Analysis of a Measured Launch*

Jon Millen, Joshua Guttman, John Ramsdell, Justin Sheehy, Brian Sniffen
The MITRE Corporation
Bedford, MA

June 5, 2007

Abstract

The design of a trusted system based on the Trusted Computing Group's Trusted Platform Module (TPM) was analyzed to understand the role and trust relationships of the TPM, firmware, and software modules involved. The objective was to confirm that the measurements stored and reported by the TPM can successfully discriminate a normal boot sequence, which leaves trusted system software in control, from an insecure one, where some trusted modules might have been replaced by malicious ones. The principal tool used in the analysis was the SMV symbolic model checker.

1 Introduction

The Trusted Platform Module (TPM) technology, developed and standardized by the Trusted Computing Group, provides a hardware *root of trust* for reliable integrity reporting. In conjunction with a public key infrastructure, TPM features support *attestation* protocols that permit a remote party to obtain a reliable report on the configuration of a platform. The ability to obtain authentic information about a remote platform is essential in a dynamic global Internet environment, where clients may seek services from remote systems that they have not previously used or authenticated.

By storing and reporting a record of the software loaded and executed during system initialization, a TPM provides information relevant to trust relationships between a client and server. Trust relationships should depend not only on the identity of the owner or user of the platform, but also on the identity, quality and integrity of the software running on it.

*Sponsored by the U.S. Dept. of Defense under contract W15P7T-07-C-F600.

A TPM does not, by itself, ensure that a platform was booted with trusted software, but it can provide a digitally signed report that shows that a certified TPM is present, and also shows what software was booted, so that an external appraiser can check it against the expected or desired configuration. It has some additional functions as well, to support secure storage and cryptographic services.

The TPM is a fairly complicated device, although the basic principles of storage and reporting of measurements are conceptually simple. In a TPM-equipped system, the boot sequence begins with firmware on the TPM chip. Each firmware or software module in the boot sequence stores a hash value of the next module into a PCR (Platform Configuration Register) in the TPM. The validity of each measurement is conditioned on the correctness of the module performing the measurement, which was itself measured. This is the "chain of trust".

In the more recent version 1.2 of the TPM, some of the responsibility for trusted operations is delegated to external firmware in the associated chipset, as provided by the CPU manufacturers. This permits some flexibility, such as a partial reboot with new low-level system software, called a "late launch". It also complicates the argument supporting the integrity of the stored measurements.

The chain of trust rationale is explained in [1] and [3]. The argument supporting the extended architecture as implemented by Intel for the TPM version 1.2 is covered in detail in [4]. An application of the AMD chipset for the TPM 1.2 is described in [9].

There is a conceptual gap between the basic idea of the chain of trust, and the implementation details, as covered carefully but informally in a reference like [4]. This led us to ask if we could check the argument for validity of measurements formally, using an abstract model of the TPM 1.2 and associated architecture. A particular concern was to model the possibly malicious misbehavior of contaminated trusted software, which might attempt to load the expected measurements into the TPM registers. With a model checker, we could attempt to prove that if the measurements are the expected ones, then the software in control does, in fact, have those measurements. The modeling activity should also expose the assumptions underlying this conclusion.

An initial version of such a model, written for the SMV model checker [5], has been developed and exercised. We have used it to check which PCRs had to be reported in order to infer that desired system software is present, and to verify an expected property of late launch, namely, that even if the BIOS is compromised, the late launch process still permits trusted system

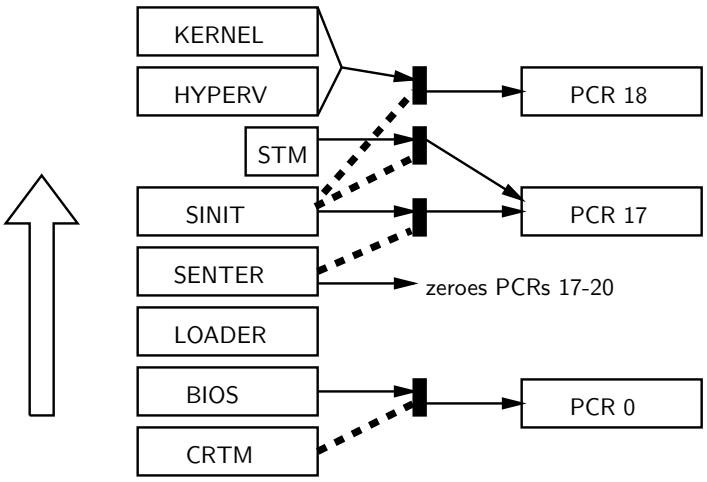


Figure 1: Modules and Measurements

software to be loaded and confirmed. The present model is not viewed as final, but rather a starting point to investigate different configurations and further implementation detail.

2 The Launch Model

The system software architecture appearing in the model is generic, and based loosely on existing experimental prototypes of which we have some knowledge. The boot sequence is constrained by the requirements of the TPM 1.2 and associated chipset, but once the sequence reaches the persistent operating system kernel that takes control of memory management, there are many alternatives. In our notional design, we assume a hypervisor that supports virtualization, with an operating system kernel above it.

The expected launch sequence is shown in Figure 1. Each box in the left side of the figure represents a firmware or software module that performs some essential functions for system startup, then transfers control to the next module higher up, loading it into memory first if necessary. The bottom module, the CRTM (Core Root of Trust for Measurement), also known as the Trusted Boot Block of the BIOS, is firmware that is guaranteed by the TPM support chipset to be executed first. It performs a self-measurement, so its measurement is not put in a PCR.

The PCRs that are used in this boot sequence are shown on the right

side of the figure. That part of the figure will be explained later. Full measurements are not necessarily stored in PCRs, but rather in a separate log. A PCR only has room for a 160-bit hash. In the case of launch modules, however, the hash of the module contents does happen to be its measurement.

Lower modules are expected to execute only once (until a reboot), while the hypervisor and OS kernel remain available. Processes above the OS kernel come and go as needed. Measurement of application software is presently not modeled, though later versions of the model might include them.

This section concludes with some reference information regarding the SMV model checker. In the next two sections, the launch model is described. The last section presents the security properties that were tested and shows how the analysis result was obtained.

2.1 SMV Sources

Source code and some binaries for the SMV model checker are available from the CMU Web site [6]. One can also use the NuSMV system available from the ITC-irst research center in Italy, from its Web site [7]. NuSMV is a re-implementation of SMV as an open source project, and has some additional features such as a SAT solver (as opposed to the BDD solver in SMV) and some extensions and options for property specification formulas. The model can be run using either tool.

3 General Features of the Model

The launch model is a high-level abstract model designed to capture the essential logical features of the launch and measurement process. The structure of the model is as general as possible to allow different configurations and assumptions to be examined and compared.

An SMV model consists of *modules* specifying component types, and a particular module called `main` specifying how a system is constructed by interconnecting instances of the component types. The main module also includes a list of properties to be tested.

The model has a module for each of the boxes shown on the left in Fig. 1. These modules are instances of a type we refer to as "executable". The model also has a module type for PCRs. Figure 1 shows the expected placement of measurements. In the figure, the measurements follow the arrows, and the dotted lines indicate which module is performing and writing the measurement.

In the notional design,the hypervisor and kernel modules are loaded from a single image, and this combined image is measured into PCR 18. The Intel description of SINIT in [4] states that SINIT extends PCR 17 with a hash of the SMI Transfer Monitor (STM) for System Management Interrupts when the STM is enabled.

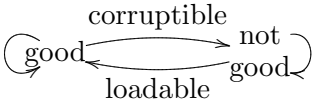
The specification of a module includes its state variables and its next-state transition relation. The normal style of state transitions in SMV is a parallel or synchronous assignment in which all variables receive their new values in a single "tick". An asynchronous option is available, but was not used in this model.

One tick in the model represents a period of time during which exactly one of the explicitly specified executable modules is in control. PCRs are passive, and may undergo a transition in parallel with any executable module. Applications are represented with a single untrusted generic executable module, not shown. Any transfer of control starts a new state.

3.1 Executable Modules

The module type `exec` represents the state of an executable program instance, especially a member of the launch sequence. Conceptually, a module instance is a contiguous portion of memory, whether it is in main memory, flash memory, or microcode, with a designated entry point. At a given time, that portion of memory may or may not contain trusted code. In the model, the state of a module is a boolean variable `good`. If the state of a trusted module is good, we can predict what it measures and launches. The variable `good` reflects the true state of the module, and part of our modeling concern is to establish a logical connection between the true state and the reported PCR values.

Instances of an executable module are characterized by three simple parameters: `init_good, loadable,` and `corruptible`. The first, `init_good`, determines the initial value of `good`. The second, `loadable`, indicates whether or not the module is capable of changing from a not-good to a good state. If this is true, the module is associated with a portion of main memory that can or must be loaded from disk, so that `init_good` should normally be false for it. On the other hand, if it is read-only microcode, its goodness value will remain at whatever the initial value was.



The third parameter, `corruptible`, indicates whether the module is capable of changing from a good to a not-good state. A false value reflects the ability of good software to protect itself from adverse modification, due to its control over DMA, page registers, interrupts. Non-corruptibility also implies that the module does not damage itself or behave in ways that violate the measurement and execution assumptions for it. The corruptibility parameter is the one that requires the most thought when assigning it for a given executable module.

The executable module instances normally receive fixed parameter values that characterize the inherent qualities of the module. One exception to this rule is that the kernel is corruptible if and only if the hypervisor is not good, since a bad hypervisor can undercut the defenses of the kernel. Thus, kernel is specified as an executable instance with corruptible parameter equal to `!hyp.good`. (A parameter value can be a state variable value from another module.)

We might make another exception to this in the future to analyze the benefits of remeasurement of the hypervisor or kernel, if their noncorruptibility is viewed as subject to failure due to unknown flaws or vulnerabilities, triggered by some untrusted software.

There is no explicit transition to force loading of a good copy of software. When a module is loadable and not good, the next state of `good` is undetermined - that is, there is a nondeterministic transition in which `good` becomes either true or false. This makes sense because it allows for two possibilities: that good software is loaded into the module memory from disk, or that the loaded software is not good because the load is unsuccessful or the disk image is corrupted or buggy.

If a module is currently good but it is corruptible, the undetermined case also applies, since it might be modified by DMA. This transition can happen whether the module is currently being executed or not.

3.2 The PCR Module

A PCR module has one state variable, representing its value symbolically, and two parameters: `static`, which is a fixed boolean value, and `input`. A PCR is dynamic (not static), according to the TPM specification, if it can be reset without a reboot. Static PCRs, 0-15, are reset only on system reboot. PCRs 16 and above are dynamic.

The value of the input parameter is either a reset command, a new value to be stored with a TPM_Extend command, or a value indicating the absence of any command or input. The input parameter value is determined by the

behavior of the executable component currently running.

Measurements are symbolized with constants `BIOSm, SINITm, STMm, SYSm, SISTm` representing expected possible measurements. A PCR might also have the symbolic values `ZERO`, the zero value caused by a static reset; `FFFF`, the all-ones value set by an SENTER reset, or `UNTm`, a value symbolically representing any unexpected value.

The TPM_Extend command does not simply write into a PCR; it combines the former value with the new one by hashing the concatenation. This behavior is carried out symbolically in the model.

4 Main Module

The module `main` identifies the instances of PCRs and executable modules being used, shows how their arguments are computed, and manages the execution sequencing via a program counter variable `pc`. It also specifies the security properties being tested.

There is a module instance for each of the boxes in Figure 1 except for the STM, which has a measurement but is not launched explicitly.

4.1 The Program Counter

The symbolic values of the variable `pc` represent the executable modules in the launch sequence. Keep in mind that the PC points to areas in microcode or main memory that may or may not contain good trusted code. The value `UNT` represents any location that is unknown or untrusted, but not one of the other named modules.

The launch sequence is determined by the initial and next-state assignment for `pc`, shown here:

```
ASSIGN
  init(pc) := CRTM;
   next(pc) := case
    pc = CRTM & crtm.good : BIOS;
    pc = BIOS & bios.good : LOADER;
    pc = LOADER & loader.good : SENTER0;
    pc = SENTER0 & senter0.good : SENTER1;
    pc = SENTER1 & senter1.good & sinit.good : SINIT;
    pc = SENTER1 & senter1.good & !sinit.good : UNT;
    pc = SINIT & sinit.good : HYP;
```

```
pc = HYP & hyp.good : KER;
pc = KER & ker.good : {HYP,KER,UNT};
1 : {BIOS,LOADER,SENTER0,SINIT,HYP,KER,UNT};
esac;
```

This assignment conveys some interesting features of the boot sequence.

The launch sequence starts with the CRTM (trusted boot block) and thereafter depends on the current `pc` value. Note that the next value of `pc` is predictable only when the current module is good. For example, if the boot loader is good, it should finish up by executing SENTER. Although SENTER is an instruction rather than a programmable module, it is treated here as a trusted module, since it has a role in the launch sequence comparable with other modules such as the CRTM and SINIT.

SENTER is split artificially into two modules, SENTER0 and SENTER1, because it has two successive effects on PCRs. First, it resets the dynamic PCRs 17-20. Then, it normally puts a measurement of SINIT into PCR 17. The normal sequencing is therefore SENTER0, SENTER1, SINIT.

Because SINIT is an *authenticated code module*, SENTER can check its measurement before launching it. The normal launch sequence aborts if SINIT has a bad measurement. (The exclamation point ! is used for negation in SMV.)

The "else" case (with boolean condition 1) is taken when running module is not in a good state, in which case the next PC value is unpredictable, and its new value can be any of the modules listed. A late-launch scenario could be modeled by changing the next state of the loader to `UNT`, representing an untrusted operating system.

The transition from hypervisor to kernel is, of course, not merely a branch, but a complex process involving the creation of a virtual machine and the initialization of the kernel. More detail could be added here, if there are multiple alternatives or cases that should be examined.

4.2 PCR Inputs

The inputs to PCRs are given as variables `p0in, p17in, p18n`. As an example, consider PCR 17. It is declared as a variable as:

```
pcr17:  pcr(0,p17in)
```

meaning that it is an instance of the `pcr` type that is not static, and it gets its input from a state variable `p17in`, defined by:

Module	Locality	Entity in control, can	Reset PCRs	Extend PCRs
	Any	Any	16,23	16,23
CRTM,BIOS, KER	0	Static CRTM, Static OS	None	0-15
	1	Dynamic OS	None	20
HYP	2	Dynamic OS	20,21,22	17-22
SINIT	3	Auxiliary trusted	None	17-20
SENTER	4	Dynamic CRTM	17-20	17,18

Figure 2: Locality Usage

```
ASSIGN
  p17in := case
    pc = SENTER0 & senter0.good : RSET;
    pc = SENTER1 & senter1.good & sinit.good : SINITm;
    pc = SENTER1 & senter1.good & !(sinit.good) : UNTm;
    pc = SINIT & sinit.good : STMm;
    pc = HYP & hyp.good : NONE;
    pc in {SENTER0,SENTER1,SINIT,HYP} : {NONE,SINITm,STMm,UNTm};
    1 : NONE;
    esac;
```

This case statement reflects the possibility that any executing module may attempt to set PCR 17. Trusted modules other than SENTER and SINIT will voluntarily leave it alone. If an untrusted or corrupted module performs TPM_Extend, it may try to put some (spurious) valid measurement into the PCR.

Locality restricts which modules may modify a PCR. In the TPM, locality restricts the memory address ranges from which certain commands affecting PCRs will be accepted. Each module has a locality of 0-4 depending on which address range it occupies. The assignments are made partly by the chipset and partly by the system architect.

The assumptions made in this version of the model are shown in Figure 2, which includes Table 1 in Section 3.1 of the TCG PC Client specification. [2] This table is embodied implicitly in the PCR input variable assignments.

5 Specification of Security Properties

The launch model is designed to be a tool for investigating a variety of questions about the relationship between reported PCR values and the state of the system. This section presents a few particular properties of interest that have been tested. We begin with a summary of the SMV provisions for specifying properties.

5.1 SMV Specification Language

Specifications to be tested are written in CTL (Computation Tree Logic). A CTL formula has the form *quantifier temporal-operator boolean-expr* where the quantifier is A or E, and the temporal operators are:

G (globally)	from this state onward
F (finally)	eventually, in this or some later state
X (next)	in the next state
U (until)	[p U q] p holds in every state until finally q

A formula without the initial quantifier is an LTL (linear temporal logic) formula. An LTL formula like Gp applies to some particular state sequence beginning with the current state. If ϕ is an LTL formula, $A\phi$ means that ϕ holds for every possible state sequence beginning with the current state, and $E\phi$ means that there exists a state sequence from the current state for which ϕ holds.

SMV checks each specification for every possible initial state. This is somewhat confusing for E properties, since it means there is an implicit "for all" in front of the property. This confusion is avoided in the launch model by initializing every variable deterministically, so that there is only one initial state for each model run.

When an AG specification fails for some initial state, SMV generates a trace showing the failure.

5.2 State of the Kernel and Hypervisor

The basic property we want is that if PCR values are normal, then the kernel and hypervisor are good. It turns out that this can be guaranteed, but only if we ask to see enough PCR values.

We might try to express this property for the hypervisor as:

```
AG (pcr18.val = SYSm -> hyp.good}
```

This says that for every reachable state, if PCR 18 has the expected measurement for the system software, then the hypervisor's "good" value is true. If we run SMV with this specification, it finds the following counterexample.

pc	CRTM	BIOS	LOADER	SENTER0	SENTER1	LOADER	HYP	BIOS
crtm.good	1	1	1	1	1	1	1	1
bios.good	1	1	1	1	1	1	1	1
loader.good	0	0	0	0	0	0	0	0
senter0.good	1	1	1	1	1	1	1	1
senter1.good	1	1	1	1	1	1	1	1
sinit.good	0	0	0	0	0	0	0	0
hyp.good	0	0	0	0	0	0	0	0
ker.good	0	0	0	0	0	0	0	0
unt.good	0	0	0	0	0	0	0	0
pcr0.val	ZERO	BIOSm	BIOSm	BIOSm	BIOSm	BIOSm	BIOSm	BIOSm
pcr17.val	FFFF	FFFF	FFFF	FFFF	ZERO	UNTm	UNTm	UNTm
pcr18.val	FFFF	FFFF	FFFF	FFFF	ZERO	ZERO	ZERO	SYSm
p0in	BIOSm	NONE	NONE	NONE	NONE	NONE	NONE	NONE
p17in	NONE	NONE	NONE	RSET	UNTm	NONE	STMm	NONE
p18in	NONE	NONE	NONE	RSET	NONE	NONE	SYSm	NONE
biosgood	1	1	1	1	1	1	1	1

The problem discovered by the model checking search is that a bad SINIT causes SENTER to return to the loader, which can invoke the bad hypervisor, which can put a spurious good SYSm into PCR 18.

This won't fool anyone who looks at PCR 17, however, to check for the expected `SISTm` value. The spec

```
AG (
( pcr17.val = SISTm
& pcr18.val = SYSm)
-> (hyp.good & ker.good))
```

is verified as true. This is the almost the result we were looking for. But we also want the hypervisor and kernel to remain in a good state when they are executed subsequently. This holds because they are self protecting, so the following stronger property is also verified as true.

```
AG ((
   pcr17.val = SISTm & pcr18.val = SYSm )
-> AG ((pc = HYP -> hyp.good) & (pc = KER -> ker.good)))
```

This specification holds even when we assume that the BIOS is not good.

It is interesting that the same specifications are also verified without assuming that SINIT is authenticated, only that it is measured like other trusted modules. That is, the program counter lines for SENTER1 can be replaced by the single line

```
pc = SENTER1 & senter1.good:  SINIT;
```

without changing the result.

The model contains a few other specifications designed to investigate the relationship between PCR values and the launch sequence. We conclude, for example, that SENTER must be executed in order to arrive at good and measured hypervisor and kernel states. Results of this kind may be useful to analysts as a way to help develop intuition about what steps in the launch sequence are necessary and how they work.

6 Conclusion

This launch model is lightweight from a performance point of view. The SMV model generates about 29,000 BDD nodes altogether, but it executes in a small fraction of a second. The model code is systematic and understandable, but a system design change typically results in model changes distributed over several places. An interface to generate model updates automatically from concise application-oriented input would be desirable. We are currently also investigating the use of SAL [10], a more recent language and modeling framework, for this analysis.

We found, as a result of analysis on the model, that good PCR values in PCRs 17 and 18 imply that good copies of the hypervisor and kernel have been loaded, and they remain good as long as they are self-protecting. It is not sufficient to check PCR 18 only. Corruption of the BIOS (measured into PCR 0) does not matter, or of non-measured software such as the boot loader. The analysis depends on the details of how features of the Intel chipset are used, in particular the way in which localities control access to PCRs. It also depends on the assumed ability of the hypervisor and kernel to protect themselves once they are correctly loaded, despite possible malicious devices or prior misbehavior of the BIOS.

This model is only the first step in a program for using this style of analysis to check the integrity arguments for a number of different architectures based on the TPM and measurement strategies.

References

[1] Trusted Computing Group TPM Main Part 1 Design Principles Specification version 1.2, Revision 94, 2006

[2] Trusted Computing Group TCG PC Client Implementation for Conventional BIOS Spec v. 1.2, Rev 1.00, 2005

[3] S. Pearson, ed., et al Trusted Computing Platforms Hewlett-Packard and Prentice Hall, 2003

[4] David Grawrock *The Intel Safer Computing Initiative* Intel Press, 2006

[5] K.L. McMillan The SMV System Carnegie Mellon Technical Report, School of Computer Science, 1992

[6] Carnegie Mellon University Model Checking Group `http://www.cs.cmu.edu/~modelcheck/smv.html`

[7] Center for Scientific and Technological Research `http://nusmv.irst.itc.it`

[8] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn Design and Implementation of a TCG-based Integrity Measurement Architecture Proc.13th USENIX Security Symposium, August, 2004

[9] J. McCune, B. Parno, A. Perrig, M. Reiter, and A. Seshadri Minimal TCB Code Execution (Extended Abstract) 2007 IEEE Symposium on Security and Privacy, IEEE Computer Society, 2007, pp. 267-272

[10] L. deMoura, Sam Owre, and N. Shankar The SAL Language Manual CSL Technical Report SRI-CSL-01-02 (Rev. 2), SRI International, August 2003