtt{extend}(\mathtt{state}(\mathtt{tpm}(y), x_1, x_2, x_3), \mathtt{tpm\_acc}, v) \to \mathtt{state}(\mathtt{tpm}(h((y, v))), x_1, x_2, x_3)$$

Intuitively, ProVerif does not terminate because it is unable to make an abstract reasoning about the introduction of the term $h((y, v))$ in the right hand side of this rewrite rule. We propose a transformation of the $\mathtt{TPM}$ process into a process $\mathtt{TPM}^b$ that allows more values to be written into the $\mathtt{PCR}$, overapproximating the effect of the problematic rewrite rule. This transformation will be sound and complete (satisfying the conditions of Proposition 1) based on the observation that, once it exceeds a certain bound, the value of the $\mathtt{PCR}$ does not matter for $\varPhi_{\mathtt{sec}}$ and $\varPhi_{\mathtt{int}}$ – thus, we can let the attacker have complete control over it.

Proposed transformation. For a given natural number b, we would like the following behaviour of the $\mathtt{TPM}^b$ process: if an extend request is received for a platform state $\mathtt{state}(\mathtt{tpm}(T_1), T_2, T_3, T_4)$ and a value V:

- if the length of the $\mathtt{PCR}$ is smaller than b, i.e. $\mathtt{length}(T_1) < b$, then execute this request normally, using the function $\mathtt{extend}$. The updated platform state returned by the $\mathtt{TPM}^b$ should now be $\mathtt{state}(\mathtt{tpm}(h((T_1, V))), T_2, T_3, T_4)$.
- if the length of the $\mathtt{PCR}$ value T_1 is greater or equal to b, i.e. $\mathtt{length}(T_1) \geq b$, then output T_1 and V to the attacker and wait for a new value T_1' as a response. If the length of T_1' is big enough, i.e. $\mathtt{length}(T_1') > b$, the updated platform state returned by the $\mathtt{TPM}^b$ should now be $\mathtt{state}(\mathtt{tpm}(T_1'), T_2, T_3, T_4)$. In a normal execution, we would have $T_1' = h((T_1, V))$. However, the attacker has the choice to set T_1' to any value.

Formally, the $\mathtt{TPM}^b$ process relies on the private function $\mathtt{is\_small}$ to detect if the value of the $\mathtt{PCR}$ is lower or higher than the bound, and treat the two cases differently. The following set of rewrite rules, for all $0 \leq i < b$, define $\mathtt{is\_small}$: $\mathtt{is\_small}(\mathtt{chain}(v_0, \ldots, v_i)) \to \mathtt{true}$, where $v_0 \in \{\mathtt{p_s}, \mathtt{p_d}\}$ and $v_1, \ldots, v_i$ are mutually distinct variables. We also need to check if some value to be extended into the $\mathtt{PCR}$ is big enough. For this, we introduce the private function $\mathtt{is\_big}$, together with the rewrite rule: $\mathtt{is\_big}(\mathtt{chain}(v_0, \ldots, v_{b+1})) \to \mathtt{true}$, where $v_0, \ldots, v_{b+1}$ are mutually distinct variables.

The only difference from the normal TPM process is in $\mathtt{PCR}^b_{\mathtt{EXTEND}}$, which first detects if the current value of the $\mathtt{PCR}$ is small or big: if it is small, the extension process proceeds normally (the process $\mathtt{TPM}^{\mathtt{SMALL}}_{\mathtt{EXTEND}}$); if it is bigger than the given bound, then the TPM requests that the operating system combines $\mathtt{pcr}$ and $\mathtt{val}$ itself (the process $\mathtt{TPM}^{\mathtt{BIG}}_{\mathtt{EXTEND}}$). Upon receiving the response from the $\mathtt{os}$, the TPM first checks that the value provided is indeed big (the compromised operating system may be cheating). Only then, it updates the $\mathtt{PCR}$ to the requested value.

We denote by $\mathcal{E}^b_{\mathtt{drt}}$ the equational theory $\mathcal{E}_{\mathtt{drt}}$ augmented with the rules for $\mathtt{is\_small}, \mathtt{is\_big}$ and $\mathtt{set\_pcr}$ introduced in this section and we assume that these new symbols are private (they are used only by $\mathtt{TPM}^b$).

```
DRTᵇ        = TPMᵇ | EXEC
TPMᵇ        = TPM { PCR_EXTEND ↦ PCRᵇ_EXTEND }
PCRᵇ_EXTEND = in(ch, (= extend_req, nonce, pf_state, val));
              let pcr = pcr(pf_state) in
              if is_small(pcr) = true then  PCRˢᴹᴬᴸᴸ_EXTEND else PCRᴮᴵᴳ_EXTEND
PCRˢᴹᴬᴸᴸ_EXTEND = let new_st = extend(pf_state, tpm_acc, val) in
              out(ch, (extend_resp, nonce, new_st))
PCRᴮᴵᴳ_EXTEND = out(os, (pcr, val)); in(os, new_pcr)
              if is_big(new_pcr) = true then
              let new_st = set_pcr(pf_state, tpm_acc, new_pcr) in
              out(ch, (extend_resp, nonce, new_st))
```

5.1   Sketch of Correctness Proofs

We have to show that, for $\Phi \in \{\Phi_{\text{sec}}, \Phi_{\text{int}}\}$, we have $\text{DRT} \models_{\mathcal{E}_{\text{drt}}} \Phi \Leftrightarrow \text{DRT}^b \models_{\mathcal{E}^b_{\text{drt}}} \Phi$.
We note that soundness (direction $\Leftarrow$) is the property that is necessary to derive
the security guarantees for DRT, while completeness is secondary: it explains
why we dont get false attacks against DRT^b with ProVerif. Since $\text{Att}(\text{DRT}) \subseteq$
$\text{Att}(\text{DRT}^b)$, soundness is easy to prove, while completeness requires careful analy-
sis of terms in $\text{Att}(\text{DRT}^b) \smallsetminus \text{Att}(\text{DRT})$. We show that such terms are roughly lim-
ited to what we explicitly release in DRT^b: state terms with big PCR values; they
cannot be used by the attacker to violate Φ_{sec} and Φ_{int}.

First, from Proposition 2 and the definition of $\mathcal{E}^b_{\text{drt}}$, we can easily translate
between $\mathcal{E}_{\text{drt}}$ and $\mathcal{E}^b_{\text{drt}}$, thus the notions and results that follow are modulo $\mathcal{E}^b_{\text{drt}}$.

Corollary 1. *For any Φ, we have* $\text{DRT} \models_{\mathcal{E}_{\text{drt}}} \Phi \Leftrightarrow \text{DRT} \models_{\mathcal{E}^b_{\text{drt}}} \Phi$.

Terms T with $\text{top}(T) = \text{state}$ are called state terms (or states). For a
state term $T = \text{state}(\text{tpm}(T_1), \text{cpu}(T_2, T_3), \text{smram}(T_3, T_4), \text{drt}(T_5, T_6, T_7))$, we let
$\text{Comp}(T) = \{T_1, \ldots, T_7\}$. For a set of terms $\mathcal{M}_1$, we say that a set of state terms
$\mathcal{M}_2$ is $\mathcal{M}_1$-saturated if for any $T \in \mathcal{M}_2$ we have $\forall U \in \text{Comp}(T) : \mathcal{M}_1 \vdash U$.

Lemma 1. *Let $\mathcal{M}_1$ be a set of terms and $\mathcal{M}_2$ be an $\mathcal{M}_1$-saturated set of state
terms. Then we have* $\text{Att}(\mathcal{M}_1 \cup \mathcal{M}_2) = \text{Att}(\mathcal{M}_1) \cup \mathcal{M}_2$.

Lemma 1 formalizes the intuition that, without access to TPM or CPU, the only
operation that an attacker can perform on a state is to extract its components.
The proof follows by a straightforward inspection of rewrite rules. To help in the
sequel, we consider several restrictions of attacker's power against DRT^b:

- $\text{Att}_0(\text{DRT}^b)$ is the set of terms that can be obtained by an attacker interacting
  with DRT^b, while not being allowed to use terms in $\text{Att}(\text{DRT}^b) \smallsetminus \text{Att}(\text{DRT})$ when
  constructing inputs for DRT^b. That is, $\text{Att}_0(\text{DRT}^b)$ can be seen as a passive
  attacker with respect to the additional functionality in DRT^b.
- $\text{Att}_1(\text{DRT}^b)$ is the knowledge of the previous attacker whose power is aug-
  mented with the ability to unseal terms from $\text{Att}_0(\text{DRT}^b)$, with TPM_UNSEAL,
  relying on state terms from $\text{Att}(\text{DRT}^b) \smallsetminus \text{Att}(\text{DRT})$. This attacker is not allowed
  to use terms from $\text{Att}(\text{DRT}^b) \smallsetminus \text{Att}(\text{DRT})$ in any other way.

- $\mathtt{Att}_2(\mathtt{DRT}^b)$ is the knowledge of a *state respecting* attacker against $\mathtt{DRT}^b$: the attacker is given unrestricted access to $\mathtt{DRT}^b$ and can use any terms from $\mathtt{Att}(\mathtt{DRT}^b) \smallsetminus \mathtt{Att}(\mathtt{DRT})$ to construct his inputs; however, the attacker can only use state terms according to the specification of an honest behaviour while interacting with the TPM, the CPU, or the equational theory.

Note that $\mathtt{Att}_0(\mathtt{DRT}^b) \subseteq \mathtt{Att}_1(\mathtt{DRT}^b) \subseteq \mathtt{Att}_2(\mathtt{DRT}^b) \subseteq \mathtt{Att}(\mathtt{DRT}^b)$. We denote by $\mathcal{M}^b$ the set of state terms returned to the attacker by the $\mathtt{PCR}_{\mathtt{EXTEND}}^{\mathtt{BIG}}$ process. Note that $\mathcal{M}^b$ is an $\mathtt{Att}(\mathtt{DRT})$-saturated set of state terms with $\forall T \in \mathcal{M}^b :$ $\mathtt{length}(\mathtt{pcr}(T)) > b$.

Lemma 2. *For any b, we have $\mathit{Att}(\mathit{DRT}) \subseteq \mathit{Att}_0(\mathit{DRT}^b) \subseteq \mathit{Att}(\mathit{DRT}) \cup \mathcal{M}^b$.*

The first inclusion follows easily from the definition of $\mathtt{DRT}^b$, which is able to simulate any normal PCR extension performed by DRT, without access to any terms in $\mathtt{Att}(\mathtt{DRT}^b) \smallsetminus \mathtt{Att}(\mathtt{DRT})$. For the second inclusion, relying on the fact that $\mathcal{M}^b$ is $\mathtt{Att}(\mathtt{DRT})$-saturated, we use Lemma 1 to deduce $\mathtt{Att}_0(\mathtt{DRT}^b) \subseteq$ $\mathtt{Att}(\mathtt{Att}(\mathtt{DRT}) \cup \mathcal{M}^b) \subseteq \mathtt{Att}(\mathtt{DRT}) \cup \mathcal{M}^b$.

Lemma 3. *For $b \geq 2$, we have $\mathit{Att}_1(\mathit{DRT}^b) \subseteq \mathit{Att}_0(\mathit{DRT}^b)$.*

By definition, $\mathtt{Att}_1(\mathtt{DRT}^b) \smallsetminus \mathtt{Att}_0(\mathtt{DRT}^b) \subseteq \{U \mid \mathtt{seal}(U,V) \in \mathtt{Att}_0(\mathtt{DRT}^b)\}$. Note that the only sealed term in $\mathtt{Att}_0(\mathtt{DRT}^b)$ that does not originate from the attacker is $\mathtt{seal}(k_{\mathtt{pp}}, \mathtt{hchain})$, with $\mathtt{length}(\mathtt{hchain}) = 2$. For any other term $\mathtt{seal}(U,V) \in \mathtt{Att}_0(\mathtt{DRT}^b)$, we have $U \in \mathtt{Att}_0(\mathtt{DRT}^b)$, and therefore $U \notin \mathtt{Att}_1(\mathtt{DRT}^b) \smallsetminus \mathtt{Att}_0(\mathtt{DRT}^b)$. From Lemma 2, the definition of $\mathtt{TPM}_{\mathtt{UNSEAL}}$, and the fact that $\forall T \in \mathcal{M}^b : \mathtt{length}(\mathtt{pcr}(T)) > b$, we also deduce that $k_{\mathtt{pp}} \notin \mathtt{Att}_1(\mathtt{DRT}^b) \smallsetminus \mathtt{Att}_0(\mathtt{DRT}^b)$, so we can conclude $\mathtt{Att}_1(\mathtt{DRT}^b) \subseteq \mathtt{Att}_0(\mathtt{DRT}^b)$.

Lemma 4. *For $b \geq 2$, we have $\mathit{Att}_2(\mathit{DRT}^b) \subseteq \mathit{Att}_1(\mathit{DRT}^b) \cup \mathcal{M}^b$.*

New terms $U \in \mathtt{Att}_2(\mathtt{DRT}^b)$ come from using a state term $V \in \mathtt{Att}_1(\mathtt{DRT}^b)$ in $\mathtt{TPM}_{\mathtt{RESET}}, \mathtt{TPM}_{\mathtt{EXTEND}}$ or CPU. From Lemmas 2 and 3, we have either $V \in \mathtt{Att}(\mathtt{DRT})$ or $V \in \mathcal{M}^b$. In both cases, we can show that $U \in \mathtt{Att}_1(\mathtt{DRT}^b) \cup \mathcal{M}^b$.

Corollary 2. *For $b \geq 2$, we have $\mathit{Att}(\mathit{DRT}) \subseteq \mathit{Att}(\mathit{DRT}^b) \subseteq \mathit{Att}(\mathit{DRT}) \cup \mathcal{M}^b \cup \mathcal{M}^f$, where $\mathcal{M}^f$ is a set of terms such that any term $T \in \mathcal{M}^f$ contains a state term T' with $\mathit{pcr}(T') > b$.*

The set $\mathcal{M}^f$ represents the additional terms that a non state respecting attacker can derive from $\mathcal{M}^b$. The property of $\mathcal{M}^f$ is due to the fact that $\mathcal{E}_{\mathtt{drt}}^b$ and the $\mathtt{DRT}^b$ process do not have effect on state terms that are used outside their intended scope. Such terms will end up as harmless subterms of attacker's knowledge.

Corollary 3. *For $b \geq 2$, DRT and DRT^b satisfy the conditions of Proposition 1 with respect to both Φ_{sec} and Φ_{int}.*

Corollary 2 shows that it is sufficient to check that conditions of Proposition 1 are satisfied for terms T in $\mathcal{M}^b \cup \mathcal{M}^f$. For $\Phi_{\texttt{sec}}$, this follows from the fact that such terms T are either state terms, or contain state terms, and therefore the key $k_{\texttt{pp}}$ cannot be among them. For $\Phi_{\texttt{int}}$, this follows from the fact that those state terms have PCR lengths bigger than 2, while the precondition of $\Phi_{\texttt{int}}$ is a state term with PCR length 2. From Corollary 3 and Proposition 1, we deduce:

Corollary 4. *For $\Phi \in \{\Phi_{sec}, \Phi_{int}\}$, we have $DRT \models_{\mathcal{E}_{drt}^b} \Phi \Leftrightarrow DRT^b \models_{\mathcal{E}_{drt}^b} \Phi$.*

From Corollaries 1 and 4, we conclude:

Theorem 1. *For $\Phi \in \{\Phi_{sec}, \Phi_{int}\}$, $DRT \models_{\mathcal{E}_{drt}} \Phi \Leftrightarrow DRT^b \models_{\mathcal{E}_{drt}^b} \Phi$.*

6 Verification

The ProVerif code for the DRT^b process and the security properties defined in Sects. 4 and 5 is available online[2]. It uses the equational theory $\mathcal{E}_{\texttt{data}} \cup \mathcal{E}_{\texttt{prog}} \cup \mathcal{E}_{\texttt{state}}^b$, with $b = 2$. The verification of each security property terminates in order of minutes, returning the expected result. From these results (implying there is no attack on DRT^b modulo $\mathcal{E}_{\texttt{drt}}^b$) and from Theorem 1 (implying there is no attack on DRT modulo $\mathcal{E}_{\texttt{drt}}$), we derive:

Theorem 2. *The DRT process satisfies, modulo $\mathcal{E}_{data} \cup \mathcal{E}_{prog} \cup \mathcal{E}_{state}$, the properties of code integrity and data secrecy defined in Sect. 4.9.*

In order to check the reachability properties $DRT \models \Phi$ defined in Sect. 4.9, we give $\neg(DRT \models \Phi)$ as input query for ProVerif - an attack with respect to this query would be a witness trace for the desired reachability property. When returning such a trace, ProVerif can either confirm that it is valid (*attack found*) or cannot confirm it. Our models fall in the latter case, and we have to further inspect the output trace to see how its steps can be used to reconstruct a valid trace: we do observe in the output trace the expected intermediary messages on the channels cpu_tpm and os, and we can follow the source of these messages up to a dynamic root of trust request, of whose validity we have to again make sure. By a similar analysis of attack traces returned by ProVerif, we can observe the attacks of [13,29] in our models, when we allow the STM to be modified arbitrarily.

7 Further Work

While our model takes into account at an abstract level the attacks and mitigations of [13,29], further refinements and soundness results are necessary in order to be able to conclude that attacks such as these or as [30,31] are not possible in practice. We need to develop models that are abstract enough to allow clear specifications and automated reasoning, and realistic enough to capture for

---

[2] www.dropbox.com/s/cvq4op3w106868t/drt.pi (using ProVerif version 1.85).

instance implementation flaws. We plan to see how the models of this paper can be expressed in richer frameworks like StatVerif [3] and SAPIC [21], in order to capture more closely the state semantics of real platforms. We think the process transformation that we have presented in Sect. 5 is an instance of a more general result, whose exploration would also be fruitful for future applications.

References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL 2001), pp. 104–115, January 2001
2. Arapinis, M., Bursuc, S., Ryan, M.D.: Reduction of equational theories for verification of trace equivalence: re-encryption, associativity and commutativity. In: Degano and Guttman [11], pp. 169–188
3. Arapinis, M., Ritter, E., Ryan, M.D.: StatVerif: verification of stateful processes. In: CSF, pp. 33–47. IEEE Computer Society (2011)
4. Arthur, W., Challener, D., Goldman, K.: A Practical Guide to TPM 2.0. APress, Berkeley (2015)
5. Barbosa, M., Portela, B., Scerri, G., Warinschi, B.: Foundations of hardware-based attested computation and application to SGX. In: IEEE European Symposium on Security and Privacy, EuroS&P, Saarbrücken, Germany, 21–24 March 2016, pp. 245–260. IEEE (2016)
6. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Computer Security Foundations Workshop (CSFW 2001) (2001)
7. Blanchet, B.: Automatic verification of correspondences for security protocols. J. Comput. Secur. **17**(4), 363–434 (2009)
8. Blanchet, B., Chaudhuri, A.: Automated formal analysis of a protocol for secure file sharing on untrusted storage. In: IEEE Symposium on Security and Privacy, pp. 417–431. IEEE Computer Society (2008)
9. Cortier, V., Degrieck, J., Delaune, S.: Analysing routing protocols: four nodes topologies are sufficient. In: Degano and Guttman [11], pp. 30–50
10. Datta, A., Franklin, J., Garg, D., Kaynar, D.: A logic of secure systems and its application to trusted computing. In: 30th IEEE Symposium on Security and Privacy, pp. 221–236. IEEE (2009)
11. Degano, P., Guttman, J.D. (eds.): POST 2012. LNCS, vol. 7215. Springer, Heidelberg (2012)
12. Delaune, S., Kremer, S., Ryan, M.D., Steel, G.: Formal analysis of protocols based on TPM state registers. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF 2011), Cernay-la-Ville, France, pp. 66–82. IEEE Computer Society Press, June 2011
13. Duflot, L., Grumelard, O., Levillain, O., Morin, B.: ACPI and SMI handlers: some limits to trusted computing. J. Comput. Virol. **6**(4), 353–374 (2010)
14. Fournet, C., Planul, J.: Compiling information-flow security to minimal trusted computing bases. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 216–235. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19718-5_12

15. Franklin, J., Chaki, S., Datta, A., McCune, J.M., Vasudevan, A.: Parametric verification of address space separation. In: Degano and Guttman [11], pp. 51–68
16. Franklin, J., Chaki, S., Datta, A., Seshadri, A.: Scalable parametric verification of secure systems: how to verify reference monitors without worrying about data structure size. In: IEEE Symposium on Security and Privacy, pp. 365–379. IEEE Computer Society (2010)
17. Garay, J.A., Jakobsson, M., MacKenzie, P.: Abuse-free optimistic contract signing. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 449–466. Springer, Heidelberg (1999). doi:10.1007/3-540-48405-1_29
18. Grawrock, D.: Dynamics of a Trusted Platform: A Building Block Approach. Intel Press, Hillsboro (2009)
19. Trusted Computing Group. TCG Architecture Overview, Specification revision 1.4 (2007). www.trustedcomputinggroup.org
20. Trusted Computing Group. TPM main specification (2011). www.trusted computinggroup.org
21. Kremer, S., Künnemann, R.: Automated analysis of security protocols with global state. In: IEEE Symposium on Security and Privacy, pp. 163–178. IEEE Computer Society (2014)
22. Küsters, R., Truderung, T.: Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In: 22nd IEEE Computer Security Foundations Symposium (CSF), pp. 157–171. IEEE Computer Society (2009)
23. Meier, S.: Advancing automated security protocol verification. PhD Thesis, ETH Zürich (2013)
24. Mödersheim, S.: Abstraction by set-membership: verifying security protocols and web services with databases. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM Conference on Computer and Communications Security, pp. 351–360. ACM (2010)
25. Paiola, M., Blanchet, B.: Verification of security protocols with lists: from length one to unbounded length. In: Degano and Guttman [11], pp. 69–88
26. RSA Security Inc., v2.20. PKCS #11: Cryptographic token interface standard, June 2004
27. Ryan, M.D., Smyth, B.: Applied pi calculus. In: Cortier, V., Kremer, S. (eds.) Formal Models and Techniques for Analyzing Security Protocols, Cryptology and Information Security Series. IOS Press (2011)
28. Schmidt, B., Meier, S., Cremers, C.J.F., Basin, D.A.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: Chong, S. (ed.) 25th IEEE Computer Security Foundations Symposium (CSF), pp. 78–94. IEEE Computer Society (2012)
29. Wojtczuk, R., Rutkowska, J.: Attacking INTEL trusted execution technology. In: Black Hat DC (2009)
30. Wojtczuk, R., Rutkowska, J.: Attacking INTEL TXT via SINIT code execution hijacking. Invisible Things Lab (2009)
31. Wojtczuk, R., Rutkowska, J., Tereshkin, A.: Another way to circumvent INTEL trusted execution technology. Invisible Things Lab (2009)
32. Yubico, A.B.: Kungsgatan 37, 111 56 Stockholm Sweden. The YubiKey manual - Usage, configuration and introduction of basic concepts (version 2.2) (2010)