PRIMA: Policy-Reduced Integrity Measurement Architecture

Trent Jaeger
SIIS Lab, CSE Dept.
Pennsylvania State University
University Park, PA
tjaeger@cse.psu.edu

Reiner Sailer
Secure Systems Group
IBM Research - Watson
Hawthorne, NY
sailer@us.ibm.com

Umesh Shankar
CS Department
UC Berkeley
Berkeley, CA
ushankar@cs.berkeley.edu

ABSTRACT

We propose an integrity measurement approach based on information flow integrity, which we call the *Policy-Reduced Integrity Measurement Architecture* (PRIMA). The recent availability of secure hardware has made it practical for a system to measure its own integrity, such that it can generate an integrity proof for remote parties. Various approaches have been proposed, but most simply measure the loaded code and static data to approximate runtime system integrity. We find that these approaches suffer from two problems: (1) the load-time measurements of code alone do not accurately reflect runtime behaviors, such as the use of untrusted network data, and (2) they are inefficient, requiring all measured entities to be known and fully trusted even if they have no impact on the target application. Classical integrity models are based on information flow, so we design the PRIMA approach to enable measurement of information flow integrity and prove that it achieves these goals. We prove how a remote party can verify useful information flow integrity properties using PRIMA. A PRIMA prototype has been built based on the open-source Linux Integrity Measurement Architecture (IMA) using SELinux policies to provide the information flow.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Information Flow Controls*

General Terms

Security, Measurement, Management

Keywords

Remote attestation, information flow, Clark-Wilson Lite integrity

1. INTRODUCTION

Distributed applications and services are essential to our future information infrastructure, but the development of a secure system foundation across a set of machines has not been achieved. While operating systems and middleware support a wide range of distributed functionality, additional mechanisms are needed for each machine to trust the others. For example, one machine may want to know that another is running a known-good version of the application code on a well-configured, trusted operating system. Without this guarantee, the remote machine may be running buggy or malicious application code, or may be improperly configured such that the trusted application can be corrupted by untrusted programs or users.

Hardware-based integrity measurement has emerged as a mechanism that enables one system to prove its integrity to other remote parties. Taking a *measurement* of something (e.g., code or data) means computing a cryptographic hash of it and extending a hardware-protected hash chain with that hash value. An example of a hardware component for integrity measurement is the Trusted Computing Group's (TCG) Trusted Platform Module (TPM) [14]. Various mechanisms have been proposed to use such hardware to generate a proof of a system's integrity, called *remote attestation* or *authenticated boot* [23]. For example, TPod implements extensions to the `grub` bootloader to measure the sequence of code loads that bring up the operating system, and it stores these measurements in the TPM to protect them from tampering by software [11]. The TPM can create signed messages that enable a remote party to verify the the code loads measured by TPod. Further, other approaches have been proposed to extend integrity measurement and verification up to the application level. One such approach, the Linux Integrity Measurement Architecture (IMA), has Linux measure code loaded and static data files (e.g., configurations) used, such that a remote party can verify that a Linux system contains no low integrity components [20]. These approaches provide a way to start with a small trusted component—the TPM—and leverage that to build a proof for a whole system by taking systematic measurements of each piece as it loads.

The extensions of TPM measurement to prove the integrity of systems at the application-level suffer from two limitations, however. First, the load-time measurements of code alone do not accurately reflect runtime behaviors, such as the use of untrusted network data. Second, existing approaches require the entire system to be trusted (more precisely, measured) even when the remote party only requires the integrity of a specific application. Generally, a remote party wants to use a particular application, that we call the *target application* of the attestation. With a suitable security configuration and operating system support, that application can be isolated in an information-flow sense from most other applications on the system. Without access to such

SACMAT'06, June 7–9, 2006, Lake Tahoe, California, USA.

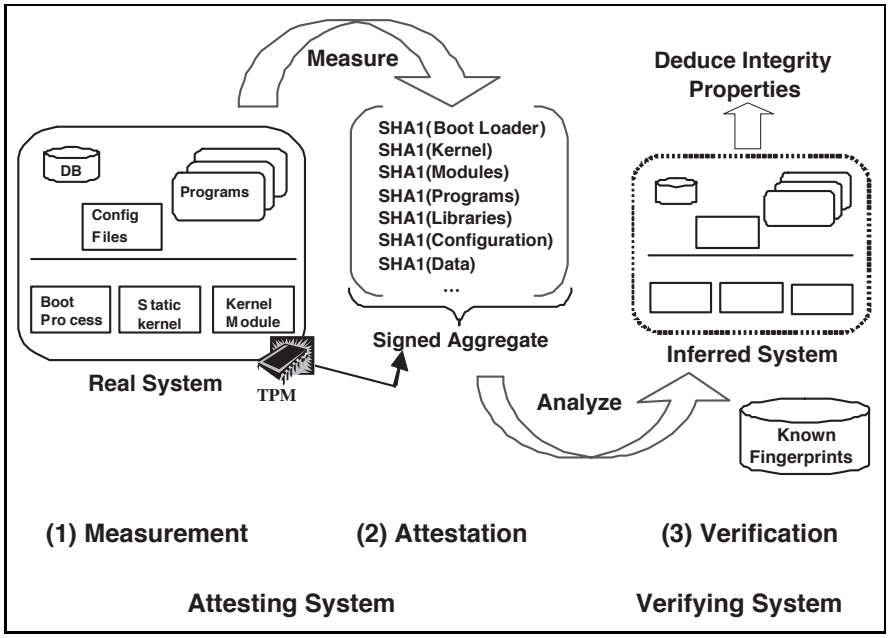

Figure 1: Linux IMA Overview: The attesting system generates system measurements (1) into a TPM-backed attestation (2) that a remote party can verify (3).

dependency information, a remote party must conclude that any unknown or untrusted program that is loaded may compromise the target application, regardless of whether any real dependency exists that may compromise it.

Historically, the integrity of applications has been evaluated using system information flows. Using information flow, an application's integrity is determined by the integrity of the inputs that it *depends on*. From the target application's perspective, we refer to inputs that are known to come from trusted sources as *high integrity*, and those that may come from untrusted sources as *low integrity*. Classical integrity models can represent such an integrity relationship, as well as more complex ones consisting of a greater number of integrity levels, as a lattice. For example, Biba integrity requires that a process (noting that an application may consist of a set of processes) receive no input that is lower integrity than itself [4]. Low-Water Mark Integrity (LOMAC) requires that a process's integrity be that of the lowest integrity input that it receives [12]. Using information flow information, we can address the two limitations of current integrity measurement approaches. We can see where runtime inputs come from, and we can optimize measurement by restricting it only to those elements on which the target application depends.

In this paper, we define the *Policy-Reduced Integrity Measurement Architecture* (PRIMA), an extension of the Linux IMA that measures not only the code that is run on a system, but also which information flows are present among processes. PRIMA's approach therefore lets us attest more sophisticated integrity guarantees, including most classical models like Biba and Clark-Wilson [6]. However, using a more practical integrity model, CW-Lite, which we defined in recent work [21], we can improve the efficiency of the attestation. We therefore use it as our example. CW-Lite is a pared-down version of Clark-Wilson integrity that relaxes its formal verification requirement and uses the system security policy's implied information flows to reduce requirements on trusted applications. From an information-flow perspective, it provides the same guarantee as the Clark-Wilson model, i.e., all flows from untrusted processes to high integrity ones must pass through a filtering/sanitizing procedure in the destination process.

In this paper, we prove that PRIMA can attest CW-Lite and describe the prototype implementation of the PRIMA system for Linux using SELinux security policies (noting that a similar approach may be used for other information-flow integrity properties). We also describe concrete threats that the PRIMA approach addresses that previous approaches do not. We find a variety of cases where previous integrity measurement approaches would generate false negative attestations (i.e., an attestation would succeed when the target may be compromised) and false positive attestations (i.e., an attestation would fail when the target is high integrity). PRIMA would reason correctly in each of these cases, with a likely decrease in the number of measurements necessary.

This paper is organized as follows. In Section 2, we examine current integrity measurement, its limitations, and how information flow can address these limitations. In Section 3, we detail the PRIMA measurement approach and show that it satisfies the integrity measurement requirements. In Section 4, we outline a prototype implementation of PRIMA that measures SELinux policies for CW-Lite integrity. In Section 5, we examine several cases where IMA and PRIMA would generate different attestation results and describe how PRIMA achieves the desired result. In Section 6, we discuss related work. In Section 7, we outline future work and conclude.

2. BACKGROUND

In this section, we provide background in integrity measurement architectures and information flow for developing solutions in subsequent sections.

2.1 Example Integrity Problem

Many employees of corporations use laptop computers as the main computing environments. On these systems, they run applications that the corporation depends on, such as purchasing, performance appraisals, expense reporting, etc. Also, employees run programs that are notorious for the vulnerabilities they enable, such as web browsing and email, that could impact the integrity of these corporate applications. A corporation would like its servers to verify the integrity of its corporate applications on these laptops – that is, that they are protected from other programs that may run on the same laptop – before permitting the corporate applications to connect to those servers.

In the past, we implemented a system that verifies remote system integrity before releasing corporate data to that system [19]. The difference here is that laptops will almost always fail that verification because low integrity or unknown software will be running, and we cannot guarantee protection of corporate data from those applications. However, the use of information flow enforced by mandatory access control in operating systems, such as the Linux Security Modules framework [24] using the SELinux module [2], can enable the isolation of key applications from such user processing. Our goal is to extend remote attestation to enable such a guarantee.

2.2 Integrity Measurement

Integrity measurement architectures aim to measure the status of a computer system, such that a remote party can prove the integrity of this system. Such architectures consist of measurement systems, attestation mechanisms, and verification mechanisms that test an integrity property. An *integrity measurement system* defines what measurements will be made, how they will be stored, and how their validity will be preserved. A *computer attestation mechanism* defines the protocol by which these measurements are conveyed to remote parties securely. Lastly, the remote party uses a *verification mechanism* to test the measurements against the expected *integrity property*. For our purposes, the most important facets are the integrity property and how it is measured and verified. For details on the measurement process, the reader is referred to prior work on the Linux Integrity Measurement Architecture (IMA) [20].

The most common integrity property of current systems is *load-time integrity*. This property requires that all code is measured at load-time, and that it is known to be of high integrity. The remote party can verify this by checking the measurements against known acceptable measurements (e.g., binaries shipped with the Fedora Core 4 distribution). This approach is used in outbound authentication [23], Palladium/NGSCB [10], and IMA [20]. Terra uses a similar approach, measuring static VM pages rather than static code and pages at the file level [13]. The BIND system takes a different approach by measuring discrete computation steps by their inputs and code, but the current examples are similar in granularity to file-level measurements [22].

For load-time integrity, both code and static data files are measured. Both are measured at load-time (i.e., prior to execution) into secure hardware, in particular the Trusted Computing Group's Trusted Platform Module (TPM), to ensure that their execution cannot hide the fact that they were loaded. A load-time measurement for code implies that the code was in a known state when it was loaded. A load-time measurement of a static file indicates that that file had a known value when it was loaded. The known state of code and data is important for verification because the remote party must also know these states in order to reason about their integrity impact. Thus, a remote party can prove that the code and static data are in known, high integrity states when they are loaded on this system.

2.3 Limits of Load-Time Measurement

Load-time measurement is limited in the runtime guarantees that can be inferred. First, load-time guarantees only state that the code is of high integrity when it was loaded. Trust in the code measured at load-time requires that all information flows be handled adequately by the program. In this context, a high integrity program is one with no known vulnerabilities. However, such programs can be compromised if an untrusted input can impact a previously unknown vulnerability.

Second, any stateful programs are dependent on dynamic state, and load-time measurement cannot ensure that this dynamic data is handled in a manner that preserves its high integrity. For example, a transaction processing system may have its customer database modified without that change being detected by load-time measurement. Many integrity measurement architectures provide what is called *authenticated boot* whereby a remote party can verify whether the system has booted with high integrity code. However, the system will still run even if low integrity code has been loaded, so dynamic data may be modified by a low integrity system, then rebooted such that authenticated boot will succeed. However, the compromised dynamic data will render the application low integrity in reality. The initial state of the system at each boot must be verified to prevent this attack.

Third, load-time measurement results in a more conservative guarantee than necessary with respect to the amount of code that must be trusted. Only code and data that the target application (i.e., the application that the remote party requires) depends upon needs to be high integrity. The integrity of code loaded on the system that has no information flows to the application need not be trusted. As above, this guarantee must be enforced across system boots.

2.4 Information Flow Integrity

Information flow integrity models explicitly represent the possible dependences of both code and data. For example, the Biba integrity model requires that code executed and data read by a process be at its integrity level or higher [4]. Thus, problems such as code injection and dependence on low integrity dynamic data will be prevented by the integrity policy. In addition, there is no need to measure the loads of lower integrity code, because this code cannot impact the target code. The same guarantee would also be provided using the Low-Water Mark (LOMAC) integrity model, where the integrity of a process is equal to the lowest integrity level of any of its inputs [12].

Our goal is to extend load-time integrity measurement to information flow integrity measurement. The following guarantees are necessary for Biba integrity measurement using information flow.

1. **Trusted Subjects**: The set of trusted subjects in the MAC policy must be trusted by the remote party.

2. **Trusted Code/Data**: All code and static data loaded for any trusted subject must correspond to known and trusted hashes by the remote party.

3. **Information Flows**: All information flows to a trusted subject must come from another trusted subject.

We must be able to distinguish the trusted subjects in the MAC policy from those that do not require trust. Trust is determined by the target application: the target application and all subjects that it must trust are in the trusted application set (i.e., form the relevant trusted computing base to the remote party). In the laptop example, the corporate applications form the target applications. If the corporate application programs are known and trusted, and there are no information flows from client applications, such as the browser and email client, to these corporate applications and the programs that they depend upon (e.g., system services), then Biba integrity is satisfied.

This is a reasonable start, but Biba-style models fail to capture a common case: high-integrity processes that much handle low-integrity inputs. For example, many UNIX services, such as `sshd`, `vsftpd`, and `inetd`, receive untrusted network input. Also, the corporate applications are connected to the network, so they must protect themselves from untrusted inputs. The Clark-Wilson integrity model expresses requirements in a manner that more closely mirrors what we are seeing [6]. The Clark-Wilson integrity model consists of several rules cover authentication, audit, and separation of duty, but two rules are particularly relevant in this context: (1) *initial verification procedures* [1] ensure that the system state is of high integrity upon each boot and (2) *transformation procedures* are the only processes that operate on high integrity data and they are assured to discard or upgrade the integrity of any low integrity inputs that they receive. In the first case, dynamic data is checked to verify that it was not compromised in prior boot cycle, such as ensuring unsealing of high integrity data only when trusted programs boot [18]. In the second case, trusted processes, such as the UNIX services above, can protect themselves from low integrity inputs.

2.5 CW-Lite Integrity

There are two practical problems with applying Clark-Wilson directly to commercial systems: (1) complete, formal assurance of high integrity applications is not practical and (2) only a small number of application interfaces are expected to handle low integrity data in practice. First, broad application of formal assurance to programs requires automated tools to verify the correctness of programs that have not emerged in the 19 years since the model was defined. Second, we find that in practice only few interfaces are open to the network or other information flows that will accept low integrity data. Only these interfaces need to be managed. We define a weaker version of Clark-Wilson integrity, called *CW-Lite* [21], that requires that: (1) only the interfaces accepting low integrity data must have filters and (2) complete, formal assurance of the program is not required, although some basis for trust in the filter interfaces' ability to discard or upgrade low integrity inputs is expected [2]. The enforcement mechanism restricts access according to Biba integrity for normal interfaces, but permits low integrity information input at filtering interfaces.

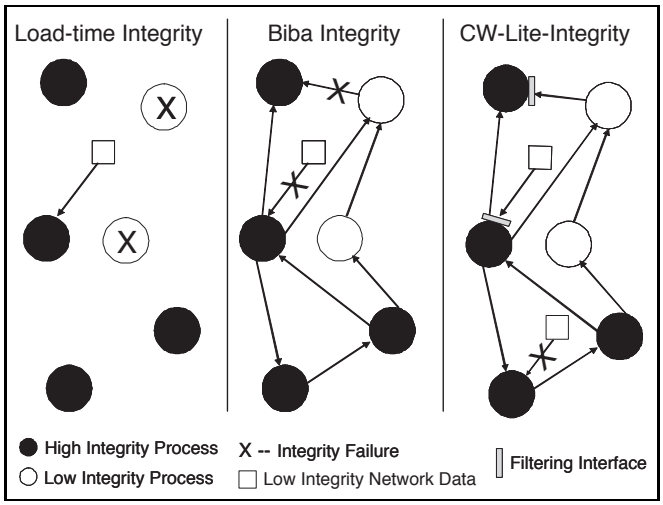

Figure 2: Integrity semantics of different integrity models: (1) Load-time measurement fails when any low integrity code is loaded, but ignores the impact of low integrity network data; (2) Biba integrity considers information flows, but fails if any low-to-high integrity flow is present; (3) CW-Lite allows some low-to-high flows via filtering interfaces only.

Figure 2 shows the different forms of integrity verification semantics offered by load-time, information flow, and CW-Lite integrity measurement. In the load-time integrity measurement, the presence of low integrity code invalidates system integrity, whereas low integrity inputs do not. The laptop example would fail load-time integrity because of the presence of untrusted client applications. In information flow integrity, the input of low integrity code or data invalidates system integrity. The laptop example would fail information flow integrity because of the input from untrusted, remote parties at a minimum. In CW-Lite, some low integrity information flows may be accepted by high integrity subjects. In the laptop example, the some interfaces of the UNIX services and corporate applications may be deemed capable of discarding or upgrading such low integrity inputs. We refer to these interfaces as *filtering interfaces*. In CW-Lite, low integrity permissions are only accessible through filtering interfaces via *filtering subjects*. That is, the permissions of filtering subjects are only available when the code of a filtering interface is run. Otherwise, the process runs with the permissions of a trusted subject and is limited to Biba integrity. See the detailed description [21] for more explanation.

We now extend the Biba integrity verification to support verification of Clark-Wilson Lite integrity by the following, additional guarantees:

4. **Initial Verification**: The initial verification procedure code must be of high integrity and the verification must be successful.

[1] Actually, they are called *integrity verification procedures* in the Clark-Wilson model, but we changed the name prevent conflict.

[2] We do not define how this is done, but the filtering interfaces in privilege-separated OpenSSH [17] serve as our motivating example.

5. **Filtering Interfaces**: Any claim that a particular interface discards or upgrades all low integrity inputs must be verifiable.

6. **Filtering Subjects**: The permissions to receive low integrity inputs must only be available to filtering subjects, and these subjects must only run within the context of filtering interfaces.

Since we are using authenticated boot, we must be able to verify the integrity of the dynamic data on boot as specified in the Clark-Wilson integrity model. Next, we identify (or build) filtering interfaces into those applications that we want to entrust to read low integrity data. We must ensure that the additional permissions to read low integrity data in the filtering subject are only used at filtering interfaces. If remote parties accept that any program that uses a filtering subject has acceptable filtering interfaces, they additionally verify that the filtering subjects are only enacted at those interfaces prior to accepting the system as meeting CW-Lite integrity.

3. POLICY-REDUCED MEASUREMENT

In this section, we detail how each of the 6 requirements are met by the Policy-Reduced Integrity Measurement Architecture (PRIMA) and describe the actual measurements that must be made.

3.1 PRIMA Requirements

PRIMA measurement generates two values: (1) a measurement list M_i containing each measurement $m_0, m_1, ..., m_i$ and (2) a hardware-protected (e.g., TPM) hash aggregate H_i where $H_0 = H(m_0), ..., H_i = H(H_{i-1}||H(m_i))$. The attesting party provides M_i and a TPM-signed H_i to a remote party (remote party must be able to reliably obtain the TPM's public key certificate for its attestation identity key), and the remote party verifies that a hash aggregate computed from the measurements in M_i corresponds to the signed hash aggregate.

Trusted Subjects. Trusted subjects are the set of MAC policy subjects, $T \subseteq S$ where S is the set of all MAC policy subjects, that must be trusted by the remote party for the integrity of the system to be verified. If the remote party does not trust one of the subjects, then the remote party must assume that target application receives a low integrity information flow. Experience has shown that this list is of modest size (about 50).

The measurement m_T for trusted subjects is value of T, and it is added to the measurement list $M_i = M_{i-1}||m_T$. The hash aggregate is extended by the hash of the measurement of m_T, $H_{i+1} = H(H_i||H(m_T))$.

Trusted Code/Data. This is the code and static data used by a trusted subject in the system. Therefore, we must measure the code or data with the subject that it pertains too. The remote party determines for each trusted subject that the code and static data measurements made for that subject correspond to known and high integrity hash values. Further, the code may indicate the role of the static data used. For example, the remote party should be able to determine that a known configuration file was used as a configuration file rather than a static input data file.

PRIMA measures the code/data, subject, and optional role if the subject is trusted. We build a measurement entry $m_e = (c||s||r)$ where c is the code or data digest, s is the trusted subject name, and r is an optional role identifier. The aggregate is $H_{i+1} = H(H_i||H(m_e))$. The measurement list is extended with the measurement entry $M_{i+1} = M_i||m_e$.

Information Flow. Information flow shows how data flows among system subjects S, both trusted and untrusted, based on the read and write operation in the MAC policy. Information flow is represented as a graph $G = (S, E)$ where S is the set of subjects that form the vertices of the graph and E is the set of edges that describe information flows operations. An edge from subject s_1 to subject s_2, $s_1, s_2 \in S$, is added if s_2 reads an object that s_1 can modify. The remote party will verify that all information flows to a trusted subject are from other trusted subjects.

Information flow is derived from the MAC policy, so we must measure the MAC policy in PRIMA. To ensure correctness of the system, the binary MAC policy is measured. It is the remote party's responsibility to build an information flow graph. We envision that standardized policies, such as the SELinux Reference Policy, may result in only the need to measure a hash of the MAC policy. Thus, the measurement of MAC policy m_P is either the policy or its digest depending on whether the latter is known. The hash aggregate is extended by $H_{i+1} = H(H_i||H(m_P))$ where P is the MAC policy. The measurement list is extended by $M_{i+1} = M_i||m_P$. This measurement must be taken only once.

Initial Verification Procedure (IVP). The IVP is the code that measures the integrity of a system at boot time and the result of the execution of that procedure. The remote party will verify that the IVP ran and that its result meets expectations. Such methods are currently domain-specific and no foolproof mechanism without hardware support has been identified.

The attesting system already measures code and its mapping to subjects, so we can identify the IVP subject and IVP code using trusted code/data measurements. Further, the IVP code can measure the result of the IVP test in the manner of static data. Thus, no new types of measurements are necessary to capture the IVP and its results.

Filtering Interfaces. Filtering interfaces are in the code of programs running as trusted subjects. This code will be trusted to switch the process' subject type temporarily to the filtering subject type, to discard or upgrade low integrity inputs, then switch back to the trusted subject.

Since all code of trusted subjects—which includes the filtering interfaces—is already measured, the attesting system needs no additional measurements for enabling verification of filtering interfaces.

Filtering Subjects. F is a set of subjects, $F \subseteq S$ where $F \cap T = \emptyset$, that are capable of filtering low integrity inputs, so they may be granted permissions to access low integrity information flows. The remote party must be aware that such filtering subjects are present to know which code to verify for use of filtering interfaces.

Each filtering subject is associated with a single trusted subject (not all trusted subjects have filtering subjects). Thus, we extend the measurement entry for trusted subjects m_T to include the indication of whether the trusted subject has an associated filtering subject. No additional measurements are made.

To Verify CW-Lite. Formally, the following must be verified as true by the remote party:

1. The remote party must verify the correspondence of the hash aggregate to the measurement list and extract the trusted subjects T and the MAC policy P. An information flow graph $G = (S, E)$ identifying trusted and untrusted subjects is built from Pp.
2. For each measurement entry in the measurement list M that is a code/data measurement m_e for a trusted subject (i.e., m_e's subject s where $s \in T$) the remote party verifies that code/data digest (c in m_e) is known and high integrity and serves the measurement's role (r in m_e) if specified.
3. For each subject $s \in S$ in the MAC policy P, the following information flow requirements must be met depending upon whether it is a trusted, filtering, or untrusted subject:
   - For a trusted subject $t \in T \subseteq S$, all information flows connected directly to the subject must be from other trusted subjects or filtering subjects. Only the edges connected directly need to be examined because the existence of any low integrity flow to any trusted subject is sufficient for failure.
   - For a filtering subject $f \in F \subseteq S$, no requirements.
   - For an untrusted subject $u \in S - (T \cup F)$, no requirements.
4. For a measurement entry m_e of an IVP subject $i \in T \subseteq S$, the code loaded c must meet the requirements of trusted subject code in #1, be trusted to perform integrity verification, measure the verification result, and the integrity verification result measurement must be positive.
5. For a measurement entry m_e of any trusted subject with an associated filtering subject (indicated in m_T), its code c must meet the requirement for the code of any trusted subject in #1 and be trusted to activate the filtering subject only within filtering interfaces that are trusted to discard or upgrade all low integrity inputs.

3.2 PRIMA Measurements

In addition to the basic integrity measurements of code and static data, we identify the following set of measurements necessary for a remote party to verify CW-Lite integrity:

1. **MAC Policy**: The mandatory access control (MAC) policy determines the system information flows.
2. **Trusted Subjects**: The set of *trusted* subjects (TCB) that interact with the target application is measured. The remote party must agree that this set contains only subjects that it trusts as well.
3. **Code-Subject Mapping**: For all code measured, record the runtime mapping between the code and the subject type under which it is loaded. For example, `ls` may be run by normal users or trusted administrators; we might want to trust only the output of trusted programs run by trusted users. If the same code is run under two subject types, then we take two measurements, but subsequent loads under a previously-used subject type are not re-measured.

At system startup, the MAC policy and the set of trusted subjects is measured. From these, the remote party constructs an information flow graph. The remote party can verify that all edges into the target and trusted applications are either from trusted subjects (that are verified at runtime only to run trusted code) or from untrusted subjects via filtering interfaces (recall that we extended the MAC system to include interface-level permissions).

Next, we measure the runtime information. Due to the information flow graph, we only need to measure the code that we depend on (i.e., trusted subjects' code). All others are assumed untrusted anyway. Also, we measure the mapping between the code loaded and the trusted subject in which the code is loaded, so the remote party can verify that the expected code is executed for the subject. This is analogous to measuring the UID a program runs as in traditional UNIX.

3.3 PRIMA Correctness

We now show that PRIMA achieves verification of CW-Lite integrity as described in Section 3.1 above. The idea is that a the verifying party receives the measurement list and its associated TPM-generated hash aggregate, the latter of which serves to authenticate the former. The verifier can check that (1) the aggregate indeed corresponds to the measurement list and (2) that the operations in the measurement list are sufficient to verify whatever integrity property is desired. In this case, verifying the CW-Lite property is the goal.

Requirement 1: High Integrity Code Loaded in Trusted Subjects. PRIMA measures all code loaded into trusted subjects and the mapping between code and the subject. This is the same as traditional integrity measurement (e.g., IMA), except that the mapping of code measurement to subject is captured and untrusted subject code is not measured.

Requirement 2: CW-Lite Information Flow Requirements. PRIMA measures the binary MAC Policy (i.e., compiled from textual rules) which defines the information flows in the system. From the binary MAC policy, an information flow graph $G = (S, E)$ can be constructed and the information flow tests described above can be executed.

Requirement 3: Initial Verification. PRIMA measures the code of trusted subjects and the IVP would run as a uniquely identifiable trusted subject. The result of the IVP would be measured in the manner of static data, and trust in the IVP would justify trust in the result measurement.

Requirement 4: Filtering Interface Correctness and Use. PRIMA measures the code of a filtering subject indirectly under its corresponding trusted subject: m_e is generated for each trusted subject and the measurement of trusted subjects m_T indicates whether an associated filtering subject is allowed. We make the assumption that any piece of loaded code that transitions to the filtering subject type also performs filtering, i.e. these two functions are never split across loadable modules. Then filtering correctness properties may be checked against the measured code binary.

4. IMPLEMENTATION

This section describes the extensions to SELinux and the Linux Integrity Measurement Architecture (IMA) necessary to develop a PRIMA implementation that measures the CW-Lite integrity property of a system. First, we describe how information flow is derived from a traditional SELinux policy. Next, we outline changes to the SELinux module and policy to enable CW-Lite policies to be defined. Third, we describe the implementation of IMA relevant to this discussion. Finally, we describe the extensions to IMA required to construct PRIMA, such that it can be applied to the CW-Lite SELinux system. Issues related to implementing PRIMA in a practical environment are discussed here.

4.1 SELinux

We apply PRIMA to the SELinux system [2] because it provides a comprehensive MAC policy implementation for Linux. Security-enhanced Linux (SELinux) is a Linux Security Module (LSM) that enforces mandatory access control (MAC) across all user-visible Linux objects for all Linux user processes. SELinux enables control of all system information flows, so it is a logical level at which to integrate information flow integrity measurement. The LSM interface defines where the Linux kernel authorizes user process operations on kernel resources (e.g., files and sockets), and SELinux implements those authorizations. SELinux defines its own MAC policy model, an extended Type Enforcement (TE) model [5]. TE labels subjects and objects as *types* and defines access of subjects to objects to perform operations in the manner of an access matrix policy. In prior work, we have shown how to convert an SELinux TE policy to an information flow policy [15, 16]. Others have also developed SELinux policy information flow analysis tools [8, 7].

The version of SELinux we used is modified to support CW-Lite as described in [21]. The main difference is that a trusted process may transition to and from its filtering subject type. This change does not imply any behaviors other than those present in the program's code.

Figure 3 summarizes the SELinux and application changes required for CW-Lite integrity. More details on the implementation are provided elsewhere [21].

4.2 Integrity Measurement Architecture

The Linux Integrity Measurement Architecture (IMA) measures files at load time to ensure that all loaded code prior to compromise has been measured. Thus, the vulnerable or malicious software or data that resulted in the compromise will be captured in the measurement list. While it may prevent new measurements, a compromised system cannot remove its past measurement without detection.

IMA is implemented as a Linux Security Module (LSM) like SELinux. The two key features are (1) how IMA generates integrity measurements and (2) the information collected in these measurements. This will serve as a basis for determining how to extend IMA to PRIMA.

IMA interposes the two operations in which the kernel loads code: (1) `file_mmap` where the a new program's code as well as dynamic library code is mapped prior to execution and (2) kernel module loads. Other code loading, such as internal code loading (e.g., databases, bash scripts) must be performed by applications. IMA provides an interface for an application to measure code before it is loaded. Further, this interface may also be used for measuring any static data that may be relevant to attestation.

An IMA measurement entry consists of a file name and hash value, as well as additional data to detect if the file has been changed. The file name enables identification of the expected hash value for the entry. The hash value is used in the computation of the TPM's PCR value used to verify the integrity of the measurements themselves.

4.3 PRIMA's Extensions to IMA

PRIMA requires changes to the IMA measurement entry to capture the mapping between code and MAC policy, and it needs to capture the MAC policy loads and other relevant policy specifications.

First, PRIMA must capture the subject label of code when it is loaded in an unforgeable manner. This means that the label must be measured with the associated code. A new measurement, $m_e = H(c||s)$, is taken of a concatenation of the code digest c and the subject type label s (note that the role of the code/data is not used in the prototype). In the prototype, caching of measurements is done at a low level, so separate measurements are currently made for the code and the code-label mapping. The remote party checks these separately. Since the remote party won't know the subject label from the file name, a new field must be added to the IMA measurement entry structure, called `subject`, to indicate the label used for the hash value.

Separate measurements for code and the code-label mapping are not fundamental to PRIMA. We can measure the combination of code and label if we can: (1) enable effective caching to prevent redundant measurements and (2) enable the remote party to verify the measurement in the measurement list. First, there are some cases (e.g., see the shared library problem below) where we do not want to measure the code multiple times even if it is loaded for different subjects, so we need to cache code measurements separately from the mapping. However, we can enable this by caching the measurement entries by the code hash and storing the relevant labels with the measurement entry (small number of labels if relevant). Second, the measurement entry containing the file name, code hash value, and label enables the remote party to verify the code integrity from the hash and build the measurement entry (m_e from Section 3.1) for computing the hash aggregate.

As noted previously, we limit PRIMA measurement only to code or data loaded on behalf of trusted subjects. This results in a conservative analysis because not all untrusted subjects may be run, so not all low integrity flows may be activated by the time of the attestation. However, the PRIMA approach assumes that the MAC policy does not permit trusted subjects to ever have any dependence on low

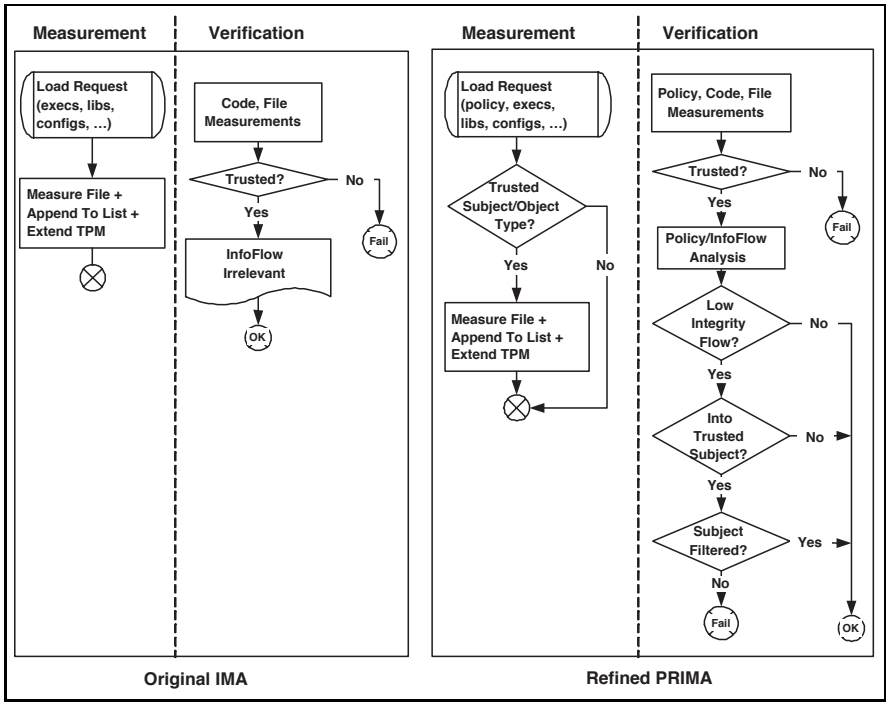

Figure 3: Differences between IMA and PRIMA in measurement and the remote party's analysis of measurements.

integrity information flows. Thus, a static, load time measurement of the MAC policy for the existence of such flows is sufficient. The runtime measurements are to ensure that the code loaded into trusted subjects has sufficient function (i.e., filtering) and is of sufficient integrity (i.e., as previously assumed for a trusted subject in IMA).

We add a new measurement for policy loading. Since this is specific to the LSM that we are using, this hook is added to SELinux's `security_load_policy` function. Unlike executable files, policies are more mutable, so it may be necessary to send the policy with the measurement list. Since SELinux policy files exceed 1MB, this would limit the practicality of PRIMA. Two viable options exist given the current state: (1) the binary policy is standardized, such that its measurement indicates a policy known to remote parties and (2) the entire policy can be measured instead. The tradeoff is efficiency (the first case) vs. flexibility (the second). Note that, for the second case, once a whole policy has been measured and properly analyzed by the verifier, the verifier can cache the policy's digest to avoid large transfers (and analysis) in the future.

The notion of trusted subjects is not explicit in SELinux at present, so this would need to be added to the policy loading process. For example, `init` can be modified to get a `trusted_subjects` file and measure this. We have not modified `init` to do this yet, but the change is straightforward.

Dynamically linked libraries are handled in a special way. They may be loaded in multiple processes, representing different subject types. For efficiency, we do not measure each combination, but rather measure the library only on its first load into a trusted subject. This is because code in untrusted subjects does not impact integrity anyway and because all trusted subjects are considered equal in their integrity impact (i.e., we allow all flows between them).

5. DISCUSSION

In this section, we briefly examine the impact of information flow integrity, in particular CW-Lite integrity, on integrity measurement accuracy and effort.

Untrusted User Code. In a client system, it may be that several user programs are being run along with a particular corporate application. If the SELinux policy enables isolation of the corporate application and the programs it depends on from other user programs, then the corporate server can use PRIMA to attest to the corporate application program on the client. The other user programs do not impact this client program, and the information flow analysis of the SELinux policy will show that.

Untrusted Code in a Trusted Subject. Suppose a trusted subject is tricked into loading a vulnerable version of an application, such as an old version of OpenSSH. Even if there are no information flow problems, PRIMA will enable detection of this problem because all code loaded into a trusted subject is measured. This is no different than IMA.

The loading of untrusted libraries and kernel modules will also be caught. Even though a library is only measured the first time that it is loaded into a trusted subject, we will see this initial load in the PRIMA measurements.

Unknown Code. Suppose a program is loaded on an attesting (i.e., integrity-measuring) server, such as an administrator's script to examine the state of system policies. Since a remote party may not be aware of the program, its presence in an integrity measurement list would likely result in a failure of the computation.

However, with PRIMA, since no target application has a

dependency on this program (i.e., it does not write any files), it need not be run by a trusted subject. Thus, it would not appear in the PRIMA measurement list.

Suppose, however, that the program was run by a trusted subject. Typically, the standard system administrator subject must be trusted. In that case, it would appear in the PRIMA measurement list and cause a false failure. SELinux enables subjects to transition when running a program, so the solution would be to add a new SELinux `type_transition` rule where the administrator subject would transition to an application subject upon executing this program [3].

6. RELATED WORK

The use of a program's load-time hash value to assess its integrity was proposed as part of the Logic of Authentication applied to the Taos operating system [1]. Here, the goal was to justify the identity of the initial system principals, which led to the notion of *secure boot* where a system is not booted unless the hashes of the code loaded meet expected values [3]. Attestation implies a slightly different guarantee, called *authenticated boot*, where it is possible for a remote party to verify the integrity of a system via the code that it loads. As Bill Arbaugh has pointed out, secure boot enables a local party to determine if a system is of high integrity, but not a remote party. On the other hand, a remote party can prove the integrity of system using authenticated boot, but the fact that the system is running does not determine its integrity.

The basic integrity semantics of authenticated boot were defined in the IBM 4758 work [23]: the code loaded must be of high integrity at load time and identifying secrets (e.g, private keys) for the code principal must be protected from leakage. Subsequent mechanisms, such as Next-Generation Secure Computing Base (NGSCB) [10], Terra [13], Linux Integrity Measurement Architecture (IMA) [20], enforced these basic semantics using the cheaper TPM hardware.

The BIND attestation system took a very different view of integrity where the dependency on inputs is made explicit [22]. A measurement consists specifically of inputs and critical code that operates on these inputs. This binds the input dependency with the code that operates on them. However, there are several issues with this approach which remain unproven: (1) Does the combination of inputs and computation required to achieve high integrity encompass nearly the entire application? (2) How are implicit flows, such as those identified by Denning [9], captured? (3) How are known, low integrity inputs handled? The initial experiments show measurement of BOINC components (i.e., process-level components) which is analogous to IMA.

7. CONCLUSION

In this paper, we have shown that an integrity measurement approach based on information flow integrity can be constructed and enables much more accurate integrity verification than existing approaches. Current integrity measurement approaches only measure the code loaded into the system and static data files, so they fundamentally provide load-time guarantees. We have found two key problems with these approaches: (1) the load-time measurements of code alone do not accurately reflect runtime behaviors, such as the use of untrusted network data, and (2) they are inefficient, requiring all measured entities to be known and fully trusted even if they have no impact on the target application. We have developed the *Policy-Reduced Integrity Measurement Architecture*, an extension of the Linux IMA system, that measures information flow integrity guarantees that can be verified by remote parties. PRIMA requires only the additional measurements of MAC policy and trusted subjects at load time and the mapping between code and MAC policy subjects at runtime to resolve both key problems, and a number of measurements are eliminated because there is no longer a need to measure the code of untrusted subjects. We described the PRIMA implementation, its integration with SELinux, and its ability to measure information flow, particularly the CW-Lite integrity property that can be achieved in practice. We also demonstrated how the key problems are resolved through PRIMA measurement. In the future, we will examine specific the specific verifications necessary for some common SELinux systems.

[3]It is somewhat more complex than this due to the use of a script interpreter for the script, but the basic idea is valid.

8. REFERENCES

[1] Martin Abadi, Edward Wobber, Michael Burrows, and Butler Lampson. Authentication in the taos operating system. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 256–269, The Grove Park Inn and Country Club, Asheville, NC, 1993. ACM Press.

[2] National Security Agency. Security-Enhanced Linux. http://www.nsa.gov/selinux/.

[3] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture, 1997.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.

[5] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, Maryland, 1985.

[6] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *In Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, California, April 1987.

[7] MITRE Corporation. MITRE - Security–Enhanced Linux. http://www.mitre.org/tech/selinux/.

[8] Tresys Corporation. SETools Policy Tools for SELinux. http://www.tresys.com/selinux/selinux_policy_tools.shtml.

[9] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[10] Paul England, Butler W. Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *IEEE Computer*, 36(7):55–62, 2003.

[11] H. Maruyama *et al.* Trusted platform on demand. Technical Report RT0564, IBM TRL, 2004.

[12] Timothy Fraser. Lomac: Low water-mark integrity protection for cots environments. In *In Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 230, Washington, DC, USA, 2000. IEEE Computer Society.

[13] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual

machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[14] Trusted Computing Group. Trusted Computing Group: TPM. https://www.trustedcomputinggroup.org/groups/tpm/.

[15] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, pages 59–74. USENIX, August 2003.

[16] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Resolving constraint conflicts. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 105–114, New York, NY, USA, 2004. ACM Press.

[17] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, USA, August 2003.

[18] David Safford. Trusted linux client. http://www.acsa-admin.org/2004/workshop/David-Safford.pdf.

[19] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 308–317, New York, NY, USA, 2004. ACM Press.

[20] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 223–238, 2004.

[21] Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward automated information-flow integrity for security-critical applications. In *In Proceedings of the* 13^{th} *Annual Network and Distributed Systems Security Symposium*. Internet Society, 2006.

[22] Elaine Shi, Adrian Perrig, and Leendert van Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *In Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 154–168, 2005.

[23] Sean W. Smith. Outbound authentication for programmable secure coprocessors. In *ESORICS '02: Proceedings of the 7th European Symposium on Research in Computer Security*, pages 72–89, London, UK, 2002. Springer-Verlag.

[24] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, USA, August 2002.