SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks **00**: 1–18 (2009)
Published online in Wiley InterScience
(www.interscience.wiley.com) DOI: 10.1002/sec.0000

Protecting the Integrity of Trusted Applications in Mobile Phone Systems

Divya Muthukumaran, Joshua Schiffman, Mohamed Hassan, Anuj Sawani, Vikhyath Rao, Trent Jaeger

Systems and Internet Infrastructure Security Lab
The Pennsylvania State University, University Park, PA 16802

Summary

Mobile phones have evolved into indispensable devices that run many exciting applications that users can download from phone vendor's application stores. However, as it is not practical to fully vet all application code, users may download malware-infected applications, which may steal or modify security-critical data. In this paper, we propose a security architecture for phone systems that protects trusted applications from such downloaded code. Our architecture uses reference monitors in the operating system and user-space services to enforce mandatory access control policies that express an approximation of Clark-Wilson integrity. In addition, we show how we can justify the integrity of mobile phone applications by using the Policy Reduced Integrity Measurement Architecture (PRIMA), which enables a remote party to verify the integrity of applications running on a phone. We have implemented a prototype on the Openmoko Linux Platform, using an SELinux kernel with a PRIMA module and user-space services that leverage the SELinux user-level policy server. We find that the performance of enforcement and integrity measurement is satisfactory, and the SELinux policy can be reduced in size by 90% (although even more reduction should ultimately be possible), enabling practical system integrity with a desirable usability model.

KEY WORDS: Mobile phones, integrity measurement, Clark-Wilson integrity, mandatory access control, reference monitor, SELinux

1. Introduction

Smart-phones have become today's gadgets of necessity. From making calls to checking email, from downloading the latest chart-topper to checking bank balances, they symbolize the convergence of technology in one small, mobile device. Recently, phone vendors have transitioned from highly customized to general purpose operating systems, such as Symbian, Windows Mobile, and Linux, making it easier for third-party developers to build applications. Further, mobile application stores like the Apple iPhone App store [3], BlackBerry App World [28] and Android Market [13] have made the user community dependent on downloading such applications.

However, enabling users to download arbitrary applications presents a threat to security-critical applications on the phone system. Since users almost always have their phones with them, many businesses envision the phone as a client in e-commerce applications, such as mobile banking [6]. These applications may be security-critical in that they must protect the integrity and secrecy of their data (e.g., account balances). There have been several documented cases of malware for phones since 2004 [5, 18, 37], so phone systems must be able to protect security-critical applications from others.

Handset manufacturers are aware of this threat, but current defenses are inadequate. Initially, Linux and

Windows-based phone systems were built assuming that all applications would be trusted, but handset manufacturers are exploring as-yet-unreleased efforts to control individual applications [20, 1]. Symbian phone systems use a hierarchical model [36], similar to Windows Vista*, where the origin of the application determines its permissions. Symbian distinguishes among Symbian, Symbian-signed, and unsigned applications, preventing unsigned applications from modifying the files of Symbian and Symbian-signed applications. However, like Windows Vista, the Symbian model does not prevent a variety of security problems [22]. We have previously shown that a malicious, unsigned application can attack a Symbian phone system by downloading secret files, installing keyloggers, and submitting malicious telephony commands [27]. Security-critical applications and the phone system itself may be compromised by these actions. For example, key banking files may be downloaded to the attacker's system, a user's PIN may be stolen by a keylogger, and unauthorized mobile banking commands may be submitted to the bank by the attacker.

Providing adequate security for security-critical applications on a system with arbitrary downloaded applications is a difficult problem. We identify two main challenges: (1) ensuring that all operations that may access security-critical data are mediated correctly and (2) designing security policies that enforce the necessary protections while still enabling downloaded applications to execute effectively. The Symbian system fails the first case, as only write access to files are mediated, so any data on the phone may be accessed by any application. Further, we have found that phone systems (Symbian and others) leverage many *user-space services* for processing security-critical data, and these services have been implemented under the assumption that all requests are from trusted software. For example, the Symbian windowing server assumes that a request for callbacks on key presses is from a trusted process, enabling the installation of a keylogger of another application.

All phone systems also fail the second requirement, as Symbian's policy is too permissive†, whereas others have not yet found an adequate policy to release in their devices. We note that simply denying untrusted applications the access to user-space services is not an option, as even untrusted applications depend on some operations provided by these services. For example, even untrusted applications require windowing support and may use telephony services for a variety of operations, such as GPRS. Our goal is to mediate all security-critical operations securely while still permitting such normal accesses.

In this paper, we develop a phone system architecture that enables the protection of security-critical applications while running arbitrary downloaded applications. Our insight is that it is possible to build phone systems targeting an approximation of classical integrity models (e.g., Biba [17] and Clark-Wilson [8]), called *CW-Lite* [32]. Using CW-Lite as a guide, we define an architecture where the phone's operating system and user-space services must be built to satisfy the requirements of an access enforcer, called the *reference monitor concept* [2] and where the access policies are verified to protect the integrity of security-critical applications. Finally, we build proofs that our phone systems satisfy CW-Lite integrity. Such proofs can be put into a remote attestation protocol [16] to be verified by remote parties (e.g., banks) to determine whether the client banking application is protected adequately on the phone system.

The result of this effort is an Openmoko phone system running an SELinux operating system and security-enforcing telephony server and installer that enforce CW-Lite integrity, using PRIMA to build proofs of such enforcement. The Openmoko/SELinux/PRIMA phone system enforces a simplified SELinux policy designed specially for the phone system environment that reduces the policy size from the SELinux reference policy by 90%. The resultant Openmoko system adds less than 0.1 ms overhead to the processing of telephony commands and can generate proof statements (hashes) in less than 0.1 seconds for nearly all phone executables. The Openmoko system does not include a root of trust measurement (e.g., MTM [38], TPM [39], or facsimile implemented via curtained memory, such as TrustZone [40]), but can easily be integrated (i.e., PRIMA has been integrated on with such hardware on desktop and server systems).

Specifically, we make the following contributions:

- **Integrity Model for Phone Systems**: We show that the CW-Lite integrity model applies to the protection of security-critical applications on phone systems by constructing an Openmoko phone system that enforces CW-Lite.

*The Symbian access control model predates the introduction of Windows Vista.

†The Symbian policy on some systems allow untrusted applications to modify key system files, such as the Bluetooth pairing database, permitting untrusted devices to upload files unbeknownst to the user [27].

DOI: 10.1002/sec

- **Access Enforcement**: We provide mediation of installation, telephony, and audio subsystem operations that access security-critical data on the phone system. The Openmoko software installer and telephony server `gsmd` are augmented to protect security-critical applications, and we leverage mediation provided by the Linux Security Modules interface [42] and SELinux conditional policy enforcement to control audio access.
- **Mandatory Access Control Policy**: We define a simplified SELinux mandatory access control policy for protecting security-critical applications using the mediation described above. This policy enables enforcement of CW-Lite integrity with a policy 90% smaller than the SELinux reference policy.
- **Integrity Measurement**: We generate a proof that the resultant system satisfies CW-Lite integrity sufficient for creating a remote attestation using trusted computing hardware. We extend the SELinux system to add the Policy-Reduced Integrity Measurement Architecture [16] (PRIMA), an extension of the Linux Integrity Measurement Architecture [31].

This paper is structured as follows. In Section 2, we outline a phone system that runs both security-critical applications and arbitrary downloaded applications. In Section 3, we discuss related work and tie our work to past work in related topics. In Section 4, we define the security architecture for enforcing protection of security-critical applications and their data. In Section 5, we describe the implementation of this architecture, highlighting the contributions listed above. In Section 6, we evaluate the security, performance, and ease-of-configuring our system. Finally, Section 7 describes future work and concludes the paper.

2. Problem Definition

In this section, we first classify the major players in a mobile phone system from a security perspective, and identify their interactions and the potential security problems that stem from these interactions. We then define the security requirements for a solution to address these problems. In general, we expect a solution that ensures protection of the security-critical (trusted) applications in their use of operating system and user-space service resources in a system that also runs untrusted (e.g., downloaded) applications. Finally, we expect that phone systems that achieve

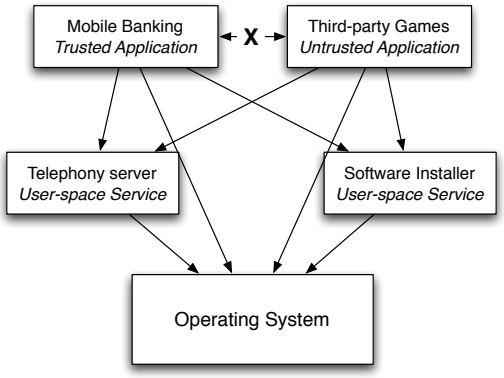

Fig. 1. Mobile phone systems consist of trusted applications, untrusted applications, user-space services that provide function to both trusted and untrusted applications, and the operating system. We prohibit untrusted applications from communicating with trusted applications

such requirements be capable of proving that to remote parties (e.g., the mobile banking client prove its phone system integrity to the bank).

We first classify the entities on the mobile phone system into four categories, as shown in Figure 1:

- **Trusted Applications**: Trusted applications are the applications that are entrusted with the processing of security-critical data. These applications must not receive any untrusted inputs as described below. If one of these applications is compromised, then the phone system is compromised. Such applications can include both pre-installed and third-party applications, such as a mobile banking application. We assume that trusted third-party applications possess a certificate of trust from an acceptable authority.
- **Untrusted Applications**: Untrusted applications are those that are not entrusted with any security-critical data. Such applications may be compromised without compromising the phone system. Such applications include third-party downloaded applications, but may also include pre-installed applications that do not perform security-critical operations.
- **User-Space Services**: Phone systems typically consist of several user-space programs that provide services to other applications. Examples of these include the software installer, the telephony server, windowing server, GPS server, etc. These services cater to both trusted and untrusted applications. Such services are

DOI: 10.1002/sec

also trusted in that if one is compromised, then the phone system is compromised.

- **Operating System**: The phone's operating system (e.g., Linux, Symbian, Windows Mobile, etc.) is also trusted.

These entities can interact as shown in Figure 1, leading to security problems as described below.

Untrusted and Trusted Interactions We prohibit untrusted applications from communicating with trusted applications directly or indirectly. This prohibition protects the secrecy and integrity of trusted applications. From an integrity perspective, such communication is undesirable because if an untrusted game or other malicious application is allowed to modify a file read by trusted application or send an IPC to a trusted application, then it may impact the integrity of that trusted application. From a secrecy perspective, we also prohibit flows from trusted applications to untrusted applications to prevent leakage security-critical data. In general, there is no need for a security-critical application to provide data to an untrusted application.

Untrusted and Service Interaction User-space services perform operations for both the trusted and untrusted applications. For example, an untrusted game may call the telephony server to check battery status or send GPRS data requests to its server, so some interaction with the telephony server must be permitted. In processing such requests, we expect that every user-space service will prohibit operations that would result in an information flow between a trusted and an untrusted application. This means that each user-space service must be able to mediate operations that may access security-critical data and that the service must enforce the expected access policy. In addition, each user-space service must protect itself from requests from untrusted applications, as user-space services are trusted by our trusted applications.

Untrusted and Operating System Interaction As for user-space services, both the trusted and untrusted applications, as well as the user-space services, may request that the operating system perform operations. For instance, the telephony server interacts with the audio subsystem in the OS to play voice from the other end. Even if an untrusted application is prevented from accessing a voice call in the telephony server, it may be able to interfere with voice call via the audio subsystem. To achieve comprehensive protection of trusted applications, it is important to ensure that the operating system mediates access to all security-critical data according to the expected policy. Some OS's provide such mediation (e.g., Linux Security Modules [42]). The challenges are to configure OS access control that supports phone systems and to handle dynamic changes, such as when a voice call terminates we may want to allow an untrusted game to play audio.

To address these concerns, we envision that the phone system will require an *integrity model* that will reflect the requirements above and will enable practical operation (e.g., access to user-space services by untrusted applications). The OS and user-space services must be capable of *enforcing* such a model, including a practical approach to *configure access policies* reflecting the model. Finally, the phone system requires a mechanism to *prove enforcement of the integrity model* to remote parties. We discuss why these are not addressed in current systems below.

3. Related Work

Below, we review related work in integrity models, mandatory access control systems, user-level enforcement, and integrity measurement that is relevant to our problem statement. In general, current integrity models are too strict for phone systems and current access control methods are not used to enforce a verifiable integrity goal. Once we define a verifiable goal, we can design an integrity measurement approach to enable verification by remote parties.

Integrity Models The main problem with classical integrity models, such as Biba [17], LOMAC [12], and Clark-Wilson [8], is their applicability to conventional systems, such as phone systems, that have trusted and untrusted processes. The Biba integrity model [17] assigns integrity labels to processes and relates these labels in an integrity lattice. It follows a "no read down, no write up" model, thereby restricting any information flow from a lower integrity process to one of a higher integrity. Unfortunately, many critical applications, including software installers, read some low integrity data (e.g., requests). To permit critical applications to receive such data, Biba requires a separate, fully-assured guard process to sanitize the inputs, but such programs are not developed for conventional systems. LOMAC [12] requires that a process drop its integrity level to that of the lowest integrity data it reads or executes, but this does not work for some processes, such as the telephony server, which must maintain high integrity after receiving low

DOI: 10.1002/sec

integrity commands. We note that we can use LOMAC for software installers, but not all high integrity processes. Clark-Wilson integrity [8] provides a more flexible alternative, by permitting processes to read low integrity data if they immediately discard or upgrade the data, but Clark-Wilson requires full formal assurance of such process's programs. Such assurance must be performed manually, so Clark-Wilson has not been applied to conventional systems.

Mandatory Access Control As mentioned in the Introduction, the developers of mobile phone systems have recognized that access control is a necessary feature, but current phone systems lack a mandatory access control approach that can be used to ensure enforcement of a precise security goal. Mandatory access control (MAC) enables the system (i.e., administrator or distributor) to define the access control policy for its processes, thus preventing a compromised process from compromising the entire system. While MAC systems originated with Multics, and there have been several implemented since, current examples of such MAC systems include Trusted Solaris, SELinux, and Apparmor (Linux), and variants of SELinux for BSDs and Mac OS X. Most prior MAC systems have focused on enforcing the multilevel security (MLS) policy, which provides verifiable enforcement of secrecy, but integrity protection is handled in an ad hoc manner. Trusted Solaris [35] continues this focus. Novell AppArmor [23] uses LSM hooks to prevent remote network attacks from compromising the system by confining network-facing daemons. However, AppArmor does not prevent attacks from downloaded applications, which is key on phone systems. SELinux is a general MAC system, which can be applied to a variety of security goals. In the past, it has been applied to least privilege (strict policy), MLS, and containment of network-facing daemons along the lines of Apparmor (targeted policy). None of these policies addresses a verifiable integrity policy, but since SELinux is general and comprehensive, we plan to apply it to our integrity goal. The SELinux approach can result in large policies, however. The current SELinux policies are approximately 3MB in size, containing over 2000 types and between 50,000 to 100,000 permission assignments.

User-Level Access Enforcement Until recently, security researchers focused solely on the enforcement provided by the operating system. However, it has become clear that there are several user-space programs that are entrusted with enforcing the system security policy. For example, SELinux/MLS identified 34 programs that are trusted by SElinux to enforce MLS on their execution. Work is underway to extend several programs with a satisfactory reference monitor [2] implementation to justify such trust, such as for Linux user space services gconf [7], XServer [41], etc. Phone systems, however, bring additional programs that deal with security-critical phone operations or user data, and these programs also will need such enforcement. We identify two such phone applications, namely the telephony server `gsmd` and the software installer `opkg`, on the Openmoko platform.

Integrity Measurement The idea of integrity measurement is to measure every event that impacts the integrity of the system, and send a proof of these events to a remote party who determines whether these events result in a high integrity system. Most integrity measurement approaches focus solely on the code run by a system [10] [31] [9] [34], although integrity models tell us that all inputs determine the integrity of a process. Thus, should a high integrity application use low integrity data and be compromised, this would not be reflected by such approaches. Integrity measurement approaches that also measure data usage, such as Bind [33] and Flicker [19], measure only a single computation and its inputs, resulting in significant restrictions in the types of systems that can be attested. We aim to develop integrity measurement for a comprehensive integrity model, enabling attestations for phone systems.

4. Architecture

Figure 2 shows our proposed security architecture layered on the phone system shown in Figure 1. The security architecture consists of the following components:

1. We apply the *CW-Lite integrity model* [32], an approximation of the Clark-Wilson integrity model [8], as our security model. The use of CW-Lite guides the development of access policies (described below) and motivates the requirement for user-space services to use filtering interfaces to securely receive input from untrusted applications. We assume that phone system operating systems also use filtering interfaces to receive input from untrusted applications (not shown).
2. We apply *reference monitors* [2] to enforce access control in user-space services and the

DOI: 10.1002/sec

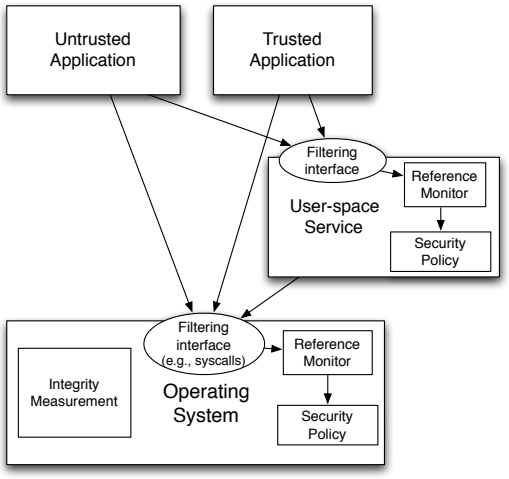

Fig. 2. Our security architecture introduces the following concepts to a mobile phone system: (1) the *CW-Lite integrity* model, which defines our integrity goal (no flow from untrusted to trusted and filtering of untrusted inputs for OS and services); (2) *reference monitoring* in user-space services as well as the OS; (3) *mandatory access control* policies for the user-space services and the OS; and (4) *integrity measurement* in the OS to build proofs of enforcement of CW-Lite integrity.

operating system. We require that any phone system operating system itself have a reference monitor that mediates access to all resources within the system. We also extend reference monitoring into user-space services by adding mediation hooks that control access to security-critical operations. For phone systems, we will show how to add reference monitoring to its installer and telephony server.

3. We design *mandatory access control* (MAC) policies that implement the CW-Lite integrity model for the operating system and user-space services. These policies isolate trusted and untrusted applications in their use of service and OS resources. Where the system only provides a single resource (e.g., audio), MAC policies ensure that no untrusted application may use that resource if any trusted application is. Unlike prior comprehensive MAC policies for practical systems, our policy design aims for simplicity.
4. We provide an *integrity measurement system* inside the operating system to enable the generation of proofs of satisfaction of CW-Lite integrity. We use the Policy-Reduced Integrity Measurement Architecture [16, 21] (PRIMA) as the approach for our integrity measurement system. PRIMA is designed to build CW-Lite proofs, and we examine the efficacy of building such proofs for phone systems.

4.1. CW-Lite Integrity Model

We propose to use the CW-Lite integrity model as the basis for the security architecture of our phone system [32]. CW-Lite is an approximation of the Clark-Wilson integrity model [8] that provides a comprehensive view of integrity that is possible to deploy on commercial systems. Using CW-Lite, we can provide comprehensive and practical integrity protection for the trusted applications, user-space services, and the operating system of a phone system.

It is difficult to ensure the integrity of phone systems (and many other modern systems) because many high integrity components are accessible to low integrity components. This is especially a problem on phone systems where arbitrary applications may be downloaded. In defining a phone systems architecture, we made an explicit distinction between trusted applications and user-space services, which we leverage for defining integrity requirements. As mentioned in Section 2, trusted applications must be isolated from untrusted applications, but user-space services and the operating system provide interfaces that both the untrusted and trusted applications may use. Yet, we depend on the user-space services and operating system to protect themselves from inputs from untrusted applications, as well as enforcing an access control policy that ensures their mutual isolation.

The CW-Lite integrity model approximates the Clark-Wilson integrity model rule that describes how a high integrity process must handle low integrity inputs securely (rule C5 from Clark-Wilson integrity model [8] paraphrased): a high integrity process that reads low integrity data must either upgrade the integrity of that data or discard that data immediately. Clark-Wilson requires full formal assurance of such high integrity processes, but such assurance has not become practical. CW-Lite aims for the spirit of this integrity rule, given the practical situation.

Definition 4.1 *CW-Lite integrity is preserved for a subject s if: (1) all high integrity objects meet integrity requirements initially (i.e., Clark-Wilson* integrity verification procedure requirement*); (2) all trusted code is identifiable as high integrity (e.g., from its hash value); and (3) all information flows from subjects of lower integrity x to a specific interface I are filtered:*

DOI: 10.1002/sec

$$flow(x, s, I) \wedge \neg filter(s, I) \rightarrow (int(x) \geq int(s))$$

where: (1) $flow(x, s, I)$ defines an information flow from x to s through interface I; (2) $filter(s, I)$ implies that subject s filters input at interface I; and (3) $int(x)$ and $int(s)$ are the integrity levels of x and s, respectively.

Our previous experiments have shown that high integrity processes receive untrusted inputs through only a small number of interfaces [32]. The idea is that these high integrity processes make these few interfaces into filtering interfaces, so these interfaces can be designed to satisfy the requirement of immediate upgrading or discarding of data. Thus, CW-Lite makes integrity comprehensive in practical systems, and motivates system implementors to focus on the risk areas, where untrusted inputs are provided. We envision that a practical, comprehensive view of integrity with known risks is far superior to an incomplete view that lacks knowledge of risks (i.e., the current situation).

The CW-Lite model satisfies the security requirements we desire for phone systems. First, the CW-Lite model ensures that there is no flow of data from untrusted to trusted applications: trusted applications have no filtering interfaces, so they cannot legally receive any untrusted inputs [‡]. Second, CW-Lite ensures that even services that accept untrusted input must only do so via filtering interfaces that protect their integrity. Third, CW-Lite ensures that the operating system must also protect itself from untrusted inputs (e.g., via system calls). We assume that operating system filtering is already present. In this paper, we focus on some specific applications and services and show how policies over these can be designed to satisfy CW-Lite.

4.2. Enforcement via Reference Monitoring

We require that the phone systems enforce CW-Lite integrity. To do this, all the mechanisms that may provide untrusted applications with access to the security-critical data of trusted applications must be controlled according to the CW-Lite integrity model. We claim that placing *reference monitoring* in user-space services and the operating system is necessary to provide proper enforcement of CW-Lite.

[‡]CW-Lite does not enforce secrecy, but we will also prevent flow from trusted applications to untrusted applications in the design of our policies.

The *reference monitor concept* [2] defines requirements for enforcing an access control policy correctly. First, access control enforcement must provide *complete mediation*. In our case, all the OS and user-space service operations that enable access to security-critical data must be mediated. Second, access control enforcement must be *tamperproof*, so the OS and user-space services must protect themselves from untrusted application requests. Also, the OS and user-space services must protect their access control policies. *Mandatory access control* policies [11], which are only configurable by the system, prevent tampering by applications. Third, the overall system must be *simple enough to prove correct* according to the integrity model. For this, we must verify that the enforcement mechanism is correct and that the MAC policy enforces the intended security goal. In our case, the integrity model will drive the design of the MAC policy.

For a phone system, we leverage SELinux as the reference monitor for the operating system and add reference monitoring to two phone system-specific entities, the installer and telephony server. The SELinux community has made significant progress on extending a number of general-purpose, user-space services with reference monitoring, such as the windowing server [41], configuration server [7], and interprocess communication server [15]. Thus, we will focus on the phone-specific services. To achieve reference monitor guarantees, we will perform the following tasks. First, we will extend an installer and a telephony server with reference monitor interfaces for which we can validate complete mediation. Second, we will verify that the deployment and execution of such services are protected from tampering through design of the MAC policy (more below). Finally, we will verify that the code of the two services satisfies a correctness property and the MAC policies satisfy CW-Lite (more below). Verifying that a program meets a general property is intractable, but we design our to services in such a way that we can claim they are correct if SELinux is correct.

4.3. Mandatory Access Control Policy

Once we have mediation, we need the reference monitoring to enforce a policy that satisfies CW-Lite. We note that we require a mandatory access control (MAC) policy to ensure that applications may not tamper with the policy (i.e., to meet the tamperproof requirement of a reference monitor). The policy must: (1) ensure that the untrusted applications cannot access trusted application data;

DOI: 10.1002/sec

(2) enable correctly synchronized access when the system provides only a single object (e.g., audio); and (3) ensure that untrusted applications can only communicate with user-space services via filtering interfaces. The resulting policy should also be much simpler than that of a comprehensive MAC system for commercial deployment (i.e., significantly simpler than the SELinux reference policy).

We propose a MAC policy consisting of three labels: `trusted`, `untrusted`, and `cw_trusted`. All trusted applications run under the `trusted` label, and all the data that they generate inherits that label[§]. Analogously, all untrusted applications run with the `untrusted` label. The `cw_trusted` label applies to the user-space services that use filtering interfaces to protect themselves from untrusted input. There are two ways that such interfaces may be used. First, a service may switch to the `cw_trusted` label prior to accessing any untrusted inputs, signaling its intention to the OS [32]. The OS then would know that the application intends to filter any input it receives while it runs under that label. A less intrusive alternative is to label the service process as `cw_trusted` for its entire run, but the service must guarantee that it always accesses low integrity data via filtering interfaces. More code must be trusted in the second case, as the application would have to determine the label of all data it accesses, preventing the access of low integrity data unless a filtering interface is used.

The resultant policy satisfies CW-Lite integrity, as trusted applications cannot access untrusted data (and vice versa), and services can only accept untrusted input via filtering interfaces. This policy model is much simpler than commercial MAC policies that enforce integrity. In these policies (e.g., SELinux [24]), each service and application would have its own label and associated policy rules. This results in many thousands of rules in the SELinux reference policy. As the phone system integrity depends on all its trusted applications, this is directly specified in this labeling, resulting in a significant simplification of MAC policy while providing comprehensive integrity.

4.4. Integrity Proofs

Finally, we want to prove that the phone system satisfies CW-Lite integrity to remote parties. Our architecture uses an integrity measurement system module located in the operating system to build such proofs. The integrity measurement system must ensure that a valid proof is only possible if a phone system actually satisfies CW-Lite integrity. It turns out that such proofs are only slightly different than those that build proofs for code integrity only [14, 9].

We use the Policy-Reduced Integrity Measurement Architecture [16] (PRIMA) to build proofs of CW-Lite integrity. PRIMA justifies CW-Lite integrity by identifying: (1) the *trusted labels* for the subjects and objects that are trusted in a system (e.g., `trusted` and `cw_trusted`); (2) the *code that runs under trusted labels* (e.g., based on the code hash); and (3) the *MAC policy* (described above) that defines the information flows in the system. With the first two sets of measurements, a remote party can verify that trusted applications and user-space services run trusted code. With the third measurement, a remote party can verify that these processes can only perform operations that result in information flows that adhere to CW-Lite integrity. Note that we do not need to measure the runtime use of filtering interfaces. The code measurement must justify that each service only accepts untrusted input via a filtering interface and that the filtering interface is acceptable. We are researching declarative specifications for filtering interfaces to ease verification.

Unlike the original PRIMA prototype [16], the MAC policies of the user-space services as well as the operating system must be measured. Fortunately, the only significant differences between the two are the types of objects and operations. These can be normalized to information flow operations, as described in the implementation.

5. Implementation

In this section, we describe the implementation of our proposed architecture on the Openmoko 2007.2 platform of the Openmoko-based Neo1973 phone [26] which uses a Samsung S3C2410 processor. Openmoko uses a Linux operating system with drivers customized for the mobile phone, and all the software on the phone is open-source. It runs a Linux 2.6 kernel, but SELinux is not enabled in the Openmoko distribution. We enabled SELinux, and designed a CW-Lite SELinux policy for the phone. We will describe our modifications to the installer (`opkg`) and telephony (`gsmd`) user-space services on the OM 2007.2 version of Openmoko to implement CW-Lite integrity. The Linux kernel

[§]If the resulting trusted computing base becomes too large, we could also separate trusted system processing from trusted phone application processing into two labels (where the system is higher integrity than applications), but that not become necessary yet.

DOI: 10.1002/sec

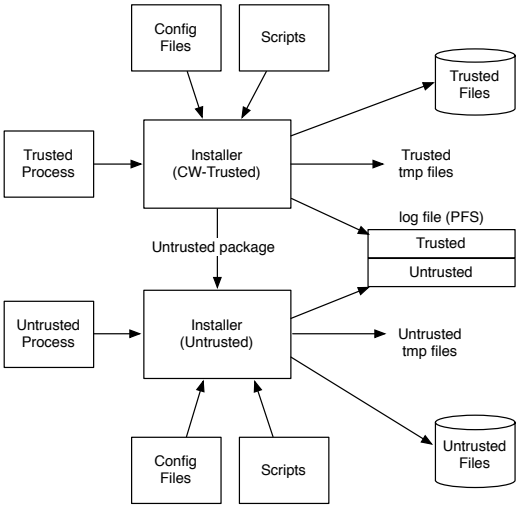

Fig. 3. The software installation process

does not yet support integrity measurement by default, but we leveraged the Linux Integrity Measurement Architecture patch to construct a PRIMA kernel library that was integrated with SELinux. While we have implemented our solution on the Openmoko Platform, it should apply to any Linux phone system, and CW-Lite is an appropriate model for guiding configuration in general.

5.1. Implementing the CW-Lite Model

5.1.1. SELinux

In order to enforce mandatory access control on the phone, we chose to use SELinux, a kernel module that implements the Linux Security Modules framework (LSM). The LSM framework consists of a set of hooks placed inside the kernel to mediate access to kernel objects. When a hook is invoked, the installed LSM (e.g., SELinux) is invoked to authorize the operation defined by the hook for the caller. There is a security server which is responsible for making access decisions using the encapsulated security policy. SELinux enforces a mandatory access control policy based on an extended Type Enforcement model [4]. The traditional TE model has subject types (e.g., processes) and object types (e.g., sockets), and access control is represented by the permissions of the subject types to the object types. All objects are an instance of a particular class (i.e., data type), which has its own set of operations. An SELinux permission associates a type, a class, and an operation set (a subset of the class's operations). Permissions are assigned to subject types using an `allow` statement.

5.1.2. Specifying a CW-Lite SELinux policy

We modified the Openmoko Linux 2.6 kernel to enable SELinux support. Normally, SELinux policies are very complex and involve thousands of types and even more access rules between those types, but our classification based on four application types in Section 2 provided us with the means to simplify the policy. We specify a coarse grained policy with four SELinux types: `trusted_t` for trusted applications, `untrusted_t` for untrusted applications, `kernel_t` for the OS, and `cwtrusted_t` for user-space services, so named because of their CW-Lite filtering.

5.1.3. CW-Lite for the Software Installer

In order to demonstrate the use of CW-Lite on the phone, we consider the OM 2007.2 software installer, `opkg`, as an example of a service that needs a filtering interface to handle untrusted input. The modified software installation process is shown in Figure 3. The software installer is entrusted with the task of installing and updating software packages. As a result, it ends up being the entrypoint for all new software on the system, so the integrity of the system depends on the integrity of the software installer. Any application can invoke the installer (`trusted_t` or `untrusted_t`), but only a trusted installer can install package files that are labeled either `trusted_t`[¶]. However, a trusted installer may also be invoked to install untrusted packages, resulting in the installer process receiving untrusted input that it must filter. A filtering interface, placed at the point where the installer accepts a package request is provided to protect the installer. When a process invokes the installer, the following sequence of actions are taken:

1. Either an `untrusted_t` or `trusted_t` process can execute an `opkg` install. We describe the two cases. First, when invoked from an untrusted process, `opkg` runs as `untrusted_t` (i.e., no transition). Second, when invoked from a trusted process, `opkg` runs as a user-space service under `cw_trusted_t`, as it can install either trusted or untrusted packages. Of course, the installer must be trusted to filter the untrusted input of an untrusted package. A combination of SELinux rules enables the resultant process to run under

¶We assume that an orthogonal tool has saved a package file, labeling it `trusted` or `untrusted` based on its requirements (e.g., signed package).

DOI: 10.1002/sec

the appropriate SELinux type depending on the situation, as shown below.

```
allow {trusted_t untrusted_t}
    priv_cw_exec_t:file {
    getattr exec };
type_transition trusted_t
    priv_cw_exec_t:process
    cw_trusted_t;
allow trusted_t priv_cw_exec_t:
    process transition;
allow cw_trusted_t
    priv_cw_exec_t:file
    entrypoint;
allow cw_trusted_t {untrusted_t
     trusted_t}:file { read };
```

The first rule states that `trusted_t` and `untrusted_t` processes can execute `opkg`, a `priv_cw_exec_t` file. However, only the trusted process can cause a transition of the resulting process to `cw_trusted_t` (i.e., an escalation of permissions to accept untrusted input) via the remaining rules. The second rule states that when a trusted process executes a `priv_cw_exec_t` file, the resultant process should attempt to transition to `cw_trusted_t`. The third rule authorizes the newly executed process's transition from `trusted_t` when executing a `priv_cw_exec_t` file. The fourth rule allows the `cw_trusted_t` process to execute the `opkg`. Finally, the fifth rule permits the `cw_trusted_t` installer to access trusted or untrusted package files.

2. The installer must filter the input arguments provided by the calling process, as it may be run at `cw_trusted_t` and its input package may be untrusted. Fortunately, `opkg` has a relatively simple interface, enabling specification of the operation (e.g., install, remove, update, etc.) and the package file. The installer knows it is filtering based on its SELinux type, `cw_trusted_t`. This filtering interface allows the installer to determine the label of the package to be installed, and reduce its privileges when installing an untrusted package by changing its subject to `untrusted_t`. This ensures that no trusted files are modified by the installation of an untrusted package. We achieve the dynamic transition using a function called `setcon` in SELinux.

The integrity of the installer is protected by its filtering interface, so any process can invoke it. The integrity of the installation process is protected by the installer lowering its privileges for the installation of untrusted packages, preventing the modification of trusted files when an untrusted package is installed.

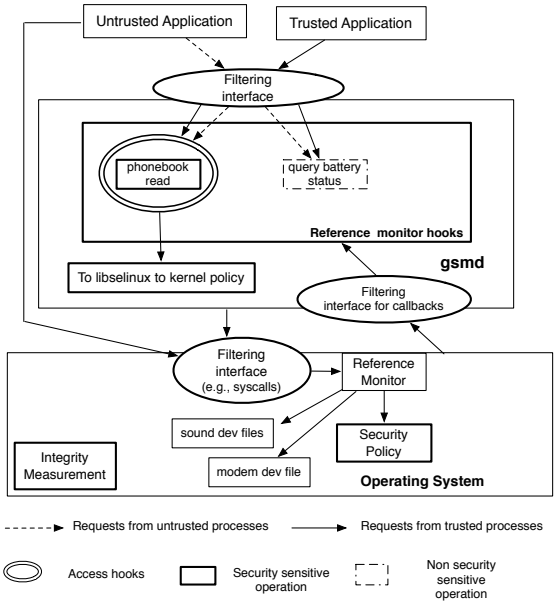

Fig. 4. This diagram shows the enforcement components in the `gsmd` and the operating system

5.2. User-Space Reference Monitors

To demonstrate how we implement a complete reference monitor into a legacy user-space service, we consider the telephony server on the Openmoko system, called the GSM daemon or `gsmd`. The architecture of the resulting enforcement is shown in Figure 4.

5.2.1. GSM Daemon Processing

The GSM daemon enables applications to use phone capabilities implemented by the GSM modem. User applications typically contact the GSM daemon using the `libgsmd` library, which is an abstraction layer that wraps the instruction protocol with a simple API. There is also a `gsmd` command shell tool that gives the user interactive control over the daemon and is used for testing and debugging. In all cases, requests are submitted over a UNIX domain socket.

A wide range of functionality is accessible through the GSM daemon, including the ability to dial and answer calls, toggle the phone's vibration mode, detect signal strength, read and send SMS messages, and retrieve the phone's battery status. Recall that both

DOI: 10.1002/sec

trusted and untrusted applications may use the GSM daemon's operations, where some operations, such as voice calling, are security-critical while others, such as checking phone battery status, are not.

5.2.2. Hook Placement Considerations

Based on the architecture of the `gsmd` code, the following approach was used to design a reference monitor interface. We use the term *hook* to refer to each function in the interface where an authorization query is submitted to be evaluated against the `gsmd` authorization policy (see Section 5.3.3).

Operation Identification Individual operations are identified by the payload processing function (e.g., the `gsmd` function `usock_rcv_voicecall`) and subtype flag (e.g., `GSMD_VOICECALL_DIAL`). Hooks are placed once these are identified. Note that the same AT command may implement multiple distinct operations. For example, the AT command `AT+COPS 0` automatically registers the phone with the network, but `AT+COPS 2` de-registers the phone. Therefore, we cannot place hooks based on the specific AT command. A simple call graph analysis shows that each combination of payload processing function and subtype flag maps to unique AT command submission, so we place the hooks after these are known.

Authorization Hook Placement A hook is placed when the subject and target object of the operation are both known. The subject must be identified from the process that submitted the UNIX domain socket request. The object identity is command-specific. We further note that the binding of the subject and object values must not change after authorization because these values must be submitted to the modem. Subjects are never changed during request processing, and only a few operations use distinct objects. Manual verification was sufficient to determine that this condition holds.

Callback Operations Some requests register a client to receive asynchronous callbacks. Hooks are also placed in callback processing to ensure that the designated client is authorized to receive the response. For example, a client can set a callback to receive notification of signal quality changes. When such an event occurs, the modem receives a response from the network, looks up the corresponding command data structure for the original request, and processes this data structure to determine the client of the response. Some of these responses may contain sensitive data, such as a call status change, change in the network operators, and a cipher status change. Prior to delivering the response, a hook is placed to authorize the response being sent to that client.

5.3. Policy Specification

In this section, we specify MAC policies for both the `gsmd` and the system. Since an SELinux policy is specified in terms of subjects accessing objects, we have to identify what the subjects and objects are within the `gsmd`.

5.3.1. Identifying Objects in the GSM Daemon

We have made a distinction between *global* and *heap* objects in the GSM Daemon. Global objects represent the collection of data created initially by GSM Daemon and shared among all subjects, if authorized. An example is the phonebook object, which is a single object that any subject may request. Unlike subjects, objects are given a finer-grained labeling, so each global object is given its own SELinux type. Heap objects are those that are created dynamically based on requests from specific subjects. An example of a heap object is a phone call. Heap objects are given the label of the client that created them. For example, a phone call object is labeled based on the client process providing the request. When the client wishes to terminate a call, it provides the object identifier, but can only terminate the call if its label is authorized to terminate calls of the call creator's label.

5.3.2. Identifying Subjects in the GSM Daemon

Subjects correspond to the clients of telephony requests to the GSM Daemon. The GSM Daemon listens on a master socket and, once a handshake is established, a callback is invoked to setup a user data structure. This structure holds information about the client, such as the socket file handler and commands masks. In addition to these fields, we have added fields to store the SELinux security information (i.e., SELinux security identifier (SID) and the SELinux security context, which includes the SELinux type of the process) for the client. We also store process state for the client, which is used to log information regarding its accesses for controlling usage of resources.

5.3.3. GSM Daemon Policy

We assign subjects to SELinux subject types, such as `trusted_t` and `untrusted_t` to distinguish between the trusted and untrusted applications. Objects are assigned to object types. For global

DOI: 10.1002/sec

objects, we assign object types based on their usage. For example, we use `phonebook_t` for phone books. For heap objects, we assign object types based on the subject type of the client that created the object. The GSM Daemon policy defines which operations `untrusted_t` may execute safely (i.e., without causing an information flow to a trusted application).

The GSM Daemon policy server uses the interface provided by `libselinux` to query access decisions from the kernel's policy server. That is, each policy request results in a kernel trap that retrieves the authorization result. A recent patch caches such requests in the user-space `libselinux`, but we do not use this patch yet.

Each application that uses SELinux is required to initialize its own policy and pass handlers for auditing and thread management. We make our audit handler forward audit messages to the `gsmd` logger.

5.3.4. GSM Daemon Enforcement

In this section, we assess how the implementation described above protects trusted applications and the `gsmd` telephony server in the processing of telephony requests. Note that the GSM Daemon runs as a CW-Lite type in SELinux, `cw_trusted_t`.

First, untrusted applications cannot access trusted application data in the modified GSM Daemon, due to the GSM Daemon's internal policy. We only give untrusted applications read access to non-sensitive GSM global objects (e.g., battery status), so the untrusted applications cannot impact the integrity of the GSM Daemon for trusted applications. Also, we only allow untrusted applications to access untrusted heap objects.

Second, untrusted applications can only communicate with the GSM Daemon via filtering interfaces. The only interface for untrusted processes to access the GSM Daemon is via UNIX domain sockets, so we ensure that the interface correctly checks the format of telephony requests. All other interfaces check that they only read trusted input (e.g., files labeled `trusted_t`).

5.3.5. SELinux System Enforcement

To ensure protection of the integrity of GSM Daemon operations, we must also consider the possibility of other processes attacking the system resources used by the GSM Daemon. The GSM Daemon depends on a communication channel to the GSM modem to submit requests and the audio subsystem to play voice audio. If an untrusted process can control either of these objects when the GSM Daemon is processing trusted application requests, an untrusted application may be able to compromise the integrity of the trusted process (e.g., by playing audio controlled by an attacker).

In Openmoko, the GSM Daemon submits GSM modem requests by writing to the device file for the modem, `/dev/ttySAC0`. In a typical UNIX system, untrusted processes can write to that file. However, we only want the GSM Daemon to write to this file on our system. To limit access, we assigned this device file an SELinux type of `trusted_t`. This permits any trusted process to write the GSM modem, but we assume that trusted processes will use the GSM Daemon for such actions. The result is that no untrusted application can submit AT commands directly to the GSM modem, prevent circumvention of the GSM Daemon authorizations.

Also, the audio subsystem is normally accessible to both trusted and untrusted applications, so we need to ensure that when a trusted application is using audio for a phone call, no untrusted applications can play audio. The audio subsystem runs in the Linux kernel and supports software mixing which allows sound from more than one application to be played at the same time. Normally, any application can send sound to the audio subsystem. We verified this on the Openmoko phone submitting an audio file to the sound device file `/dev/snd`. When a voice call is active, sound files are provided by the GSM Daemon, but an attacker may be able to slip its own sound file between actual voice call sound files, compromising voice communications on the phone. Mediating the audio subsystem requires us to use a feature of SELinux called Conditional Policy Extensions [25]. We create a boolean variable in the SELinux policy called `untrustedaudio` to correspond to whether untrusted applications are allowed to play audio or not. The access rules are then specified as follows:

```
bool untrustedaudio true
If untrustedaudio {Rules to allow
    untrusted access to audio
    subsystem}
```

This indicates that untrusted applications will be allowed access only if `untrustedaudio` is `true`. When a trusted application wants exclusive access it sets this boolean to `false` and so untrusted applications can no longer access the sound subsystem. To control this, we enable the GSM Daemon to toggle this boolean when a trusted application initiates a phone call. The GSM Daemon invokes the SELinux Policy Management Server to

DOI: 10.1002/sec

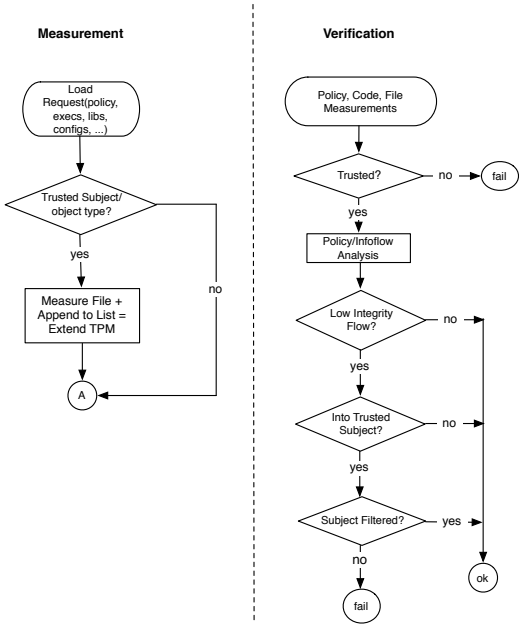

Fig. 5. The PRIMA integrity measurement process

change the value of policy booleans[‖]. When the phone call completes, the GSM Daemon resets the boolean. The result is that while a phone call from a trusted application is active, no untrusted process can play sound.

5.4. Integrity Measurement

In this section, we discuss how we use integrity measurement to build a proof that the resultant phone system enforces CW-Lite integrity. We use the Policy Reduced Integrity Measurement Architecture (PRIMA) integrity measurement approach described in Section 4.4 and shown in Figure 5.

PRIMA functions like a Linux kernel library that can be called from SELinux. In order to integrate integrity measurement functionality into SELinux, Linux IMA callback functions were added after each of SELinux's authorization hooks, so they are invoked only if SELinux authorizes the policy decision. There is a list of trusted subjects in PRIMA. This list specifies the SELinux subject types that run trusted software only. Prior to the loading of this list, all of the subjects that are part of the early boot phase are trusted. A `sysfs` file `/selinux/ts_load` was created to load and view the trusted label list. Measurement is done as follows:

- `file_mmap` was modified to obtain the subject of the current context. Whenever a code that is memory mapped as an executable is called, PRIMA checks the trusted label list for a match with the subject of the current context. On finding a match, the code is measured.
- PRIMA then measures the concatenation of the code hash above with the SELinux subject type.
- To measure the SElinux policy, PRIMA provides another `sysfs` file `/selinux/measurereq`, which enables us to pass a pointer to a MAC policy. We measure an information flow representation of the policy, generated as specified in Section 5.4.1.
- PRIMA also measures the aggregate of pre-kernel integrity measurements generated during PRIMA's initialization. Because the policy is not available at this point, no subject binding measurement is done.
- PRIMA also performs measurement, including the subject binding, whenever a kernel module is loaded into the kernel.

5.4.1. Building the Information Flow Graph

We created a program that parses the SELinux policy binary to extract information flows. The aim is to simplify verification for the remote party. We want to identify what flows are possible from and to the different types, especially the trusted types, that are specified in the policy. This is done in the following manner:

- We first extract the mapping between type numbers and their string representations - Type_number:Type_name map
- We build a hash table of *access vectors* (AV). These access vectors represent the `allow` rules that define access rights in the system.
- We run through the hashtable and identify whether the permissions between the source and target type correspond to reads or writes or both and this defines the flow between them. We collect this in an information flow graph after mapping the subject and object type numbers to the names using the map will built in the first step. An entry in the information flow graph looks like this

  `trusted_t untrusted_t 2`

  where trusted_t is the source type, untrusted_t is the target type and 2 indicates a flow only from the subject to the object.

[‖] Only trusted processes can contact the Policy Management Server.

5.4.2. `init` Modification

PRIMA requires the following modifications to init:

- `init` needs to load the trusted label list before loading the policy. First the contents of the trusted label list stored in `/etc/selinux/trusted` is written to `measurereq` for measurement, and then written to `/selinux/ts_load` to load into kernel memory.
- A call to generate the information flows is also added to `init` and the graph generated is passed to `measurereq` for measurement.

6. Evaluation

In this section, we evaluate the implementation of our security architecture on the Openmoko 2007.2 platform of the Openmoko-based Neo1973 phone [26]. We perform the evaluation in three steps:

- We evaluate the security of the resultant Openmoko system in terms of *reference monitor guarantees* [2] and show how our system achieves these guarantees.
- We evaluate the function and performance of the resultant Openmoko phone system and show that the performance overhead is negligible.
- We evaluate the use of PRIMA to generate integrity proofs by showing that such proofs are practical to build and incur a modest overhead on executable and library loads.

6.1. Security Evaluation

In our system, we leverage an existing SELinux implementation and add reference monitoring to some key services, such as the telephony server and installer, to enforce comprehensive mandatory access control over phone system use. An implementation that enforces mandatory access control must satisfy *reference monitor guarantees* [2]. Below, we examine whether the resultant system satisfies these three guarantees in turn.

Complete Mediation This guarantee requires that all security-sensitive operations run in the system be mediated to enforce access control. To ensure this, we need to show that for the applications we are securing, we have added access hooks to all security-sensitive operations they can execute. Within the `gsmd`, all requests are converted to AT commands based on the payload processing function and subflag, so we added security hooks for each security-sensitive combination. The `gsmd` also receives callbacks for requests that must be delivered to authorized receivers, so mediation of callbacks is also necessary. Fortunately, callbacks are filtered through a single function that also identifies the possible operations. Some broadcasts are for specific clients, so we have to save context for these callbacks. There is only one call context allowed on the Openmoko phone, so it is trivial to match the context to the callback. The software installer application is even simpler, in that it accepts untrusted input from only a couple of interfaces (install and update), so we added filtering code to mediate that interface by detecting whether the operation is for trusted or untrusted code.

Tamperproof Tamperproof means that the code and data (including policy) of the enforcing OS and user-space services cannot be modified by untrusted parties. Tampering can take place at rest (e.g., through malicious modification of files) or at runtime (e.g., through malicious inputs). For example, if an untrusted process can modify the `gsmd` executable, configuration, or policy files, then the reference monitor in the `gsmd` can be tampered. On the other hand, runtime tamperproofing involves validation that all access to untrusted processing is filtered, that is, verification of CW-Lite integrity. As this is the same requirement as for any trusted processing, we verify that our enforcing OS and user-space services are protected at rest at this stage, and below we verify that all trusted processing is protected by a policy that satisfies CW-Lite integrity.

To verify that a trusted program cannot be tampered at rest, we use a technique that we previously have developed [30]. In this approach, we use program packages to define the set of files in a program**, and verify that only trusted processes can modify such files. To implement this verification, we built a policy analysis tool, called PALMS [29], which converts SELinux policy to a Prolog representation that is queried to identify any information flows that violate Biba integrity. We apply PALMS by determining whether any untrusted process (i.e., process other than the installer, kernel, and package process itself) can modify any file once installed from the package (i.e., `opkg` packages for Openmoko). No access that violates this tamperproofing requirement was found for `gsmd`. For the installer, we had to take a slightly

**System libraries are not included in packages, but they must also be protected from tampering. We verify the protection of these files separately.

DOI: 10.1002/sec

different route, as the installer does not have a package definition (i.e., a chicken-and-egg problem, since it needs an installer). Instead, we used `strace` to determine the files loaded when the installer is built and installed on an Openmoko system. Based on this set of files, we apply PALMS, and once again found that the tamperproofing requirement was met.

Verifiability The reference monitor must be simple enough to verify that: (1) its code correctly enforces mandatory access control and (2) that the policy is correct relative to the system security goal, CW-Lite integrity.

Verifying that a program is correctly implemented is not yet a practical undertaking for conventional systems, so we verify a more modest property, showing that the enforcement of mandatory access control is dependent on the correctness of SELinux[††]. In the case of `gsmd`, the `opkg` installer, and the audio subsystem, we simply added SELinux enforcement (e.g., calls to the SELinux enforcement mechanism via `avc_has_perm`) at the necessary filtering and mediation points, so indeed, the system's correctness depends on SELinux.

To verify policy correctness, we once again apply our PALMS policy analysis tool. In this case, the verification requirement is CW-Lite integrity, where trusted processes (e.g., those installed as trusted by the installer) are protected via Biba integrity and that user-space services (e.g., the telephony server and installer) that receive input from untrusted processes be capable of filtering that input.

In designing the SELinux policy for the phone system, our aim was to reduce the policy size and complexity significantly. The key means for reducing the policy size was the use of four subject types, representing the four types of system components: trusted applications (`trusted_t`), untrusted applications (`untrusted_t`), user-space services (`cw_trusted_t`), and the operating system (`kernel_t`). However, while configuring the policy for this experiment, we found that there were many dependencies between SELinux types in the policy, which required us to retain some types in the existing reference policy. While we were able to compile a policy of 100 types, other SELinux types, particularly types for device objects, appear to be necessary for the system to function properly. We are still experimenting with the policy to see how many of these types can be eliminated, but currently, our policy has approximately 700 SELinux types. This is still a significant improvement over the 2000 types that are in the SELinux reference policy. In particular, the SELinux policy binary is reduced from 3MB to less than 300KB, resulting in greater than a 90% reduction in policy size that made it easy to verify.

We then applied the PALMS tool to verify CW-Lite integrity for the resulting policy. With the addition of these other SELinux types to the policy, more types needed to be considered for trusted applications and user-space services. First, the types for the initial process (`init_t`) and some system processes (`logrotate`, `ifconfig`, and SELinux types, such as `setfiles`, although others may be warranted in more complex configurations) must be considered as trusted, as well. Second, some user-space services already have SELinux types, such as `dbus`. Further, the `matchbox` window server must be considered a user-space service. We note that dbus and the X window server already have reference monitor implementations for SELinux, so we would require

Operation	Meaning	Category	Subject Type	Outcome
AT+COPS	Network Register	S	U	D
ATD	Voicecall Dial	S	U	D
ATD	Voicecall Dial	S	T	A
AT+CGMM	Get phone model number	N	U	A
AT+CBC	Get Battery Info	N	U	A
AT+CBC	Get Battery Info	N	T	A
alsactl -f /etc/stereoout.state restore	Set audio codec to playback mode	S	U	D
alsactl -f /etc/stereoout.state restore	Set audio codec to playback mode	S	T	A
cu −l /dev/ttySAC0	Command interface to send text to the modem	S	T	D
cu −l /dev/ttySAC0	Command interface to send text to the modem	S	U	D

Table I. This table shows some operations that we tried with trusted(T) and untrusted(U) clients and the result of an access check i.e., allowed(A) or denied(D). The trusted application is allowed access to all security-sensitive(S) operations except for directly accessing the modem. Untrusted applications are allowed access to all non-sensitive(N) operations like phone model information (AT+CGMM). They are denied access to all of the strictly sensitive operations like network registration (AT+COPS). Additionally they are also not allowed to toggle the audio codec mode.

[††]Correctness is also dependent on correct placement of SELinux code to completely mediate security-sensitive operations, which we demonstrated in the Complete Mediation paragraph above.

DOI: 10.1002/sec

a reference monitor for phone window server in the future. In this configuration, PALMS verified no information flow from untrusted to any of the trusted SELinux types, given that the user-space service types are trusted to filter untrusted inputs correctly. Thus, CW-Lite integrity for the phone system is verified.

6.2. System Function and Performance

To test the functionality of the modified installer, we created two different clients, one a trusted client with type `trusted_t` that can perform all telephony commands, and an untrusted client `untrusted_t` that can only query for the phone version and battery status. The trusted client could perform all commands, whereas when the untrusted client attempted other operations, it was denied (i.e., an SELinux denial message was logged). We performed operations that were both security-sensitive and not, and show some of the results in Table I.

We then allowed untrusted processes to perform SMS messaging in order to test our ability to limit misuse of other objects, in this case SMS messages. Untrusted clients were then able to send SMS messages, but the number of SMSes sent collectively by untrusted applications was limited to one message every five minutes and logged as `controlled access granted`. Beyond this limit, all SMSes were blocked, so we saw SELinux denial messages being logged for these rejections. A blacklist of SMS numbers were also created, and access was denied to these specific numbers for untrusted subjects.

The audio subsystem was protected by restricting access to the sound device file to the trusted processes when the `untrustedaudio` flag was false. The GSM Daemon set this flag to false whenever a voice call was authorized (via the Policy Management Server). We found that untrusted clients could not play sound files during this time.

Implementation overhead was tested by running local GSM functions with and without the `gsmd` hooks. The GSM Daemon command `pr` requests that the GSM Daemon read the local phone book. This command was used for testing because it can be handled locally on the phone. Commands that do not use the network minimize variance due to network delay. The `gsmd` hooks involve a system call to the kernel to determine the authorization result for the requested operation. Only one authorization was necessary for each client request. We found that on average overhead was less than 0.1 ms between the original `gsmd` and the modified `gsmd` for executing `pr` requests, which is not out of line for system call processing on the phone system. A recent patch enables caching of authorizations in `libselinux`, which will remove the need for repeated calls to the kernel to authorize the same operation. It is future work to integrate the modified `gsmd` server with this mechanism.

Vanilla kernel	1 min 39.0 seconds
IMA Kernel	1 min 55.0 seconds
PRIMA Kernel	1 min 52.7 seconds

Table II. Boot time of Vanilla, IMA and PRIMA kernel (in seconds) on the Openmoko 2007.2 platform of the Neo1973 phone.

6.3. PRIMA Measurement and Performance

In this section, we show that PRIMA integrity proofs can be generated for phone systems and that the performance overhead for collecting measurements for such proofs is acceptable. Figure 6 shows a sample PRIMA measurement list for an Openmoko phone. This measurement list includes the following fields: (1) the PCR extended by the hash (faked since there is no TPM or MTM hardware); (2) the hash of that measurement; (3) the filename of the file measured; and (4) the SELinux subject type into which that file is being executed. PRIMA stores each measurement in the order they were performed. This forms a list that is stored in kernel memory and can be inspected through the `sysfs` file `ascii_runtime_measurements`.

Recall that PRIMA takes two measurements for executables (executable file hash and SELinux type), and only one measurement for libraries (library file hash, as it is only relevant that the library was loaded in a trusted subject type). The first executable measurement includes the file hash only (e.g., as in traditional integrity measurement, such as IMA [31]). This first measurement enables the remote party to identify the executable file uniquely, since they are identified by their hash value. The second measurement binds the executable file to the subject type in which it is loaded. This enables the remote party to verify that the expected code is being run for that subject type. Libraries only require one measurement, as we are only concerned that high integrity libraries are used, not which subject they are bound to.

Table II shows the boot time with a vanilla, IMA and PRIMA Linux kernels on the Openmoko phone. The overhead for IMA and PRIMA kernels indicates the additional cost for collecting integrity measurements (i.e., hash value computation). The time to update hardware is not included (since there is no such hardware), but as such updates are performed

DOI: 10.1002/sec

```
10 ffffffffffffffffffffffffffffffffffffffff boot_aggregate
10 e56018bfcc61405d9def6a595d2e40b7b11c506a boot_aggregate kernel
10 6f6eb4425481a71ca77d0f1daf66fd15aa8f8767 init
10 2db507746ded4f5d96aaa8ae9b581c020bbc6c82 init kernel_t
10 607923211824a0896681a3905462d686e31efed6 ld-uClibc.so.0
10 72ee17e727640c366694d4688bf1eb8211490139 libselinux.so.1
10 5353b8942212fff22bf8d58cb3a7f95f099633c0 libc.so.0
10 431130f8b5339f70ac96450e2978ad4084b2a5be libsepol.so.1
10 ebf6d1687c36e7bf53cdc62bc1adab62468de21f libgcc_s.so.1
10 0d8a05330cdf2b02a65cf102809a6dd496b2cfff sh
10 b6f76619ca02b186a09a90878aa465e4ce1d331e sh init_t
10 ee7f114040e114012e25c8259d886dd8e8f71aca libcrypt.so.0
10 847a13e34caa35dd7049494e6f474253f1e53c9d syslogd initrc_t
10 57b62f6b521f644093e0b975ea4fd5070f99c28b ipkg-cl
10 bfe18be303892a8cce1cbc2623d2dd150af2742d ipkg-cl init_t
10 8c727828829ab478e7c77fd499543fde9abc52bf libipkg.so.0.0.0
```

Fig. 6. Example of a PRIMA measurement list. Each line consists of a (a) PCR Location, (b) SHA-1 Hash, (c) filename and (d) subject type.

asynchronously, the additional overhead should be negligible. Table II shows that the overall boot time is increased 13.8% by the integrity measurements of the PRIMA kernel run on the Openmoko phone. Note, however, that the increase is reduction from the IMA kernel, as several applications processes that are not part of the system's trusted computing base are started at boot time. As the Openmoko boot time is rather slow for a phone, additional work to improve boot time, such as reducing the number of processes that must start before the phone is usable, should be undertaken. Note that such measures would also reduce the cost of integrity measurement prior to boot.

Table III shows the time taken to measure files (i.e., compute their hash values) of different sizes. We tested the performance of our integrity measurement system on the files of sizes 150 Kb, 1.2 Mb, 4.1 Mb and 16.8 Mb and found that the time to measure these were 0.03, 0.16, 0.52 and 2.08 seconds, respectively, showing a linear growth. The Openmoko distributors take pains to keep the size of executables modest, so over 90% of the executable files (in `/bin`, `/usr/bin`, and `/usr/sbin`) are under 100K in size and only our modified `gsmd` is over 1M (probably due to our build process since the original `gsmd` is only 68K). Thus, we would expect overheads of less than 50 milliseconds for over 90% of code loads and no overhead greater than 0.2 seconds. Libraries are generally larger in size, with nearly 1/3 of those in `/usr/lib` exceeding 100K, but only 2 exceed 1M (libc and libgtk). While `libc` is used in many processes, recall that we only need to measure a library the first time that it is loaded into a trusted subject process. Thus, the cost to load all 180 libraries in `/usr/lib`, assuming 2/3 incur the cost of hashing a 150K file and 1/3 incur the cost of hashing a 1.2M file (which is conservative), would be 13 seconds total. We note that this cost appears largely in the system boot, so the development of an approach to measure all standard libraries in one step, such as by measuring a file that specifies the standard library hashes, should significantly improve boot times.

File Size	**Time Taken**
150 Kb	0.03
1.2 Mb	0.16
4.1 Mb	0.52
16.8 Mb	2.08

Table III. Time taken to measure files of different sizes (in seconds) on the Openmoko 2007.2 platform of the Neo1973 phone.

7. Conclusion

In this paper, we provide a solution for protecting and measuring the integrity of security-critical applications running on mobile phones. Our insight is to employ the *CW-Lite* integrity model to protect the integrity of security-critical applications, while permitting practical system function via user-space services. CW-Lite dictates that we augment user-space services with reference monitoring to enforce mandatory access control policy and filtering interfaces to protect it from untrusted input. Further, we integrate access enforcement between user-space services, a software installer and telephony server, and the operating system to ensure that comprehensive mediation of CW-Lite integrity. In particular, we integrate SELinux enforcement with the extended

DOI: 10.1002/sec

telephony server to ensure that attackers cannot use system resources, such as direct access to the modem or audio subsystem, to compromise voice calls. Finally, we incorporate the PRIMA integrity measurement architecture to build proofs that a remote party can verify CW-Lite integrity of the applications on the phone. We achieve all this with low overhead on the Linux-based Openmoko phone, and hope that our approach provides guidance for how to build secure mobile phone systems.

References

1. O. Aciicmez, L. Afshin, J.-P. Seifert, and X. Zhang. A trusted mobile phone prototype. In *Proceedings of the 5th IEEE Consumer Communications and Networking Conference (CCNC)*, pages 1208–1209, 2008.
2. J. P. Anderson. Computer security technology planning study. Technical report, ESD-TR-73-51, Oct. 1972.
3. Apple Inc. Apple app store. http://www.apple.com/iphone/appstore/.
4. W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.
5. F-Secure Computer Virus Information Pages: Cabir. website, 2006. http://www.f-secure.com/v-descs/cabir.shtml.
6. S. Carew. Cingular launches U.S. mobile banking. http://www.reuters.com/article/technology-media-telco-SP/idUSN2719455220070327.
7. J. Carter. Using gconf as an example of how to create an userspace object manager. *2007 SELinux Symposium*, 2007.
8. D. D. Clark and D. Wilson. A comparison of military and commercial security policies. In *1987 IEEE Symposium on Security and Privacy*, May 1987.
9. P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.
10. H. M. *et al*. Trusted platform on demand. Technical Report RT0564, IBM, Feb. 2004.
11. P. A. L. *et al*. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.
12. T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *2000 IEEE Symposium on Security and Privacy*, May 2000.
13. Google. Android market. http://www.android.com/market/.
14. IBM. Integrity Measurement Architecture for Linux. http://www.sourceforge.net/projects/linux-ima.
15. J. Palmieri. Get on d-bus. http://www.redhat.com/magazine/003jan05/features/dbus/#security.
16. T. Jaeger, R. Sailer, and U. Shankar. PRIMA: Policy-reduced integrity measurement architecture. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, pages 19–28, June 2006.
17. K.J.Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, June 1975.
18. F-Secure Computer Virus Information Pages: Mabir.A. http://www.f-secure.com/v-descs/mabir.shtml, 2005.
19. J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal TCB code execution. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 267–272. IEEE Computer Society, 2007.
20. Motorola. Protection must be pervasive. http://www.motorola.com/content.jsp?globalObjectId=6693.
21. D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring integrity on mobile phone systems. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 155–164, New York, NY, USA, 2008. ACM.
22. R. Naraine. Russinovich: Malware will thrive, even with vista's uac. http://blogs.zdnet.com/security/?p=175.
23. Novell. AppArmor Linux Application Security. http://www.novell.com/linux/security/apparmor/.
24. Security-Enhanced Linux. http://www.nsa.gov/selinux.
25. NSA. SELinux conditional policy language extensions. http://www.crypt.gen.nz/selinux/conditional_policy.html.
26. openmoko.com. http://www.openmoko.com/, 2008.
27. V. Rao. Security in mobile phones - handset and networks perspective. Master's thesis, The Pennsylvania State University, 2007.
28. Research In Motion Limited. Blackberry app world. http://na.blackberry.com/eng/services/appworld/.
29. S. Rueda. SELinux policy analyzer. http://www.cse.psu.edu/~ruedarod/res/analyzer_v4.tar.bz2.
30. S. Rueda, D. H. King, and T. Jaeger. Verifying compliance of trusted programs. In *USENIX Security Symposium*, pages 321–334, 2008.
31. R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, USA, Aug. 2004.
32. U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the 2006 Networked and Distributed Systems Security Symposium*, Feb. 2006.
33. E. Shi, A. Perrig, and L. van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 154–168, 2005.
34. S. Shrivastava. Satem: Trusted service code execution across transactions. In *Reliable Distributed Systems, 2006. SRDS '06. 25th IEEE Symposium on*, pages 337–338, Oct. 2006.
35. Sun Microsystems. Trusted Solaris Operating System. http://www.sun.com/software/solaris/trustedsolaris/index.xml.
36. Symbian Limited. Symbian signed. http://www.symbiansigned.com.
37. Trifinite.org – home of the trifinite.group. http://trifinite.org/trifinite_stuff.html, 2008.
38. Trusted Computing Group. Trusted computing group: Mobile. https://www.trustedcomputinggroup.org/groups/mobile.
39. Trusted Computing Group. TCG TPM specification version 1.2 revision 85, Feb 2005. https://www.trustedcomputinggroup.org/groups/tpm/.
40. Trustzone technology overview. http://www.arm.com/products/security/trustzone/, 2007.
41. E. Walsh. Application of the Flask architecture to the X window system server. 2007.
42. C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, August 2002.

DOI: 10.1002/sec