IBM Research Report

Design and Implementation of a TCG-Based Integrity Measurement Architecture

Reiner Sailer, Xiaolan Zhang, Trent Jaeger, Leendert Van Doorn IBM Research Division Thomas J. Watson Research Center P.O. Box 704 Yorktown Heights, NY 10598



Research Division Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publication, its distributionoutside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. payment of royalties). Copies may be requested from IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

Design and Implementation of a TCG-based Integrity Measurement Architecture

Reiner Sailer and Xiaolan Zhang and Trent Jaeger and Leendert van Doorn IBM T. J. Watson Research Center 19 Skyline Drive, Hawthorne, NY 10532 {sailer,cxzhang,jaegert,leendert}@watson.ibm.com

Abstract

We present the design and implementation of a secure integrity measurement system for Linux. All executable content that is loaded onto the Linux system is measured before execution and these measurements are protected by the Trusted Platform Module (TPM) that is part of the Trusted Computing Group (TCG) standards. Our system is the first to extend the TCG trust concepts to dynamic executable content from the BIOS all the way up into the application layer. In effect, we show that many of the Microsoft NGSCB guarantees can be obtained on today's hardware and today's software and that these guarantees do not require a new CPU mode or operating system but merely depend on the availability of an independent trusted entity, a TPM for example. We apply the measurement architecture to a web server application where we show how our system can detect undesirable invocations, such as rootkit programs, and that measurement is practical in terms of the number of measurements taken and the performance impact of making them.

1 Introduction

With the introduction of autonomic computing, grid computing and on demand computing there is an increasing need to be able to securely identify the software stack that is running on remote systems. For autonomic computing, you want to determine that the correct patches have been installed on a given system. For grid computing, you are concerned that the services advertised really exist and that the system is not compromised. For on demand computing, you may be concerned that your outsourcing partner is providing the software facilities and performance that have been stipulated in the service level agreement. Yet another scenario is where you are interacting with your home banking or bookselling webservices application and you want to make sure it has not been tampered with.

The problem with the scenarios above is, who do you trust to give you that answer? It cannot be the program itself because is could be modified to give you wrong answers. For the same reason we cannot trust the kernel or the BIOS on which these programs are running since they may be tampered with too. Instead we need to go back to an immutable root to provide that answer. This is essentially the secure boot problem [1], although for our scenarios we are interested in an integrity statement of the software stack rather than ensuring compliance with respect to a digital signature.

The Trusted Computing Group (TCG) has defined a set of standards [2] that describe how to take integrity measurements of a system and store the result in a separate trusted coprocessor (Trusted Platform Module) whose state cannot be compromised by a potentially malicious host system. This mechanism is called trusted boot. Unlike secure boot, this system only takes measurements and leaves it up to the remote party to determine the system's trustworthiness. The way this works is that 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.

The TCG trusted boot process is composed of a set of ordered sequential steps and is only defined up to the bootstrap loader. Conceptually, we would like to maintain the chain of trust measurements up to the application layer, but unlike the bootstrap process, an operating system handles a large variety of executable content (kernel, kernel modules, binaries. shared libraries, scripts, plugins, etc.) and the order in which the content is loaded is seemingly random. Furthermore, an operating system almost continuously loads executable content and measuring the content at each load time incurs a considerable performance overhead.

The system that we describe in this paper addresses these concerns. We have modified the Linux kernel and the runtime system to take integrity measurements as soon as executable content is loaded into the system, but before it is executed. We keep an ordered list of measurements inside the kernel. We change the role of the TPM slightly and use it to protect the integrity of the in-kernel 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 tampered with and, once the list is validated, what kind of trust it associates with the measurements. To minimize the performance overhead, we cache the measurement results and eliminate future measurement computations as long as the executable content has not been altered. The amount of modifications we made to the Linux system were minimal, about 1000 lines of code.

Our enhancement keeps track of all the software components that are executed by a system. The number of unique components is surprisingly small and the system quickly settles into a steady state. For example, the workstation used by this author which runs RedHat 9 and whose workload consists of writing this paper, compile programs and browse the web does not accumulate more than 500 measurement entries. On a typical web server the accumulated measurements are about 250. Thus, the notion of completely fingerprinting the running software stack is surprisingly tractable.

Contributions: This paper makes the following contributions:

- A non-intrusive and verifiable remote software stack attestation mechanism that uses standard (commodity) hardware.
- An efficient measurement system for dynamic executable content.
- A tractable software stack attestation mecha-

nism that does not require new CPU modes or a new operating system.

Outline: Next, we introduce the structure of a typical run-time system, for which we will establish an integrity-measurement architecture throughout this paper. In Section 3, we present related work in the area of integrity protecting systems and attestation. In Sections 4 and 5, we describe the design of our approach and its implementation in a standard Linux operating environment. Section 6 describes experiments that highlight how integrity breaches are made visible by our solution when validating measurement-lists. It also summarizes runtime overhead. Finally, Section 7 sketches enhancements to our architecture that are being implemented or planned. Our results show and validate that our architecture is efficient, scales with regard to the number of elements, successfully recognizes integrity breaches, and offers a valuable platform for extensions and future experiments.

2 Problem Statement

To provide integrity verification services, we first examine the meaning of system integrity, in general. We then describe a web server example system to identify the types of problems that must be solved to prove integrity to a remote system with a high degree of confidence. We show that the operating system lacks the context to provide the level of integrity measurement necessary, but with a hardware root of trust, the operating system can be a foundation of integrity measurement. Currently, we surmise that it is more appropriate for finding integrity bugs than full verification, but we aim to define an architecture that can eventually be extended to meet our measurement requirements.

2.1 Integrity Background

Our goal is to enable a remote system (the *challenger*) to prove that a program on another system (the *attesting system* owned by the *attestor*) is of sufficient integrity to use. The *integrity* of a program is a binary property that indicates whether the program and/or its environment have been modified in an unauthorized manner. Such an unauthorized modification may result in incorrect or malicious behavior by the program, such that it would be unwise for a challenger to rely on it.

While integrity is a binary property, integrity is a relative property that depends on the verifier's view of the ability of a program to protect itself. Biba defines that integrity is compromised when a program depends on (i.e., reads or executes) low integrity data [3]. In practice, programs often process low integrity data without being compromised (but not all programs, all the time), so this definition is too restricted. Clark-Wilson define a model in which integrity verification procedures verify integrity at system startup and high integrity data is only modified by *transformation procedures* that are certified to maintain integrity even when their inputs include low integrity data [4]. Unfortunately, the certification of applications is too expensive to be practical.

More recent efforts focus on measuring code and associating integrity semantics with the code. The IBM 4758 explicitly defines that the integrity of a program is determined by the code of the program and its ancestors [5]. In practice, this assumption is practical because the program and its configuration are installed in a trusted manner, it is isolated from using files that can be modified by other programs, and it is assumed to be capable of handling low integrity requests from the external system. To make this guarantee plausible, the IBM 4758 environment is restricted to a single program with a welldefined input state and the integrity is enforced with secure boot. However, even these assumptions have not been sufficient to prevent compromise of applications running on the 4758 which cannot handle low integrity inputs properly [6]. Thus, further measurement of low integrity inputs and their impact appear to be likely.

The key differences in this paper are that: (1) we endeavor to define practical integrity for a flexible, traditional systems environment under the control of a potentially untrusted party and (2) the only special hardware that we leverage is the root of trust provided by the Trusted Computing Group's Trusted Platform Module (TCG/TPM). In the first case, we may not assume that all programs are loaded correctly simply by examining the hash because the untrusted party may try to change the input data that the program uses. For example, many programs enable configuration files to be specified in the command line. Ultimately, applications define the semantics of the inputs that they use, so it is difficult for an operating system to detect whether all inputs have been used in an appropriate manner by an application if its environment is controlled by an untrusted party. However, a number of vulnerabilities can be found by the operating system alone, and it is fundamental that the operating system collect and protect measurements.

Second, the specialized hardware environment of the IBM 4758 enables secure boot and memory lockdown, but such features are either not available or not practical for current PC systems. Secure boot is not practical because integrity requirements are not fixed, but defined by the remote challengers. If remote parties could determine the secure boot properties of a system, systems would be vulnerable to a significant denial-of-service threat. Instead the TCG/TPM supports trusted boot, where the attesting system is measured and the measurements are used by the challengers to verify their integrity requirements. Since trusted boot does not terminate a boot when a low integrity process is loaded, all data could be subject to attack during the "untrusted" boot. Since multiple applications can run in a discretionary access control environment currently, it is difficult to determine whether the dynamic data of a system (e.g., a database) is still acceptable. Discretionary integrity mechanisms, such as sealed storage [7], do not solve this problem in general.

2.2 Example

We use as an example a server machine running an Apache Webserver and Tomcat Web Containers that serve static and dynamic content to sell books to clients running on remote systems. The system is running a RedHat 9.0 Linux environment. Figure 1 illustrates the runtime environment that affects the Web server.



Figure 1: Runtime System Components

The system is initiated by booting the operating system. The boot process is determined by the BIOS, grub bootloader, and kernel configuration file (/boot/grub.conf). The first two can alter the system in arbitrary ways, so they must be measured. An interesting point is that measurement of configuration files, such as grub.conf, is not necessary as long as they do not: (1) modify code already loaded and (2) all subsequent file loads can be seen by the measurement infrastructure. Since the BIOS and grub bootloader are unaffected, we only need to ensure that the kernel and other programs whose loads are triggered by the configuration are measured.

The boot process results in a particular kernel being run. There are a variety of different types of kernels, kernel versions, and kernel configurations that determine the actual system being booted. For example, we load Linux 2.4.21 from /boot/vmlinuz-2.4.21-tcg which includes a TCPA driver and our measurement hooks. Further, the kernel may be extended by loadable kernel modules. The measurement infrastructure must be able to measure the kernel and any modules that are loaded. The challenger must be able to determine whether this specific kernel booted and the dynamically loaded modules meet the desired integrity requirements.

Once the kernel is booted, then user-level services and applications may be run. In Linux, a program execution starts by loading an appropriate interpreter (i.e., a dynamic loader, such as 1d.so) based on the format of the executable file. Loads of the target executable's code and supporting libraries are done by the dynamic loader. Executables include the following files on our experimental system:

- Apache server (apachectl, httpd, ...)
- Apache modules (mod_access.so, mod_auth.so, mod_cgi.so, ...)
- Tomcat servlet machine (startup.sh, catalina.sh, java, ...)
- Dynamic libraries (libjvm.so, libcore.so, libjava.so, libc-2.3.2.so, libssl.so.4, ...)

All of this code impacts system integrity, so we need to measure them, but the operating system does not see a distinction between these files and typical data files (i.e., they are all opened and read). An advantage is that the dynamic loader (for that file format) understands that these files are being loaded as executable code. Dynamic loaders are trusted by the system, typically, so they trigger the measurement of library files.

Some other files loaded by the application itself also define its execution behavior. For example, the Java class files that define servlets and web services must be measured because they are loaded by the Tomcat server to create dynamic content, such as shopping cart or payment pages. Application configuration files, such as the startup files for Apache (httpd.conf) and Tomcat (startup scripts) may also alter the behavior of the Web server. These files in our example system include:

- Apache configuration file (httpd.conf)
- Java virtual machine security configuration (java.security, java.policy)
- Servlets and web services libraries (axis.jar, servlet.jar, wsdl4j.jar, ...)

While each of these files may have standard contents that can be identified by the challenger, it is difficult to determine which files are actually being used by an application and for what purpose. Even if http.conf has the expected contents, it may not be loaded as expected. For example, Apache has a command line option to load a different file, links in the file system may result in a different file being loaded, and races are possible between when the file is measured and when it is loaded. Thus, a Tripwire-like [8] measurement of the key system files is not sufficient because the users of the attesting system (attestors) may change the files that actually determine its integrity, and these users are not necessarily trusted by the challengers. As in the dynamic loader case, the integrity impact of opening a file is only known to the requesting program. However, unlike the case for the dynamic loader, the problem of determining the integrity impact of application loads involves instrumentation of many more programs, and these may be of varying trust levels.

The integrity of the Web server environment also depends on dynamic, unstructured data that is consumed by running executables. The key issue is that even if the application knows that this data can impact its integrity, its measurement is useless because the challenger cannot predict values that would preserve integrity. In the web server example, the key dynamic data are: (1) the various kinds of requests from remote clients, administrators, and other servlets and (2) the database of book orders. The sorts of things that need to be determined are whether the order data or administrator commands can be modified only by high integrity programs (i.e., Biba) and whether the low integrity requests can be converted to high integrity data or rejected (i.e., Clark-Wilson). Sealed storage is insufficient to ensure the first property, information flow based on mandatory policy is necessary in general, and enforcement of the second property requires trusted upgraders or trust in the application itself.

2.3 Measuring Systems

Based on the analysis of the web server example, we list the types of tasks that must be accomplished to achieve a Clark-Wilson level of integrity verification.

- Verification Scope: Unless information flows among processes are under a mandatory restriction, the integrity of all processes must be measured. Otherwise, the scope of integrity impacting a process may be reduced to only those processes upon which it depends for high integrity code and data.
- Executable Content: For each process, all code executed must be of sufficient integrity regardless of whether it is loaded by the operating system, dynamic loader, or application.
- Structured Data: For each process, data whose content has an identifiable integrity semantics may be treated in the same manner as executable content above. However, we must be sure to capture the data that is actually loaded by the operating system, dynamic loaders, and applications.
- Unstructured Data: For each process, the data whose content does not have an identifiable integrity semantics, the integrity of the data is dependent on the integrity of the processes that have modified it or the integrity may be upgraded by explicit upgrade processes or this process (if it is qualified to be a transformation procedure in the Clark-Wilson sense).

The first statement indicates that for systems that use discretionary policy (e.g., NGSCB), the integrity of all processes must be measured because all can impact each other. Second, we must measure all code including modules, libraries, and code loaded in an ad hoc fashion by applications to verify the integrity of an individual process. Third, some data may have integrity semantics similar to code, such that it may be treated that way. Fourth, dynamic data cannot be verified as code, so data history, security policy, etc. are necessary to determine its integrity. The challengers may assume that some code can handle low integrity data as input. The lack of correct understanding about particular code's ability to handle low integrity data is the source of many current security problems, so we would ultimately prefer a clear identification of how low integrity data is used.

Further, an essential part of our architecture is the ability of challengers to ensure that the measurement list is:

- fresh and complete, i.e., includes all measurements up to the point in time when the attestation is executed,
- unchanged, i.e., the fingerprints are truly from the loaded executable and static data files and have not been tampered with.

An attestor that has been corrupted can try to cheat by either truncating measurements or delivering changed measurements to hide the programs that have corrupted its state. Replaying old measurement lists is equivalent to hiding new measurements.

This analysis indicates that integrity verification for a flexible systems environment is a difficult problem that requires several coordinated tasks. Rather than tackle all problems at once, a more practical approach is to provide an extensible approach that can identify some integrity bugs now and form a basis for constructing reasonable integrity verification in the future. This approach is motivated by the approach adopted by static analysis researchers in recent work [9]. Rather than proving the integrity of a program, these tools are design to find bugs and be extensible to finding other, more complex bugs in the future. Finding integrity bugs is also useful for identifying that code needs to be patched, illegal information flows, or cases where low integrity data is used without proper safeguards. For example, a challenger can verify that an attesting system is using high integrity code for its current applications.

In this paper, we define operating systems support for measuring the integrity of code and structured data. The operating system ensures that the code loaded into every individual user-level process is measured, and this is used as a basis for dynamic loaders and applications to measure other code and data for which integrity semantics may be defined. Thus, our architecture ensures that the breadth of the system is measured (i.e., all user-level processes), but the depth of measurement (i.e., which things are subsequently loaded into the processes) is not complete, but it is extensible, such that further measurements to increase confidence in integrity are possible. At present, we do not measure mandatory access control policy, but the architecture supports extensions to include such measurements and we are working on how to effectively use them.

3 Related Work

Related work includes previous efforts to measure a system to improve its integrity and/or enable remote integrity verification. The key issues in prior work are: (1) the distinction between *secure boot* and *authenticated boot* and (2) the semantic value of previous integrity measurement approaches.

Secure boot enables a system to measure its own integrity and terminate the boot process if an action compromises this integrity. The AEGIS system by Arbaugh [1] provides a practical architecture for implementing secure boot on a PC system. It uses signed hash values to identify and validate each layer in the boot process. It will abort booting the system if the hashes cannot be validated. Secure boot does not enable a challenging party to verify the integrity of a boot process (i.e., authenticated boot) because it simply measures and checks the boot process, but does not generate attestations of the integrity of the process.

The IBM 4758 secure coprocessor [10] implements both secure boot and authenticated boot, albeit in a restricted environment. It promises secure boot guarantees by verifying (flash) partitions before activating them and by enforcing valid signatures before loading executables into the system. A mechanism called *outgoing authentication* [5] enables attestation that links each subsequent layer to its predecessor. The predecessor attests to the subsequent layer by generating a signed message that includes the cryptographic hash and the public key of the subsequent layer. To protect an application from flaws in other applications, only one application is allowed to run at a time. Thus, the integrity of the application depends on hashes of the code and manual verification of the application's installation data. This data is only accessible to trusted code after installation. Our web server example runs in a much more dynamic environment where multiple processes may access the same data and may interact. Further, the security requirements of the challenging party and the attesting party may differ such that secure boot based on the challenging party's requirements is impractical.

The Trusted Computing Group [11] is a consortium of companies that together have developed an open interface for a Trusted Platform Module, a hardware extension to systems that provides cryptographic functionality and protected storage. By default, the TPM enables the verification of static platform configurations, both in terms of content and order, by collecting a sequence of hashes over target code. For example, researchers have examined how a TPM can be used to prove that a system has booted a valid operating system [12]. The integrity of applications running on the operating system is outside the scope of this work and is exactly where we look to expand the application of the TPM.

Terra [13] and Microsoft's Next Generation Secure Computing Base (NGSCB [7]) are based on the same hardware security architecture (TCG/TPM) and are similar in providing a "whole system solution" to authenticated boot. NGSCB partitions a platform into a trusted and untrusted part each of which runs its own operating system. Only the trusted portion is measured which limits the flexibility of the approach (not all programs of interest should be fully trusted) and it depends on hardware and base software not yet available.

Terra is a trusted computing architecture that is built around a trusted virtual machine monitor that -among other things- authenticates the software running in a VM for challenging parties. Terra tries to resolve the conflict between building trusted customized closed-box run-time environments (e.g., IBM 4758) and open systems that offer rich functionality and significant economies of scale that, however, are difficult to trust because of their flexibility. As such, Terra tries to solve the same problem as we do, however in a very different way. Terra measures the trusted virtual machine monitor on the partition block level. Thus, on the one hand, Terra produces about 20 Megabyte of measurement values (i.e., hashes) when attesting an exemplary 4

Gigabyte VM partition. On the other hand, because those measurements are representative of blocks, it is difficult to interpret varying measurement values. Thus, our system measures selectively those parts of the system that contribute to the dynamic runtime system; it does so on a high level that is rich in semantics and enables remote parties to interpret varying measurements on a file level.

4 Design of an Integrity Measurement Architecture

Our integrity Measurement architecture consists of three major components:

- The *Measurement Mechanism* on the attested system determines what parts of the run-time environment to measure, when to measure, and how to securely maintain the measurements.
- An *Integrity Challenge Mechanism* that allows authorized challengers to retrieve measurement lists of a computing platform and verify their freshness and completeness.
- An *Integrity Validation Mechanism*, validating that the measurement list is complete, non-tampered, and fresh as well as validating that all individual measurement entries of runtime components describe trustworthy code or configuration files.

Figure 2 shows how these mechanisms interact to enable remote attestation. Measurements are initiated by so-called measurement agents, which induce a measurement of a file, (a) store the measurement in an ordered list in the kernel, and (b) report the extension of the measurement list to the TPM.



Figure 2: TPM-based Integrity Measurement

The integrity challenge mechanism allows remote challenger to request the measurement list together with the TPM-signed aggregate of the measurement list (step 1 in Fig 2). Receiving such a challenge, the attesting system first retrieves the signed aggregate from the TPM (steps 2 and 3 in Fig 2) and afterwards the measurement list from the kernel (step 4 in Fig 2). Both are then returned to the attesting party in step 5. Finally, the attesting party can validate the information and reason about the trustworthiness of the attesting system's run-time integrity in step 6.

4.1 Assumptions

Before we describe these three components of our architecture, we establish assumptions about the attacker model because without such restrictions, there would always be attackers that are able to fool a remote client.

We use services and protection offered by Trusted Computing Project Architecture the (TCPA [11]) in order to: (1) enable challenging parties to establish trust into the platform configuration of the attesting system (measurement environment) and (2) ensure challengers that the measurement list compiled by the measurement environment has not been tampered with. We assume that the TPM hardware works according to the TPM specifications [11] and that the TPM is embedded correctly into the platform, ensuring the proper measurement of the BIOS, bootloader, and following system environment parts.

The TPM cannot prevent direct hardware attacks against the system, so we assume that these are not part of the threat model.

We assume that code measurements are sufficient to describe its behavior. Thus, self-changing code can be evaluated because the intended ability of code to change itself is reflected in the measurement and can be taken into account in verification. The same holds for the kernel code that is thought to be changed only through loading and unloading modules. Kernel changes based on malicious DMA transfers overwriting kernel code are not addressed; however, the code setting up the DMA is measured and thus subject to evaluation.

We also assume that the challenging party holds a valid and trusted certificate binding a public RSA

identity key AIK_{pub} of the attesting system's TPM. AIK_{pub} will be used by the challenging party to validate the quoted register contents of the attesting system's TPM before using those registers to validate the measurement list.

We assume that there are no confidentiality requirements on measurement data that cannot be satisfied by controlling the access to the attestation service.

Finally, for the interpretation of system integrity measurements, we rely on the challenger's run-time because the validation results must be securely computed, interpreted, and acted upon. We assume that the challenger can safely decide which measurements to trust either by comparing them to a list of trusted measurements or by off-loading the decision to trusted parties that sign trusted measurements according to a common policy (i.e., common evaluation criteria).

4.2 Measurement Mechanism

Our measurements mechanism consists of a base measurement when a new executable is loaded and the ability to measure other executable content and sensitive data files. The idea is that BIOS and bootloader measure the initial kernel code and then enable the kernel to measure changes to itself (e.g., module loads) and the creation of user-level processes. The kernel uses the same approach with respect to user-level processes, where it measures the executable code loaded into processes (e.g., dynamic loader and httpd loaded via execve). Then, this code can measure subsequent security sensitive inputs it loads (e.g., libraries by the dynamic loader and scripts by httpd). The challenger's trust is dependent on its trust in the measured code to measure its security sensitive inputs, protect itself from unmeasured inputs, and protect data it is dependent upon across reboots. The operating system can provide further protection of applications through mandatory access control policy which can limit the sources of malicious, unmeasured inputs and protect data from modification. However, the use of such policy is future work.

In this section, we discuss how measurements are made. The application of these measurements to a complete measurement system is described in Section 5.

To uniquely identify any particular executable con-

tent, we compute a SHA1 hash over the complete contents of the file. The resulting 160bit hash value unambiguously identifies the file's contents. Different file types, versions, and extensions can be distinguished by their unique fingerprints.

The individual hashes are collected into a *measurement list* that represents the integrity history of the attesting system. Modifications to the measurement list are not permissible as that would enable an attacker to hide integrity-relevant actions. As our architecture is non-intrusive, it does not prevent systems from being corrupted and then tamper with the measurement list. However, to prevent such malicious behavior from going unnoticed (preventing corrupted systems from cheating), we use a hardware extension on the attesting system, known as Trusted Platform Module, to make modifications of the measurement list visible to challenging parties.

The Trusted Computing Group's (TCG [11]) Trusted Platform Module (TPM) provides some protected data registers, called Platform Configuration Registers, which can be changed only by two functions: The first function is rebooting the platform, which clears all PCRs (value 0). The second function is the TPM_extend function, which takes one 160bit number n and the number i of a PCR register as arguments and then aggregates n and the current contents of PCR[i] by computing a SHA1(PCR $[i] \parallel$ n). This new value is stored in PCR[i]. There is no other way for the system to change the value of any PCR register, based on our assumptions that the TPM hardware behaves according to the TCG specification and no direct physical attacks occur.

We use the Platform Configuration Registers to maintain an integrity verification value over all measurements taken by our architecture. Any measurement that is taken is also aggregated into a TPM PCR (using TPM_extend) before the measured component can affect and potentially corrupt the system. Thus, any measured software is recorded before taking control directly (executable) or indirectly (static data file of the configuration). For example, if *i* measurements $m_1..m_i$ have been taken, the aggregate in the chosen PCR contains $SHA1(..SHA1(SHA1(0||m_1)||m_2)..||m_i)$. The protected storage of the TPM prevents modification by devices or system software. While it can be extended with other chosen values by a corrupted system, the way that the extension is computed (properties of SHA1) prevents a malicious system from adjusting the aggregate in the PCR to represent a prescribed

system. Once a malicious component gains control, it is too late to hide this component's existence and fingerprint from attesting parties.

Thus, corrupted systems can manipulate the measurement list, but this is detected by re-computing the aggregate of the list and comparing it with the aggregate stored securely inside the TPM.

4.3 Integrity Challenge Mechanism

The Integrity Challenge protocol describes how challenging parties securely retrieve measurements and validation information from the attesting system. The protocol must protect against the following major threats when retrieving attestation information:

- Replay attacks: a malicious attesting system can replay attestation information (measurement list + TPM aggregate) from before the system was corrupted.
- Tampering: a malicious attesting system or intermediate attacker can tamper with the measurement list and TPM aggregate before or when it is transmitted to the challenging party.
- Masquerading: a malicious attesting system or intermediate attacker can replace the original measurement list and TPM aggregate with the measurement list and TPM aggregate of another (non-compromised) system.

Fig. 3 depicts the integrity challenge protocol used by the challenging party C to securely validate integrity claims of the attesting system AS. In steps 1 and 2, C creates a non-predictable 160bit random *nonce* and sends it in a challenge request message ChReq to AS. In step 3, the attesting system loads a protected RSA key AIK into the TPM. This AIKis encrypted with the so-called Storage Root Key (SRK), a key known only to the TPM. The TPM specification [11] describes, how a 2048bit AIK is created securely inside the TPM and how the corresponding public key AIK_{pub} can be securely certified by a trusted party. This trusted party certificate links the signature of the PCR to a specific TPM chip in a specific system. Then, the AS requests a Quote from the TPM chip that now signs the selected PCR (or multiple PCRs) and the *nonce* originally provided by C with the private key AIK_{priv} . To complete step 3, the AS retrieves the ordered list of all measurements (in our case from the kernel). Then, AS responds with a challenge response message ChRes in step 4, including the signed aggregate and nonce in *Quote*, together with the claimed complete measurement list ML.

- 1. C: create non-predictable 160bit nonce
- 2. $C \rightarrow AS$: ChReq(nonce)
- 3a. AS: load protected AIK_{priv} into TPM
- 3b. AS : retrieve $Quote = sig\{PCR, nonce\}_{AIK_{priv}}$
- 3c. AS : retrieve Measurement List ML
- 4. $AS \rightarrow C$: ChRes(Quote, ML)
- 5a. C: determine trusted $cert(AIK_{pub})$
- 5b. C: validate $sig\{PCR, nonce\}_{AIK_{priv}}$ 5c. C: validate nonce and ML using PCR

Figure 3: Integrity Challenge Protocol

In step 5a, C first retrieves a trusted certificate $cert(AIK_{pub})$. This AIK certificate binds the verification key AIK_{pub} of the QUOTE to a specific system and states that the related secret key is known only to this TPM and never exported unprotected. Thus masquerading can be discovered by the challenging party by comparing the unique identification of AS with the system identification given in $cert(AIK_{pub})$. This certificate must be verified to be valid, e.g., by checking the certificate revocation list at the trusted issuing party. C then verifies the signature in step 5b.

In step 5c, C validates the *freshness* of the QUOTE and thus the freshness of the PCR (the measurement aggregate). Freshness is guaranteed if the nonces match as long the *nonce* in step 2 is unique and not predictable. As soon as AS receives a nonce twice or can predict the nonce (or predict even a small enough set into which the nonce will fall), it can decide to replay old measurements or request TPM-signed quotes early using predicted nonces. In both cases, the quoted integrity measurements ML might not reflect the actual system status, but a past one. If the nonce offers insufficient security, then the validity of the signature keys can be restricted, because the replay window for signed aggregates is also bound to using a valid signature key.

Validating the signature in step 5b, C can detect *tampering* with the TPM aggregate, because it will invalidate the signature (assuming cryptographic properties of a digital 2048bit-signature today, assuming the secret key is known only to the TPM, and assuming no hardware tampering of the TPM).

Tampering with the measurement list is made visible in step 5c by walking through the measurement list ML and re-computing the TPM aggregate (simulating the TPM extend operations as described in Section 4.2) and comparing the result with the TPM aggregate PCR that is included in the signed Quote received in step 4. If the computed aggregate matches the signed aggregate, then the measurement list is valid and untampered, otherwise it is invalid.

4.4 Integrity Validation Mechanism

The challenging party must validate the individual measurements of the attesting party's platform configuration and the dynamic measurements that have taken place on the attesting system since it has been rebooted. The aggregate for the configuration and the measurement list has already been validated throughout the integrity challenge protocol and is assumed here. The same holds for the validity of the TPM aggregate.

Concluding whether to trust or distrust an attesting system is based on testing each measurement list entry independently, comparing its measurement value with a list of trusted measurement values. More sophisticated validation models can relate multiple measurements to reach an evaluation result. Testing measurement entries is logically the same regardless of whether the entry is code or data. The idea is that the entry matches some predefined value that has known integrity semantics. Unknown fingerprints can result from new program versions, unknown programs, or otherwise manipulated code. As such, fingerprints of program updates can be measured by the challenging party and added to the database; in turn, old program versions with known vulnerabilities [14] might be reclassified to distrusted.

The challenging party must have a policy in place that states how to classify the fingerprints and how to proceed with unknown or distrusted fingerprints. Usually, a distrusted fingerprint leads to distrusting the integrity of the whole attesting system if no additional policy enforcement mechanisms guarantee isolation of the distrusted executable. Alternatively, trustworthy fingerprints can be signed by trusted third parties, e.g., regarding their suitability to enforce certain security targets (Common Criteria Evaluation) related to their purpose.

Transaction Integrity Usually, the integrity of the attesting system is of interest when it processes a

transaction that is important to a challenging party. To verify the integrity of a transaction that is taking place between the challenging and the attesting party (e.g., a Web request), the challenging party can challenge the integrity of the attesting system before and after the transaction was processed, e.g., before sending the Web request and after receiving the Web response. If the attesting system is trusted both times, then– so it seems –the transaction can be trusted, too.

This is, however, not entirely true because it assumes that both measurements have taken place in the same epoch (validity period), i.e., that any system change throughout the transaction would have been recorded in the second measurement. However, the attesting system could have been compromised just after the first challenge and before the transaction took place. Then, the attesting system could have rebooted before the second challenge took place. Thus, though trusted at two points in time, the reboot covered the distrusted attesting system state against the challenger. Even if the possibility seems small, systems can reboot very fast and actually come up into an exactly pre-defined state (thus exhibiting the same measurement list as in earlier measurements) 1 .

Fortunately, there is a way to discover if an epoch changes, i.e., whether the system rebooted between two attestations. For this purpose, we can use socalled TPM counters. As opposed to the PCRs, these counters are never cleared or decreased but can only increase throughout the lifetime of a TPM. Increases of one of these counters could be triggered by the BIOS each time the system reboots. The BIOS is also responsible to disable the TPM as soon as the counter has reached its maximum value. Typical TPM have multiple counters that can be combined and thus are sufficient for normal platform lifetimes². Thus, a trusted kernel including such a counter into the measurement list ensures that the prefixes of two measurement lists differ at least in this single counter measurement once the system is rebooted.

Consequently, in this enhanced version, transaction integrity can be validated by ensuring that the measurement list validated at the first challenge before

¹This is used in another TPM mechanism allowing to seal a secret to a platform configuration, though originally this did not include any dynamic measurements.

 $^{^{2}}$ The TPM specification [11] demands that the externally accessible counters must allow for 7 years of increments every 5 seconds without causing a hardware failure.

the transaction is a prefix of the measurement list validated at the second challenge after the transaction. Then, the system did not reboot and thus (given our assumptions) any distrusted system component potentially impacting the transaction on the attesting system, would show in the measurement list of the second challenge. In effect, our architecture does not offer predictable security as long as it is non-intrusive, yet it can offer retrospective assurance of the integrity state of a system.

5 Implementation

This section describes the enhancements we made to the Linux system to implement the measurement functionalities. Before any of our dynamic measurements are initiated (i.e., before linuxrc or init are started), our kernel pre-loads its measurement list with the expected measurements for BIOS, bootloader, and kernel and uses the aggregate of the real boot process, found in a pre-defined TPM PCR, as the starting point for our own measurement aggregate. If the actual boot process differs from the expected one, the validation of the measurement list will fail. We focus on the stages following the initial OS boot and refer interested readers to another paper [12] that covers detailed measurement steps before booting as well as their validation.

Our prototype implementation is done on a RedHat 9.0 Linux distribution running a 2.4.21 kernel ³. The prototype implementation is divided into three major components: inserting measurement points into the system that measures a given file (Section 5.1), the actual measurement (Section 5.2), and how to validate the measurements to ensure that an implementation of our architecture is actually in place on the attesting system (Section 5.3).

5.1 Inserting Measurement Points

The actual measurement is performed in the kernel and is exported to user-level applications via a new measure system call. The main task is thus to identify points in the system (be it user or kernel level) where execution-related contents are loaded, and insert a measure system call (or a direct call to the measurement code when in kernel) at those places. In Section 4.2, we outline the approach to measurement, where the load of measured code gives the opportunity to more fine-grained measurement to that code. If that code is not of high integrity, it will be detected (because it is already in the measurement list). If it is of high-integrity, then it may be trusted to measure its loads. As we discussed in Section 2, the operating system call interface lacks the semantic richness to express the integrity impact of reading a file.

At present, we have implemented the following measurement points discussed below. We have not yet explored adding measurements within an application. The fundamental mechanism would be the same, but we anticipate the need for more support for application programmers.

Kernel Modules: Kernel modules are extensions to the kernel that can be dynamically loaded after the system is booted. The basic tool for loading a kernel module is *insmod*. *insmod* functions as a relocating linker/loader – it resolves external references in the kernel module and makes the module known to the kernel. One can invoke insmod directly as a regular command to explicitly load a module. The kernel can also *implicitly* load a module via the *modprobe* mechanism if automatic module loading is enabled. In the latter case, when the kernel detects a module is needed, it automatically finds and loads the appropriate module by invoking *modprobe* in the context of a user process. Typically, modprobe is just a symbolic link to insmod. To measure kernel modules, we instrument the *insmod* program with a measure system call at the point between the successful opening of the object file containing the kernel module and the loading of the kernel module.

User-level Executables: User-level executables are loaded through the user-level *loader*. When a binary executable is invoked via the system call execve, the kernel calls the binary handler routine, which then interprets the binary and locates the appropriate loader for the executable. The kernel then maps the loader into the memory and sets up the environment such that when the execve system call returns, execution resumes with the loader. The loader in turn performs normal loading functionalities and finally passes control to the main function of the target executable. In the case of a statically linked binary, the only file being loaded is the target binary itself. To measure such binaries, we added a call to the measurement routine in the execve system call. The case of a dynamically linked binary is

³The mechanisms presented here are sufficiently generic that porting to a Unix-like system should be straightforward.

more complicated and is elaborated below.

Dynamically Loadable Libraries: A dynamically linked binary typically requires loading of additional libraries that it depends on. This process is done by the user-level loader and is transparent to the kernel. Thus, to measure these dynamically loaded libraries, we instrument the loader with a measure system call after each successful loading of a dynamic library and before it can be used.

Scripts: Scripts are loaded through the execve system call exactly like a binary executable, except that the loader in this case is the interpreter of the particular script language. For example, if the script language is PERL, then the interpreter */bin/perl* will be invoked.

Script interpreters may load additional code, so we would prefer that the script interpreters also be capable of measuring their integrity-relevant input. At present, we have not instrumented them yet, so we restrict the scripts that we trust.

5.2 Taking Measurements

This section describes the implementation of the **measure** system call used at the measurement points to initiate the measurement of a file. The **measure** system call takes one argument, namely, the descriptor of the file to be measured. From the file descriptor one can look up the corresponding inode and data blocks, and take a SHA1 over the data blocks.

The consistency between the measurement and what is loaded depends on: (1) accurate identification of the inode loaded and (2) detection of any subsequent writes to the inode. Clearly, the relationship between a file descriptor and an inode can be changed between when the **measure** system call is made and the actual contents are read. However, we have measured the code requesting the measurement, so if it is of high integrity, it is trusted not to swap the inode associated with a file descriptor. The replacement of the file by a malicious attestor will not change the mappings of the process, so this is not a problem either. Second, there could be a race between the measure and read and a write system call that modifies the data. While the measurement may not be exactly what is loaded, the subsequent modification of the inode will be measured by our system as described below. Once a file is measured, any subsequent modifications are captured. Remote NFS files cannot be measured dependably unless the file's complete contents are cached on the local system. We do not implement such caching at present.

In addition to the content of the loaded file, the name of the file, user or group ID of the loading entity, as well as the file system type (local, remote NFS) might be useful environment information for later evaluation of the impact of loading this file or matching it with local security policies. At this time, our implementation supports gathering this additional data for information purposes but does not include it in the measurement.

A naive implementation would be to take a fingerprint for every measure system call. This approach would, however, incur significant performance overhead (see Section 6.2).

Instead, we use caching to reduce performance overhead. The idea is to keep a cache of measurements that have already been performed, and take a new measurement only if the file has not been seen before (cache-miss) or the file might have changed since last measurement. For the latter case, we only record a new file measurement if the file has actually changed. Recording identical measurements each time an application runs would have severe impact on the management (storage, retrieval, validation) of the list.

We store all measurements in a singly-linked, ordered list. The order of measurements is essential to detect any modification to the measurement list. If the measurements are not checked in order, then the aggregate hash will not match the TPM aggregate that results from the TPM_extend operation.

For efficiency reasons, we overlay the linked list with two hash tables, one keyed with the inode number and device number of the measured file, the second keyed with the resulting fingerprint (SHA1 value) of the measured file. Thus, each measurement entry can be reached by traversing the measurement list, by its inode, or by its fingerprint. Each measurement entry contains a dirty flag bit, indicating whether the file is CLEAN (not modified), or DIRTY (possibly modified). The measure system call uses the inode corresponding to the file descriptor of the target file to quickly look up the file in the hash table and see if it has been measured before.

Measuring new files: If the file is not found in the inode-keyed hash table, then we measure the file by computing a SHA1 hash over its complete content.

At this point, we use the computed fingerprint to check whether it is present in the hash table keyed by the SHA1 hash value of existing measurements. If the measured fingerprint is not found, then we create a new measurement entry, and add it to the list and adjust the hash table structures. We finally extend the relevant Platform Configuration Register in the protected TPM hardware by the SHA1 hash before returning from the call and allowing the loading of the executable content. If the fingerprint was already measured before, then we return from the system call without extending the TPM or the measurement list. This can happen if executable files are copied and thus yield the same fingerprint. In this case, we assume for our purpose that both executables are equivalent.

Remeasuring files: If the file is found in the inodekeyed hash table, then it was measured before. If the dirty flag of the found measurement entry is CLEAN (clean-hit), then nothing needs to be done, and the system call returns. If the dirty flag bit is DIRTY (dirty-hit), then we compute the SHA1 value of the file. If the measured fingerprint is identical to the one stored in the measurement list, then we set the dirty flag to CLEAN. We do not extend the PCR or record this measurement as it is known already.

If the measured fingerprint differs from the one stored in the found measurement entry for the inode, then we look up the new fingerprint in the hash table using the SHA1 value as the key. If the SHA1 value exists, then the same file contents was measured before (copy of the current file). We return without recording the measurement, as above. If the SHA1 value does not exist in the hash table, then the current file has changed. A new measurement entry is created and added to the table, and the PCR is extended before the system call returns.

Dirty flagging: We set the dirty flag bit to DIRTY whenever the target file (a) was opened with write, create, truncate, or append permission, (b) was located on a file system we can't control access to (e.g., NFS), or (c) belongs to a file system which was unmounted. This seems a bit conservative, since an open for write (or unmounting a file) does not necessarily result in modifications to the file. The SHA1-keyed hash table enables us to clear the dirty flag if a file did not change after an open with write permission. If we control access to the file, then we clear the dirty flag in such cases. Experiments show that on a non-development system using local file systems, the percentage of dirty-hits on the cache is

far less than 1%.

In summary, the performance optimization requires us to modify two system calls, namely **open** and **umount** to implement the dirty flagging.

Fail-safe mechanisms: To protect our *dirty flagging*, we set the dirty-flag of a measurement to DIRTY as soon as we detect a writable open on the related partition's device file, e.g., /dev/hda1. For this purpose, as well as for dirty-flagging of unmounted file systems, we keep the superblock pointer of a file in the file's measurement structure. Walking through the whole measurement list to dirty-flag entries related to the mount point imposes overhead, but this happens rarely (e.g., on shutdown) on most correctly setup and configured systems and the measurement lists are usually not very large (<1000 entries).

In case of any error throughout the recording of measurements, e.g., caused by out-of-memory errors when allocating a new measurement structure, we invalidate the TPM aggregate by extending it with random values without extending the measurement list and deleting the random value to protect it from later use. Thus, from this time on, validations of the aggregate will fail against the measurement list. We do not interfere with the system (non-intrusive) but we disable such a system from successful attestation until it reboots.

In case of *suspicious behavior*, e.g., when recognizing writable access to /dev/kmem, we also invalidate the TPM aggregate as described above. This is necessary to prevent the kernel from being changed without this change being measured. The only allowed way to change the kernel is through loading modules, in which case the loaded modules are measured and can be evaluated by challenging parties. Such suspicious cases are rarely necessary or observed in normal systems. Thus, disabling successful attestation in these cases seems justified.

5.3 Validating Measurements

Our architecture uses the TPM's protected storage of the TPM aggregate to protect the integrity of the measurement list. Once a measurement is taken, it cannot be changed or deleted without causing the aggregate hash of the measurement list to differ from the TPM aggregate. However, the challenging party must also ensure that the attesting system has the measurement architecture correctly in place so that all necessary measurements are actually initiated and carried out. As our architectural components are measured as well when they are executed, challenging parties can determine whether the architecture is in place by inspecting these measurements.

The major portion of the measurement architecture is in the static kernel. Thus, the challenging party trusts only such kernels that implement the kernel part of our measurement architecture. Other kernels will be unacceptable to challenging parties because they can skip important measurements.

When validating individual measurements, the challenging party must enforce the following rules:

Insmod program: The instrumented insmod program measures kernel modules before they are loaded into the kernel. Only insmod programs instrumented with the **measure** call are acceptable. If a fingerprint of any other program with insmod functionality is seen, then it must not be trusted and thus the validation fails.

Dynamic library loader: The instrumented dynamic loader ensures that all dynamically loaded libraries are measured. Only dynamic loaders instrumented with the measure call are thus acceptable. A fingerprint of any other dynamic loader must not be trusted and thus the validation fails if such a fingerprint is included in the measurement list.

Fingerprints of any other part of the system can be trusted according to known vulnerabilities of corresponding programs or libraries as described in Section 4.4. As any unknown fingerprint could result from a changed insmod or dynamic loader, we cannot trust any fingerprint whose corresponding program's functionality is unknown.

6 Results

6.1 Experiments

To test our system's ability in detecting possible attacks, we construct a small experiment using lrk5, a popular Linux rootkit. We start with a perfectly good target system, and take measurements of this system. Then we launch a rootkit attack against the target system, and take measurements again after the attack. Figure 4(a) shows a (partial) list of measurements for the good system, and Figure 4(b) shows the corresponding list of the same system that is compromised by a rootkit. The italicized entries show that after the attack, the signature of the **syslogd** program is different, indicating that the rootkit had replaced the original **syslogd** with a trojan version. This example illustrates how such attacks can be discovered reliably using our system.

#000: D6DC07881A7EFD58EB8E9184CCA723AF4212D3DB boot_aggregate
#001: CD554B285123353BDA1794D9ABA48D69B2F74D73 linuxrc
#002: 9F860256709F1CD35037563DCDF798054F878705 nash
#003: 84ABD2960414CA4A448E0D2C9364B4E1725BDA4F init
#004: 194D956F288B36FB46E46A124E59D466DE7C73B6 ld-2.3.2.so
#005: 7DF33561E2A467A87CDD4BB8F68880517D3CAECB libc-2.3.2.so
...
#110: F969BD9D27C2CC16BC668374A9FBA9D35B3E1AA2 syslogd
...
(a)

#110: F969BD9D27C2CC16BC668374A9FBA9D35B3E1AA2 syslogd ...

#525: 4CA3918834E48694187F5A4DAB4EECD540AA8EA2 syslogd

(b)

Figure 4: Detecting a Rootkit Attack.

6.2 Performance Evaluation

We measure the runtime performance with a set of micro-benchmarks. We first measure the latencies of the measure system call in three different cases, namely, no_SHA1, SHA1, SHA1+extend. no_SHA1 represents the case when the target file is found in the cache and is clean. The measure system call thus returns without further work. In the SHA1 case, the target file is remeasured and the SHA1 fingerprint is recalculated. However, the TPM is not extended because the fingerprint is found to be already in the cache. SHA1+extend represents the case when a brand new file is measured and the resulting fingerprint needs to be extended. Since the goal is to measure the latency, we use a test file size of 2 bytes. Implementation of the micro-benchmarks is based on the HBench framework [15].

Table 1 shows the results. For reference purposes, we include the running times of two normal system calls (call pairs): gettimeofday, and open/close. It is clear from the table that the overhead for the

| System Call | | Overhead (stdev) |
|-------------|--------------|-----------------------|
| measure | no_SHA1 | $0.5 \ \mu s \ (0.0)$ |
| | SHA1 | $2.7 \ \mu s \ (0.0)$ |
| | SHA1+extend | 9971.8 μ s (0.3) |
| reference | gettimeofday | $0.6 \ \mu s \ (0.0)$ |
| | open/close | $2.6 \ \mu s \ (0.0)$ |

Table 1: Overhead of the Measure System Call.

measure system call in the case of a clean cache hit (no_SHA1) is minimal - it takes 0.5 microseconds to run, less time than the gettimeofday system call, which is often quoted as the basic system call layer overhead (entering and leaving the kernel mode). Fortunately, our experiences indicate that this is the majority case, even for servers that tend to run for a long time, accounting for more than 99.9% of all measure system calls.

When the file is remeasured, the measure system call takes about 2.7 microseconds, about the same time it takes for opening and closing a file. This case shows the overhead of setting up the file for measurement, and searching the hash table for a matching fingerprint. Notice that this case does not measure the overhead of the fingerprinting itself, since the file size is only 2 bytes. Fingerprinting performance will be discussed later. The extend operation is clearly the most expensive, taking about 10 milliseconds to execute. This is understandable, because the extend operation interacts with the TPM chip. As mentioned before, these two cases adding together represent less than 0.1% of all measure system calls. Thus we are confident that the performance penalty our system imposes upon the user will be negligible.

Next we present the fingerprinting performance as a function of file sizes. We measure the measure system call's running time in the SHA1 case, varying the input file sizes. The results are shown in Table 2. When the file size is large, the fingerprinting overhead can be significant. For example, measuring a 128 Kilobytes file takes about 2 milliseconds. The running time increases close to a linear fashion as the size of file increases. These latencies translate to a throughput performance of about 55MB per second. These experiments were run with a hash table containing about 1000 entries. Our experiences show that a standard RedHat 9.0 Linux system including the Xwindow server and the Gnome Desktop system accumulates about 500 measurement entries after running about one week. Thus we believe our

| File Size (Bytes) | Overhead (stdev) |
|-------------------|-------------------------|
| 2 | $2.7 \ \mu s \ (0.0)$ |
| 512 | 11.5 μs (0.0) |
| 1K | $20.1 \ \mu s \ (0.1)$ |
| 16K | $280.5 \ \mu s \ (0.3)$ |
| 128K | 2226.3 μ s (1.7) |
| 1M | 18312.3 μ s (18.1) |

Table 2: Performance of the SHA1 FingerprintingOperation as a Function of File Sizes.

performance results are representative of a normal Linux environment.

7 Discussion

Our architecture is non-intrusive and does not prevent systems from running malicious programs. However, we modify our approach to *enforce security* as well. In this case, we pre-load the measurement cache with a set of expected fingerprints for trusted programs. The measurement call then fingerprints the file to be measured and compares it to the set of expected fingerprints. If the fingerprint does not match any of them, it aborts the load and reports the illegal fingerprint. Note that the attesting system's enforcement requirements may be different than those of the challenger, so the challenger still needs to perform a validation.

Our measurement architecture is not restricted to measuring executable code. Adding measurement hooks into applications, we can include *structured input data*, such as configuration files and java classes, into our measurements. Changes are simple– instrumenting applications, such as Apache or the Java classloader, means adding a measurement call before loading relevant files.

In order to establish confidence in a system, *privacy* is impacted by our approach. The attestation protocol releases detailed information of the attesting system to allow challengers or trusted third parties to establish trust. However, the attesting system has full control over the release of this information, and can run code that it trusts not to release such information. Also, a system agent could be configured to release attestations to authenticated challengers and the operating system could only provide quotes to that agent. Inducing frequent changes in loaded executable files can cause the measurement list to grow beyond practical limits, resulting in a *denial of service* attack. To prevent this attack, a maximum length of the measurement list can be configured. Any additional measurement is aggregated into the TPM-protected PCR register, but the measurement is not stored in the kernel. Consequently, a system that exceeds this maximum number of measurements will not be able to successfully convince challenging parties of its integrity because the measurement list will not validate against the aggregate any more.

8 Conclusions

We presented the design and implementation of a secure integrity measurement system for Linux. This system extends the TCG trust concepts from the BIOS all the way up into the application layer for a general operating system. We extend the operating system with hooks to measure when the first code is loaded into a process (execve), provide a measure system call for subsequent measurements, and detect when changes to measured inodes are taken. This mechanism enables the measurement of dynamic loaders, shared libraries, and kernel modules in addition to the executed files. Further, the approach is extensible, such that applications can measure their specialized loads. The result is that we show that many of the Microsoft NGSCB guarantees can be obtained on today's hardware and today's software and that these guarantees do not require a new CPU mode or operating system but merely depend on the availability of an independent trusted entity. Such a system can already detect a variety of integrity issues, such as the presence of rootkits or vulnerable software. Our measurements show that the non-development systems can be practically measured and that the measurement overhead is reasonable.

The measurement system is extensible and we believe that we can ultimately achieve guarantees beyond those of Microsoft NGSCB. The application of mandatory access control policy can ensure that dynamic data cannot be modified except by trusted sources [16]. Identification of low integrity data flows can enable the possibility of control over whether these flows should be allowed, whether effective restriction can be put on them at the system-level or within applications.

References

- W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A Secure and Reliable Bootstrap Architecture," in *IEEE Computer Society Conference on Security and Privacy*. IEEE, 1997, pp. 65–71.
- [2] "Trusted Computing Group," http://www.trustedcomputinggroup.org.
- [3] K. J. Biba, "Integrity considerations for secure computer systems," Tech. Rep. MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.
- [4] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *IEEE Symposium on Security and Privacy*, 1987.
- [5] S. W. Smith, "Outgoing authentication for programmable secure coprocessors," in *ESORICS*, 2002, pp. 72–89.
- [6] M. Bond, "Attacks on cryptoprocessor transaction sets," in Proceedings of the 2001 Workshop on Cryptographic Hardware and Embedded Systems, May 2001.
- [7] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A Trusted Open Platform," *IEEE Computer*, vol. 36, no. 7, pp. 55–62, 2003.
- [8] G. Kim and E. Spafford, "Experience with Tripwire: Using Integrity Checkers for Intrusion Detection," in System Administration, Networking, and Security Conference III. USENIX, 1994.
- [9] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking systems rules using system-specific, programmer-written compiler extensions," in *Proceedings of the 4th Sympo*sium on Operating Systems Design and Implementation (OSDI 2000), October 2000.
- [10] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 Secure Coprocessor," *IEEE Computer*, vol. 34, no. 10, pp. 57–66, 2001.
- [11] Trusted Computing Group, Trusted Platform Module Main Specification, Part 1: Design Principles, Part 2: TPM Structures, Part 3: Commands, October 2003, Version 1.2, Revision 62, http://www.trustedcomputinggroup.org.
- [12] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetho, and S. Yoshihama, "Trusted Platform on demand (TPod)," in *Technical Report, Submitted for Publication*, 2004, In submission.
- [13] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *Proc. 9th ACM Symposium* on Operating Systems Principles, 2003, pp. 193–206.
- [14] CERT Coordinatin Center, "CERT/CC Advisories," http://www.cert.org/advisories.
- [15] A. B. Brown and M. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," in Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, June 1997, pp. 214–224.
- [16] T. Jaeger et. al., "Leveraging information flow for integrity verification," in SUBMITTED for publication, 2004.