RC23898 (W0603-030) March 3, 2006
Computer Science

IBM Research Report

PRIMA: Policy-Reduced Integrity Measurement Architecture

Trent Jaeger
Pennsylvania State University

Reiner Sailer
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Umesh Shankar
University of California at Berkeley

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

PRIMA: Policy-Reduced Integrity Measurement Architecture

Trent Jaeger
tjaeger@cse.psu.edu
Pennsylvania State University

Reiner Sailer
sailer@us.ibm.com
IBM T. J. Watson Research Center

Umesh Shankar
ushankar@cs.berkeley.edu
University of California at Berkeley

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.

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 middle-ware 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 so that the trusted application can be corrupted by untrusted programs or users.

Hardware-based integrity measurement has recently 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 it. 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* [21]. 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 [18]. 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 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.

Indeed, historically, the integrity of applications has been evaluated using system information flows. Using informa-

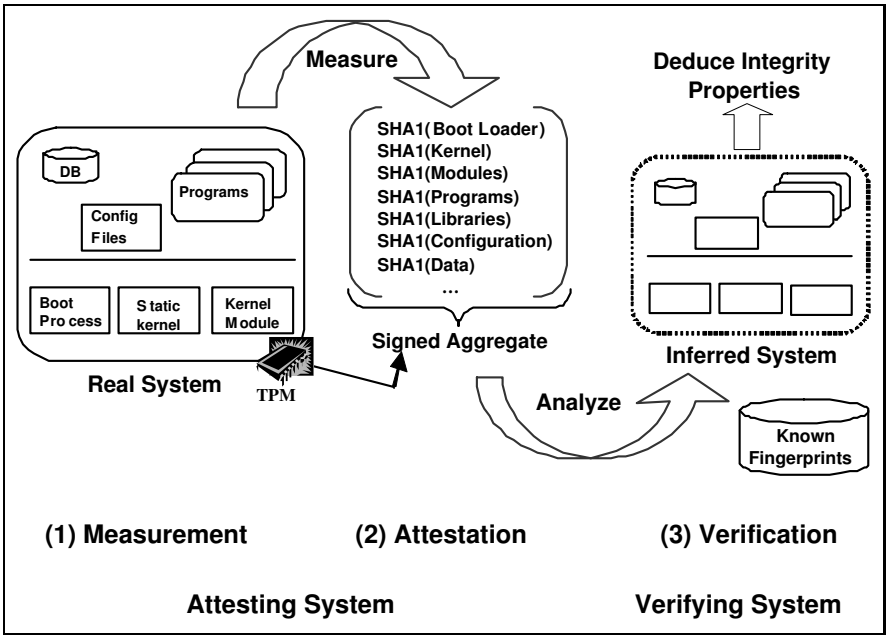

Figure 1: **Linux IMA Overview**: The attesting system generates system measurements (a) into a TPM-backed attestation (b) that a remote party can verify (c).

tion 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 source 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 [19], 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. 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 it 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 Sec-

tion 6, we discuss related work. In Section 7, we outline future work and conclude.

2 Background

In this section, we describe integrity measurement architectures and information flow background necessary to outline what must be done to measure and verify an information flow integrity goal.

2.1 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 IMA [18].

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 [21], Palladium/NGSCB [10], and the Linux Integrity Measurement System [18]. 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 [20].

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 at load time. A load time measurement of a static file indicates that a file of a known value that a process depends upon 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 data are in known, high integrity states when they are loaded on this system.

2.2 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 at load time. The load-time measurement implies that information flows that may result in code injection attacks are 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 data they depend on (e.g., configurations) have been modified maliciously or an untrusted input can impact a previously unknown vulnerability.

Second, any stateful, trusted 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 detection 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 attesting 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.

2.3 Information Flow Integrity and Measurement

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].

accurate and more flexible, but also can be practical in current systems.

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

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 must trust are in the trusted application set. Fortunately, our experience is that there is a specific group of system services that must be trusted by all applications.

This is a reasonable start, but Biba-style models fail to capture a common case: high-integrity processes that much handle low-integrity 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 [17]. In the second case, trusted processes, such as the UNIX services above, can protect themselves from low integrity inputs.

There are two practical problems with applying Clark-Wilson directly to commercial systems: (1) 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

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

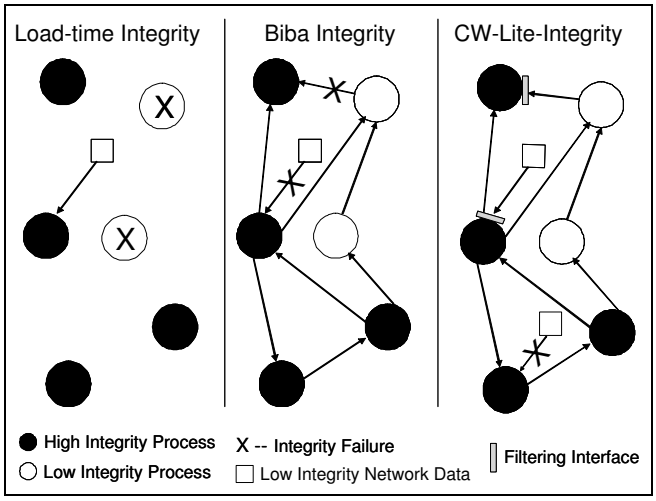

Figure 2: **Integrity semantics of different integrity models**: (1) Load-time measurement fails when any low integrity code is loaded, but ignores the impact of low integrity network data; (2) Biba integrity considers information flows, but fails if any low-to-high integrity flow is present; (3) CW-Lite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

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* [19], 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 would be desirable. The enforcement mechanism restricts access according to Biba integrity for normal interfaces, but permits low integrity information input at filtering interfaces.

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 load integrity inputs do not. In information flow integrity, the input of low integrity code or data invalidate system integrity. In CW-Lite, some low integrity information flows may be accepted by high integrity processes using filtering interfaces.

We now extend the basic information flow verification to support verification of Clark-Wilson Lite by the following additional measurements:

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

be successful.

5. **Filtering Subjects**: The subjects entrusted with filtering their own low integrity inputs must be trusted by the remote party to do so.

6. **Filtering Interfaces**: A filtering subject must only run within the execution of filtering interface code.

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 those subjects that are capable of filtering low integrity inputs to ensure safe handling of low integrity data. Lastly, we must ensure that filtering subjects only access low integrity inputs via filtering interfaces. Note that not all interfaces of a program need to be filtering interfaces, but the filtering subject should only be active for filtering interfaces.

If each component of a distributed application meets these requirements, then the integrity of distributed application as a whole can be guaranteed within the application members and also to external parties that may rely on the application.

3 Policy-Reduced Integrity

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

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.

The attesting system (i.e., the system doing the measurements) must explicitly collect the list of trusted subjects and measure them. This is typically not specified in the MAC policy directly, at least not for Type Enforcement policies like SELinux, so it must be a special measurement.

Trusted Code/Data is the code and static data used by a trusted subject in the system. That is, for each code or static data measurement $m \in M$, functions $S(m)$ and $M(s)$ can be defined that determines a unique subject under which that code or data was loaded and the measurements pertaining to a particular subject, respectively. Typically, this is how a remote party will evaluate trust in a trusted subject. The remote party determines for each trusted subject is the code and static data measurements made for that subject correspond to known and high integrity hash values for that subject. Further, the code should 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.

The attesting system must explicitly measure the mapping of code and static data to the trusted subject under which it is used. Currently, code and static data are measured in integrity measurement systems, so it is the mapping between code/data and subjects that must be added to compute $S(m)$.

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 the attesting system must measure the MAC policy. Various optimizations of this measurement are possible depending on conditions. For example, if MAC policies are standardized, then the remote party can use a standard representation of the information flow graph precomputed for that policy to verify information flow properties.

Initial Verification Procedure (IVP) is the code that is 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 from existing measurements. Further, the IVP code can measure the result of the IVP test in the manner of static data. Thus, no further measurements are necessary to capture the IVP and its results.

Filtering Subjects F is a set of subjects $F \subseteq S$ which are entrusted to handle low integrity inputs for some trusted subjects. The remote party must be aware that such filtering subjects are present to know which code to verify for use of filtering interfaces.

The attesting system measures filtering subjects with the set of subjects in the MAC policy. Such subjects are given special names with *cwl* prepended to indicate mapping. No

further measurements are necessary to capture the filtering subjects.

Filtering Interfaces are in the code of programs running as trusted subjects that have associated filtering subjects. Such code must be trusted to: (1) use the filtering subject only at filtering interfaces and (2) have filtering code that is effective in either discarding or upgrading low integrity inputs as prescribed in the Clark-Wilson integrity model.

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

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

1. For any trusted subject $t \in T \subseteq S$, the code loaded $M(t)$ must have known hashes and be trusted to be high integrity.

2. For each subject $s \in S$, 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 or filtering subjects in $T \cup F$. 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.

3. For the IVP subject $i \in T \subseteq S$, the code loaded $M(i)$ 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.

4. For any filtering subject $f \in F \subseteq S$, the code loading into its corresponding trusted subject $t \in T$, $M(t)$, 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.

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 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. That is, $M(t) = M(f)$ where t is a trusted subject and f is the corresponding filtering subject. However, the presence of a filtering subject for a trusted subject indicates the presence of filtering interfaces in the code previously measured. It is the remote party's responsibility (perhaps delegated to a trusted third party) to determine if the code's hash value justifies the requirements that the filtering subject only be activated at filtering interfaces and that the filtering interface can be trusted to discard or upgrade all low integrity inputs.

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].

4.2 SELinux and Application Changes

SELinux does need to be changed to support CW-Lite subjects, however. Two things need to be done: (1) a new CW-Lite subject needs to be constructed and added to the SELinux MAC policy and (2) SELinux needs to recognize when to use the CW-Lite subject rather than the original subject (i.e., the one that adheres to Biba integrity).

First, since the MAC policy easily supports the addition of a new subject, the addition of a new CW-Lite principal to the MAC policy is a straightforward policy update. The original subject's permission assignments, plus the new low integrity input permissions, are assigned to the new CW-Lite version of the subject. SELinux then permits access when the CW-Lite subject is activated. It is more economical in policy space to have a single subject and its extended CW-Lite rights, because we would not need a redundant copy of the original subject's rights, but we would then have to change the entire SELinux module to use the extended rights without additional performance overhead. Such work is beyond the scope of the prototype.

Second, the SELinux module is modified to recognize transitions between the original and CW-Lite subjects. We add SELinux module calls via `sysfs` where the application can tell SELinux to activate or deactivate the CW-Lite subject. The application is entrusted with the responsibility of deciding when to activate CW-Lite rights as part of the filtering interface. Trust in the filtering interface includes trust in activating the low integrity permissions.

Also, applications must be changed to inform SELinux when the CW-Lite subject is activated and deactivated. The SELinux module calls are wrapped in a macro called `DO_FILTER` that activates the CW-Lite subject, performs a low integrity read, and deactivates the CW-Lite subject. Such filtering as is necessary occurs outside the scope of `DO_FILTER` which is acceptable as the low integrity permissions are no longer required.

Figure 3 summarizes the SELinux and application changes required for CW-Lite integrity. More details on the implementation are provided elsewhere [19].

4.3 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

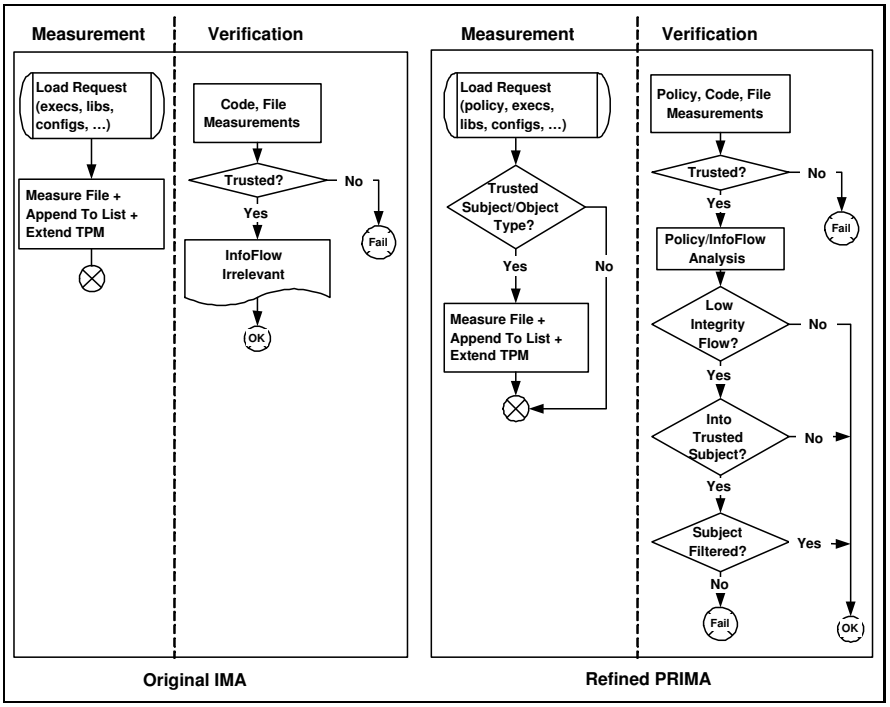

Figure 3: Differences between IMA and PRIMA in measurement and the remote party's analysis of measurements.

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 primarily. Other values are tracked to determine if a measured file is modified. 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.

4.4 PRIMA Extensions

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, $m = H(m_c + t)$, is taken of a concatenation of the code measurement m_c and the subject type label t. Since the remote party won't know the subject label from the file name, a new field must be added to the measurement entry, called `subject`, to indicate the label used for the hash value. Note that a separate measurement for the code is needed for caching (e.g., see the shared library problem below) and for the remote party to verify the code integrity [2].

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 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

[2]Despite the additional measurement per code load, we still believe that the measurement is "policy-reduced" through elimination of unnecessary measurements, but any term may be used as long as resulting acronym is PRIMA.

(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, rather than to Linux in general, 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 source policy can be measured instead. In the first case, the remote party can retrieve the source policy for the binary and perform analysis on that. In the second case, the remote party must trust that the binary and source policies match. We expect that some indirect processing of measurements on the attesting system will be necessary for scalability and privacy reasons, but a formal basis for such trust is not yet present.

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.

We found a problem during the PRIMA implementation. Dynamically linked libraries are loaded in multiple processes, as multiple subjects, but there impact on integrity does not change. We only measure a `.so` file once, when it is loaded in its first process running as a trusted subject.

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 banking client. If the SELinux policy enables isolation of the banking client program and the programs it depends on from other user programs, then the bank server can use PRIMA to attest to the banking client program. 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].

Filtering Inputs Several system services, such as OpenSSH, inetd, etc., accept requests from the Internet, potentially from malicious users. For OpenSSH, Provos *et al.* constructed a privilege-separated version of OpenSSH which accepts only well-formatted requests in permitted orders. Such an interface is an example of a filtering interface, thus enabling permissible access to the network. A remote party could verify that OpenSSH program's filtering interface is acceptable, and that this is the only place where the filtering subject is activated. When PRIMA provides this program measurement for the OpenSSH subject, the remote party could then accept the integrity of this system.

For other programs, such as inetd, not as much effort has gone into a filtering interface, but we envision that an understanding of where security problems are can motivate filtering interface implementation, including software tools to assist in development.

Other Low Integrity Inputs OpenSSH uses temporary files to enable local user data to be input to the OpenSSH server. We are not sure where such a function is valuable,

[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.

but clearly this is a low integrity information flow. PRIMA and the CW-Lite extensions enable two ways to handle this.

First, if access to the temporary file is given to the OpenSSH server's trusted subject, then it can open the file from interfaces other than the filtering interfaces. This could lead to a compromise of the system. The PRIMA policy measurement will include this permission with the trusted subject, and the policy analysis by the remote party will find the low integrity input to a trusted subject. The attestation will fail in this case.

Second, if access to the temporary file is given to the OpenSSH server's filtering subject, then the file can only be opened from filtering interfaces. Thus, any other attempt by the server to access the file will be denied because the trusted subject does not have access. The PRIMA policy measurement will show that the trusted subject does not have the low integrity information flow of the previous case. The filtering interface must be implemented to handle all low integrity inputs.

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 where defined in the IBM 4758 work [21]: 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) [18], 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 [20]. 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.

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 8^{th} 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] David Safford. Trusted linux client. http://www.acsa-admin.org/2004/workshop/David-Safford.pdf.

[18] 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.

[19] 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.

[20] 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.

[21] 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.