

The Role of TPM in Enterprise Security

Reiner Sailer, Leendert van Doorn, James P. Ward

Establishing trust in a remote computer system is an essential building block for distributed systems. Unfortunately trust is a hard property to achieve without appropriate hardware support. In this article we describe our trusted computing platform where we extend the hardware rooted trust guarantees of TCG technology to the (Linux) operating system and all its applications and allow remote parties to verify these trust guarantees.



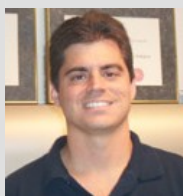
Reiner Sailer
Research Staff Member
IBM Research
Secure Systems

E-Mail: [sailer@watson.ibm.com]



Leendert van Doorn
Senior Manager
IBM Research
Secure Systems

E-Mail: [leendert@watson.ibm.com]



James P. Ward
President Trusted
Computing Group
Senior Technical
Staff Member, IBM
Software Group

E-Mail: [jpward@us.ibm.com]

Introduction

Establishing trust between entities is becoming an increasingly important requirement in today's highly connected and distributed Enterprise business environment. Business critical capabilities such as remote access, distributed workforce, dynamic outsourcing, and business portals all implicitly depend on mechanisms for verifiably establishing the authenticity and integrity of the connected devices, processes, and services. Emerging business models and architectures such as GRID, On Demand, and Utility Computing will further emphasize the need for determining these trust attributes in a standardized and interoperable manner.

The Trusted Computing Group (TCG, [1]) specifications are intended to provide an open set of security related building blocks for enhancing the trust associated with a computing platform. These common building blocks are developed to be platform and vendor agnostic such that they can be applied into any device type (i.e. PCs, servers, mobile phones, embedded devices), operating system (e.g., Linux, Windows, UNIX, Solaris), or solution framework (Web Services, etc). The TCG trust model is based on establishing a common assurance root and function definition for these trust characteristics. The TCG Trusted Platform Module, or TPM, serves as the starting point, or root, for this transitive trust model. The TPM as core root of Trust for measurement, or CRTM, can measure additional system attributes and then later verifiably report them as a basis for determining the overall trustworthiness of a platform. The TPM must be and is designed to be initially trusted because it represents the start of the trust chain. The measure, record, and report process is cumulatively referred to as attestation. The TCG function does not qualitatively assess what the information means in terms of trustworthiness. Rather, information reported via the TCG building blocks can be considered trustworthy. The manner this information is used to deter-

mine trust is a function of policy outside of the existing TCG standards.

The TCG standardizes the measurement and reporting of attributes covering trust establishment into the boot process of a system. However, this does not yield information about the trustworthiness of the runtime environment built on top of it. We solve this problem by extending the measurement from the static boot sequence into the dynamic runtime of the operating system (OS) and enabling the attestation of properties of a system's runtime.

The remaining of the paper is organized as follows: in Section 1, we introduce attestation based on TPM hardware; in Section 2, we present our architecture in detail and explain how we extend attestation into the system runtime. Section 3 describes the major results and shows at an example how we can detect a compromised system runtime using attestation. We conclude in Section 4 describing future work in this area.

1 TPM-based Attestation

The TPM represents a separate trusted co-processor, whose state cannot be compromised by potentially malicious host system software. TPM-based attestation represents a powerful tool for establishing the trust attributes of a system. Attestation based information about the device hardware, firmware, operating system, and applications can all be dynamically assessed to determine if the system should be trusted prior to granting a privilege (network / resource access, service, etc).

Unlike secure boot, which loads only signed and verified software, the TCG trusted boot process only takes measurements up to the bootstrap loader and leaves it up to the remote party to determine the system's trustworthiness. Thus, when the system is powered on it transfers control to an immutable base. This base will measure the next part of BIOS by computing a SHA1 secure hash over its contents and

protect the result by using the TPM. This procedure is then applied recursively to the next portion of code until the OS has been bootstrapped.

We adjust the role of the TPM by using it to protect the integrity of the in-kernel measurement list rather than holding measurements directly. To prove to a remote party what software stack is loaded, the system needs to present the TPM state using the TCG attestation mechanisms and this ordered list. The remote party can then determine whether the ordered list has been manipulated and, once the list is validated, what kind of trust it associates with the measurements. We have modified the Linux kernel and the runtime system to take integrity measurements as soon as executable content is loaded into the system, before it is executed [2]. We maintain the ordered list of measurements inside the Linux kernel.

Unlike existing approaches, such as secure boot or authenticated boot [3], where a system is instrumented to boot only signed and verified software, or secure coprocessors [4], which offer a closed environment to run certified and signed software in a protected environment, our approach is non-intrusive and does not change the behavior of the system that is being attested. It can be used in open environments where a large spectrum of software runs and changes regularly.

2 Mutual Attestation

In this Section, we demonstrate the power of TPM-based remote attestation in the process of making informed decisions about trust for Web services, and help to clarify how these concepts could be used in an open environment. Mutual platform attestation is the process by which peers in a transaction, with potentially no previous knowledge of each other, can establish trust based on the integrity of each other's computing environment – e.g., that a peer is truly running the service being offered in an environment that is acceptable.

Prior to the actual transaction, peers exchange integrity measurements of each other's environment – e.g., fingerprints of all the software running on each system. As trust decisions are then based on these measurements, authenticity is a critical factor. Our architecture uses the Trusted Platform Module (TPM) to protect and assure the validity of the integrity measurement for each executable that is loaded into the OS

runtime. Thus, coupled with the integrity measurements of the boot process from system firmware through OS program load (which also uses TPM), remote parties are assured the authenticity of the integrity measurements since system boot and can make appropriate trust decisions based on service requirements. This approach is applicable in a host of other scenarios including remote Systems Management and Service Level Agreement or Quality of Service verification on-demand.

Goal: Our goal is to measure what is useful and necessary for a challenging party to regenerate the software stack of an attested system securely and to determine trusted properties of the attested system. We instrumented the Linux kernel to create and store such measurements as well as protect them against compromised systems by using the TPM hardware. Our approach is non-intrusive in the sense that it will prevent a system neither from becoming compromised nor from manipulating the kernel-held measurements. However, we do prevent such a system from posing as a non-compromised system by allowing challenging parties to independently validate the attested party's integrity by means of the received measurement list.

Assumptions: We assume that the TPM works correctly and its operation is not manipulated using physical attacks (which it is not designed to withstand at this time). We also assume that the challenging party possesses a valid certificate to the signature key used by the TPM of the attested system.

Limitations: As we measure data when it is loaded, vulnerabilities propagated by running software – e.g. through buffer-overflow exploits – will not be represented in the measurements. However, the known potential of executables to become compromised during run-time is well represented in the measurements. Challenging parties can derive the identity of the program and its version from the measurement and relate it to known vulnerabilities, e.g., using CERT [10] data bases when deciding whether to trust this part of the software stack.

2.1 Experiment Setup

Figure 1 gives an overview of the process followed in our experiment. We distinguish two independent systems: the *attested system* – the system whose software-stack is to be validated – and the *challenging party*

system that is going to validate the software-stack of the attested system.

The attested system is instrumented to produce evidence (called measurements, see Fig. 1, step 1) that allow challenging parties to re-create its run-time safely. The attested system contains a TPM security chip that protects the integrity of the created evidence even if the attested system should become compromised later on. In response to the demand of an authorized challenging party, the attested system returns its evidence and also provides related contents of the security chip that allows the challenging party to validate the integrity of the provided evidence (i.e., the measurements).

The challenging system requests from the attested system the actual measurement list as well as the integrity value over the list, which is stored inside the TPM, validates the integrity of the list (step 2), evaluates the individual measurement, and finally reconstructs an image of the attested system's software stack (step 3). Based on this image, the challenging system concludes about properties of the attested system's runtime (step 4). Exchanging the roles of attested and challenging system then implements mutual attestation.

The following subsections describe the instrumentation of the attested system to produce and protect evidence about the loaded software stack since reboot (2.2), the establishment of trust into the evidence of a system (2.3), and the interpretation of evidence to conclude properties of the software stack of the attested system (2.4).

2.2 System Instrumentation

We have instrumented the Linux kernel [2] of the attested system to produce measurements of post-boot events that affect the run-time of the system. Our measurements are taken in a way that allows (remote) parties to securely reconstruct what was actually loaded into the software stack of a system and to determine if this system can be trusted according to the local security policy. To establish trust into the instrumented Linux kernel, preceding boot stages produce measurements through the TPM hardware, the BIOS, and the Grub boot loader stages, each stage in turn gathering and storing information about the attested system's next boot stage up to the running kernel.

As opposed to other approaches, e.g., terra [8] measuring whole partition contents

of virtual machines, we aim at representative measurements of software stack components that are rich in semantic value and allow challenging parties to reconstruct functional properties of the actual software stack. Therefore, we instrumented the Linux kernel running on the attested system to create such evidence based on which the software stack can be reconstructed.

What we measure: The goal is to create *representative evidence* that can be interpreted by a challenging party in order to decide whether loading the represented data maintains or breaks the trust into the overall software stack of a system.

We consider the following information about loading data into the run-time as being representative evidence:

- Kernel modules – they potentially affect the measurement architecture in the kernel
- Executables and shared libraries – they don't change often and can be related to functionality as well as known vulnerabilities
- Configuration files – they don't change often once the system is correctly configured. Additionally, configurations can be decisive for the trustworthiness of the program consuming and interpreting them.
- Other important input files that affect trust into run-time software stack, e.g., Bash command files, Java Servlets, and java libraries.

We don't consider measuring dynamic input data such as user input, web requests, and remote commands because they would not lead to representative semantic value. Vulnerabilities based on such data are better addressed by operating system access control mechanisms (e.g., SELinux [11]), which are represented in the kernel measurement and available to attesting parties for trust establishment.

How we measure: A measurement is implemented as the computation of the 160bit result of a SHA1 hash function (fingerprint) applied to the file that contains data or executables loaded into the run-time. The slightest difference in a data file will generate a distinguished fingerprint and, hence, variations in programs (e.g., due to viruses or Trojan horses) are easily detected by differing measurement values.

In order to prevent attested systems from unnoticed cheating, we have integrated functions of the TPM into the measurement architecture:

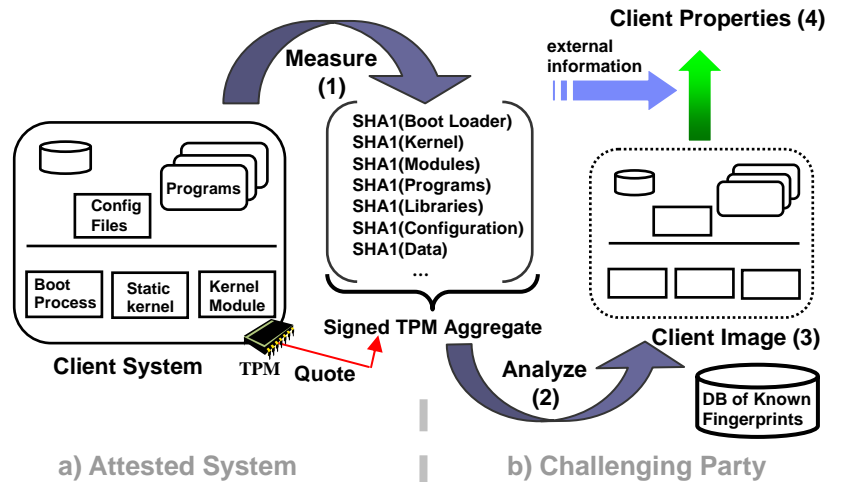


Figure 1: Attestation Architecture Overview

- We use the TPM to maintain an integrity value (stored in its protected hardware) over the current measurement list that is kept in the kernel. This protects the measurement list from being manipulated unnoticedly even if the system and kernel become corrupted.
- We create measurements of files before they are loaded and potentially affect the system. Thus, once loaded data can corrupt the measurement list, it is too late to cover its own traces from the TPM.

Our TPM (Version 1.1) offers 16 platform configuration registers (PCR) that allow extending 160 bit numbers (length of a SHA1 value) into them. These PCR are reset to 0 whenever the system is reset (e.g., reboot). The first 8 PCRs (PCR0 – PCR7) are used for attesting the booting steps, the remaining 8 PCRs (PCR8 – PCR15) are allocated for use by the booted system [9]. We use a configurable PCR number greater than 7 (e.g., 10) to maintain an integrity value over the current measurement list after system boot. If a new measurement is added to the measurement list, we also write its 160bit measurement value into TPM PCR 10. The TPM computes the new register content by building a SHA1 over the current content concatenated with the new 160bit number written into the PCR. The cryptographic properties of SHA1 (being collision-free) guarantee protection against the adaptation of a TPM PCR to match a manipulated measurement list by compromised systems later on.

We refer the interested reader to [2] for further details of the integrity measurement

architecture and its implementation. Figure 2 shows a partial snapshot of a measurement list for a Redhat Linux system including executables, shared libraries, kernel modules, bash command files (e.g., server initialization scripts) and bash source files (e.g., bash configuration files). We include some additional information in our kernel-held measurement list, such as the file name of the measured file. Our Web-based project description [5] includes a complete measurement list including measurements collected during system boot.

#	SHA1(160bit)	File	Type
000:	D6DC...D3DB	n/a	[boot aggregate]
001:	84AB...DA4F	init	[exec]
002:	9ECF...BE3D	ld-2.3.2.so	[library]
003:	3365...2342	libc-2.3.2.so	[library]
004:	A4DC...C12B	bash	[exec]
...			
027:	2AC8...980D	clock	[bash-src]
028:	C0F7...9A3D	hwclock	[exec]
...			
070:	01B3...9A1E	rc	[bash-cmd]
071:	CEBA...1AA4	runlevel	[exec]
072:	2998...8ED4	egrep	[bash-cmd]
073:	6846...B72D	kudzu	[bash-cmd]
...			
080:	147D...8168	parport	[module]
081:	F940...0115	parport_pc	[module]
...			
244:	D312...DA7C	rc.local	[bash-cmd]
245:	BB2C...AAB3	mingetty	[exec]

Figure 2: Measurement List Example

The measurement list is always initialized with the boot aggregate representing the measurements of the boot stages up to and including the running kernel. The actual measurements – aggregated into the boot aggregate – are stored in the BIOS as the kernel is not yet running. They are protected

