New abstractions in applied pi-calculus and automated verification of protected executions

Shiwei Xu
Wuhan Digital Engineering Institute
Wuhan 430074, China

Sergiu Bursuc
School of Computer Science
University of Bristol, UK

Julian P. Murphy
Centre for Secure Information Technologies
Queen's University of Belfast, UK

Abstract—**Protocols for the protected execution of programs, like those based on a hardware root of trust, will become of fundamental importance for computer security. In parallel to such protocols, there is therefore a need to develop models and tools that allow formal specification and automated verification of the desired security properties. Still, current protocols lack realistic models and automated proofs of security. This is due to several challenges that we address in this paper.**

We consider the classical setting of applied pi-calculus and ProVerif, that we enrich with several generic models that allow verification of protocols designed for a given computing platform. Our contributions include models for specifying platform states and for dynamically loading and executing protected programs. We also propose a new method to make ProVerif terminate on a challenging search space - the one obtained by allowing an unbounded number of extensions and resets for the platform configuration registers of the TPM.

We illustrate our methods with the case study of a protocol for a dynamic root of trust (based on a TPM), which includes dynamic loading, measurement and protected execution of programs. We prove automatically with ProVerif that code integrity and secrecy of sealed data hold for the considered protocol.

I. Introduction

Motivation. Malware is one of the most important problems that computer security research is facing today. The aim is to ensure that the computing platforms are trustworthy and that personal data is not lost to intruders. A hardware root of trust, including dynamic measurement and protected execution, is a promising concept in this context [19]. It relies on the idea that hardware is more difficult to compromize than software and therefore it can play a crucial role in a protocol 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.

Still, from this basic idea to a secure design and implementation there is a long way to go, as various attacks show [14], [27]. In particular, we need models and tools that allow automated verification of desired properties against trusted computing protocols and implementations.

Protocols for protected execution open new and significant challenges for automated verification. The most obvious of them is the sheer size of platforms to be verified. Furthermore, messages of such protocols consist not only of data, but also of programs that can be supplied by an attacker or honest participant to be executed on the platform. Modeling the platform configuration registrers (PCR) of the trusted platform module (TPM) [20] is another problem, because they can be arbitrarily extended. Even the most efficient symbolic methods struggle with the structure of the resulting search space [5], [12].

Related work. Two papers that are particularly relevant in this context are [9] and [12]. However, as we will discuss in section II, both of them have limitations. Firstly, the analysis of [9] is formal, but manual. We will also argue that, while their approach is useful to discover flaws and to provide initial guarantees, it is nevertheless too abstract. It relies on axioms that are simply taken as granted, while we believe the respective logical rules are not basic enough and should be proved. Furthermore, we will argue that their security definitions should also be made more precise. An orthogonal issue is that [9] does not model the sealing functions of the TPM, which are essential to keep secrets for the protected programs.

The analysis of [12] is automated and their axioms are derived from basic operations. In some sense, however, their analysis of a (static) trusted boot is even more abstract than the one in [9]: loaded programs are simply represented by constants, with no associated computational content, and there is only one honest program (one constant) and one dishonest program (another constant). This is certainly not realistic. Their security definitions do not cover code integrity for the protected program. The number of extensions for the TPM registers is bounded in order to make ProVerif terminate. All this necessarily limits the type of applications and properties that can be verified.

Our contributions.

1. We propose a generic model for platform states as terms in applied pi-calculus. The read and write actions on platform states are formalized as an equational theory in the algebra of state terms. This model allows different level of abstractions by simply refining the term that represents the platform state and adding or removing access equations (section IV-A).

2. We propose a model for dynamically loaded *and* protected programs in applied pi-calculus. It is very simple and does not require heavy encodings, being based on the classic idea of *processes as data*. In the setting of protected execution, we need in addition to take care that once a caller has loaded a program, it can be certain to communicate and to allow certain execution capabilities precisely to that program (section IV-B). We further connect loaded programs to the corresponding platform state by using additional equations to model their execution abilities (section IV-C).

3. Relying on contributions 1 and 2, we model dynamic measurement and protected execution in applied pi-calculus, following the general structure of INTEL's Trusted Execution Technology and AMD's Secure Virtual Machine. We rely on reachability [5] and on correspondence assertions [6] to formally express the desired security properties. This is the first formal model that can take into account the attacks of [14], [27] and where proposed solutions can be succesfully verified (section V).

4. We propose a new abstraction to model the extension of PCR registers of the TPM that allows automated verification for a larger class of protocols than the one in [12]. Instead of bounding the number of PCR extensions, we show how to alter the model such that the structure of the search space is simplified, without losing possible attacks or introducing false attacks. The main idea is to notice that we can let the attacker set the PCR to *any* value, as long as it is big enough (section VI).

5. Putting it all together, we obtain the first automated verification for a realistic model of dynamic measurement and protected execution. 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). This is more than [9] and [12] prove, in a more realistic model, and with a higher degree of automation (section VII).

II. Related work

A logic of secure systems [9]. A programming language and a logic for expressing trusted computing properties are proposed in [9]. Their setting is quite expressive and it allows verification of protocols similar to the ones that we study in this paper. They do not consider the seal/unseal functions of the TPM, but we believe their language could be extended to capture them. However, considering the complexity of proofs involved, the lack of automation is a serious limitation of [9]. Further, more subtle, problems are related to the foundations of their formal model. Firstly, there is no clear separation between the operational semantics of programs and the logical framework for proving security properties. Although not a problem in itself, this can lead however to properties that are taken for granted in the logical framework. Take the example of the following axiom (section 3.2 in [9]):

$$\vdash LateLaunch(m, I)@t \supset IsLocked(pcr, I)@t$$

It states that, whenever a late launch (in our terminology, a dynamic root of trust) happens on thread I, the PCR is locked for I. Yet, this is based on a protocol between the calling thread and the TPM, and it will be true only if the attacker can not interfere with that protocol. Hence, it should not be an axiom. Similar arguments apply to other, more complex, axioms present in [9].

We believe that the formal definition of code integrity proposed in [9] should also be reviewed. It is stated as two separate properties: a verifier can correctly read the PCR values (section 4.1.3) and that a dynamic root of trust can be executed as expected (section 4.2.1). In fact, these two properties should be linked: the values recorded in the PCR values should correctly reflect the dynamic root of trust.

On the other hand, we model the dynamic root of trust as a process, outlining each execution step in the protocol, and we express code integrity as a correspondence assertion in applied pi-calculus, which takes into accout the relation between PCR registers and the state of the platform.

TPM registers [12]. The analysis of [12] is automated with ProVerif and is based on a Horn clause model. They show that Microsoft's Bitlocker protocol preserves the secrecy of some data sealed against a static sequence of PCR values. However, their setting is quite abstract and can only be considered as a first step towards a more detailed analysis. For instance, because they do not have a model of dynamically loaded programs, they can only verify a static root of trust with a particular execution chain. Furthermore, there is no way to express a program that has access to some data in a protected environment. This entails that, in order to preserve the security property in their model, there is no way for the attacker to obtain a state with an expected PCR value (which is possible by simply letting the platform execute the expected honest programs). In fact, there is no model of state, and thus code integrity properties can not be expressed either. Our models address these issues.

An interesting result shown in [12] is that, for a class of Horn clauses, it is sound to bound the number of extensions of PCR registers. Since our model is in applied pi-calculus, we can not directly rely on their result, and it is not clear if their syntactic or semantic criteria on Horn clauses would hold in a more general setting. We propose a different approach: instead of decreasing the power of the attacker, we propose to carefully increase it in a way that allows ProVerif to make an efficient abstraction, while avoiding false attacks.

Information-flow security and the TPM. [15] presents a secure compiler for translating programs and policies into cryptographic implementations, distributed on several machines equipped with TPMs. They do not consider the problem of how to secure the source program in the first place, which may already involve the TPM. Their model is computational and quite specific. Our approach is symbolic and we can rely on existing verification tools, while it could be instantiated in any computational setting.

Unbounded search space. Several works are similar in spirit to our reduction result for ProVerif, but technically they are all based on principles that can not be translated to PCR registers. In [23], 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 [8], 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 [17], [16]. Their reductions are based on the principle of data independence for memory stores. Their results are complementary to ours and can be seen as a guarantee that, once loaded with a dynamic root of trust, the protected program (a supervisor in their case) can properly enforce a desired security policy.

In [21] and respectively [2], it is shown how to handle an unbounded number of Diffie-Hellman exponentiations and respectively re-encryptions in ProVerif. Surprisingly, the underlying associative-commutative properties of Diffie-Hellman help in [21], 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 [7]. In order to obtain an automated proof, [7] 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 [22]. 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.

Models with state [3]. Applied pi-calculus is extended with an explicit notion of persistent state in [3]. While verification of protocols relying on TPM are among their motivations, it is not present in their case studies. This illustrates the difficult balance between expressiveness and efficiency. We believe however that our abstractions could be combined with theirs to perform automated verification of more applications.

III. Preliminaries

A. Dynamic root of trust (DRT) protocols

We consider a computer platform equipped with a TPM and a CPU which supports the technology of dynamic measurement and protected execution, for instance Intel's *Trusted Execution Technology* (TXT) or AMD *Secure Virtual Machine* (SVM). In the following, we sketch the main ideas of the functionality and the desired security properties for these platforms.

1) Trusted platform module: In the context of a dynamic root of trust, the *platform configuration registers* (PCRs) of the TPM play a fundamental part. Their role is to store the measurement of loaded programs and in this way to provide evidence about the state of the platform. The application interface of the TPM allows the PCRs to be *reset* only by some privileged instruction of the CPU or by a system reset. On the other hand, they can be *extended* at any time 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. When a program is loaded, the *measurement* (i.e. application of a hash function) of its binary code is extended into a PCR. A sequence of such extensions will result in a PCR value which demonstrates that a particular sequence of programs was run. Secret information can be *sealed* against a set of PCR values, and can only be unsealed by the TPM when its PCRs record the specified value. This allows a loaded program to have access to sensitive data only if the PCRs attest that the platform is in the expected configuration. For simplicity and without loss of generality, we assume that there is only one PCR.

2) Dynamic measurement and protected execution: Assume a program, that we will call $\mathsf{DRT}_{\mathsf{PP}}$ (called *measured launch environment* on INTEL and *secure kernel* on AMD), needs to be loaded in a secure environment. The first entry point of the DRT protocol is a privileged instruction of the CPU, that we will call $\mathsf{DRT}_{\mathsf{CPU}}$ (called *GETSEC[SENTER]* on INTEL and *SKINIT* on AMD). The basic assumption here is that there is no way for the attacker to compromise the code of $\mathsf{DRT}_{\mathsf{CPU}}$, which is protected by hardware. $\mathsf{DRT}_{\mathsf{CPU}}$ resets the PCR to a specific constant p_d marking the start of a protected execution. It is the only process that has the ability to perform such a reset. To help with the establishment of a protected environment, $\mathsf{DRT}_{\mathsf{CPU}}$ then loads and executes another program, that we will call $\mathsf{DRT}_{\mathsf{INIT}}$ (called *SINIT authenticated code module* on INTEL and *secure loader* on AMD). A powerful software attacker could in principle compromise $\mathsf{DRT}_{\mathsf{INIT}}$, and that is why $\mathsf{DRT}_{\mathsf{CPU}}$ extends its measurement into the TPM. Finally, $\mathsf{DRT}_{\mathsf{INIT}}$ creates a protected environment for $\mathsf{DRT}_{\mathsf{PP}}$, extends its measurement into the TPM and loads it.

The execution sequence can be summarized as follows: 1) The $\mathsf{DRT}_{\mathsf{CPU}}$ function receives a DRT request containing the $\mathsf{DRT}_{\mathsf{INIT}}$ code and the $\mathsf{DRT}_{\mathsf{PP}}$ code. The PCR is first reset to p_d and then it is extended with the measurement of the $\mathsf{DRT}_{\mathsf{INIT}}$ code. The communication between the CPU and the TPM is performed on a dedicated private channel named *locality 4*. 2) The system interrupts are disabled. This ensures that no device with direct memory access privileges can interfere with the dynamic root of trust. 3) The $\mathsf{DRT}_{\mathsf{INIT}}$ program is loaded and it computes the measurement of the $\mathsf{DRT}_{\mathsf{PP}}$ program, extending it into the PCR. The communication between $\mathsf{DRT}_{\mathsf{INIT}}$ and the TPM is performed on the private channel *locality 2*. $\mathsf{DRT}_{\mathsf{INIT}}$ also allocates protected memory for the execution of $\mathsf{DRT}_{\mathsf{PP}}$ and invokes it. 4) The execution of $\mathsf{DRT}_{\mathsf{PP}}$ can start. It can establish further protections for its memory space and re-enable interrupts once appropriate interrupt handlers are set. Furthermore, the $\mathsf{DRT}_{\mathsf{PP}}$ program can now request the TPM to unseal data that has been sealed against the current PCR value, and have access to that data in a protected environment. The communication between $\mathsf{DRT}_{\mathsf{PP}}$ and the TPM is performed on the private channel locality 2. 5) Before ending its execution, the $\mathsf{DRT}_{\mathsf{PP}}$ program extends the PCR with a dummy value, to record that the platform state is not to be trusted any more.

3) System management interrupts and software transfer monitor: For efficiency reasons, when a system interrupt is handled, all physical memory can be accessed by the system management interrupt (SMI) handler. In particular, if a dynamic root of trust is running when the interrupt request is received, the SMI handler could access the memory of the protected program $\mathsf{DRT}_{\mathsf{PP}}$. Therefore, it is important that the SMI handler can not be compromised by an intruder, and that is why it is stored in a protected memory area called SMRAM.

However, as is shown in [14] and [27], in the context of a larger system these security safeguards can be bypassed by making use of the CPU caching mechanism. Roughly, these attacks work by noticing that the protection of the SMRAM is not carried on to its cached contents. Then, the attacker can first cache the code of the SMI handler (e.g. by performing an interrupt), then modify the cached version, and finally write back to the main memory a compromised version of the SMI handler.

A possible protection 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 can not be cached while a DRT is running (special registers of the CPU should ensure that), and its role is to protect regions of memory from the SMI handler. As part of this solution, the STM code is also measured and extended into the PCR by the $\mathsf{DRT_{CPU}}$ program, in order to ensure that it is not compromised. We are not aware whether this solution is currently implemented by INTEL or AMD.

4) Security goals: Let us summarize the two main security goals of the dynamic root of trust. Assume given an honest $\mathsf{DRT_{INIT}}$ program P_{init}, an honest STM program P_{stm} and an honest $\mathsf{DRT_{PP}}$ program P_{pp}.

Code integrity: In any execution of the platform, if the measurements recorded in the PCR value of the TPM correspond to the sequence $(p_d, h(P_{init}), h(P_{stm}), h(P_{pp}))$, then the platform is indeed running a dynamic root of trust for the protected execution of P_{pp} in the context of P_{init} and P_{stm}.

Secrecy of sealed data: Any secret data that is sealed only against a PCR value recording $(p_d, h(P_{init}), h(P_{stm}), h(P_{pp}))$, is only available to the program P_{pp}, in any execution of the platform.

B. Applied pi-calculus and ProVerif

The ProVerif calculus [5], [6] is a language for modelling distributed systems and their interactions. It is a dialect of applied pi calculus [1], [24]. In this section, we briefly review the basic notions of applied pi-calculus and ProVerif.

1) Terms and equational theories: We assume given an infinite set of *names*, $a, b, c, k, n \ldots$, an infinite set of *variables*, $x, y, z, \ldots$ and a possibly infinite *signature* $\mathcal{F}$. A signature is a set of *function symbols*, each with an associated arity. Then, names and variables are basic terms and new terms can be built by applying a function symbol $f \in \mathcal{F}$ to names, variables and other terms. Thus, if $t_1, \ldots, t_n$ are terms and $f \in \mathcal{F}$ is of arity n (fact denoted by f/n subsequently), then $f(t_1, \ldots, t_n)$ is a term. We define $top(f(t_1, \ldots, t_n)) = f$. The set of terms (also called messages) built from a set of names $\mathcal{N}$, a set of variables $\mathcal{X}$ and a signature $\mathcal{F}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{X}, \mathcal{N})$. The set of messages that do not contain names is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Terms without variables are called ground. For a term t, we denote by $fn(t)$ the names that occur in t and by $st(t)$ the subterms of t.

Furthermore, $\mathcal{F}$ is split into a set of *public* functions $\mathcal{F}^{pub}$ and a set of *private* functions $\mathcal{F}^{priv}$: $\mathcal{F} = \mathcal{F}^{pub} \cup \mathcal{F}^{priv}, \mathcal{F}^{pub} \cap \mathcal{F}^{priv} = \emptyset$. Public functions can be applied by anyone, including the attacker, whereas the private functions can be applied only as specified by the protocol. When $\mathcal{F}^{priv}$ is not explicitly specified, we assume that all functions are public.

A substitution σ is a partial function from the set of variables to the set of terms. The application of a substitution σ to a term u (resp. a proces P) is the term $u\sigma$, called an instance of u, (resp. the process $P\sigma$, called an instance of P) obtained by replacing every variable x of u (resp. of P) with the corresponding term $x\sigma$. The domain of a substitution σ is denoted by $\mathsf{dom}(\sigma)$, and its range is denoted by $\mathsf{ran}(\sigma)$. We say that a substitution σ' extends σ if $\mathsf{dom}(\sigma) \subseteq \mathsf{dom}(\sigma')$ and $\forall x \in \mathsf{dom}(\sigma).\ x\sigma' = x\sigma$.

We consider a special set of names $\Upsilon = \{\epsilon, \epsilon_1, \epsilon_2, \ldots\}$ that will be used to specify contexts. A *context* is a term C in $\mathcal{T}(\mathcal{F}, \mathcal{X}, \Upsilon)$, denoted by $C[\epsilon_1, \ldots, \epsilon_1]$, when $fn(C) \cap \Upsilon = \{\epsilon_1, \ldots, \epsilon_n\}$. An instance of a context $C[\epsilon_1, \ldots, \epsilon_n]$ is obtained by replacing each name ϵ_i with a term t_i, for all $1 \leq i \leq n$, and is denoted by $C[t_1, \ldots, t_n]$.

The semantics of terms is given by a set $\mathcal{E}$ of equations $u_1 = v_1, \ldots, u_n = v_n$, where $u_1, \ldots, u_n, v_1, \ldots, v_n$ are terms with variables. The set of equations should capture the logical properties of algorithms that are modeled by function symbols. Then, two terms u, v are equal, denoted by $u =_{\mathcal{E}} v$ (or simply $u = v$ when $\mathcal{E}$ is clear from the context), if one term can be derived from the other by applying any number of times some of the given equations [4], [13]. A pair formed of a signature and a set of equations, $(\mathcal{F}, \mathcal{E})$, will be called an equational theory. $\mathcal{E}$ will also be called an equational theory when $\mathcal{F}$ is clear from the context.

ProVerif specifics. ProVerif has a set of special symbols that allows the construction/destruction of tuples: for all n, one can derive $(t_1, \ldots, t_n)$ from $t_1, \ldots, t_n$, and conversely. We will assume the presence of this symbol implicitly throughout the paper.

2) Processes and operational semantics: Processes of the calculus are built according to the grammar given in figure 1, where u, v are terms, n is a name and x is a variable. The operational semantics of these processes is quite standard and we refer to [6], [24] for formal details. Let us explain informally the less obvious constructs. Replication allows the creation of any number of instances of a process: formally, $!P$ is equivalent with $P \mid !P$. The process $\mathsf{let}\ u = v\ \mathsf{in}\ P\ \mathsf{else}\ Q$ executes as follows: if there is a substitution σ such that $u\sigma = v$, then $P\sigma$ is executed; otherwise, Q is executed. The frame element $\{u\}$ represents a message u that has been previously sent to the attacker on a public channel. We define the frame of a process P, denoted by $fr(P)$, to be the union of all the frame elements of P.

Names that are introduced by a new construct are called *bound* or *private*, and they represent the creation of fresh and secret data. Variables that are introduced in the term x of an input or the term u of a let construct are called *bound*, and they represent the reception or computation of fresh data. Names and variables that are not bound are called *free*, or *public*. We denoted by $fv(P)$ and respectively $fn(P)$ the set of free variables and respectively free names of P.

$P, Q, R ::=$	processes
0	null process
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\, n; P$	name restriction
$\mathsf{in}(v, x); P$	message input on v
$\mathsf{out}(v, u); P$	message output on v
$\{u\}$	frame element
$\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
$\mathsf{let}\ u = v\ \mathsf{in}\ P\ \mathsf{else}\ Q$	term evaluation

Fig. 1. Process algebra

The result of applying several execution steps to a process P is called a trace of P, denoted by $P \xrightarrow{w} Q$, where Q is the process obtained after the trace is executed and w is a sequence of labels recording the actions (e.g. input on channel c, output on channel c) that have determined the trace. All messages sent on public channels in the trace $P \xrightarrow{w} Q$ are recorded in $fr(Q)$ - the frame of Q.

ProVerif specifics. ProVerif allows a syntactic sugar that we will use in the specification of our case study. We can write a process of the form $\mathsf{in}(v, (A_1, \ldots, A_n)); P$ where each A_i is either a variable, or an expression of the form $= t_i$, for some term t_i. Let us define $\phi(A_i) = A_i$, if A_i is a variable, and $\phi(A_i) = t_i$ if A_i is of the form $= t_i$. Then, the process described above is an abbreviation for the process $\mathsf{in}(v, x); \mathsf{let}\ x = (\phi(A_1), \ldots, \phi(A_n))\ \mathsf{in}\ P$. This allows us to specify filters on tuples received on a channel v.

3) Security properties: In this paper, we will rely on secrecy properties [5] and on correspondence assertions [6]. First, we define the computational power of the attacker.

Deducibility. Assume given an equational theory $(\mathcal{F}, \mathcal{E})$. The ability of an attacker to obtain new knowledge by performing operations on known messages is captured by the notion of deducibility. For a set of terms T and a term t, we define the deducibility relation $T \vdash_{\mathcal{E}} t$ (or simply $T \vdash t$ when $\mathcal{E}$ is clear from the context) as being true if and only if

- there exists a term $t' \in T$ such that $t =_{\mathcal{E}} t'$ or
- there are terms $t_1, \ldots, t_n$ such that $T \vdash_{\mathcal{E}} t_1, \ldots, T \vdash_{\mathcal{E}} t_n$ and a function symbol $f \in \mathcal{F}^{pub}$ such that $f(t_1, \ldots, t_n) =_{\mathcal{E}} t$

Secrecy. For any term t and process P, the ability of an attacker to deduce t by interacting with P is captured by the formula $P \models_{\mathcal{E}} t$, or simply $P \models t$ when $\mathcal{E}$ is clear from the context. By definition, $P \models t$ is true when there exists an execution trace $P \xrightarrow{w} Q$ such that $fr(Q) \vdash t\sigma$, for some substitution σ. The formula $P \not\models t$ is true when $P \models t$ is false. Thus, if $P \not\models t$ is true, then t is kept secret in any execution of P.

Correspondence assertions. In this paper we only need a simplified version of correspondence assertions, that we explain in the following. A (simple) correspondence assertion is a formula of the form $P \models_{\mathcal{E}} t \rightsquigarrow u = v$, for some terms u, v, t. It is satisfied if, whenever the attacker can deduce t by interacting with P, it is the case that u equals v. Formally, we have $P \models_{\mathcal{E}} t \rightsquigarrow u = v$ if and only if for any execution trace $P \xrightarrow{w} Q$ and substitution σ such that $fr(Q) \vdash_{\mathcal{E}} t\sigma$, we have $u\sigma' =_{\mathcal{E}} v\sigma'$, for some substitution σ' that extends σ.

IV. Platform states and protected programs in applied pi-calculus

In this section we present our proposed generic models of platform execution (subsection IV-A) and of protected, dynamically loaded programs (section IV-B). We also link the two models, by showing how to assign execution capabilities to loaded programs (section IV-C).

A. Platform execution as an equational theory

Definition 1: A *state structure* is a pair $(\mathcal{F}, \mathcal{S})$, where $\mathcal{F} = \mathcal{F}_{struct} \cup \mathcal{F}_{data}$ is a signature and $\mathcal{S}$ is a term such that

- $\mathcal{F}_{struct} \subseteq \mathcal{F}^{priv}$ and
- $\mathcal{S} \in \mathcal{T}(\mathcal{F}_{struct}, \mathcal{X})$

A term t_s for which there are substitutions σ, θ such that $t_s\sigma = \mathcal{S}\theta$ is called a *state term.* □

The idea is that the term $\mathcal{S}$ represents the structure of a platform state. Its symbols are assumed to be private so that an attacker can not arbitrarily construct platforms states, but only reach them following the specification of the platform (via equations and interactions with other agents, as shown below). The variables of $\mathcal{S}$ represent the mutable elements of the platform state, and the attacker or other agents may be able to modify them. A particular state is represented by a ground instance of $\mathcal{S}$. A state term t_s represents the set of states formed of all its ground instances.

Example 1: Consider the state structure where

$$\begin{aligned} \mathcal{F}_{struct} &= \{state/3, cpu/2, ram/1, tpm/1\} \\ \mathcal{F}_{data} &= \{f/1, a/0, b/0, c/0\} \\ \mathcal{S} &= state(cpu(x_1, x_2), ram(x_3), tpm(x_4)) \end{aligned}$$

Here, $\mathcal{S}$ represents a state where two CPU registers have values x_1, x_2, a zone of the RAM memory has value x_3 and a PCR register of the TPM has value x_4.

The term $state(cpu(a, f(a)), ram(f(f(b))), tpm(c))$ represents a particular platform state, while $state(y_1, ram(x_2), y_2)$ and $state(y_1, y_2, tpm(b))$ are state terms. □

Definition 2: An equational theory $(\mathcal{F}_{get}, \mathcal{E}_{get})$ provides *read access* for a state structure $(\mathcal{F}_{struct} \cup \mathcal{F}_{data}, \mathcal{S})$, if each equation in $\mathcal{E}_{get}$ is of the form:

$$C_{get}[t_{state}] = t$$

where t_{state} is a state term, $C_{get}[\epsilon]$ is a context in $\mathcal{T}(\mathcal{F}_{get} \cup \mathcal{F}_{data}, \mathcal{X}, \{\epsilon\})$ and $t \in st(t_{state})$. □

This way, for any substitution σ, the context $C_{get}\sigma[\epsilon]$ represents the actions and credentials that allow the access to the part $t\sigma$ of the platform state $t_{state}\sigma$.

Example 2: Continuing example 1, the following equations can be part of $\mathcal{E}_{get}$:

$$\begin{aligned} get\_cpu(state(cpu(x_1, x_2), y_1, y_2), (cpu\_acc, one)) &= x_1 \\ get\_ram(state(y_1, ram(x), y_2)) &= x \\ get\_pcr(state(y_1, y_2, tpm(x))) &= x \end{aligned}$$

where $cpu\_acc/0 \in \mathcal{F}_{get}^{priv}$. Thus, the read access to CPU is restricted, while the access to RAM and TPM is public. □

Definition 3: An equational theory $(\mathcal{F}_{set}, \mathcal{E}_{set})$ provides *write access* for a state structure $(\mathcal{F}_{struct} \cup \mathcal{F}_{data}, \mathcal{S})$ if each equation in $\mathcal{E}_{set}$ is of the form

$$C_{set}[t_{state}] = t'_{state}$$

where t_{state}, t'_{state} are state terms and $C_{set}[\epsilon]$ is a context in $\mathcal{T}(\mathcal{F}_{set} \cup \mathcal{F}_{data}, \mathcal{X}, \{\epsilon\})$. □

This way, for any substitution σ, the context $C_{set}\sigma[\epsilon]$ represents the actions, credentials and update values that alllow the modification of the platform state from $t_{state}\sigma$ to $t'_{state}\sigma$.

Example 3: Continuing example 2, we can have the following equations as part of $\mathcal{E}_{set}$:

$$\begin{aligned} set\_cpu(state(cpu(x_1, x_2), y_1, y_2), (cpu\_acc, one), xv) = \\ state(cpu(xv, x_2), y_1, y_2) \\ set\_pcr(state(y_1, y_2, tpm(x)), tpm\_acc, xv) = \\ state(y_1, y_2, tpm(xv)) \end{aligned}$$

where $\mathcal{F}_{set}^{priv} = \mathcal{F}_{get}^{priv} \cup \{tpm\_acc/0\}$. Thus, while the read access to the TPM is public, the write access is restricted by the private constant tpm_acc. □

Platform execution. Now, as we will see in section V, the participants of a protocol executing on a platform can simply obtain a platform state by reading the corresponding term on a public channel, modify the platform state according to their capabilities, and update the platform state by transmiting the new term on a public channel.

B. Protected programs as protected data

Higher-order process calculi. We have argued in the introduction and illustrated in preliminaries that we need a model for dynamically loaded programs, in order to have a faithful representation of the dynamic root of trust. It seems then that we need to make use of higher-order process calculi, similar to the ones in e.g. [26], [25]. Then, we would have a rule of the form:

$$\mathsf{out}(c, Q) \mid \mathsf{in}(c, X).P \to P \mid Q$$

In fact, it has been shown [26], [25] that such behavior can be encoded in standard first-order process calculi. The same should be the case for higher-order applied pi-calculus. However, this kind of behaviour is not sufficient to model what we want in terms of program protection. For instance, once a program Q is "loaded", there is no guarantee that Q is executed and not an arbitrary program Q'. Indeed, we have

$$Q' \mid \mathsf{out}(c, Q) \mid \mathsf{in}(c, X).P \to P \mid Q' \mid Q$$

and there is no way in the model to make a difference between Q and Q'. Furthermore, the "loaded" program Q should get something in return, like privileged access to some elements of the platform state.

Desired functionality. Informally, we would like to have a rule of the form

$$\mathsf{out}(c, Q) \mid \mathsf{in}(c, X).P \to P\ [\ Q\ ]$$

where the semantics of $P[\ Q\ ]$ would capture the properties that we hinted at above: there is a particular channel that allows P to communicate with Q and (possibly another channel) that allows Q to have access to the platform state. While this is interesting in theory, we believe an encoding at the first order level is possible, following the same lines as [26], [25]. Therefore, we propose directly a first-order model, so that we can rely on ProVerif for automated verification. We leave a precise link between the following model and a higher order calculus out of the scope of this paper.

Protected and dynamically loaded programs. We consider a new public function symbol $f_{prog}/1$ and an infinite signature of private constants $\mathcal{F}_{\mathcal{P}}$, containing a different constant n_P for every possible process P.

Definition 4: For a process P, free variables $x_1, \ldots, x_n \in fv(P)$ and a public name c, we define the process

$$P_{prog,c}^{x_1,\ldots,x_n} = \mathsf{out}(c, f_{prog}(n_P)); \mathsf{in}(n_P, (x_1, \ldots, x_n)); P$$

where n_P is the constant from $\mathcal{F}_{\mathcal{P}}$ that corresponds to P. □

Intuitively, the term $f_{prog}(n_P)$ is a public and unique identifier for the program P (for instance, $f_{prog}(n_P)$ may be computed from the source code of P). On the other hand, the constant n_P represents a *private entry point* for the program P. It will be used for communication between loaded instances of P and the loader. The variables $x_1, \ldots, x_n$ represent the parameters that the program P expects to receive after loading.

Definition 5: An equational theory $(\mathcal{F}_{prog}, \mathcal{E}_{prog})$ models *program access* if $\mathcal{F}_{prog}^{pub} = \{f_{prog}/1\}$, $\mathcal{F}_{prog}^{priv} = \{f_{entry}/1\}$ and $\mathcal{E}_{prog} = \{f_{entry}(f_{prog}(x)) = x\}$ □

The idea is that a trusted loader of programs (the CPU in our case) will have access to the private function f_{entry}. $\mathcal{E}_{prog}$ will then permit the loader to gain access to the private entry point of any program. Now, for a program P to be loaded, the first step is to replace P with $P_{prog,c}^{x_1,\ldots,x_n}$, allowing the loader to obtain n_P. Next, the loader will prepare parameters $t_1, \ldots, t_n$ for P and send them on the private channel n_P, that is now shared between the loader and the loaded program P.

This model achieves dynamical loading of programs (any program can play the role of P) and their protected execution (by relying on the private channel that is set up during loading). Furthermore, if we consider a state structure as introduced in section IV-A, we can now store identifiers of programs in the platform state and we can be certain that, if the loader is honest, then the identity $prog(n_P)$ stored in the platform state indeed corresponds to the program running on the platform.

Example 4: Continuing example 3, let us model a program that, once loaded, gets access to the TPM and sets its PCR to a particular value $f(a)$. For simplicity, we assume the loaded program is stored in one of the CPU registers.

$$\begin{aligned} P = {} & \mathsf{out}(c, f_{prog}(n_P)); \mathsf{in}(n_P, (x_{pf\_state}, x_{tpm\_acc})); \\ & \mathsf{let}\ x'_{pf\_state} = set\_pcr(x_{pf\_state}, x_{tpm\_acc}, f(a))\ \mathsf{in} \\ & \mathsf{out}(c, x'_{pf\_state}) \end{aligned}$$

$$\begin{array}{l} Q = \mathsf{in}(c, x_p); \mathsf{in}(c, x_{pf\_state}) \\ \quad \mathsf{let}\ x'_{pf\_state} = set\_cpu(x_{pf\_state}, (cpu\_acc, one), x_p)\ \mathsf{in} \\ \quad \mathsf{let}\ x_{entry} = f_{entry}(x_p)\ \mathsf{in} \\ \quad \mathsf{out}(x_{entry}, (x'_{pf\_state}, tpm\_acc)); \mathsf{out}(c, x'_{pf\_state}) \end{array}$$

C. Platform states and loaded programs

Looking back at examples 1-4, we may note that the access to the TPM is modeled too liberally. Indeed, once a process gets the TPM access in a state, it can use it to access the TPM in any state. One way to solve this problem is to model the TPM access with a term that contains more information, and not simply with a constant. Yet, in this subsection we propose another model, which allows concise specifications of loaded program abilities. A context whose instances are state terms will be called a *state context*.

Definition 6: Let $(\mathcal{F}, \mathcal{S})$ be a state structure and $(\mathcal{F}_{prog}, \mathcal{E}_{prog})$ be an equational theory that models program access. We say that an equational theory models *loaded program abilities* if it is defined by a set of equations of the form:

$$\begin{array}{l} C_{get}[C[f_{prog}(x_1), \ldots, f_{prog}(x_n)], x_1, \ldots, x_n] = t \\ C_{set}[C[f_{prog}(x_1), \ldots, f_{prog}(x_n)]), x_1, \ldots, x_n] = t_s \end{array}$$

where C is a state context, C_{set}, C_{get} are contexts, t is a term in $st(C[f_{prog}(x_1), \ldots, f_{prog}(x_n)])$ and t_s is a state term. □

The idea is that, if the programs $f_{prog}(x_1), \ldots, f_{prog}(x_n)$ have been loaded in certain locations of the platform state (specified by the context C), then these programs can access or write some elements of the platform state by relying on their private entry points, and possibly some other private data specified by C_{get}, C_{set}.

We use a conjunction of entry points because in some cases one program of the platform state may monitor another program and prevent it from accessing some protected memory. In that case, an intruder needs to control both programs in order to modify the respective portion of the platform state.

Example 5: Continuing example 4, instead of having Q send tpm_acc to P, we may rely on the following equation to allow P to modify the contents of the TPM only while it is loaded in the first of the CPU registers:

$$\begin{array}{r} set\_pcr(state(cpu(f_{prog}(x), y), y_1, tpm(y_2)), x, xv) = \\ state(cpu(f_{prog}(x), y), y_1, tpm(xv)) \end{array}$$

V. Formal models for the case study

A. Attacker model, assumptions and simplifications

We assume the presence of a powerful attacker who controls the operating system and is able to execute a dynamic root of trust any number of times with any programs. In particular, it can compromise the $\mathsf{DRT_{INIT}}$ program and, by relying on the CPU cache, it can compromise the STM and the SMI handler. It can furthermore use the compromised STM and SMI handler to access protected memory. The attacker has access to all TPM functions, and is able to perform any number of static PCR resets and extend the PCR any number of times with any data. It can request the unseal of any data, which will only be unsealed if the values of the PCR permit it.

To achieve the desired security properties, we assume that the attacker can not compromise the CPU and the TPM - this is the fundamental DRT assumption. In particular, only the CPU can execute a dynamic reset of the PCR. Furthermore, the attacker can not compromise the STM while a dynamic root of trust is running. As usual, we assume that cryptography can not be broken.

For simplicity, and without loss of generality, we omit the so called TPM authdata, which permits users to authenticate to the TPM. This only means that we extend the attacker's power, by providing all authdata to the attacker. We also ommit some functions of the TPM that are not relevant for our purposes and have been handled elsewhere, such as those for loading, certifying and wrapping keys [12], [11].

We have abstract models of memory and of the CPU cache: we only model locations where our elements of interest lie, and we express by equations how these can be changed, directly or by flushing the cache.

B. Model of data

We consider

$$\begin{array}{rl} \mathcal{F}^{pub}_{data} = \{ & p_s/0, p_d/0, true/0, false/0, \\ & h/1, senc/2, sdec/2, seal/2\} \end{array}$$

$\mathcal{F}^{priv}_{data} = \{unseal/2\}$ and the set of equations $\mathcal{E}_{data}$:

$$\begin{array}{rcl} sdec(x, senc(x, y)) & = & y \\ unseal(seal(x_{pcr}, x_{val}), x_{pcr}) & = & x_{val} \end{array}$$

The constant p_d (resp. p_s) represents the result of a dynamic (resp. static) PCR reset. A dynamic reset is especially relevant for our purposes, because it marks the start of a dynamic root of trust. The functions $senc$ and $sdec$, and the corresponding equation, model symmetric key encryption. We use a free symbol h to represent a hash function. To model sealing and unsealing functions of the TPM, we have the the symbols $seal$ and $unseal$. Anyone can seal a value, while the corresponding equation and $unseal \in \mathcal{F}^{priv}$ ensure that only the TPM can unseal it by providing the PCR values that have been used for sealing.

C. Model of the platform

1) State structure: As instance of definition 1, we consider the state structure defined by $(\mathcal{F}_{struct} \cup \mathcal{F}_{data}, \mathcal{S})$, where $\mathcal{F}_{data}$ is defined above and

$$\begin{array}{rcl} \mathcal{F}_{struct} & = & \{state/4, tpm/1, cpu/2, drt/3, smram/2\} \\ \mathcal{S} & = & state(\ tpm(x_{pcr}), cpu(x_{int}, x_{cache}), \\ & & \quad drt(x_{init}, x_{pp}, x_{lock}), smram(x_{stm}, x_{smi})) \end{array}$$

where x_{pcr} stands for the value of the PCR register of the TPM; x_{int} represents a register of the CPU showing if interrupts are enabled; x_{cache} represents the contents of the CPU cache; x_{lock} is showing if a dynamic root of trust is running; x_{init} represents the $\mathsf{DRT_{INIT}}$ program; x_{pp} represents the protected program $\mathsf{DRT_{PP}}$; located in SMRAM, x_{smi} is the SMI handler and x_{stm} is the STM program.

Fig. 2. Equations for modifying the platform state

$$
\begin{aligned}
\mathit{reset\_pcr}(& \mathit{state}(\mathit{tpm}(y), x_1, x_2, x_3), \mathit{tpm\_acc}, p_s) \\
= & \mathit{state}(\mathit{tpm}(p_s), x_1, x_2, x_3) \\
\mathit{reset\_pcr}(& \mathit{state}(\mathit{tpm}(y), x_1, x_2, x_3), \mathit{tpm\_acc}, p_d) \\
= & \mathit{state}(\mathit{tpm}(p_d), x_1, x_2, x_3) \\
\mathit{extend\_pcr}(& \mathit{state}(\mathit{tpm}(y), x_1, x_2, x_3), \mathit{tpm\_acc}, v) \\
= & \mathit{state}(\mathit{tpm}(h((y, v))), x_1, x_2, x_3) \\
\mathit{set\_int}(& \mathit{state}(x_1, \mathit{cpu}(y_1, y_2), x_2, x_3), \mathit{cpu\_acc}, v) \\
= & \mathit{state}(x_1, \mathit{cpu}(v, y_2), x_2, x_3) \\
\mathit{cache}(& \mathit{state}(x_1, \mathit{cpu}(y_1, y_2), x_2, x_3), v) \\
= & \mathit{state}(x_1, \mathit{cpu}(y_1, v), x_2, x_3) \\
\mathit{flush\_stm}(& \mathit{state}(x_1, \mathit{cpu}(y_1, y_2), \mathit{drt}(w_1, w_2, \mathit{false}), \\
& \qquad \mathit{smram}(z_1, z_2))) \\
= & \mathit{state}(x_1, \mathit{cpu}(y_1, v), \mathit{drt}(w_1, w_2, \mathit{false}) \\
& \qquad \mathit{smram}(y_2, z_2))) \\
\mathit{flush\_smi}(& \mathit{state}(x_1, \mathit{cpu}(y_1, y_2), x_2, \mathit{smram}(z_1, z_2))) \\
= & \mathit{state}(x_1, \mathit{cpu}(y_1, y_2), x_2, \mathit{smram}(z_1, y_2)) \\
\mathit{set\_init}(& \mathit{state}(x_1, x_2, \mathit{drt}(y_1, y_2, y_3), x_3), \mathit{cpu\_acc}, v) \\
= & \mathit{state}(x_1, x_2, \mathit{drt}(v, y_2, y_3), x_3) \\
\mathit{set\_pp}(& \mathit{state}(x_1, x_2, \mathit{drt}(y_1, y_2, y_3), x_3), \mathit{cpu\_acc}, v) \\
= & \mathit{state}(x_1, x_2, \mathit{drt}(y_1, v, y_3), x_3) \\
\mathit{set\_lock}(& \mathit{state}(x_1, x_2, \mathit{drt}(y_1, y_2, y_3), x_3), \mathit{cpu\_acc}, v) \\
= & \mathit{state}(x_1, x_2, \mathit{drt}(y_1, y_2, v), x_3)
\end{aligned}
$$

2) Read access: The read access is universal: we simply assume that any agent who has access to a platform state can read any of its components. Let $\mathcal{W}_{\mathcal{S}} = \{\mathit{pcr}, \mathit{int}, \mathit{cache}, \mathit{init}, \mathit{pp}, \mathit{lock}, \mathit{stm}, \mathit{smi}\}$. As instance of definition 2, we consider the equational theory $(\mathcal{F}_{get}, \mathcal{E}_{get})$, where

$$
\begin{aligned}
\mathcal{F}_{get} &= \{\mathit{get\_w}/1 \mid \text{ for all } w \in \mathcal{W}_{\mathcal{S}}\} \\
\mathcal{E}_{get} &= \{\mathit{get\_w}(\mathcal{S}) = x_w \mid \text{ for all } w \in \mathcal{W}_{\mathcal{S}}\}
\end{aligned}
$$

and all symbols of $\mathcal{F}_{get}$ are public.

3) Write access: As instance of definition 3, we have the equational theory $(\mathcal{F}_{set}, \mathcal{E}_{set})$, where $\mathcal{F}_{set}^{priv} = \{\mathit{cpu\_acc}/0, \mathit{tpm\_acc}/0\}$ and $\mathcal{E}_{set}$ is defined in figure 2. The private constants $\mathit{cpu\_acc}$ and $\mathit{tpm\_acc}$ allow only the CPU and the TPM to change certain components of the state. Anyone can cache a value and then request the cache to be copied into the SMRAM. However, our equation for $\mathit{flush\_stm}$ ensures that the STM can be modified in this way only if a DRT is not running. We can modify the equation for $\mathit{flush\_stm}$ to allow changing the STM in any state, and then we recover the attacks of [14], [27] in our model.

4) Abilities of loaded programs: Furthermore, some components of the state can also be changed by other programs running on the platform. To model this ability, we rely on an instance of definition 6 described in figure 3, where the symbol $\mathit{prog}/1$ plays the role of f_{prog}. The $\mathsf{DRT_{INIT}}$ program can modify the $\mathsf{DRT_{PP}}$ program. The $\mathsf{DRT_{PP}}$ program can set/unset the DRT lock and enable/disable interrupts. If the interrupts are enabled, the STM and the SMI handler together can modify the protected program and the DRT lock.

Fig. 3. Abilities of loaded programs

$$
\begin{aligned}
\mathit{set\_pp}(\quad & \mathit{state}(&& x_1, x_2, \mathit{drt}(\mathit{prog}(y_1), y_2, y_3), x_3), y_1, v) \\
= \quad & \mathit{state}(&& x_1, x_2, \mathit{drt}(\mathit{prog}(y_1), v, y_3), x_3) \\
\mathit{set\_pp}(\quad & \mathit{state}(&& x, \mathit{cpu}(\mathit{true}, z), \mathit{drt}(y_1, y_2, y_3), \\
& && \mathit{smram}(\mathit{prog}(z_1), \mathit{prog}(z_2))), (z_1, z_2), v) \\
= \quad & \mathit{state}(&& x, \mathit{cpu}(\mathit{true}, z), \mathit{drt}(y_1, v, y_3), \\
& && \mathit{smram}(\mathit{prog}(z_1), \mathit{prog}(z_2))) \\
\mathit{set\_lock}(\quad & \mathit{state}(&& x_1, x_2, \mathit{drt}(y_1, \mathit{prog}(y_2), y_3), x_3), y_2, v) \\
= \quad & \mathit{state}(&& x_1, x_2, \mathit{drt}(y_1, \mathit{prog}(y_2), v), x_3) \\
\mathit{set\_lock}(\quad & \mathit{state}(&& x, \mathit{cpu}(\mathit{true}, z), \mathit{drt}(y_1, y_2, y_3), \\
& && \mathit{smram}(\mathit{prog}(z_1), \mathit{prog}(z_2))), (z_1, z_2), v) \\
= \quad & \mathit{state}(&& x, \mathit{cpu}(\mathit{true}, z), \mathit{drt}(y_1, y_2, v), \\
& && \mathit{smram}(\mathit{prog}(z_1), \mathit{prog}(z_2))) \\
\mathit{set\_int}(\quad & \mathit{state}(&& x_1, \mathit{cpu}(y, z), \mathit{drt}(z_1, \mathit{prog}(z_2), \mathit{true}), x_2), \\
& && \qquad z_2, v) \\
= \quad & \mathit{state}(&& x_1, \mathit{cpu}(v, z), \mathit{drt}(z_1, \mathit{prog}(z_2), \mathit{true}), x_2)
\end{aligned}
$$

D. Processes for the dynamic root of trust

We use a public channel os to communicate platform states and any other messages that may be intercepted and modified by the intruder. We use the program access signature $\{\mathit{prog}/1, \mathit{get\_entry}/1\}$ to model loading of programs and access to their entry points as described in definitions 4 and 5. Corresponding to locality 4, a private name $\mathit{cpu\_tpm}$ models the secure channel between the CPU and the TPM. Corresponding to locality 2, a private function $\mathit{tpm\_ch}/1$ models the ability of the CPU to establish a private channel between a running program and the TPM. Generally, these channels will be of the form $\mathit{tpm\_ch}(\mathit{prog}(t))$ and the CPU will send this term both to the program represented by $\mathit{prog}(t)$ (on channel t) and to the TPM (on channel $\mathit{cpu\_tpm}$). We also use message tags that should be clear from the context. We describe the actions of each process in comments.

1) The CPU role: The process $\mathsf{DRT_{CPU}}$, specifying the execution of a dynamic root of trust on the CPU, is defined in figure 4. Note that the measurement extended in to the TPM includes not only the $\mathsf{DRT_{INIT}}$ program, but also the running STM, to ensure that it is not compromised. The process also establishes a shared private channel $\mathit{tpm\_ch}(\mathit{drt\_init})$ between $\mathsf{DRT_{INIT}}$ and TPM. This will be used by $\mathsf{DRT_{INIT}}$ to securely extend the PCR with the measurement of $\mathsf{DRT_{PP}}$. After the $\mathsf{DRT_{INIT}}$ program has measured the $\mathsf{DRT_{PP}}$ program and loaded it into memory, the CPU gets back the new platform state and sets up the private channel for communication between the loaded $\mathsf{DRT_{PP}}$ and the TPM.

2) The $\mathsf{DRT_{INIT}}$ *role:* We specify the behaviour of an honest $\mathsf{DRT_{INIT}}$ program, relying on the process $\mathsf{EXP_{INIT}}$ from figure 5. We assume a private constant $\mathit{exp\_init} \in \mathcal{F}_{\mathcal{P}}$ to be its private entry point. Note that we make use of the special equation that allows the running $\mathsf{DRT_{INIT}}$ program to set the $\mathsf{DRT_{PP}}$ program.

3) An example $\mathsf{DRT_{PP}}$ *program:* We illustrate the execution of a $\mathsf{DRT_{PP}}$ program with an example. Furthermore, this

Fig. 4. The $\mathsf{DRT_{CPU}}$ process

```
(*** Receive a DRT request ***)
in$(os, (= drt\_req, drt\_init, drt\_pp, pf\_state))$
if $get\_lock(pf\_state) = false$ then
(*** Disable interrupts and set the DRT lock ***)
let $s'_0 = set\_int(pf\_state, cpu\_acc, false)$ in
let $s_0 = set\_lock(s'_0, cpu\_acc, true)$ in
(*** Reset the PCR ***)
new $nonce$; out$(cpu\_tpm, pcr\_reset\_req, nonce, s_0)$;
in$(cpu\_tpm, pcr\_reset\_resp, = nonce, s_1)$;
(*** Extend the PCR with the measurement ***)
let $m = (h(drt\_init), h(get\_stm(pf\_state)))$ in
out$(cpu\_tpm, (pcr\_extend\_req, nonce, s_1, m))$;
in$(cpu\_tpm, (= pcr\_extend\_resp, = nonce, s_2))$;
(*** Load $\mathsf{DRT_{INIT}}$ and establish TPM access ***)
let $s_3 = set\_init(s_2, cpu\_acc, drt\_init)$ in
let $einit = get\_entry(drt\_init)$ in
out$(einit, (drt\_req, nonce, s_3, tpm\_ch(drt\_init), drt\_pp))$;
out$(cpu\_tpm, (drt\_channel, tpm\_ch(drt\_init))))$;
in$(einit, (= drt\_resp, = nonce, new\_state))$;
(*** Establish TPM channels for the loaded $\mathsf{DRT_{PP}}$ ***)
let $epp = get\_entry(get\_pp(new\_state))$ in
out$(epp, (drt\_start, new\_state, tpm\_ch(prog(epp))))$;
out$(cpu\_tpm, (drt\_channel, tpm\_ch(prog(epp))))$;
out$(cpu\_tpm, (drt\_start, new\_state, tpm\_ch(prog(epp))))$
```

Fig. 5. The $\mathsf{EXP_{INIT}}$ process: an honest $\mathsf{DRT_{INIT}}$ program

```
out$(os, prog(exp\_init)))$;
(*** Receive $\mathsf{DRT_{PP}}$ and TPM channel from the CPU ***)
in$(exp\_init, (= drt\_req, nonce_0, pf\_state, tpmc, drt\_pp))$;
(*** Measure and extend $\mathsf{DRT_{PP}}$ into the PCR ***)
let $m = h(drt\_pp)$ in new $nonce$;
out$(tpmc, (pcr\_extend\_req, nonce, pf\_state, m))$;
in$(tpmc, (= pcr\_extend\_resp, = nonce, ext\_state))$;
(*** Load $\mathsf{DRT_{PP}}$ on the platform state ***)
let $new\_state = set\_pp(ext\_state, exp\_init, drt\_pp)$ in
(*** Pass the control back to CPU ***)
out$(exp\_init, (drt\_resp, nonce_0, new\_state)))$;
(*** Make the new platform state public ***)
out$(os, new\_state)$
```

will allow us to verify that secret data that is sealed for use by a particular program remains secret. Assume our $\mathsf{DRT_{PP}}$ program has the public identity $prog(exp\_pp)$ and private entry point $exp\_pp \in \mathcal{F_P}$. We consider a fresh symmetric key k_{pp} and assume that this key has been sealed against the expected $\mathsf{DRT_{PP}}$, the expected $\mathsf{DRT_{INIT}}$ and the expected STM (with identity $prog(exp\_stm)$). This is represented by

Fig. 6. The $\mathsf{EXP_{PP}}$ process: an example $\mathsf{DRT_{PP}}$ program

```
out$(os, prog(exp\_pp)))$;
(*** Receive TPM access from the CPU ***)
in$(exp\_pp, (= drt\_start, pf\_state_0, tpmc))$;
(*** Re-enable interrupts ***)
let $pf\_state = set\_int(pf\_state_0, exp\_pp, true)$ in
out$(os, pf\_state)$;
(*** Unseal the key $k_{pp}$ and decrypt the private message ***)
in$(os, x_{seal})$; in$(os, x_{enc})$;
out$(tpmc, (tag\_unseal, x_{seal}))$; in$(tpmc, (= tag\_plain, x_k))$;
let $mess = sdec(x_k, x_{enc})$ in out$(os, mess)$;
(*** Ending the execution ***)
new $n$; out$(tpmc, (pcr\_extend\_req, n, pf\_state, zero))$;
in$(tpmc, (= pcr\_extend\_resp, = n, exts))$;
let $ends = set\_lock(exts, exp\_pp, false)$ in out$(os, ends)$
```

the term

$$\begin{aligned} seal(h((h((\; & p_d, \\ & (h(prog(exp\_init)), h(prog(exp\_stm))))), \\ & h(prog(exp\_pp)))), \\ k_{pp})) & \end{aligned}$$

which we assume publicly available. Recall that p_d represents a reset constant which can only be the initial value of the PCR when the CPU requests a new dynamic root of trust. This will be ensured by the TPM process below.

We also assume that some private message $hello_{pp}$ has been encrypted with k_{pp}: $senc(k_{pp}, hello_{pp})$ is publicly available. Now, in the context of a dynamic root of trust with the expected parameters, the example program $prog(exp\_pp)$ will be able to unseal the key k_{pp} and get access to $hello_{pp}$. We assume that it will send this message on a public channel. This way, we can verify both the secrecy of k_{pp} and, by showing that $hello_{pp}$ is not secret, the correct execution of $prog(exp\_pp)$.

The example $\mathsf{DRT_{PP}}$ program is modeled by the process $\mathsf{EXP_{PP}}$ in figure 6. After re-enabling interrupts, the program receives a sealed blob and the encrypted private message from the operating system, unseals the blob relying on the TPM to obtain k_{pp}. It then decrypts the ciphertext to obtain the value $hello_{pp}$, which it outputs back to the operating system. Before the execution of $\mathsf{EXP_{PP}}$ ends, not only the DRT lock is set to false, but also the PCR is extended with a dummy value in order to leave the PCR in a state which is not to be trusted anymore.

4) The TPM role: First we define generic functions for reseting and extending the PCR value. The requests come on a channel ch, which will be instantiated to different values depending on whether the request comes from the CPU, from some running DRT programs, or from the operating system. The value to which the PCR is reset, rv, will also depend on the context: only the CPU can reset the PCR to the specific value p_d marking a dynamic root of trust. Reset requests from

Fig. 7. Reset and extend functions of the TPM

$$
\begin{aligned}
\mathsf{TPM}_{\mathsf{RESET}} \quad &= \\
&\mathsf{let}\ (ch, rv) = (cpu\_tpm, p_d)\ \mathsf{in}\ !\mathsf{PCR}_{\mathsf{RESET}}\ | \\
&\mathsf{let}\ (ch, rv) = (os, p_s)\ \mathsf{in}\ !\mathsf{PCR}_{\mathsf{RESET}} \\
\mathsf{TPM}_{\mathsf{EXTEND}} \quad &= \\
&\mathsf{let}\ ch = cpu\_tpm\ \mathsf{in}\ !\mathsf{PCR}_{\mathsf{EXTEND}}\ | \\
&\mathsf{let}\ ch = os\ \mathsf{in}\ !\mathsf{PCR}_{\mathsf{EXTEND}}\ | \\
&!(\mathsf{in}(cpu\_tpm, (= drt\_channel, tpmc)); \\
&\ \ \mathsf{let}\ ch = tpmc\ \mathsf{in}\ !\mathsf{PCR}_{\mathsf{EXTEND}})
\end{aligned}
$$

Fig. 8. The $\mathsf{TPM}_{\mathsf{UNSEAL}}$ process

(*** Private unseal on $tpmc$ for the running $\mathsf{DRT}_{\mathsf{PP}}$ ***)
$!\mathsf{in}(cpu\_tpm, (= drt\_start, pf\_state, tpmc));$
$!(\mathsf{in}(tpmc, (= tag\_unseal, blob));$
$\mathsf{let}\ value = unseal(blob, get\_pcr(pf\_state))\ \mathsf{in}$
$\mathsf{out}(tpmc, (tag\_plain, value))$
$)\ |$
(*** Public unseal ***)
$!(\mathsf{in}(os, (tag\_unseal, pf\_state, blob));$
$\mathsf{if}\ get\_lock(pf\_state) = false\ \mathsf{then}\ ($
$\mathsf{let}\ value = unseal(blob, get\_pcr(pf\_state))\ \mathsf{in}$
$\mathsf{out}(os, (tag\_plain, value)))).$

other parties will result in a PCR value of p_s.

$$
\begin{aligned}
\mathsf{PCR}_{\mathsf{RESET}} \quad &= \\
&\mathsf{in}(ch, (= pcr\_reset\_req, nonce, st)); \\
&\mathsf{let}\ new\_st = reset\_pcr(st, tpm\_acc, rv)\ \mathsf{in} \\
&\mathsf{out}(ch, (pcr\_reset\_resp, nonce, new\_st)) \\
\mathsf{PCR}_{\mathsf{EXTEND}} \quad &= \\
&\mathsf{in}(ch, (= pcr\_extend\_req, nonce, st, v)); \\
&\mathsf{let}\ new\_st = extend\_pcr(st, tpm\_acc, v)\ \mathsf{in} \\
&\mathsf{out}(ch, (pcr\_extend\_resp, nonce, new\_st))
\end{aligned}
$$

Then, we have

$$\mathsf{TPM} \ = \ !\mathsf{TPM}_{\mathsf{RESET}}\ |\ !\mathsf{TPM}_{\mathsf{EXTEND}}\ |\ !\mathsf{TPM}_{\mathsf{UNSEAL}}$$

where $\mathsf{TPM}_{\mathsf{RESET}}$, $\mathsf{TPM}_{\mathsf{EXTEND}}$ are defined in figure 7 and $\mathsf{TPM}_{\mathsf{UNSEAL}}$ in figure 8.

Note that the unseal process has two parts. The first part represents the unseal functionality that is available for a running $\mathsf{DRT}_{\mathsf{PP}}$ program. For this, the TPM first receives from the CPU on the cpu_tpm channel the corresponding platform state and the channel $tpmc$ for communicating with the protected program. Subsequently, it can handle any number of unseal requests on the channel $tpmc$, as long as the PCR values of sealed blobs match those recorded in the platform state. On the other hand, the second part of the unseal process allows the attacker to unseal any blob in any state, as long as the PCR values match and, crucially, if there is no dynamic root of trust running in that state.

Fig. 9. The DRT process

$\mathsf{new}\ k_{pp}; \mathsf{new}\ hello_{pp}; \mathsf{out}(os, senc(k_{pp}, hello_{pp}));$
$\mathsf{out}(os,$

$$
\begin{aligned}
\ \ seal(h((h((\quad &p_d, \\
&(h(prog(exp\_init)), h(prog(exp\_stm))))), \\
&h(prog(exp\_pp)))), \\
&k_{pp}))
\end{aligned}
$$

$\mathsf{out}(os, prog(exp\_stm));$
(*** Initial state ***)
$\mathsf{in}(os, x_{stm}); \mathsf{in}(os, x_{smi});$
$\mathsf{out}(os, state(tpm(p_s), cpu(true, null),$
$\qquad drt(null, null, false), smram(x_{stm}, x_{smi}));$
(*** Run a DRT with any loaded programs ***)
$\mathsf{in}(os, drt\_init); \mathsf{in}(os, drt\_pp); \mathsf{in}(os, pf\_state);$
$\mathsf{out}(os, (drt\_req, drt\_init, drt\_pp, pf\_state));$
$(\ !\mathsf{DRT}_{\mathsf{CPU}}\ |\ !\mathsf{EXP}_{\mathsf{INIT}}\ |\ !\mathsf{EXP}_{\mathsf{PP}}\ |\ \mathsf{TPM}\ )$

5) Execution of the platform: To model the execution of the platform, we put all these processes together to obtain the DRT process in figure 9. We create a private key and a private message for the protected program and publish the sealed key and the encrypted message. We output the identity of the honest STM program to the attacker, but we assume the attacker does not obtain its entry point. The attacker can load into the initial state of the platform any STM and any SMI handler, for instance at system reboot. We show explicitly how the attacker can request a dynamic root of trust with any $\mathsf{DRT}_{\mathsf{INIT}}$ program and any $\mathsf{DRT}_{\mathsf{PP}}$ program.

E. Security properties in the formal model

1) Reachability: The reachability of a state in the platform can be expressed as a (non-)secrecy property: a state is reachable when a corresponding term $state(t_{tpm}, t_{cpu}, t_{drt}, t_{smram})$ can be obtained by the attacker after interacting with the process DRT. We can express this as a formula of the form $\mathsf{DRT} \models state(t_{tpm}, t_{cpu}, t_{drt}, t_{smram})$.

We verify the adequacy of our model by checking that an expected DRT state can been reached (running the DRT programs $\mathsf{EXP}_{\mathsf{INIT}}$,$\mathsf{EXP}_{\mathsf{PP}}$ and the expected STM):

$$
\begin{aligned}
&\mathsf{DRT} \models \\
&\ state(\quad tpm(h((h((\quad p_d, \\
&\qquad\qquad (h(prog(exp\_init)), h(prog(exp\_stm))))), \\
&\qquad\qquad h(prog(exp\_pp))))), cpu(true, x), \\
&\qquad drt(prog(exp\_init), prog(exp\_pp), true), \\
&\qquad smram(prog(exp\_stm), prog(y)))
\end{aligned}
$$

and that the program $\mathsf{EXP}_{\mathsf{PP}}$ has succeeded to unseal the key k_{pp} and thus can output the private message $hello_{pp}$ on the public channel os: $\mathsf{DRT} \models hello_{pp}$.

2) 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 $h(prog(exp\_init)), h(prog(exp\_stm))$ and $h(prog(exp\_pp))$, then only the corresponding $\mathsf{DRT_{PP}}$, $\mathsf{DRT_{INIT}}$ and STM are running on the platform, and they can not be modified. This can be expressed by the following correspondence assertion:

$$
\begin{aligned}
&\mathsf{DRT} \models \\
&\quad state(\quad tpm(h((h((\quad p_d, \\
&\qquad\qquad (h(prog(exp\_init)), h(prog(exp\_stm))))), \\
&\qquad\qquad h(prog(exp\_pp))))), cpu(x, y), \\
&\qquad\quad drt(x_{init}, x_{pp}, true), smram(x_{stm}, x_{smi})) \\
&\quad \rightsquigarrow \ (x_{init}, x_{pp}, x_{stm}) = \\
&\qquad (prog(exp\_init), prog(exp\_pp), prog(exp\_stm))
\end{aligned}
$$

Note that we ensure the property only for honest programs. Indeed, if any of $\mathsf{DRT_{PP}}$,$\mathsf{DRT_{INIT}}$ or STM is dishonest, 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 the first dishonest program in the chain of trust. In particular, our property shows that dishonest DRT programs can not make the PCR values record the measurement of honest programs.

3) Secrecy of sealed data: We also verify that secret data sealed for $\mathsf{EXP_{PP}}$, i.e. the key k_{pp}, stays secret:

$$\mathsf{DRT} \not\models k_{pp}$$

VI. Abstraction of PCR extensions

ProVerif does not terminate for the equational theory that we propose in section V. The main reason is the equation that allows an unbounded number of PCR extensions, reflecting the same problem as the one first noticed in [12]. In this section, we propose a general way to replace this equation with a set of equations that can simulate the same behaviour, while simplifying the abstraction task for ProVerif.

For simplicity of our soundness proofs and without much loss of generality, we assume that all the values to be extended into the PCR are available to the attacker. This is the case in all applications that we are aware of. In our case study, these values are either public program identities, or other values chosen by the attacker.

A. Intuitions and definitions

Notation. A term of the form $h((\dots h((t_0, t_1))), \dots, t_n))$ will be denoted by $pcr(t_0, \dots, t_n)$, and $pcr(t_0)$ is t_0. We define $length(pcr(t_0, \dots, t_n)) = n$.

Proposed abstraction. For a given natural number n_b, we would like the following to hold in the equational theory

$$
\begin{aligned}
extend\_pcr(\quad & state(tpm(t_1), t_2, t_3, t_4), tpm\_acc, v) = \\
& state(tpm(h((t_1, v))), t_2, t_3, t_4) \\
& \qquad\qquad \text{if and only if } length(t_1) < n_b \\
extend\_pcr(\quad & state(tpm(t_1), t_2, t_3, t_4), tpm\_acc, v) = \\
& state(tpm(v), t_2, t_3, t_4) \\
& \text{if and only if } length(t_1) \geq n_b \text{ and } length(v) > n_b
\end{aligned}
$$

Intuitively, the first equality means that the PCR can be extended normally only a constant number of times, given by the bound n_b. Note however that the second equality allows the PCR to be set to any value whose length is higher than the bound n_b. This means that, if any party wants to extend the PCR value say $n_b + n$ times, it can simply use the second equation by supplying the desired final value of the PCR. Since we assume that all the values extended into the PCR are public, anyone can obtain the desired final extended value and set it as the current PCR value (via the TPM).

The abstraction is tight. The chain of PCR extensions that leads to a desired protected configuration is obviously finite, and its length will give us the value of the bound n_b. For instance, in our case study from section V, the desired PCR value has length 2:

$$
\begin{aligned}
pcr(p_0, \quad & (h(prog(exp\_init)), h(prog(exp\_stm))), \\
& h(prog(exp\_pp)))
\end{aligned}
$$

If we would allow the PCR to be set to any value whose length is smaller or equal to n_b, we would have a false attack in the formal model, and the abstraction would not be useful. Indeed, the attacker would for instance be able to unseal any data that is sealed for a protected configuration of length n_b. However, it is not helpful for the attacker to set the PCR to any value whose length is bigger than n_b, since there is no way to decrease the PCR and thus violate a security property.

The abstraction is sound. Formally, we will show in section VI-B that any state (and thus any attack) that is reachable in the initial model, is also reachable in the abstracted model, by simulating the normal PCR extension using the newly introduced equations.

The abstraction is simpler to verify. In presence of the extend equation in the original model, 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 the equation. On the other hand, in presence of the equational theory that we propose as replacement, we will let ProVerif deduce that the actual value of $h((y, v))$ does not matter after a certain point.

Encoding the abstraction in the equational theory. Given a bound $n_b \geq 1$, we simply replace the $extend\_pcr$ equation from section V:

$$
\begin{aligned}
extend\_pcr(\quad & state(tpm(y), x_1, x_2, x_3), tpm\_acc, v) \\
= \quad & state(tpm(h((y, v))), x_1, x_2, x_3)
\end{aligned}
$$

with the set of equations $\mathcal{E}_{bound} \cup \mathcal{E}_{any}$. The set $\mathcal{E}_{bound}$ is formed of equations:

$$
\begin{aligned}
extend\_pcr(\quad & state(tpm(pcr(v_0, \dots, v_i)), x_1, x_2, x_3), \\
& \quad tpm\_acc, v) \\
= \quad & state(tpm(pcr(v_0, \dots, v_i, v)), x_1, x_2, x_3)
\end{aligned}
$$

for all $0 \leq i < n_b$, where $v_0 \in \{p_s, p_d\}$ and $v_1, \dots, v_i, v$ are variables. The reason v_0 is a constant is to ensure that the length of the extended PCR value is exactly i. The set $\mathcal{E}_{any}$ is formed of the single equation

$$
\begin{aligned}
extend\_pcr(\quad & state(tpm(pcr(y_0, \dots, y_{n_b})), x_1, x_2, x_3), \\
& \quad tpm\_acc, (v_0, \dots, v_{n_b+1})) \\
= \quad & state(tpm(pcr(v_0, \dots, v_{n_b+1})), x_1, x_2, x_3)
\end{aligned}
$$

where y_i, v_i and x_i are variables, for all i. Therefore, if an agent intends to reach a PCR value of $pcr(t_0, \ldots, t_n)$, with $n > n_b$, it will use (through the TPM) the equation in $\mathcal{E}_{any}$ with values $v_0 = pcr(t_0, \ldots, t_{n-n_b-1})$, $v_1 = t_{n-n_b}, \ldots, v_{n_b+1} = t_n$.

B. Soundness proofs

Our abstraction should be sound in a general setting, but to keep the paper concise we only prove it for the particular model presented in section V. The equational theory of section V is denoted by $\mathcal{E}_1$ and the equational theory of this section is denoted by $\mathcal{E}_2$. For a set of terms T, we denote by $pcrv(T)$ the set of values that have been extended into the PCRs of the set T, formally defined in appendix A.

First we show that our abstraction is sound at the term deducibility level. As mentioned above, we provide to the abstracted theory the set of values that have been extended into the PCR:

Lemma 1: For any $n_b \geq 0$, set of terms T and term t, we have

$$T \vdash_{\mathcal{E}_1} t \implies T, pcrv(T) \vdash_{\mathcal{E}_2} t$$

Proof sketch (details in appendix A): We do the proof by induction on the number of derivation steps in $T \vdash_{\mathcal{E}_1} t$. The non-trivial case is when the function symbol f that is applied in the last derivation step is $extend\_pcr$, with arguments $state(tpm(u_1), u_2, u_3, u_4)$, tpm_acc and a term t_3, to derive $state(tpm(h((u_1, t_3))), u_2, u_3, u_4)$. Then, we consider two subcases: when $length(u_1) < n_b$, we apply an equation from $\mathcal{E}_{bound} \subseteq \mathcal{E}_2$; when $length(u_1) \geq n_b$, we apply the equation from $\mathcal{E}_{any} \subseteq \mathcal{E}_2$, by relying also on $pcrv(T)$ to obtain any desired PCR value. □

Now we extend the result at the process level. The DRT process ensures that all values extended into the PCR are public, and we can rely on lemma 1 for the proof.

Proposition 1: For all $n_b \geq 2$ and term t, we have

$$\mathsf{DRT} \models_{\mathcal{E}_1} t \implies \mathsf{DRT} \models_{\mathcal{E}_2} t$$

Proof sketch (details in appendix A): We prove that for any trace $\mathsf{DRT} \xrightarrow{w}_{\mathcal{E}_1} P$, we have

1) $\mathsf{DRT} \xrightarrow{w}_{\mathcal{E}_2} P$
2) for all term u, $fr(P) \vdash_{\mathcal{E}_1} u \implies fr(P) \vdash_{\mathcal{E}_2} u$
3) for all term v in $pcrv(fr(P))$, $fr(P) \vdash_{\mathcal{E}_2} v$

From definitions, the points 1 and 2 will allow us to conclude the proposition. We also note that the point 2 follows from lemma 1 combined with point 3, so it is sufficient to show points 1 and 3. We do the proof by induction on the length of the trace $\mathsf{DRT} \xrightarrow{w}_{\mathcal{E}_1} P$. The point 3 follows easily from the fact that any value extended into the PCR can be traced to a request coming from the attacker, relying also on the point 2 of the induction hypothesis to switch from $\mathcal{E}_1$ to $\mathcal{E}_2$.

To prove the point 1, we consider all the possible cases for the last atomic action in the trace $\mathsf{DRT} \xrightarrow{w}_{\mathcal{E}_1} P$. When the last action is an output on a public channel or an internal communication on a private channel, it can be trivially matched in the theory $\mathcal{E}_2$. When it is an input on a public channel, we rely on the point 2 of the induction hypothesis to deduce that the same term can be provided by the attacker in presence of $\mathcal{E}_2$. The most difficult case is when the last action is an equality test performed internally in a process Q, and where moreover the function $extend\_pcr$ has been used. In that case, if the PCR value is higher than n_b, we show that $\mathcal{E}_{any} \subseteq \mathcal{E}_2$ can be used, by relying on the point 3 of the induction hypothesis. The assumption $n_b \geq 2$ helps us to deduce that it is the attacker who is trying to extend the PCR above the bound n_b, which is helpful to construct a different trace leading to the same process. □

VII. Verification results

The ProVerif code for the DRT process and the security properties defined in section V is available in appendix B and online. It uses the equational theory $\mathcal{E}_2$ obtained from $\mathcal{E}_1$, from section V, by following the abstraction from section VI with $n_b = 2$. The verification of every property terminates in less than 10 minutes, returning the expected result. From these results (implying there is no attack modulo $\mathcal{E}_2$) and from proposition 1 (implying there is no attack modulo $\mathcal{E}_1$), we derive:

Theorem 1: The DRT process satisfies, modulo $\mathcal{E}_1$, the property of code integrity, defined in section V-E2, and the property of sealed data secrecy, defined in section V-E3. □

We also show that our models indeed encode the desired functionality, by verifying the reachability properties defined in section V-E1. Note that we can not rely on proposition 1 to transfer reachability results from $\mathcal{E}_2$ to $\mathcal{E}_1$. We would need completeness in order to do that (which should be true if properly stated). However, it is sufficient to check the trace returned by ProVerif to convince ourselves that the states are indeed reachable in the original model. In fact, since they correspond to the standard execution of the protocol, these traces can even be derived by hand without any difficulty. Similarly, we are able to discover the attacks of [14], [27] in our model, when we allow the STM to be modified arbitrarily.

VIII. Future work

While our model takes into account at an abstract level the attacks and mitigations of [14], [27], further refinements and soundness results are necessary in order to be able to conclude that attacks such as these or as [29], [28] 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 instance implementation flaws.

Another direction of research is to extend the methods proposed in this paper to the protection of programs without a hardware root of trust. The challenge there is that the platform state is even more open to intrusions from a software attacker.

We think the abstraction of PCR registers that we have presented in section VI is an instance of a more general result, whose exploration would be fruitful for future applications, helping to resolve the tension between an exploding search space and its automated analysis.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, January 2001.

[2] M. Arapinis, S. Bursuc, and M. D. Ryan. Reduction of equational theories for verification of trace equivalence: Re-encryption, associativity and commutativity. In Degano and Guttman [10], pages 169–188.

[3] M. Arapinis, E. Ritter, and M. D. Ryan. StatVerif: Verification of Stateful Processes. In *CSF*, pages 33–47. IEEE Computer Society, 2011.

[4] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[5] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Computer Security Foundations Workshop (CSFW'01)*, 2001.

[6] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[7] B. Blanchet and A. Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *IEEE Symposium on Security and Privacy*, pages 417–431. IEEE Computer Society, 2008.

[8] V. Cortier, J. Degrieck, and S. Delaune. Analysing routing protocols: Four nodes topologies are sufficient. In Degano and Guttman [10], pages 30–50.

[9] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 221–236. IEEE, 2009.

[10] P. Degano and J. D. Guttman, editors. *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, volume 7215 of *Lecture Notes in Computer Science*. Springer, 2012.

[11] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel. A formal analysis of authentication in the TPM. In P. Degano, S. Etalle, and J. Guttman, editors, *Revised Selected Papers of the 7th International Workshop on Formal Aspects in Security and Trust (FAST'10)*, volume 6561 of *LNCS*, pages 111–125, Pisa, Italy, September 2010. Springer.

[12] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel. Formal analysis of protocols based on TPM state registers. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*, pages 66–82, Cernay-la-Ville, France, June 2011. IEEE Computer Society Press.

[13] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

[14] L. Duflot, O. Grumelard, O. Levillain, and B. Morin. ACPI and SMI handlers: some limits to trusted computing. *Journal in computer virology*, 6(4):353–374, 2010.

[15] C. Fournet and J. Planul. Compiling information-flow security to minimal trusted computing bases. In G. Barthe, editor, *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 216–235. Springer, 2011.

[16] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan. Parametric verification of address space separation. In Degano and Guttman [10], pages 51–68.

[17] J. Franklin, S. Chaki, A. Datta, and A. Seshadri. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In *IEEE Symposium on Security and Privacy*, pages 365–379. IEEE Computer Society, 2010.

[18] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 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.trustedcomputinggroup.org.

[21] R. Küsters and T. Truderung. Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 157–171. IEEE Computer Society, 2009.

[22] S. Mödersheim. Abstraction by set-membership: verifying security protocols and web services with databases. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 351–360. ACM, 2010.

[23] M. Paiola and B. Blanchet. Verification of security protocols with lists: From length one to unbounded length. In Degano and Guttman [10], pages 69–88.

[24] M. D. Ryan and B. Smyth. Applied pi calculus. In V. Cortier and S. Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, Cryptology and Information Security Series. IOS Press, 2011.

[25] D. Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT*, volume 668 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1993.

[26] B. Thomsen. A calculus of higher order communicating systems. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 143–154. ACM, 1989.

[27] R. Wojtczuk and J. Rutkowska. Attacking INTEL trusted execution technology. Black Hat DC, 2009.

[28] R. Wojtczuk and J. Rutkowska. Attacking INTEL TXT via SINIT code execution hijacking. *Invisible Things Lab*, 2009.

[29] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another way to circumvent INTEL trusted execution technology. *Invisible Things Lab*, 2009.

Appendix A
Soundness proofs

For simplicity of proofs, we orient each equation $l = r$ in $\mathcal{E}_1$ and $\mathcal{E}_2$ from left to right, obtaining a rewrite rule $l \to r$. It is easy to see that we obtain this way two convergent (i.e. terminating and confluent) rewrite systems, $\mathcal{R}_1$ and respectively $\mathcal{R}_2$, that represent $\mathcal{E}_1$ and respectively $\mathcal{E}_2$. We also let $\mathcal{R}_{bound}$ and $\mathcal{R}_{any}$ be the rewrite systems that correspond to $\mathcal{E}_{bound}$ and $\mathcal{E}_{any}$ respectively. The application of a rewrite rule from a rewrite system $\mathcal{R}$ to a term u in order to obtain a term v is denoted by $u \to_{\mathcal{R}} v$. If zero, one or more rewrite steps are applied to obtain v from u, we denote this by $u \to^*_{\mathcal{R}} v$. A term is in normal form if no rewrite rule can be applied to it. We refer to [4], [13] for more details on rewriting.

Now, for a set of terms in normal form T and a term in normal form t, we have $T \vdash_{\mathcal{E}_i} t$ if and only if:

- *Base case:* either $t \in T$
- *Derivation step:* or there is a public function symbol f and a set of terms $t_1, \ldots, t_n$ such that $T \vdash_{\mathcal{E}_i} t_1, \ldots, T \vdash_{\mathcal{E}_i} t_n$ and $f(t_1, \ldots, t_n) \to^*_{\mathcal{R}_i} t$

for all $i \in \{1, 2\}$

For a set of terms T, we denote by $pcrv(T)$ the set of values that have been extended into the PCRs of the set T. Formally, we have

- $pcrv(\{t_1, \ldots, t_n\}) = pcrv(t_1) \cup \ldots \cup pcrv(t_n)$
- $pcrv(f(t_1, \ldots, t_n)) = pcrv(t_1) \cup \ldots \cup pcrv(t_n)$, if $f \neq tpm$
- $pcrv(tpm(t)) = pcrv_1(t)$
- $pcrv_1(h((t_1, t_2))) = \{t_2\} \cup pcrv_1(t_1) \cup pcrv(t_2)$
- $pcrv_1(t) = \{t\} \cup pcrv(t)$, if $top(t) \neq h$

Lemma 1: For any $n_b \geq 0$, set of terms T and term t, we have

$$T \vdash_{\mathcal{E}_1} t \implies T, pcrv(T) \vdash_{\mathcal{E}_2} t$$

Proof: Without loss of generality, we assume that the terms in T and the term t are in normal form. We do the proof by induction on the number of derivation steps used to deduce $T \vdash_{\mathcal{E}_1} t$. If $t \in T$, then we can immediately conclude from definitions that $T, pcrv(T) \vdash_{\mathcal{E}_2} t$.

Assume now that $T \vdash_{\mathcal{E}_1} t_1, \ldots, T \vdash_{\mathcal{E}_1} t_n$ and $f(t_1, \ldots, t_n) \to^*_{\mathcal{R}_1} t$, for some terms in normal form

$t_1, \ldots, t_n$. By induction hypothesis, we have $T, pcrv(T) \vdash_{\mathcal{E}_2} t_1, \ldots, T, pcrv(T) \vdash_{\mathcal{E}_2} t_n$. Note also that the shape of rewrite rules in $\mathcal{R}_1$ ensures that there is at most one rewrite step in the rewriting sequence $f(t_1, \ldots, t_n) \rightarrow^*_{\mathcal{R}_1} t$. We consider several cases:

Case when then number of rewrite steps in $f(t_1, \ldots, t_n) \rightarrow^*_{\mathcal{R}_1} t$ is zero: if f is any other symbol than $extend\_pcr$, we can conclude immediately, because if no rule in $\mathcal{R}_1$ can be applied, then no rule in $\mathcal{R}_2$ can be applied. In the case of the $extend\_pcr$ symbol, we note that the left-hand side of the extend rule in $\mathcal{R}_1$ is more general than the left-hand side of any rule in $\mathcal{R}_{bound}$ or $\mathcal{R}_{any}$. Therefore, if $t = extend\_pcr(t_1, t_2, t_3)$ can not be reduced using $\mathcal{R}_1$, it can not be reduced using $\mathcal{R}_2$ either, and we can conclude $T \vdash_{\mathcal{E}_2} t$.

Case when $f = extend\_pcr$: then $n = 3$ and we have $t_1 = state(tpm(u_1), u_2, u_3, u_4)$, $t_2 = tpm\_acc$ and $t = state(tpm(h((u_1, t_3)), u_2, u_3, u_4)$. We distinguish two subcases depending on the PCR length of u_1:

Subcase when $length(u_1) < n_b$: this means the PCR value has not reached the extension bound yet and we can continue to extend it normally, using the rules in $\mathcal{E}_{bound}$. Note that, because the PCR value in a state can be set or modified only through the $reset\_pcr$ and $extend\_pcr$ functions, we must have $u_1 = pcr(s_0, \ldots, s_i)$ for some $s_0 \in \{p_s, p_d\}$, some terms $s_1, \ldots, s_i$ and $i < n_b$. Therefore, using a rewrite rule in $\mathcal{R}_{bound}$, we have $extend\_pcr(t_1, t_2, t_3) \rightarrow_{\mathcal{R}_2} state(tpm(h((u_1, t_3))), u_2, u_3, u_4) = t$. Thus, we can conclude $T, pcr(T) \vdash_{\mathcal{E}_2} t$.

Subcase when $length(u_1) \geq n_b$: this means the PCR value has reached the extension bound and we can use the extend rule in $\mathcal{R}_{any}$ to set the PCR to any value. To this end, we have to first recover the desired components of the final PCR value, either from the set $pcr(T)$ or from previously deduced terms. Assume $u_1 = pcr(s_0, \ldots, s_m)$, for some $s_0 \in \{p_s, p_d\}$ and some terms $s_1, \ldots, s_m$, with $m \geq n_b$. Since the PCR of any state can only be modified via $reset\_pcr$ or $extend\_pcr$,

- either there exists an index l, $0 \leq l \leq m$, such that $pcr(s_0, \ldots, s_l) \in pcrv(T)$ and $s_{l+1}, \ldots, s_m$ have been consequently used as the paramater of a PCR extension during the derivation of $T \vdash_{\mathcal{E}_1} t_1$.
- or there has been a PCR reset to s_0 and $s_1, \ldots, s_m$ have been subsequently used as as the paramater of a PCR extension during the derivation of $T \vdash_{\mathcal{E}_1} t_1$.

For the values s_i that have been used as parameters for the PCR extension during the derivation of $T \vdash_{\mathcal{E}_1} t_1$, we have $T \vdash_{\mathcal{E}_1} s_i$, using a smaller number of derivation steps. Therefore, we can apply the induction hypothesis to deduce that $T, pcrv(T) \vdash_{\mathcal{E}_2} s_i$. Recall also that, by induction hypothesis, we have $T, pcrv(T) \vdash_{\mathcal{E}_2} t_3$. In both cases outlined above, we can therefore deduce that $T, pcrv(T) \vdash_{\mathcal{E}_2} pcr(s_0, \ldots, s_m, t_3)$.

Now, we can apply the $extend\_pcr$ equation from $\mathcal{E}_{any}$ to deduce:

$$\begin{aligned} & extend\_pcr(t_1, t_2, pcr(s_0, \ldots, s_m, t_3)) \\ & \rightarrow_{\mathcal{R}_2} state(tpm(pcr(s_0, \ldots, s_m, t_3)), u_2, u_3, u_4) \\ & = state(tpm(h((u_1, t_3))), u_2, u_3, u_4)) = t \end{aligned}$$

and we can conclude $T, pcrv(T) \vdash_{\mathcal{E}_2} t$.

Any other case: let $l \rightarrow r \in \mathcal{R}_1$ be the rule that is applied in the rewrite step $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{R}_1} t$. Since $t_1, \ldots, t_n$ are in normal form, we deduce that the rewrite rule $l \rightarrow r$ is applied at the top of the term, and therefore $top(l) = f$. Since $f \neq extend\_pcr$, we deduce that $l \rightarrow r \in \mathcal{R}_2$ and thus $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{R}_2} t$ and we can conclude. □

Proposition 1: For all $n_b \geq 2$ and term t, we have

$$\mathsf{DRT} \models_{\mathcal{E}_1} t \implies \mathsf{DRT} \models_{\mathcal{E}_2} t$$

Proof sketch: Without loss of generality, we assume that constructed terms are always put in normal form before any other action that involves them. For a process or term A we denote by $A\{t_1 \mapsto t_2\}$ the process or term obtained by replacing all occurences of t_1 in A by t_2. For two sets of terms T_1, T_2, we write $T_1 \vdash T_2$ if for all $t \in T_2$, we have $T_1 \vdash t$.

We prove that for any trace $\mathsf{DRT} \xrightarrow{w}_{\mathcal{E}_1} P$ we have:

1) $\mathsf{DRT} \xrightarrow{w}_{\mathcal{E}_2} P$
2) for all term u, $fr(P) \vdash_{\mathcal{E}_1} u \implies fr(P) \vdash_{\mathcal{E}_2} u$
3) for all term v in $pcrv(fr(P))$, $fr(P) \vdash_{\mathcal{E}_2} v$

From definitions, the points 1 and 2 will allow us to conclude the proposition. Note that the point 2 follows from lemma 1 and the point 3, so it is sufficient to show points 1 and 3. We proceed by induction on the length of the trace $\mathsf{DRT} \xrightarrow{w}_{\mathcal{E}_1} P$.

Base case: In this case, we have $P = \mathsf{DRT}$ and the point 1 is immediate. Moreover, $fr(P)$ is empty and the point 3 also trivially holds.

Induction step: Assume now $\mathsf{DRT} \xrightarrow{w'}_{\mathcal{E}_1} P' \xrightarrow{\alpha}_{\mathcal{E}_1} P$, for some process P' and some atomic transition α. By induction hypothesis, we have

1) $\mathsf{DRT} \xrightarrow{w}_{\mathcal{E}_2} P'$
2) for all term u', $fr(P') \vdash_{\mathcal{E}_1} u' \implies fr(P') \vdash_{\mathcal{E}_2} u'$
3) for all term v' in $pcrv(fr(P))$, $fr(P) \vdash_{\mathcal{E}_2} v'$

We consider the possible cases for the action α: it can be an output, an input or an internal action τ of P' (communication on a private channel, an assignement via let, or an if test).

Case $\alpha = \mathsf{out}(w, s)$: we obviously have $P' \xrightarrow{\alpha}_{\mathcal{E}_2} P$, thus the point 1. Note that $fr(P) = fr(P') \cup \{s\}$ and thus $pcrv(fr(P)) = pcrv(fr(P')) \cup pcrv(s)$. By induction hypothesis, we have $fr(P') \vdash_{\mathcal{E}_2} pcrv(fr(P'))$, thus to show the point 3 we only have to show that $fr(P) \vdash_{\mathcal{E}_2} pcrv(s)$. Now, the only process who can extend PCR values is the TPM: all the values v in $pcrv(s)$ are either in $\{p_s, p_d\}$ or have been *previously* received by the TPM on a public or on a private channel. In the former case, we can conclude from the point 2 of the induction hypothesis. In the latter case, from the design of honest programs, the value v can only be equal to $(h(p_1), h(p_2))$ or $h(p_3)$, for some $\mathsf{DRT_{INIT}}$ program identity p_1, STM program identity p_2 and $\mathsf{DRT_{PP}}$ program identity

p_3. Then, p_1,p_2,p_3 are available to the attacker, and therefore so is the value v.

Case $\alpha = \mathsf{in}(w, x)$**:** then $P = P'\{x \mapsto s\}$, for some term s that is deducible by the attacker from the frame of P', i.e. $fr(P') \vdash_{\mathcal{E}_1} s$, and is sent on some channel w. Then, from the point 2 of the induction hypothesis, we have $fr(P') \vdash_{\mathcal{E}_2} s$ and thus the term s can be sent on channel w in presence of $\mathcal{E}_2$ as well. Therefore, we deduce $P' \xrightarrow{\alpha}_{\mathcal{E}_2} P$ and we conclude the point 1. The point 3 follows immediately from induction hypothesis, because $fr(P) = fr(P')$.

Case $\alpha = \tau$**:** we have $fr(P') = fr(P)$, so the point 3 is immediate. When τ represents communication on a private channel, the point 1 also trivially holds. The cases of a let assignement and of an if test are similar, so let us consider only the former. Then, we have $P' = C[\mathsf{let}\ t_1 = t_2\ \mathsf{in}\ Q]$ and $P = C[Q\sigma]$, for some process context $C[\_]$, terms t_1, t_2, process Q and substitution σ such that $t_1\sigma =_{\mathcal{E}_1} t_2\sigma$. If the function $extend\_pcr$ does not occur in t_1 and in t_2 or if it only occurs with a term different from tpm_acc as a second argument, we can easily deduce that $t_1\sigma =_{\mathcal{E}_2} t_2\sigma$, and we can conclude.

Otherwise, it must be the case that $\mathsf{let}\ t_1 = t_2\ \mathsf{in}\ Q$ is a subprocess of the $\mathsf{PCR}_{\mathsf{EXTEND}}$ process, with $t_1 = new\_st$ and t_2 in place of $extend\_pcr(st, tpm\_acc, v)$:

$$\mathsf{let}\ new\_st = extend\_pcr(st, tpm\_acc, v)\ \mathsf{in}\ Q$$

Then t_1 is a variable and

$$t_2 \quad = \quad extend\_pcr(state(tpm(s_0, \ldots, s_n), s', s'', s'''), \\ tpm\_acc, t_v)$$

where t_v is the term to be extended into the PCR. If n is smaller than the PCR bound n_b, we can easily deduce $t_1\sigma =_{\mathcal{E}_2} t_2\sigma$, by relying on the corresponding equation from $\mathcal{E}_{bound}$. Otherwise, let us take one step back in the process $\mathsf{PCR}_{\mathsf{EXTEND}}$ to the input action that supplied $state(tpm(s_0, \ldots, s_n), s', s'', s''')$ and t_v, that is

$$\mathsf{in}(ch, (= pcr\_extend\_req, nonce, st, v));$$

(Recall that this is a syntactic sugar collating an input and a let, but we can treat it as an atomic action in our proof). Note that ch is either one of cpu_tpm, $tpm\_ch(prog(exp\_pp)), tpm\_ch(prog(exp\_init))$ or a public channel. In fact, by definition of $\mathsf{CPU}_{\mathsf{DRT}}, \mathsf{EXP}_{\mathsf{INIT}}$ and $\mathsf{EXP}_{\mathsf{PP}}$, they have control of a measured platform state and would never attempt to extend the PCR above the bound n_b. Hence, ch must be a public channel.

Let P_0 be the process that makes the input transition on ch and P_1 be the resulting process. We have $P \xrightarrow{w_1} P_0 \xrightarrow{\mathsf{in}(ch,.)} P_1 \xrightarrow{w_2} P'$. From the point 3 of the induction hypothesis, we have $fr(P_0) \vdash_{\mathcal{E}_2} s_0, \ldots, fr(P_0) \vdash_{\mathcal{E}_2} s_n$. Furthemore, we have $fr(P_0) \vdash_{\mathcal{E}_1} t_v$, because t_v was supplied on a public channel as a value to be extended into the PCR. From the point 2 of the induction hypothesis, we deduce $fr(P_0) \vdash_{\mathcal{E}_2} t_v$. Therefore, we have $fr(P_0) \vdash_{\mathcal{E}_2} t'_v$, where $t'_v = pcr(s_0, \ldots, s_n, t_v)$. Thus, we have $P_0 \xrightarrow{\mathsf{in}(ch,.)}_{\mathcal{E}_2} P_1\{t_v \mapsto t'_v\}$. Furthermore, since the value v is local to the process $\mathsf{PCR}_{\mathsf{EXTEND}}$, the replacement of t_v with t'_v does not affect any other transitions: we have $P_1\{t_v \mapsto t'_v\} \xrightarrow{w_2} P'\{t_v \mapsto t'_v\}$. Now, we can rely on the equation $\mathcal{E}_{any}$ to deduce that

$$t_2\{t_v \mapsto t'_v\}\sigma = \\ extend\_pcr(state(tpm(s_0, \ldots, s_n), s', s'', s'''), tpm\_acc, t'_v)\sigma \\ =_{\mathcal{E}_2} state(pcr(s_0, \ldots, s_n, t_v), s', s'', s''')\sigma = t_2\sigma$$

Therefore, we have $P'\{t_v \mapsto t'_v\} \xrightarrow{\tau}_{\mathcal{E}_2} P$ and we can conclude $\mathsf{DRT} \xrightarrow{w}_{\mathcal{E}_2} P$. ☐

APPENDIX B
PROVERIF CODE FOR THE CASE STUDY

The ProVerif code for our case studies is presented on the following pages and is also available online at https://www.dropbox.com/s/7d8vpp3m8umklba/drt-csf2013.pi

Comments are starting with (* and ending with *).

```
(***
ABBREVIATIONS:
DRT - DYNAMIC ROOT OF TRUST
DRT_INIT - THE SINIT (INTEL) OR SLB (AMD) PROGRAM
DRT_PP - THE PROTECTED PROGRAM: MLE(INTEL) OR SK(AMD)
***)
param reconstructTrace = false.
(*CHANNELS*)
free os. (* PUBLIC CHANNEL FOR THE OPERATING SYSTEM CONTROLLED BY THE INTRUDER *)
private free cpu_tpm.  (* PRIVATE CHANNEL FOR THE COMMUNICATION BETWEEN CPU AND TPM *)
private fun tpm_ch/1. (* tpm_ch(x) - PRIVATE CHANNEL USED BY A PROGRAM x TO COMMUNICATE WITH TPM *)
(*CRYPTO*)
fun h/1. (* HASH FUNCTION *)
fun senc/2. (* SYMMETRIC KEY ENCRYPTION *)
reduc sdec(x,senc(x,y)) = y.
fun ps/0.  (* STATIC RESET VALUE OF THE PCR *)
fun pd/0.  (* DYNAMIC RESET VALUE OF THE PCR *)
fun false/0. fun true/0. (* BOOLEAN VALUES *)
(* TPM SEAL/UNSEAL*)
fun seal/2.
private reduc unseal(seal(xpcr,xvalue), xpcr) = xvalue.

(* STATE STRUCTURE: state(tpm(PCR),cpu(INT,CACHE), drt(INIT,PP,LOCK),smram(STM,SMI) *)
private fun state/4. private fun tpm/1. private fun cpu/2. private fun drt/3.  private fun smram/2.

(* PRIVATE CONSTANTS FOR THE PRIVILEGED ACCESS THAT THE CPU AND TPM HAVE TO THE PLATFORM STATE *)
private fun cpuAccess/0. private fun tpmAccess/0.

(* ABSTRACTION FOR DYNAMICALLY LOADING PROGRAMS*)
fun program/1. private reduc getENTRY(program(x)) = x.

(*** ACCESSING THE PLATFORM STATE ***)
reduc getPCR (state(tpm(y),x1,x2,x3)) = y.        reduc getINT(state(x1,cpu(y1,y2),x2,x3)) = y1.
reduc getCACHE(state(x1,cpu(y1,y2),x2,x3)) = y2.  reduc getINIT(state(x1,x2,drt(y1,y2,y3),x3)) = y1.
reduc getPP(state(x1,x2,drt(y1,y2,y3),x3)) = y2.  reduc getLOCK(state(x1,x2,drt(y1,y2,y3),x3)) = y3.
reduc getSTM (state(x1,x2,x3,smram(y1,y2))) = y1. reduc getSMIH (state(x1,x2,x3,smram(y1,y2))) = y2.

(*** MODIFYING THE PLATFORM STATE + ABILITIES OF LOADED PROGRAMS ***)
(* TPM *)
reduc resetPCR (state(tpm(y),x1,x2,x3),tpmAccess,pd) =state(tpm(pd),x1,x2,x3);
      resetPCR (state(tpm(y),x1,x2,x3),tpmAccess,ps) =state(tpm(ps),x1,x2,x3).
reduc extendPCR(state(tpm(pd),x1,x2,x3), tpmAccess, value) =state(tpm(h((pd,value))),x1,x2,x3);
      extendPCR(state(tpm(ps),x1,x2,x3), tpmAccess, value) =state(tpm(h((ps,value))),x1,x2,x3);
      extendPCR(state(tpm(h((pd,y1))),x1,x2,x3), tpmAccess, value) =
      state(tpm(h((h((pd,y1)),value))),x1,x2,x3);
      extendPCR(state(tpm(h((ps,y1))),x1,x2,x3), tpmAccess, value) =
      state(tpm(h((h((ps,y1)),value))),x1,x2,x3);
      extendPCR(state(tpm(h((h((y0,y1)),y2))),x1,x2,x3), tpmAccess, (v0,v1,v2,v3)) =
      state(tpm(h((h((h((v0,v1)),v2)),v3))),x1,x2,x3).

(* CPU *)
reduc setINT(state(x1,cpu(y1,y2),x2,x3),cpuAccess,value) = state(x1,cpu(value,y2),x2,x3);
      setINT(state(x1,cpu(y1,y2),drt(z1,program(z2),true),x2),z2,value) =
      state(x1,cpu(value,y2),drt(z1,program(z2),true),x2).
reduc cache(state(x1,cpu(y1,y2),x2,x3),value) = state(x1,cpu(y1,value),x2,x3).
reduc flush_smi(state(x1,cpu(y1,y2),x2,smram(z1,z2))) = state(x1,cpu(y1,y2),x2,smram(z1,y2)).
reduc flush_stm(state(x1,cpu(y1,y2),drt(w1,w2,false),smram(z1,z2))) =
      state(x1,cpu(y1,y2),drt(w1,w2,false),smram(y2,z2)).
(* TO OBTAIN THE ATTACK, ADD THE EQUATION: *)
(* flush_stm(state(x1,cpu(y1,y2),x2,smram(z1,z2))) = state(x1,cpu(y1,y2),x2,smram(y2,z2)). *)

(* DRT *)
reduc setINIT(state(x1,x2,drt(y1,y2,y3),x3),cpuAccess,value)  =state(x1,x2,drt(value,y2,y3),x3).
reduc setPP(state(x1,x2,drt(y1,y2,y3),x3),cpuAccess,value)    =state(x1,x2,drt(y1,value,y3),x3);
      setPP(state(x1,x2,drt(program(y1),y2,y3),x3),y1,value)=state(x1,x2,drt(program(y1),value,y3),x3);
      setPP(state(x1,cpu(true,z),drt(y1,y2,y3),smram(program(z1),program(z2))),(z1,z2),value)
      =state(x1,cpu(true,z),drt(y1,value,y3),smram(program(z1),program(z2))).
reduc setLOCK(state(x1,x2,drt(y1,y2,y3),x3),cpuAccess,value) =state(x1,x2,drt(y1,y2,value),x3);
      setLOCK(state(x1,x2,drt(y1,program(y2),true),x3),y2,value)
                 =state(x1,x2,drt(y1,program(y2),value),x3);
      setLOCK(state(x, cpu(true,z),drt(y1,y2,y3),smram(program(z1),program(z2))),(z1,z2),value)
                 =state(x,cpu(true,z),drt(y1,y2,value),smram(program(z1),program(z2))).
```

```
(** MESSAGE TAGS **)
free drt_request,drt_response,pcr_extend_request,pcr_extend_response,
     pcr_reset_request,pcr_reset_response, drt_start, tag_unseal, tag_plain, drt_channel.
(* FUNCTION FOR CREATING NONCES *)
private fun fnonce/1.
(** DRT PROCESSES **)
let DRT_CPU = (* GET A DRT REQUEST FROM THE OPERATING SYSTEM *)
     in(os, (=drt_request, drt_init, drt_pp, pf_state));
  (* ONLY ACCEPT THE REQUEST IF NOT ALREADY RUNNING A DYNAMIC ROOT OF TRUST *)
  if getLOCK(pf_state) = false then (
   (* DISABLE INTERRUPTS *)
   let s0'=setINT(pf_state,cpuAccess, false) in
   (* UPDATE THE LOCK *)
   let s0 = setLOCK(s0', cpuAccess, true) in
   (* RESET THE PCR *)
   (* DESIRED CODE: *)
   (* new nonce; *)
   (* CLASSIC ABSTRACTION THAT RUNS FASTER: NONCES ARE A FUNCTION OF THEIR CONTEXT *)
   let nonce = fnonce((drt_init,drt_pp, getSTM(pf_state))) in
   out(cpu_tpm, (pcr_reset_request, nonce, s0));
   in(cpu_tpm, (=pcr_reset_response,=nonce,s1));
   (* EXTEND THE PCR WITH THE MEASUREMENT *)
   let measurement = (h(drt_init),h(getSTM(pf_state))) in
   out(cpu_tpm, (pcr_extend_request, nonce, s1, measurement));
   in(cpu_tpm, (=pcr_extend_response, =nonce,s2));
   (* LOAD DRT_INIT AND ESTABLISH TPM CHANNELS *)
   let s3 = setINIT(s2,cpuAccess, drt_init) in
   out(cpu_tpm, (drt_channel, tpm_ch(drt_init)));
   let entry_init = getENTRY(drt_init) in
   out(entry_init, (drt_request, nonce, s3, tpm_ch(drt_init), drt_pp));
   (* THE drt_init PROGRAM HAS MEASURED AND SET UP THE drt_pp PROGRAM*)
    in(entry_init, (=drt_response, =nonce, new_state));
   (* SETUP TPM CHANNELS FOR THE LOADED DRT_PP *)
    let entry_pp = getENTRY(getPP(new_state)) in
    out(entry_pp, (drt_start, new_state, tpm_ch(program(entry_pp))));
    out(cpu_tpm, (drt_channel, tpm_ch(program(entry_pp))));
    out(cpu_tpm, (drt_start, new_state, tpm_ch(program(entry_pp))))
   ).

(* THE TWO EQUATIONS BELOW TOGETHER WITH THE CACHE PROCESS ARE A CONSEQUENCE OF EQUATIONS
FOR cache, flush_smi AND flush_stm. WRITING THEM EXPLICITLY HELPS PROVERIF
TERMINATE 5 MINUTES FASTER *)
reduc setSTM (state(x1,x2,x3,smram(y1,y2)),cpuAccess,value)=state(x1,x2,x3,smram(value,y2)).
reduc setSMIH(state(x1,x2,x3,smram(y1,y2)),cpuAccess,value)=state(x1,x2,x3,smram(y1,value)).
let CACHE =
  ( in(os, (pf_state,xsmi));
  let new_state = setSMIH(pf_state,cpuAccess, xsmi) in
  out(os, new_state) ) |
  ( in(os,(pf_state,xstm));
  if getLOCK(pf_state) = false then
  let new_state = setSTM(pf_state,cpuAccess,xstm) in
  out(os, new_state) ).
```

```
private fun expected_init/0.
let EXPECTED_INIT = out(os, program(expected_init));
    (* RECEIVE DRT_PP AND TPM ACCESS FROM THE CPU *)
    in(expected_init, (=drt_request, nonce0, pf_state, tpmc, drt_pp));
    (* MEASURE AND EXTEND DRT_PP INTO THE PCR *)
    let measurement = h(drt_pp) in
    (* DESIRED CODE: *)
    (* new nonce; *)
    (* CLASSIC ABSTRACTION THAT RUNS FASTER: NONCES ARE A FUNCTION OF THEIR CONTEXT *)
    let nonce = fnonce(drt_pp) in
    out(tpmc, (pcr_extend_request,nonce, pf_state, measurement));
    in(tpmc, (=pcr_extend_response,=nonce, ext_state));
    (* LOAD DRT_PP ON THE PLATFORM STATE *)
    let new_state = setPP(ext_state,expected_init, drt_pp)  in
    (* PASS THE CONTROL BACK TO THE CPU *)
    out(expected_init, (drt_response, nonce0, new_state));
    (* MAKE THE NEW PLATFORM STATE PUBLIC *)
    out(os, new_state).

private fun expected_pp/0.
let EXPECTED_PP = (* DECRYPT A SEALED BLOB, RELYING ON COMMUNICATION WITH TPM*)
  out(os, program(expected_pp));
  in(expected_pp, (=drt_start,pf_state0,tpmc));
  (* RE-ENABLE INTERRUPTS *)
  let pf_state = setINT(pf_state0,expected_pp,true) in
  out(os,pf_state);
  (* UNSEAL THE KEY AND DECRYPT THE PRIVATE MESSAGE *)
  in(os,xSealedBlob);  in(os,xEncBlob);
  out(tpmc,(tag_unseal,xSealedBlob));
  in(tpmc,(=tag_plain,xSymKey));
  let xMessage = sdec(xSymKey,xEncBlob) in
  out(os,xMessage);
  (* ENDING THE EXECUTION: THE LOCK IS SET TO FALSE AND THE PCR VALUE IS DESTROYED *)
  (*new nonce; *)
  (* ABSTRACTION THAT RUNS FASTER *)
  let nonce = fnonce(drt_pp) in
  out(tpmc, (pcr_extend_request, nonce, pf_state, zero));
  in(tpmc, (=pcr_extend_response, =nonce, ext_state));
  let end_state = setLOCK(ext_state,expected_pp,false) in
  out(os,end_state).

let TPM = !TPM_RESET | !TPM_EXTEND | !TPM_UNSEAL.
let TPM_RESET = let (channel, reset_type) = (cpu_tpm,pd) in !PCR_RESET |
                let (channel, reset_type) = (os,ps) in !PCR_RESET.
let PCR_RESET = in(channel, (=pcr_reset_request, nonce, pf_state));
                let new_state = resetPCR(pf_state,tpmAccess,reset_type) in
                out(channel, (pcr_reset_response, nonce, new_state)).
let TPM_EXTEND = let channel = cpu_tpm in !PCR_EXTEND |
                 let channel = os in !PCR_EXTEND |
                 !in(cpu_tpm, (=drt_channel, drtc)); let channel = drtc in !PCR_EXTEND.
let PCR_EXTEND=  in(channel, (=pcr_extend_request,nonce,pf_state,value));
                 let new_state = extendPCR(pf_state,tpmAccess,value) in
                 out(channel, (pcr_extend_response,nonce,new_state)).
let TPM_UNSEAL =
               (* PRIVATE UNSEAL ON LOC2 FOR THE RUNNING DRT_PP *)
               !in(cpu_tpm, (=drt_start, pf_state, tpmc));
               !(in(tpmc,(=tag_unseal, blob));
                      let value = unseal(blob,getPCR(pf_state)) in
                      out(tpmc,(tag_plain,value))
                ) |
            (* PUBLIC UNSEAL *)
             !(in(os, (tag_unseal, pf_state, blob));
                 if getLOCK(pf_state) = false then
                 let value = unseal(blob, getPCR(pf_state)) in out(os,(tag_plain,value))).

(* QUERIES *)
(*A. THE EXPECTED STATE HAS BEEN REACHED *)
query attacker:state(tpm(h((h((pd,(h(program(expected_init)),h(program(expected_stm))))),
                     h(program(expected_pp))))), cpu(true,x),
                     drt(program(expected_init),program(expected_pp),true),
                     smram(program(expected_stm),program(y))).
(* QUERY RESULT: FALSE => THE ATTACKER HAS THE TERM *)

(*B. THE PROTECTED PROGRAM SUCCESFULLY DECRYPTS THE PRIVATE MESSAGE
USING THE SEALED KEY AND MAKES IT PUBLIC*)
query attacker:hello_pp.
(* QUERY RESULT: FALSE => THE ATTACKER HAS THE TERM *)

(*C. WHENEVER THE EXPECTED PCR IS SET, THE PLATFORM HAS THE EXPECTED STATE *)
query attacker:state( tpm(h((h((pd,(h(program(expected_init)),h(program(expected_stm))))),
                      h(program(expected_pp))))), cpu(x,y),
                      drt(xi,xp,true), smram(xstm,xsmih))
==> (xi,xp,xstm)=(program(expected_init),program(expected_pp),program(expected_stm)).
(*QUERY RESULT: TRUE => THE ASSERTION IS VALID *)

(*D. THE ATTACKER DOES NOT HAVE ACCESS TO THE SEALED KEY *)
query attacker:k_pp.
(* QUERY RESULT: TRUE => THE ATTACKER DOES NOT HAVE k_pp *)

(* THE MAIN PROCESS *)
free null.
private fun expected_stm/0.
private free k_pp. (* SECRET KEY WHICH SHOULD ONLY BE KNOWN BY THE PROTECTED PROGRAM *)
private free hello_pp. (* PRIVATE MESSAGE ENCRYPTED WITH k_pp *)
process (* ENCRYPTED PRIVATE MESSAGE FOR PP *)
out(os,senc(k_pp,hello_pp));
(*ASSUME THAT THE BLOB SEALING THE SECRET KEY IS PUBLIC*)
out(os,seal(h((h((pd,(h(program(expected_init)),h(program(expected_stm))))),
                                           h(program(expected_pp)))),k_pp));

out(os, program(expected_stm));
(* INITIAL STATE LOADED UPON A SYSTEM RESET *)
in(os,(xInitStm,xInitSmih));
out(os, state(tpm(ps),cpu(true,null),drt(null,null,false),smram(xInitStm,xInitSmih)));
(* REQUESTING A DYNAMIC ROOT OF TRUST WITH ANY LOADED PROGRAMS *)
in(os, drt_init); in(os,drt_pp); in(os,pf_state);
out(os, (drt_request, drt_init, drt_pp, pf_state));
(*EXECUTING THE DRT PROCESSES *)
( !DRT_CPU | !CACHE | !EXPECTED_INIT | !EXPECTED_PP | TPM)
```