Dynamic measurement and protected execution: model and analysis

Shiwei Xu[1], Ian Batten[2], and Mark Ryan[2]

[1] Wuhan Digital Engineering Institute, Wuhan, China
[2] School of Computer Science, University of Birmingham, UK

Abstract. Useful security properties arise from sealing data to specific units of code. Modern processors featuring Intel's TXT and AMD's SVM achieve this by a process of measured and protected execution. Only code which has the correct measurement can access the data, and this code runs in an environment protected from observation and interference. We present a modelling language with primitives for protected execution, along with its semantics. We characterise an attacker who has access to all the capabilities of the hardware. In order to achieve automatic analysis of systems using protected execution without attempting to search an infinite state space, we define transformations that reduce the number of times the attacker needs to use protected execution to a pre-determined bound. Given reasonable assumptions we prove the soundness of the transformation: no secrecy attacks are lost by applying it. We then describe using the StatVerif extensions to ProVerif to model the bounded invocations of protected execution. We show the analysis of realistic systems, for which we provide case studies.

1 Introduction

Modern hardware often includes security features that support the ability to *seal data to program code*. This allows a data owner to impose a policy (embodied as code) about how their data is to be processed: the hardware guarantees that the policy will be enforced. Sealing data to code is a very powerful mechanism that enables a wide variety of applications, in mobile computing and cloud computing alike. For example, mobiles can store cryptographic keys for exclusive use by certain applications, such as payment or banking applications. In the cloud, servers can guarantee to remote users that the data they uploaded is being processed only in accordance with their wishes.

This paper aims to analyse the specific mechanisms for sealing data to code provided on commodity processors from AMD and Intel. These processors already ship with off-the-shelf computers and are becoming ubiquitous. They allow code to be executed in a protected environment outside the influence of malware or untrusted software (including the operating system) that may be present on the host computer. Specifically, we analyse the architecture and mechanisms provided by Flicker [11]. The idea of Flicker is to define very small programs that handle security-sensitive data, and are run in protected execution mode directly

under the control of the hardware security features. The bulk of the software which is security-*in*sensitive runs on top of an operating system, as usual. For example [11], a *certificate authority* could be structured as a small secure base that handles the signing key, along with a bigger untrusted part that deals with I/O, the user interface, etc.

The hardware mechanisms which support this architecture are Intel's *trusted execution technology* (TXT), or the roughly-equivalent *secure virtual machine* (SVM) technology from AMD. Both of these technologies rely on the presence of a *trusted platform module* to store and use cryptographic keys. Our analysis therefore involves cryptographic protocols; and since the TPM uses persistent state registers called *platform configuration registers* (PCRs), it also involves statefulness.

For these reasons, we wish to use StatVerif [2] for our analysis; it is an extension of ProVerif which can deal with protocols that have persistent state. Unfortunately, StatVerif does not reliably terminate when the state space is infinite, such as in the case studies we are using in this paper. We therefore have to develop appropriate abstractions. There are two sources of infinite state in our application. The first one arises from an operation called PCR *extension*. We already found a suitable abstraction in earlier work [6], and we re-use it here. The second source of infinite state arises because an attacker can unboundedly often reset the hardware and invoke a new session that uses the hardware primitives. This paper is devoted to developing an abstraction to overcome this obstacle to the analysis; the details of our analysis are available as a technical report.

1.1 Contribution

A platform with a TPM and a processor that implements TXT or SVM supports *protected execution* and the ability to seal data to program code. We provide a formal model of this protected execution so that we can prove the security properties it offers. Specifically:

- We define TXML, a language with primitives for protected execution.
- We formalise a model for an adversary who is attacking protected execution.
- We state and prove a theorem that allows us to restrict attention to finite attacker strategies, paving the way to automated verification.
- We briefly describe the use of StatVerif [2] (which is an extension of ProVerif) to model some applications that use protected execution from the literature [11].
- We demonstrate that the applications indeed satisfy the security goals, in the context of our attacker model.

1.2 Attacker Model

Our attacker is a powerful attacker, inspired by the capabilities against a network protocol of a Dolev-Yao attacker. The attacker can:

– Perform arbitrary offline computation using their own resources in order to plan their strategy.
– Execute arbitrary code on the system they are attacking, including using supervisor, kernel or other privileged modes. This includes running either their own or the defender's code in a protected execution environment.
– Access the TPM as though they were the owner of the TPM, with full knowledge of the authorisation data.

The restrictions we impose exclude low-level hardware attacks on the platform, and assume that the hardware correctly implements the hardware design. We also assume that the cryptographic primitives are sound. Our attacker therefore cannot:

– Extract keys or other information from the TPM other than by using the published API. This excludes hardware attacks on the TPM, and assumes that the TPM correctly implements the API.
– Bypass the published protections of TXT/SVM by, for example, attacking the memory controller on the platform.
– Successfully guess cryptographic keys, perform offline decryption given only the cipher text, find collisions in hash functions.

1.3 Related work

There have been previous formal analyses of TPM and dynamic measurement. Lin [10] uses the theorem prover Otter and the model finder Alloy to analyse the security of the TPM when presented with invalid sequences of API calls. Lin considered modelling PCR state, but was unable to do this with Otter. Gurgens *et al* [9] describe an analysis of the TPM API using a finite state automata, but the model fragment given does not appear to consider PCR state and the analysis in the paper is predominantly informal. Coker *et al* [3] focus on the analysis of TPM APIs for remote attestation, but their SAL model is not yet publicly available. Delaune *et al* [5] analyse a fragment of the TPM, using the applied pi calculus as a modelling language and using the ProVerif tool to automate their verification, but they too do not consider PCR state.

Much of the previous formal work on dynamic measurement is abstract. Millen [12] uses LTL to model the roles and trust relationships in a dynamic measurement system; Datta [4] proposes a logic for reasoning about secure systems built on dynamic measurement; Fournet and Planul [7] reason in the cryptographic model about the security of sealed data. Our methods complement those of [4,7], because they are automatic and scalable. We demonstrate this with our case studies.

Arapinis *et al* [2] extend the process language of ProVerif to allow the modelling of global state. Their work allows the description of the TPM and dynamic measurement in a process language with state, and the automatic generation of Horn clauses from this model. However, ProVerif will not terminate on these clauses. To assist termination, Delaune *et al* [6] show a first-order model based

on Horn clauses. This model focuses on PCR state and related API commands. They place an upper bound on the number of times a PCR needs to be extended between two resets. They show that if there is an attack using an unlimited number of extensions, then there is also an attack which requires a bounded number of PCR extensions. They also show that this upper bound is small enough to be tractable using ProVerif. Their model solves the non-termination problem caused by PCR extension; however, in some applications further termination problems are caused by multiple PCR resets as the result of multiple invocations of dynamic measurement. Bounding the number of PCR extensions does not solve the problem caused by multiple PCR resets, which we characterise and solve in this paper.

2 Background to Trusted Computing

Hardware primitives A TPM can provide evidence that a system is running in a particular configuration through the use of *platform configuration registers* (PCRs). The TPM only allows the PCRs to be reset to initial values by privileged instructions or at system reset. They can however be *extended* at any time. Extension involves hashing the current value of the PCR with another value; a PCR containing value p is extended with x by concatenating p with x, hashing the result and making this the new value of the PCR.

When a software module is loaded, its *measurement*, conventionally consisting of a secure hash of the code, can be extended into a PCR. A sequence of such extensions will result in a PCR value that is unique to the set of modules so measured, and the order in which they were measured, starting from an initial measure of trusted code loaded at boot-time.

Secret information can be *sealed* against a set of PCR values. The TPM encrypts the information using a key that the TPM controls, and the TPM will perform the matching decryption if and only if the PCRs are in the state against which the data has been sealed.

Only the loading of a unique sequence of modules will produce a particular PCR. Any change to any one of the modules will produce a different value. In practice, this limits the measurement of the whole configuration; the process of booting a general purpose operating system from initial power-on to user login has too many variable elements, and changes too frequently. Additionally, because code is measured at the point of loading, an attacker who can access memory (which is usually a reasonable assumption) can overwrite already-measured code without altering the PCR value.

To address these issues, version 1.2 of the TPM specification introduced the concept of a dynamic root of trust. Rather than tracing execution back to power-on, a privileged instruction can be used to reset PCRs in a new bank (17–22), which were initialised at power-on to all-ones (u_1), directly to all-zeroes (u_0). These *resetable* PCRs are then available to be extended with measurements from the reset onwards, rather than from power-on. This functionality is the

basis for dynamic measurement technology, such as Intel's *Trusted Execution Technology* (TXT) [8] and AMD's *Secure Virtual Machine* (SVM) [1].

As the Intel and AMD technologies operate in a similar fashion, for simplicity we only consider the AMD SVM technology; the differences do not affect our argument in any material way. SVM provides a privileged command SKINIT, which creates a protected environment in which code can execute free from external influences. The unit of code to be executed with this protection is called a Secure Loader Block (SLB). When the SKINIT instruction is executed, interrupts and DMA are disabled to prevent access to the SLB and the resetable PCRs are reset. The SLB is sent to the TPM for measurement (using a hash function) and the result is extended into PCR17. The SLB is then executed.

Flicker architecture Flicker [11] provides a mechanism for executing pieces of code with specific guarantees of privacy and integrity by making use of dynamic measurement. It uses Intel TXT or AMD SVM technology to measure and protect an SLB, which consists of initialisation and clean-up (SLB Core) and application functionality (Pieces of Application Logic, PAL); the measurement allows the SLB to unseal private data, while the protection prevents interference or observation with manipulation of the private data. Appropriate kernel services are made available to enable execution.

Although their use is not mandatory, the standard SLB template provides some additional features. Firstly, it provides a standard mechanism for passing arguments in to the PAL and passing out results. Secondly, after execution of the PAL has completed, but before the SLB exits, the SLB measures the inputs and the outputs of the PAL and extends their value into PCR17, and then extends a fixed public constant into PCR17. One effect of this is to leave the PCRs in a state which is of no use to an attacker who is attempting to unseal confidential data; the extension with the fixed public constant leaves a value against which no data will have been sealed. It also provides a verifiable chain to prove the execution: PCR17 is left as the result of successively extending an initial u_0 with the measurements of the SLB, any inputs, any outputs and finally the fixed public constant. A later TPM_Quote operation on PCR17 provides an attestation as a verifiable link between the inputs, the outputs and the SLB for a verifier.

3 A Model of Protected Execution

In this section, we model the functionality of dynamic measurement and the TPM. We introduce a modelling language, TXML, along with its semantics. We describe a theorem that allows us to bound the length of traces such that models can be verified with available tools. To illustrate our model, we introduce a simple Flicker-based decryption oracle. We then formally define our model.

3.1 Simplifications and abstractions

We simplify both Flicker and the TPM, but in ways which grant additional powers to the attacker. If the attacker cannot compromise our simplified system, he also cannot compromise the full system.

- We omit TPM authdata, normally required to authenticate access to the TPM. Permitting access to the TPM without authdata is equivalent to assuming that the attacker can obtain the authdata from the running machine; this is in keeping with our attacker model.
- We assume that all keys used in the model have been created and are permanently loaded in the TPM. This does not alter the power of the attacker but simplifies the model substantially.
- For simplicity, we consider a TPM with only PCR17, as it is sufficient to model the facilities we are using.
- Finally, as we are analysing the secrecy properties of the system rather than the correctness of attestations, we omit the SLB's extension of its input and output into PCR17. This allows the attacker to replay previous executions as though they were fresh.

3.2 An introductory example

Our introductory example is a decryption oracle implemented as an SLB. Any object supplied to the SLB is decrypted using symKey and returned to the user. The intent is that symKey is never revealed outside the oracle. In order to prevent the oracle being used to decrypt arbitrary ciphertexts, a check is made for the presence of a pre-arranged tag. Our concern is the privacy of the decryption key in the face of a powerful attacker, so the tag is made available to the attacker.

We represent the oracle SLB as slbD which receives as input the sealed symKey and an object which is to be treated as ciphertext. slbD attempts to unseal symKey. If that succeeds it uses symKey to decrypt the cipher-text. If that succeeds, and the tag is found in the plaintext, the plaintext is output. Finally, the public fixed constant fpc is extended into PCR17 to revoke access to the secrets.

Assuming the correctness of the TPM unsealing function, symKey can only be unsealed when PCR17 value is $\mathsf{h}(\mathsf{u}_0, \mathsf{slbD})$. Due to the operation of dynamic measurement, PCR17 can be set to $\mathsf{h}(\mathsf{u}_0, \mathsf{slbD})$ only by the execution of slbD with an SKINIT instruction. slbD itself uses, but does not output, symKey. Consequently, symKey is not exposed outside the decryption oracle.

3.3 Trusted Execution Modelling Language

We present a formal model of protected execution. This models a machine equipped with a simplified TPM which offers sealing, unsealing, resetting, reading and extension of the PCR. The machine also allows users to execute programs in the protected execution environment provided by dynamic measurement.

We introduce the syntax and semantics of a language, TXML[3]. This language describes actions that can be taken either by the attacker, as part of a strategy to attack the security properties of the defender's system, or by the defender, in order to implement protected execution. We model the decryption oracle example to show its use in describing a defender's SLB. We then define transformations from one strategy, which is used to model a list of commands in TXML, to another strategy. We show that the result of this transformation is both equivalent to the original and tractable for analysis.

TXML is not a complete language. It lacks any looping constructs, and has only rudimentary conditionals. The attacker would not design an attack strategy using TXML. However, any sequence of actions the attacker carries out can be retrospectively expressed in TXML. Any sequence of operations that involves loops or conditionals can be expressed, once the number of iterations in each loop and the outcome of each conditional has been determined, by a simpler sequence of operations that does not involve loops and conditionals, starting from the same initial state. The attacker is given the ability to perform arbitrary offline computation; this allows him to construct one or more sequences of TXML commands to perform their attack.

Syntax Suppose sets N of names including 0, 1, $\mathsf{sk}_{\mathsf{Srk}}$ (the storage root key of the TPM), tpmPf (the TPM Proof inserted into a seal so that the TPM can confirm that it was sealed on this TPM), ...; V of variables with typical elements x, y, z, ... The letters u, v, w, ... range over $\mathrm{V} \cup \mathrm{N}$. Typical constructor function symbols, including at least $\mathsf{h}/2$, $\mathsf{senc}/2$, $\mathsf{aenc}/3$, $\mathsf{pk}/1$ and $\mathsf{measure}/1$, are represented as f. These represent respectively the SHA1 hash function, symmetric and asymmetric encryption, the derivation of a public key from a private key, and the measurement with SHA1 of a fragment of program text. Typical destructor function symbols, including at least $\mathsf{sdec}/2$ and $\mathsf{adec}/2$, are represented as g. These represent respectively symmetric and asymmetric decryption. We define terms t_1,t_2,... over $\mathrm{V} \cup \mathrm{N}$ in the usual way. The rewrite rules $t_1 \to t_2$ where t_1,t_2 are over variables include at least $\mathsf{sdec}(x, \mathsf{senc}(x, y)) \to y$ and $\mathsf{adec}(x, \mathsf{aenc}(\mathrm{pk}(x), y, z)) \to z$, linking associated encryption and decryption operations.

The syntax of TXML is shown in Figure 1 and described as follows:

- $x := \mathsf{f}(u_1, \ldots, u_n)$ and $x := \mathsf{g}(u_1, \ldots, u_n)$ are applications of constructors and destructors.
- $x := \mathsf{seal}(u, v)$, $x := \mathsf{unseal}(u)$, $\mathsf{extend}(u)$ and reset are the actions of sealing and unsealing data, extending a PCR and resetting the TPM.
- $\mathsf{check}\ u = v$ confirms the equality of two terms.
- skip is the null action.
- $x := \mathsf{SKINIT}\{\mathrm{list(statement)}; \mathsf{rtn}\ u\}$ is a list of statements that are executed protected by SKINIT, which return u.

[3] Trusted eXecution Modelling Language

statement ::=
$x := \mathsf{f}(u_1, \ldots, u_n)$ |
$x := \mathsf{g}(u_1, \ldots, u_n)$ |
$x := \mathsf{seal}(u, v)$ |
$x := \mathsf{unseal}(u)$ |
$\mathsf{extend}(u)$ | reset |
$\mathsf{check}\ u = v$ | skip

command ::=
statement |
$x := \mathsf{SKINIT}\{\text{list(statement)}; \mathsf{rtn}\, u\}$ |
$x := \mathsf{SUBR}\{\text{list(statement)}; \mathsf{rtn}\, u\}$

program ::=
list(command)

Fig. 1. Syntax of TXML

– $x := \mathsf{SUBR}\{\text{list(statement)}; \mathsf{rtn}\, u\}$ is analogous to $x := \mathsf{SKINIT}\{\ldots; \mathsf{rtn}\, u\}$. However, it does not modify the PCR value. SUBR is not available to the programmer or the attacker, and cannot appear in the definition of an SLB; it is present in TXML as a technical convenience that we use during transformation.

Semantics We will be using TXML as the basis for our proof that we can bound the number of SKINITs used by the attacker. We therefore need to define its semantics. We consider configurations (K, p), where the attacker's knowledge base $\mathsf{K} : \mathrm{V} \to$ ground terms is a partial function and p is a ground term. We assume K is extended to V∪N, as the identity function on N. The initial configuration is $(\mathsf{K}_{init}, 1)$. Transitions between configurations are labelled by programs. We assume side conditions that $\mathsf{K}(x)$ is defined and r is non-deterministically chosen whenever we write $\mathsf{K}(x)$ and r. We also assume an injective function $\mathsf{measure} : \mathrm{TXML}^* \to N$ taking a sequence of TXML commands and returning a name.

$(\mathsf{K}, p) \xrightarrow{C} (\mathsf{K}', p')$ means that when the knowledge base is K and the PCR value is p, and the attacker performs command C, his new knowledge base will be K', and the new PCR value will be p'. The relations $\xrightarrow{C}$, which relates to individual commands, and $\overset{S}{\Rightarrow}$, which relates to sequences of commands, are defined in Figure 2. A PCR value of $\perp$ represents a PCR value against which no blob is sealed.

Figure 2 shows the semantics of each command in TXML. We clarify some of the more complex rules:

– A sealed blob consists of the encryption of (tpmPf, p, t), where tpmPf is a constant known only to the TPM, p is the PCR state which must be current for an unsealing operation to succeed, and t is the data which has been sealed. The encryption is done with a public key, whose private part is available only within the TPM. The rules for unseal add the secret t to the attacker's knowledge base if the required PCR value in the sealed blob matches the current PCR value.
– In the rule for $\mathsf{SKINIT}\{L; \mathsf{rtn}\, u\}$, we first compute the effect of L when run from knowledge base K and a PCR value reflecting the measurement of L.

$(\mathsf{K}, p) \xrightarrow{\mathsf{skip}} (\mathsf{K}, p)$

$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{f}(u_1,\ldots,u_n)} (\mathsf{K}[x \to \mathsf{f}(\mathsf{K}(u_1), \ldots, \mathsf{K}(u_n))], p)$

$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{g}(u_1,\ldots,u_n)} (\mathsf{K}[x \to t], p)$ if $\mathsf{g}(t_1, \ldots, t_n) \to t_{n+1}$ is a reduc and $\mathsf{K}(u_i) = t_i\sigma$ and $t = t_{n+1}\sigma$

$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{seal}(u,v)} (\mathsf{K}[x \to \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, \mathsf{K}(u), \mathsf{K}(v)))], p)$ if $\mathsf{K}(u) \neq \bot$

$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{unseal}(u)} (\mathsf{K}[x \to t], p)$ if $p \neq \bot$ and $\mathsf{K}(u) = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, p, t))$

$(\mathsf{K}, p) \xrightarrow{\mathsf{extend}(u)} (\mathsf{K}, \mathsf{h}(p, \mathsf{K}(u)))$ if $p \neq \bot$

$(\mathsf{K}, \bot) \xrightarrow{\mathsf{extend}(u)} (\mathsf{K}, \bot)$

$(\mathsf{K}, p) \xrightarrow{\mathsf{reset}} (\mathsf{K}, 1)$

$(\mathsf{K}, p) \xrightarrow{\mathsf{check}\ u=v} (\mathsf{K}, p)$ if $\mathsf{K}(u) = \mathsf{K}(v)$

$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{SKINIT}\{\mathrm{L};\mathsf{rtn}\ u\}} (\mathsf{K}[x \to \mathsf{K}'(u)], p')$ if $(\mathsf{K}, \mathsf{h}(0, \mathsf{measure}(\mathrm{L}))) \overset{\mathrm{L}}{\Rightarrow} (\mathsf{K}', p')$

$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{SUBR}\{\mathrm{L};\mathsf{rtn}\ u\}} (\mathsf{K}[x \to \mathsf{K}'(u)], p)$ if $(\mathsf{K}, \mathsf{h}(0, \mathsf{measure}(\mathrm{L}))) \overset{\mathrm{L}}{\Rightarrow} (\mathsf{K}', p')$

Fig. 2. This defines the relation $\xrightarrow{C}$. Let S be a TXML program. The relation $\overset{S}{\Rightarrow}$ is defined as $(\mathsf{K}, p) \overset{\emptyset}{\Rightarrow} (\mathsf{K}, p)$ in the case that S is the null program. In other cases, $(\mathsf{K}, p) \overset{C;S}{\Longrightarrow} (\mathsf{K}', p')$ if $(\mathsf{K}, p) \xrightarrow{C} (\mathsf{K}'', p'')$ and $(\mathsf{K}'', p'') \overset{S}{\Rightarrow} (\mathsf{K}', p')$.

– SUBR is similar to SKINIT, except that the final PCR value is p rather than p'. As mentioned, SUBR is not used in source TXML programs; we use it in our transformation.

Modelling the introductory example Two objects are supplied to the decryption oracle: a sealed blob containing the symmetric key pre-sealed against PCR17 with the value of u_0 extended with the measurement of the decryption oracle's program, and some atomic message encrypted with the symmetric key. Therefore, the initial knowledge base is:

$$\mathsf{K}_{init\_DO} = \{x_{sdata} = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, \mathsf{h}(0, \mathsf{measure}(\mathsf{slbD})), \mathsf{symKey})),\ x_{EncBlob} = \mathsf{senc}(\mathsf{symKey}, \mathsf{message})\}$$

where slbD is the program:

```
result := SKINIT {
  xSymKey := unseal(xSData);
  xMessage := sdec(xSymKey, xEncBlob);
  extend(fpc);
  rtn xMessage;
}.
```

The security property we are checking is the secrecy of the symmetric key symKey. To attempt to obtain the key, the attacker can adopt any strategy, as described in §1.2. The desired security property is therefore that there is no

TXML program S, knowledge base K', PCR values p, p' and variable x such that $(K_{\text{init\_DO}}, p) \stackrel{S}{\Rightarrow} (\mathsf{K}', p')$ and $\mathsf{K}'(x) = \text{symKey}$.

Transformations of strategies The security property above requires reasoning over all possible TXML programs S. We show that it is sufficient to consider only programs involving a bounded number of SKINITs and resets. To achieve this, we transform any strategy S into a new strategy S' that is equivalent to S and has a number of SKINITs and resets which is bounded by a value derivable from the initial knowledge base. This allows us to use automatic tools to search for strategies that achieve a certain goal and model at most this number of SKINITs and resets; if none are found, we may conclude that there are no longer strategies that achieve the goal.

The transformation performs two changes. First, unseal operations that are not necessary are replaced by an equivalent assignment. An unseal operation is not necessary if there is another variable in the knowledge base that has the value of the unsealed item. Second, operations that reset the PCR, namely, reset and SKINIT, are removed if there is no necessary unseal operation between the current point and the next reset or SKINIT. This reflects the fact that PCR values are required to be correct only in order for unseals to work. The transformation uses configurations (K, p) as before, but with the extension that p may take the special value $\perp$ which signifies that any value will do; the PCR value is not needed. The transformation is such that $p = \perp$ if and only if there is no unseal between the current position in the program and the next reset or SKINIT.

The result of this transformation does not weaken the attacker. The attacker can perform computation with the defender SLB using any number of inputs. The attacker can run arbitrary code with the PCR in the state left by the execution of the defender SLB. The attacker can attempt to unseal data sealed by the defender. The attacker already knows the contents of sealed data sealed by the attacker. If the attacker wishes to obtain more results from the defender SLB, these are available to him from the transformed strategy.

The transformations we use are shown in Figure 3. In the definition of $\frac{C'}{C}\!\!\rightarrow$, the truth or falsity of $p = \perp$ enforces a global constraint on the way the transformation works, and makes the transformation deterministic. The two rules for each of SKINIT and reset appear to be non-deterministic, but in fact only one of them may be chosen; which one is chosen depends on whether there is an unseal before the next SKINIT or reset, as expressed by the rule for unseal which requires $p \neq \perp$. The apparent free choice of whether $q = p$ or $q = \perp$ in the rule for unseal is similarly constrained by the remainder of the program S being transformed.

SUBR is introduced into S′ by the second rule for SKINIT, which is invoked if and only if there is no unseal between now and the next SKINIT or reset, as indicated by the $\perp$ value on the right hand side.

The relation $(\mathsf{K}, p) \xrightarrow[C']{C} (\mathsf{K}', p')$ indicates that when in configuration (K, p) the execution of command C yields the new configuration (K', p') and adds the command C' to the transformed strategy.

$(\mathsf{K}, p) \xrightarrow[\mathsf{skip}]{\mathsf{skip}} (\mathsf{K}, p)$

$(\mathsf{K}, p) \xrightarrow[x:=\mathsf{f}(u_1,\ldots,u_n)]{x:=\mathsf{f}(u_1,\ldots,u_n)} (\mathsf{K}[x \to \mathsf{f}(\mathsf{K}(u_1), \ldots, \mathsf{K}(u_n))], p)$

$(\mathsf{K}, p) \xrightarrow[x:=\mathsf{g}(u_1,\ldots,u_n)]{x:=\mathsf{g}(u_1,\ldots,u_n)} (\mathsf{K}[x \to t], p)$ if $\mathsf{g}(t_1, \ldots, t_n) \to t_{n+1}$is a reduc and $\mathsf{K}(u_i) = t_i\sigma$ and $t = t_{n+1}\sigma$

$(\mathsf{K}, p) \xrightarrow[x:=\mathsf{seal}(u,v)]{x:=\mathsf{seal}(u,v)} (\mathsf{K}[x \to \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, \mathsf{K}(u), \mathsf{K}(v)))], p)$

$(\mathsf{K}, p) \xrightarrow[x:=y]{x:=\mathsf{unseal}(u)} (\mathsf{K}[x \to \mathsf{K}(y)], p)$ if $\mathsf{K}(u) = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, p', \mathsf{K}(y)))$

$(\mathsf{K}, p) \xrightarrow[x:=\mathsf{unseal}(u)]{x:=\mathsf{unseal}(u)} (\mathsf{K}[x \to t], q)$ otherwise, if $\mathsf{K}(u) = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, p, t))$ and $p \neq \bot$ and $(q = p$ or $q = \bot)$

$(\mathsf{K}, p) \xrightarrow[\mathsf{extend}(u)]{\mathsf{extend}(u)} (K, \mathsf{h}(p, \mathsf{K}(u)))$ if $p \neq \bot$

$(\mathsf{K}, \bot) \xrightarrow[\mathsf{extend}(u)]{\mathsf{extend}(u)} (K, \bot)$

$(\mathsf{K}, \bot) \xrightarrow[\mathsf{reset}]{\mathsf{reset}} (\mathsf{K}, 1)$

$(\mathsf{K}, \bot) \xrightarrow[\mathsf{skip}]{\mathsf{reset}} (\mathsf{K}, \bot)$

$(\mathsf{K}, p) \xrightarrow[\mathsf{check}\ u=v]{\mathsf{check}\ u=v} (\mathsf{K}, p)$ if $\mathsf{K}(u) = \mathsf{K}(v)$

Suppose $(\mathsf{K}, \mathsf{h}(0, \mathsf{measure}(\mathrm{P}))) \xrightarrow[\mathrm{P}']{\mathrm{P}} (\mathsf{K}', p')$.

If P is a defender SLB:

$(\mathsf{K}, \bot) \xrightarrow[x:=\mathsf{SKINIT}\{\mathrm{P};\mathsf{rtn}\ u\}]{x:=\mathsf{SKINIT}\{\mathrm{P};\mathsf{rtn}\ u\}} (\mathsf{K}[x \to \mathsf{K}'(u)], p')$

$(\mathsf{K}, \bot) \xrightarrow[x:=\mathsf{SUBR}\{\mathrm{P};\mathsf{rtn}\ u\}]{x:=\mathsf{SKINIT}\{\mathrm{P};\mathsf{rtn}\ u\}} (\mathsf{K}[x \to \mathsf{K}'(u)], \bot)$

Suppose P is an attacker SLB:

$(\mathsf{K}, p) \xrightarrow[x:=\mathsf{SUBR}\{\mathrm{P'};\mathsf{rtn}\ u\}]{x:=\mathsf{SKINIT}\{\mathrm{P};\mathsf{rtn}\ u\}} (\mathsf{K}[x \to \mathsf{K}'(u)], \bot)$

Fig. 3. Given a knowledge base $\mathsf{K}_{\mathrm{init}}$, a strategy S is transformed to S' if $(\mathsf{K}_{init}, 1) \overset{S}{\underset{S'}{\Longrightarrow}} (\mathsf{K}, \bot)$ for some K, where the relation $\overset{S}{\underset{S'}{\Longrightarrow}}$ (S and S' are strategies) is defined from $\xrightarrow[C']{C}$ above as follows: $(\mathsf{K}, p) \overset{\emptyset}{\underset{\emptyset}{\Longrightarrow}} (\mathsf{K}, p)$ where $\emptyset$ is the empty strategy; $(\mathsf{K}, p) \overset{C;S}{\underset{C';S'}{\Longrightarrow}} (\mathsf{K}'', p'')$ if there exists (K', p') such that $(\mathsf{K}, p) \xrightarrow[C']{C} (\mathsf{K}', p')$ and $(\mathsf{K}', p') \overset{S}{\underset{S'}{\Longrightarrow}} (\mathsf{K}'', p'')$.

Bounding the number of SKINITs and resets The ability to seal data against arbitrary PCR values is very flexible, and can lead to situations more complex than the simple sealing of one piece of data against one set of PCR values. A piece of data sealed against one set of PCR values can contain another piece of data sealed against another set of values. A piece of data may be sealed against an SLB which will only release the decrypted version once some other conditions are met. To reach a state where these conditions are true might include the decrementing of a counter, or transition through some state machine.

We define data as *boundedly sealed* if there is a finite bound to the number of SKINITs required to extract it.

Definition 1. *1. A piece of sealed data B is* sealed against program P *if*

$$B = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), r, (\mathsf{tpmPf}, p, t))$$

and

$$p = \mathsf{h}(\ldots \mathsf{h}(\mathsf{h}(0, \mathsf{measure}(P)), t_2), \ldots t_n).$$

2. A knowledge base K produces K' *using* P *if*

$$\mathsf{K}' = \mathsf{K} \cup \{B \mid \textit{on inputs from } K, P \textit{ can output } B\}$$

3. We then define

$$\overline{\mathsf{K}} = \bigcup_{K \textit{ produces* } K'} \mathsf{K}'.$$

where produces is the reflexive transitive closure of produces.*
K *has only bounded seals if the number of sealed blobs in* $\overline{\mathsf{K}}$ *is finite.*

Theorem 1 Suppose knowledge base K has only bounded seals. Let m be the number of sealed blobs in $\overline{\mathsf{K}}$, as in the definition 1. Let S be any strategy, and suppose that $(\mathsf{K}, \bot) \overset{S}{\underset{S'}{\Rightarrow}} (\mathsf{K}', \bot)$. We have the following properties:

1. $(\mathsf{K}, \bot) \overset{S}{\Rightarrow} (\mathsf{K}', p)$ implies $\exists S'.(\mathsf{K}, \bot) \overset{S}{\underset{S'}{\Rightarrow}} (\mathsf{K}', \bot)$.
2. S' simulates S; that is, $(\mathsf{K}, \bot) \overset{S}{\underset{S'}{\Rightarrow}} (\mathsf{K}', \bot)$ implies $\exists q.(\mathsf{K}, \bot) \overset{S'}{\Rightarrow} (\mathsf{K}', q)$.
3. S' uses only the data present in S that is, every name in S' is in S.
4. The number of SKINITs plus the number of resets in S' is at most m.

The proof of the theorem is given in Appendix A.

This theorem enables us to undertake practical verification of systems that use protected execution. Without the theorem, we would need to consider attacker strategies of unbounded length, or accept weak results based on a significantly weakened attacker who can only use strategies of a fixed length. Our theorem removes these restrictions.

The theorem allows us to bound the number of SKINIT operations that need to be considered to give the attacker access to the full range of PCR states.

However, a typical SLB will perform some unsealing, and will leave the PCR in some new state, but it performs these operations in order to enable some calculation which returns a result. If we bound the attacker's ability to perform these calculations, then we substantially reduce their capability.

We therefore introduce the SUBR operation. Although it does not modify the PCR state (because we have shown that we can bound the number of such modifications we need to consider) it returns the same result as the corresponding SKINIT. So in the typical case where data is simply sealed so that we need only consider one SKINIT, the attacker can nonetheless make unlimited use of the SUBR to get results from the defender's SLB. Our theorem shows that it is sound to transform a strategy that uses an arbitrary number of SKINIT operations into a strategy which performs some bounded number of SKINIT operations, together with an arbitrary number of uses of the SUBR.

We are now in a position to verify the security properties of our decryption oracle. We have one piece of sealed data in our knowledge base, and by definition 1 it is boundedly sealed: no sealed blobs are added to the knowledge base by the operation of the SLB. $\overline{\mathsf{K}_{init\_DO}}$ therefore contains only one seal, x_{sdata}. We apply theorem 1 to K_{init_DO}, with $m = 1$.

A StatVerif model was constructed, which includes:

- A process modelling a TPM, complete with PCR state and seal, unseal, reset and extend operations;
- A process modelling the use of the SLB via SKINIT, which leave the PCR values in a deterministic state;
- A process modelling the use of the SLB via SUBR, which leave the PCR values in an indeterminate state.

This model is then bounded appropriately and run to test the secrecy property of the symmetric key. The property is confirmed.

4 Case Study: Password Authentication for SSH

Description An additional authentication mechanism for OpenSSH is proposed by McCune *et al* [11]. The goal is to prevent any malicious code on the server from learning the user's password, even if the server is compromised. The prevents an attacker from making use of the password to pose as the legitimate user.

A keypair is shared between the authentication SLB and the client. The private part of the keypair ($\mathsf{sk}_{\mathsf{slb}}$) is sealed against PCR17 with the value u_0 extended with the measurement slbA. The public part is conveyed to the user in a way which allows him to confirm the key generation was done correctly (see [11] §6.3.1 for details).

An authentication proceeds as follows. A nonce is generated by the server and sent to the client. The client encrypts this nonce and their password (pwd) using the public key that they hold and sends it to the server. The server sends

the nonce and the cipher text to the SLB, together with the sealed key and the salt (salt) extracted from the password file.

The server invokes the SLB using SKINIT. The SLB unseals the private part of the keypair, and uses that to decrypt the message from the client. The nonce contained in that message is compared with the nonce supplied by the server to confirm freshness. The password extracted from the message is hashed together with the salt provided by the server to form a value that the server can compare with the copy of the hash from the password file. The plaintext of the password is not available outside the SLB; the hash is available more widely.

As with the decryption oracle, we can see that no sealed data is output. Although the output is not plaintext (it is the hash of a password together with some salt) it will does not contain the TPM proof and is not encrypted.

Modelling We model the SSH password authentication application in TXML. As well as the salt and the public part of all keys, we assume that the attacker has the private part of $\mathsf{sk}_{\mathsf{Slb}}$ sealed against u_0 extended with the measurement of slbA.

$$\begin{aligned}\mathsf{K}_{init\_SSH} = \{&x_{salt} = \mathsf{salt},\\ &x_{pksrk} = \mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}),\\ &x_{pkslb} = \mathsf{pk}(\mathsf{sk}_{\mathsf{Slb}}),\\ &x_{sdata} = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, \mathsf{h}(0, \mathsf{measure}(\mathsf{slbA})), \mathsf{sk}_{\mathsf{Slb}}))\ \}\end{aligned}$$

where slbA is the program:

```
result := SKINIT {
  xsk_Slb := unseal(xSdata);
  xTemp := adec(xsk_Slb, xCipher);
  xPwd := fst2(xTemp);
  xNonce' := snd2(xTemp);
  check xNonce = xNonce';
  hash := md5(xSalt,xPwd);
  extend(fpc);
  rtn hash;
}.
```

Result of our analysis Because slbA does not output any seals (it only outputs an MD5 hash), $\overline{\mathsf{K}_{init\_SSH}}$ contains only one seal (x_{sdata} from the initial knowledge). Therefore, we can apply theorem 1 to K_{init_SSH} with $m = 1$.

We wrote a StatVerif model based on the above description. As previously stated, the SLB cannot output a sealed blob, as its only output is a hash of the password. We can bound the number bof extensions with the results in [6], and our theorem allows us to bound the number of resets and SKINITs. The complete StatVerif code for these examples, along with some supporting scripts to simplify the running of ProVerif and StatVerif, a description of the methodology used and some further background information, is available for download. The location is given in the bibliography.

5 Case Study: A Certification Authority

Description This certification authority example is also taken from [11]. It consists of two SLBs, one to perform key generation, the other to perform key signing.

The key generation SLB constructs a keypair ($\mathsf{sk}_{\mathsf{SignKey}}$) suitable for use in signing other keys, and the private part of $\mathsf{sk}_{\mathsf{SignKey}}$ is sealed against u_0 extended with the measurement of the second SLB.

For the signing SLB (slbC), the client forms a certificate signing request (CSR) containing a public key along with details of the client's identity. The client submits this to the key signing SLB, which has access to the sealed form of its own private key. The SLB checks the signing policy, then unseals its private part of the keypair in order to sign the CSR. The result is returned to the client.

Modelling We model the CA application in TXML, after making some abstractions. Firstly, we check that the signing SLB maintains the secrecy of any signing key with which it is used. This allows us to leave the key-generation SLB unmodelled and use a simple process that produces sealed keys instead. Secondly, as the required security property is the secrecy of the CA's signing key rather than the authenticity of the CSRs, the signing policy is not modelled.

We assume that the attacker has the public parts of the storage root key and the signing key $\mathsf{sk}_{\mathsf{SignKey}}$. We also assume that the attacker has the private part of $\mathsf{sk}_{\mathsf{SignKey}}$ sealed against u_0 extended with the measurement of slbC.

As in the previous example, we are able to determine that $m = 1$, as the application does not output any new sealed objects.

$$
\begin{aligned}
&\mathsf{K}_{init\_CA} = \{ \\
&\qquad x_{pksrk} = \mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \\
&\qquad x_{pkSignKey} = \mathsf{pk}(\mathsf{sk}_{\mathsf{SignKey}}), \\
&\qquad x_{sdata} = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, \\
&\qquad\quad (\mathsf{tpmPf}, \mathsf{h}(0, \mathsf{measure}(\mathsf{slbC})), \mathsf{sk}_{\mathsf{SignKey}})) \\
&\}
\end{aligned}
$$

where slbC is the program:

```
result := SKINIT {
  xskSignKey := unseal(xSdata);
  xCert := sign(xskSignKey,xCSR);
  extend(fpc);
  rtn xCert;
}.
```

Result of our analysis The security property we are checking is the secrecy of the CA signing key $\mathsf{sk}_{\mathsf{SignKey}}$. As a partial check that the model is correct we

also check for the existence of certificates signed by $\mathsf{sk}_{\mathsf{SignKey}}$. The queries are written in the StatVerif calculus as follows:

$$\begin{array}{ll} \mathsf{query}\ \mathsf{att}(u, \mathsf{sk}_{\mathsf{SignKey}}) & (F_5) \\ \mathsf{query}\ \mathsf{att}(u, \mathsf{sign}(\mathsf{sk}_{\mathsf{SignKey}}, x_{CSR})) & (F_6) \end{array}$$

We bound the number of PCR extensions as in §4. ProVerif then terminates with F_6 reachable, which shows that the model does in fact produce signed certificates, and F_5 unreachable, which shows that there are no short attacks on the secrecy of the CA signing key $\mathsf{sk}_{\mathsf{SignKey}}$. Based on K_{init_CA} and slbC, the model conforms to the conditions of Theorem 1. Therefore there is no attack on the secrecy of $\mathsf{sk}_{\mathsf{SignKey}}$.

6 Conclusion

Protected execution on x86 platforms involves a stateful model with a state space that in unbounded in two ways. First, a PCR value may be extended with arbitrary data an arbitrary number of times. Second, the PCR value is reset each time a protected execution session is begun, and this too can happen an arbitrary number of times. We proved that it is nonetheless sound to consider only attacker strategies that are bounded in both these senses. This allows us to use StatVerif to analyse protected execution, which we have done for some examples.

Hardware-based security mechanisms, such as TPM, TXT, virtualisation and Hardware Security Modules are an important part of defending computing platforms. Formal analyses of their often complicated APIs are therefore timely. Developing abstractions of the kind described in this paper is a step in extending the ProVerif methodology to hardware-based security mechanisms. In future work, we intend to explore some more of these mechanisms.

References

1. Advanced Micro Devices: Secure Virtual Machine Architecture Reference Manual. Advanced Micro Devices (2005)
2. Arapinis, M., Ritter, E., Ryan, M.D.: Statverif: Verification of stateful processes. In: Proc. of the 24th IEEE Computer Security Foundations Symposium. pp. 33–47. IEEE Computer Society Press (2011)
3. Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., O'Hanlon, B., Ramsdell, J., Segall, A., Sheehy, J., Sniffen, B.: Principles of remote attestation. International Journal of Information Security 10(2), 63–81 (2011)
4. Datta, A., Franklin, J., Garg, D., Kaynar, D.: A logic of secure systems and its application to trusted computing. In: Proc. of the 30th IEEE Symposium on Security and Privacy. pp. 221–236. IEEE Computer Society Press (2009)
5. Delaune, S., Kremer, S., Ryan, M., Steel, G.: A formal analysis of authentication in the TPM. Formal Aspects of Security and Trust pp. 111–125 (2011)

6. Delaune, S., Kremer, S., Ryan, M., Steel, G.: Formal analysis of protocols based on TPM state registers. In: Proc. of the 24th IEEE Computer Security Foundations Symposium. IEEE Computer Society Press (2011)
7. Fournet, C., Planul, J.: Compiling information-flow security to minimal trusted computing bases. Programming Languages and Systems pp. 216–235 (2011)
8. Grawrock, D.: Dynamics of a Trusted Platform: A Building Block Approach. Intel Press (2009)
9. Gürgens, S., Rudolph, C., Scheuermann, D., Atts, M., Plaga, R.: Security evaluation of scenarios based on the TCG's TPM specification. In: Biskup, J., Lopez, J. (eds.) ESORICS 2007, pp. 438–453. Springer, Berlin / Heidelberg (2007)
10. Lin, A.: Automated analysis of security APIs. Ph.D. thesis, MIT (2005)
11. McCune, J., Parno, B., Perrig, A., Reiter, M., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. In: ACM SIGOPS Operating Systems Review. vol. 42(4), pp. 315–328. ACM (2008)
12. Millen, J., Guttman, J., Ramsdell, J., Sheehy, J., Sniffen, B.: Analysis of a measured launch. `http://www.mitre.org/work/tech_papers/tech_papers_07/07_0843/07_0843.pdf`, Accessed 7 December 2011 (2007)

The StatVerif files corresponding to our experiments are available for download at:

`http://markryan.eu/research/projects/ProtectedExecution/`

A Proof of Theorem 1

Suppose K contains B_1, B_2, ..., B_n. Let m be the number of sealed blobs in $\overline{\mathsf{K}}$, as in definition 1. Let S be a strategy such that $(K, \bot) \stackrel{S}{\Rightarrow} (K', p)$. Then:

1 $\exists S'$ such that $(K, \bot) \underset{S'}{\stackrel{S}{\Rightarrow}} (K', \bot)$.

2 S' simulates S; that is $\exists q$ such that $(K, \bot) \stackrel{S'}{\Rightarrow} (K', q)$.

3 S' uses only data present in S, i.e. names$(S') \subseteq$ names(S).

4 The number of SKINITs plus the number of resets in S' is at most m.

We first prove a number of lemmas.

Lemma 1. $(K, \bot) \stackrel{S}{\Rightarrow} (K', p')$ *implies* $\forall p \exists p''(K, p) \stackrel{S}{\Rightarrow} (K', p'')$.

Proof. We show this by induction on S.

Base case $S = \emptyset$. The proof is obvious, setting $p'' = p$.

Inductive case $S = C; S_1$. We have $(K, \bot) \xrightarrow{C} (K', p') \stackrel{S_1}{\Rightarrow} (K'', p''')$. Take any p.

Taking each case of C

$\mathsf{skip}, x := f(), x := g(), x := \mathsf{seal}(u, v), \mathsf{check}\ u = v$ We have $p' = \bot$, and $(K, p) \xrightarrow{C} (K', p)$. Apply IH with p to obtain p'', and we have $(K, p) \xrightarrow{C} (K', p) \stackrel{S_1}{\Rightarrow} (K'', p'')$, i.e. $(K, p) \stackrel{S}{\Rightarrow} (K'', p'')$.

$\mathsf{unseal}(v)$ This situation is impossible.

$\mathsf{extend}(v)$ We have $p' = \bot$ and $(K, p) \xrightarrow{C} (K', h(p, v))$. Apply IH with $h(p, v)$ to obtain p'', and we then have $(K, p) \xrightarrow{C} (K', h(p, v)) \stackrel{S_1}{\Rightarrow} (K'', p'')$ i.e. $(K, p) \stackrel{S}{\Rightarrow} (K', p'')$.

$\mathsf{reset}, \mathsf{SKINIT}\{P; \mathsf{rtn}\, u\}$ We have $p' = 1$ and $(K, p) \xrightarrow{C} (K', 1)$, and $(K, p) \xrightarrow{C} (K', 1) \stackrel{S_1}{\Rightarrow} (K'', p''')$, i.e. set $p'' = p'''$ and we have $(K, p) \stackrel{S}{\Rightarrow} (K', p'')$.

Lemma 2. $(K, p) \xrightarrow{C} (K', p')$ *implies* $\exists C'.(K, p) \underset{C'}{\xrightarrow{C}} (K', p')$.

Proof. Consider each case of C in turn.

Lemma 3. $(K, p) \xrightarrow{C} (K', p')$ *implies* $\exists C'.(K, p) \underset{C'}{\xrightarrow{C}} (K', \bot)$ *or* $(K, \bot) \underset{C'}{\xrightarrow{C}} (K', \bot)$.

Proof. Consider each case of C in turn. For *unseal*, we prove the left disjunct. For all other cases, we prove the right disjunct.

Lemma 4. *If* $p \neq \bot$ *then* $(K,p) \xrightarrow[C']{C} (K',p')$ *implies* $(K,p) \xrightarrow{C'} (K',p')$.

Proof. Consider each case of C in turn.

Lemma 5. $(K,p) \xrightarrow[C']{C} (K',\bot)$ *implies* $p \neq \bot, (K,p) \xrightarrow{C'} (K',p)$ *or* $\exists p'.(K,p) \xrightarrow{C'} (K',p')$.

Proof. Consider each case of C in turn. For *unseal*, we prove the left disjunct. For all other cases, we prove the right disjunct.

Part 1 of Theorem 1

$(K,\bot) \overset{S}{\Rightarrow} (K',p')$ implies $\exists S'.(K,\bot) \underset{S'}{\overset{S}{\Longrightarrow}} (K',\bot)$.

Proof. We prove something more general:
$(K,p) \overset{S}{\Rightarrow} (K',p')$ implies $\exists S'.(K,p) \underset{S'}{\overset{S}{\Longrightarrow}} (K',\bot)$ or $(K,\bot) \underset{S'}{\overset{S}{\Longrightarrow}} (K',\bot)$.

Base case $S = \emptyset$ is obvious.

Inductive case $S = C; S_1$
Suppose $(K,p) \overset{S}{\Rightarrow} (K',p'')$. RTP $\exists S'.(K,p) \underset{S'}{\overset{S}{\Longrightarrow}} (K',\bot)$ or $(K,\bot) \underset{S'}{\overset{S}{\Longrightarrow}} (K',\bot)$.
Expanding: $(K,p) \xrightarrow{C} (K',p') \overset{S_1}{\Rightarrow} (K'',p'')$. By inductive hypothesis, $\exists S_1'$.

- either $(K,p') \underset{S_1'}{\overset{S_1}{\Longrightarrow}} (K',\bot)$. From $(K,p) \xrightarrow{C} (K',p')$, by Lemma 2, $\exists C'.(K,p) \xrightarrow[C']{C}$ (K',p'). So set $S' = C'; S_1'$. Then $(K,p) \xrightarrow[S']{S} (K',\bot)$.
- or $(K,\bot) \underset{S_1'}{\overset{S_1}{\Longrightarrow}} (K',\bot)$. From $(K,p) \xrightarrow{C} (K',p')$, by Lemma 3, either $(K,p) \xrightarrow[C']{C}$ $(K',\bot)$, so $(K,p) \underset{S'}{\overset{S}{\Longrightarrow}} (K',\bot)$, or $(K,\bot) \xrightarrow[C']{C} (K',\bot)$, so $(K,\bot) \underset{S'}{\overset{S}{\Longrightarrow}} (K',\bot)$ where again, $S' = C'; S_1'$.

Part 2 of Theorem 1

$(K,\bot) \underset{S'}{\overset{S}{\Longrightarrow}} (K',\bot)$ implies $\exists p'.(K,\bot) \overset{S'}{\Rightarrow} (K,p')$.

Proof. We prove something stronger:
$(K,p) \underset{S'}{\overset{S}{\Longrightarrow}} (K',\bot)$ implies $\exists p'.(K,p) \overset{S'}{\Rightarrow} (K',p')$.
We prove this using induction on S.

Base case $S = \emptyset$ is obvious.

Inductive case $S = C; S_1$. Inductive hypothesis: $(K, \bot) \overset{S_1}{\underset{S_1'}{\Longrightarrow}} (K', \bot)$ implies $\exists p'.(K, \bot) \overset{S_1'}{\Longrightarrow} (K', p')$.

We want to prove $(K, p) \overset{S}{\underset{S'}{\Longrightarrow}} (K', \bot)$ i.e. $(K, p) \underset{C'}{\overset{C}{\longrightarrow}} (K', p') \overset{S_1}{\underset{S_1'}{\Longrightarrow}} (K'', \bot)$.

- Either $p' = \bot$
  - either $p = \bot$: $(K, \bot) \underset{C'}{\overset{C}{\longrightarrow}} (K', \bot) \overset{S_1}{\underset{S_1'}{\Longrightarrow}} (K'', \bot)$, so by Lemma 4 and IH $\exists p''.(K, \bot) \overset{C'}{\longrightarrow} (K', \bot) \overset{S_1'}{\Longrightarrow} (K'', p'')$, i.e. $(K, \bot) \overset{S'}{\Longrightarrow} (K'', p'')$.
  - or $p \neq \bot$: $(K, p) \underset{C'}{\overset{C}{\longrightarrow}} (K', \bot) \overset{S_1}{\underset{S_1'}{\Longrightarrow}} (K'', \bot)$. by Lemma 5 and IH $\exists p''$.
    - either $(K, p) \overset{C'}{\longrightarrow} (K', p), (K, \bot) \overset{S'}{\Longrightarrow} (K'', p'')$, Then by Lemma 1, $\exists p'''.(K, p) \underset{C'}{\overset{C'}{\longrightarrow}} (K', p) \overset{S_1'}{\Longrightarrow} (K''', p''')$, i.e. $(K, p) \overset{S'}{\Longrightarrow} (K''', p''')$.
    - or $\exists p_1'.(K, p) \overset{C'}{\longrightarrow} (K', p_1'), (K', \bot) \overset{S_1'}{\Longrightarrow} (K'', p'')$. Then by Lemma 1, $\exists p'''.(K, p) \overset{C'}{\longrightarrow} (K', p) \overset{S_1'}{\Longrightarrow} (K'', p''')$, i.e. $(K, p) \overset{S'}{\Longrightarrow} (K'', p''')$.
- or $p' \neq \bot$ then $(K, p) \overset{C'}{\longrightarrow} (K', p')$ by Lemma 4, and so by IH $\exists p''.(K, p') \overset{S_1'}{\Longrightarrow} (K', p'')$, i.e. $(K, p) \overset{S'}{\Longrightarrow} (K'', p'')$.

Part 3 of Theorem 1

S' uses only the data present in S that is, every name in S' is in S.

Part 3 of the theorem is readily proved by inspection of the transformation.

Part 4 of Theorem 1

The number of SKINITs plus the number of resets in S' is at most m.

Part 4 follows from the facts that:

- at most m plaintext-distinct sealed blobs can be produced from the initial data;
- the transformed strategy S' runs at most one SKINIT for each blob sealed to a PCR value rooted in 0 (other invocations are run as SUBRs);
- the transformed strategy S' runs at most one reset for each sealed blob rooted in 1 (other resets are transformed into skips).