Measuring Integrity on Mobile Phone Systems

Divya Muthukumaran
Systems and Internet Infrastructure Security Laboratory
Pennsylvania State University
University Park, PA 16802
dzm133@psu.edu

Anuj Sawani
Systems and Internet Infrastructure Security Laboratory
Pennsylvania State University
University Park, PA 16802
axs1003@psu.edu

Joshua Schiffman
Systems and Internet Infrastructure Security Laboratory
Pennsylvania State University
University Park, PA 16802
jschiffm@cse.psu.edu

Brian M. Jung
Secure Systems Group
Samsung Electronics Co., Ltd.
Suwon-City, Gyeonggi-Do,
Korea, 443-742
brian.m.jung@samsung.com

Trent Jaeger
Systems and Internet Infrastructure Security Laboratory
Pennsylvania State University
University Park, PA 16802
tjaeger@cse.psu.edu

ABSTRACT

Mobile phone security is a relatively new field that is gathering momentum in the wake of rapid advancements in phone system technology. Mobile phones are now becoming sophisticated smart phones that provide services beyond basic telephony, such as supporting third-party applications. Such third-party applications may be security-critical, such as mobile banking, or may be untrusted applications, such as downloaded games. Our goal is to protect the integrity of such critical applications from potentially untrusted functionality, but we find that existing mandatory access control approaches are too complex and do not provide formal integrity guarantees. In this work, we leverage the simplicity inherent to phone system environments to develop a compact SELinux policy that can be used to justify the integrity of a phone system using the Policy Reduced Integrity Measurement Architecture (PRIMA) approach. We show that the resultant policy enables systems to be proven secure to remote parties, enables the desired functionality for installing and running trusted programs, and the resultant SELinux policy is over 90% smaller in size. We envision that this approach can provide an outline for how to build high integrity phone systems.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection — *Access Control*

SACMAT'08, June 11–13, 2008, Estes Park, Colorado, USA.

General Terms

Security

Keywords

Integrity Measurement, Mobile Phones, SELinux

1. INTRODUCTION

Cellular communication is changing. Mobile phones have become smaller, lighter, and more powerful, and support a wide variety of applications, including text messaging, e-mail, web surfing and even multimedia transmissions. Smart phones that are a hybrid of cell phones and PDAs that can handle voice and data communications, in essence functioning as a "tiny computer." This transformation motivated the transition from small, custom operating environments to more powerful, general purpose environments that are based on personal computer environments, such as Windows Mobile [33] and Linux phone OS projects [19, 23].

Third-party developers now provide many multimedia applications that users can easily download onto these powerful new phones. The flexibility of supporting third-party applications presents security concerns for other applications that handle critical user data. For example, mobile banking applications have been created for such phones [2], providing attackers with a valuable target. Worm attacks [16, 4] have been launched against the market-leading Symbian mobile platform [27], a variety of vulnerabilities on this platform have been identified [7, 29], and a large number of users (over 5 million in March 2006 [11]) download freeware games (i.e., potential malware) to their mobile devices. As a result, it seems likely that mobile phones, including Linux and Windows phones, will become targets for a variety of malware.

Security architectures for phone systems are emerging, but they make no concrete effort to justify critical application integrity. The Symbian security architecture distinguishes between its installer, critical applications, and untrusted applications. The Symbian approach has been effective at protecting its kernel, but some critical resources, such as phone

contacts and Bluetooth pairing information, can be compromised by untrusted applications [24]. A mandatory access control framework has been developed for Linux, the Linux Security Modules (LSM) framework [34], but LSM-based approaches (e.g., SELinux [22] and AppArmor [21]) do not ensure integrity. The SELinux LSM focuses on enforcing *least privilege*, and its policies on personal computer systems are too complex to understand integrity completely. The AppArmor LSM focuses on confining network-facing daemons, which may prevent integrity problems from untrusted network requests, but not from untrusted programs running on the system.

Our goal is to protect the integrity of critical phone applications from the untrusted code and data of downloaded third-party applications. The mobile banking application above is one critical phone application. The aim is to install and execute such trusted applications under the control of a phone policy for which precise integrity guarantees can be made. We believe that mandatory access control policies are the foundation for providing such guarantees, but the policies developed thusfar are inadequate because they are too complex or are focused on the wrong goal.

In this paper, we define a MAC policy for a Linux phone system and enable a remote party to verify the integrity of our phone systems using integrity measurements. We use the SELinux LSM as the starting point, but we reduce the policy to focus on integrity goals. In designing our phone policy, we use the CW-Lite integrity model [25], a weakened, but more practical, version of the Clark-Wilson integrity model [6] to define our precise integrity goals. Focusing on integrity, we find that the SELinux LSM policy can be reduced dramatically, by over 90% in size thusfar, although we believe that much greater reductions are possible. We also show that the resultant policy is suitable for justifying the integrity of such critical applications to remote parties using the PRIMA integrity measurement architecture [13]. PRIMA measures the trusted code and the information flows generated by the MAC policy to ensure that the integrity of the trusted code is protected from low integrity inputs according to the CW-Lite integrity policy. We envision that this approach can provide an outline for how to build high integrity phone systems in the future.

The structure of the paper is as follows. In Section 2, we review the background of phone systems, SELinux, formal integrity models, and integrity measurement that form the basis for this work. In Section 3, we define the phone system architecture, outline our policy design goals, and show that these goals satisfy integrity requirements while permitting the necessary function. In Section 4, we describe the implementation of our system on an evaluation board using to prototype phone software. We show how our policies are implemented, and how integrity measurements are generated for this system. We also provide results showing the performance of the system, when performing integrity measurement. In Section 5, we specify other related work, and we conclude with Section 6.

2. BACKGROUND

In this section, we provide background for phone systems security, SELinux, integrity models, and integrity measurement approaches that motivate our work.

2.1 Mobile Phone Security

Historically, mobile phone systems have been standalone devices with custom operating systems. These consumer electronics devices were installed with software in the factory and no user interfaces were provided for typical users to update the software.

As more functional, "Smart" phones began to appear, the operating system functionality requirements increased. A consortium of phone manufacturers created the Symbian operating system [27], a general-purpose, embedded operating system targeted specifically at the phone market.

The Symbian operating system is most noteworthy for not having a known kernel compromise in its history, but it also implements an interesting security model. The Symbian system defines three distinct subjects: the installer, Symbian-signed subjects, and untrusted subjects [28]. Each process is assigned to one of these three subjects depending upon which of the three categories the originating program file belongs. The three subjects essentially form a Biba hierarchy with installer being the highest integrity level. However, the choice of how files are assigned to integrity-levels is somewhat ambiguous. For example, some system files, such as the Bluetooth pairing database can be modified by untrusted code, permitting untrusted devices to upload files unbeknownst to the user [24]. Although we like the small number of subjects, the integrity protections provided are insufficient.

Recently, Windows and Linux-based phone systems have begun to emerge, eating into the Symbian market share, although it is still the operating system in over 50% of the phone devices sold. Windows and Linux systems bring both applications and security issues to the phone market. Security in the initial versions of these phones was nearly nonexistent. For early Linux phones, if an attacker could get a user to download her malware to the phone, it would be trivially compromised. But, most modern phones provide users with easy mechanisms to upload new programs. As a result, many phone system vendors are seeing that they need to add security enforcement. Motorola Linux phones, such as the A1200, include a mandatory access control module called MotoAC [19] and Samsung Research has explored SELinux on phones [35].

The challenge for phone security is becoming similar to the personal computer. Do the phone system vendors provide so much flexibility that the phones become impossible to manage? Or can a model of security that permits the secure use of untrusted code be created? We explore the answers to these questions in this paper.

2.2 SELinux

SELinux is a reference monitor for the Linux operating system [22]. SELinux enforces a mandatory access control policy based on an extended Type Enforcement model [3]. 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 labeled with a type. All objects are an instance of a particular class (i.e., data type) which has its own set of operations. A 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.

SELinux also permits domain transitions that allow a process to change its label (e.g., when it executes a new pro-

gram). Domain transitions are important because an unprivileged program could not invoke a privileged program without such transitions. For example, `passwd` would not be able to change a user's password in the `/etc/shadow` file when called from a user's shell unless a transition permitted `passwd` to invoke its own rights. Domain transitions are also relevant to security because a privileged program that does not protect itself from invocations by untrusted subjects will be a security liability to the system. In SELinux, a subject type must have a `transition` permission to the resultant subject type in order to effect a domain transition.

SELinux provides a fine-grained model in which virtually any policy could be defined. As a result, we believe that the SELinux model can be used to implement a policy that we can use to verify the integrity of critical phone applications. However, the development of SELinux policies to date have focused on defining least privilege permissions to contain services. Also, SELinux policies have grown to be very complex. A typical SELinux policy is approximately 3MB in size containing over 2000 types and between 50,000 to 100,000 permission assignments. While there have been efforts to shrink the SELinux policy, we believe that a different view of policy and function is necessary for the phone system. If we can get a simple SELinux policy that provides effective functionionality, then we might get a handle on security before the phone systems get out of control. We believe that to do this we need to focus on the integrity protection of critical applications.

2.3 Integrity Models

Protecting the integrity of critical system applications has always been a goal of security practitioners. However, the integrity models that have been proposed over the years seem not to match the practical environment. Our challenge in the development of phone system policies is to find a practical integrity model.

The Biba integrity model [15] assigns integrity labels to processes and relates these labels in an integrity lattice. Biba integrity requires that normal processes not read data at labels of lower integrity in the lattice. Also, Biba integrity does not permit normal processes to write data at labels of higher integrity in the lattice. As such, no lower integrity data could reach our critical, high integrity application in a Biba system. Unfortunately, many critical applications, including software installers, read some low integrity data.

Efforts to allow processes to read lower integrity data without compromising their integrity have not found acceptance either. LOMAC [8] requires that a process drop its integrity level to that of the lowest integrity data it reads, but some critical phone processes, such as the telephony servers, must be permitted to accept commands from low integrity subjects, but execute at high integrity. In general, we find LOMAC too restrictive, although we implement a variant of it for software installers (see Section 4.2). Clark-Wilson integrity [6] provides a more flexible alternative, by permitting subjects to read low integrity data if the immediately discard or upgrade the data, but Clark-Wilson requires full formal assurance of such processes.

We have previously proposed a compromise approach to integrity, called the *CW-Lite* integrity model. CW-Lite is weaker than Clark-Wilson in that it doesn't require full formal assurance, but CW-Lite requires processes to have filtering interfaces that immediately upgrade or discard low integrity data as Clark-Wilson prescribes. The focus then moves to identifying where low integrity data may be read and ensuring that programs use filtering interfaces to read such data. We aim to apply this view of integrity to phone systems.

2.4 Integrity Measurement

Given the inherently untrustworthy nature of remote parties, it is desirable to be able to validate that a system is of high integrity. More specifically, there should be some guarantee that the remote machine is only running programs that are trusted to behave properly and that the security policy is correct. A proposed method of establishing these guarantees uses integrity measurement [26, 17, 10, 5, 12]. Integrity measurements consist of cryptographic hashes that uniquely identify the components that define system integrity (i.e., code and data). Remote parties verify the integrity of a system by verifying that the integrity measurements taken are consistent with the remote party's view of integrity. Such measurements are conveyed in a messages signed by an authority trusted to collect the measurement, and a signed integrity measurement is called an *attestation*.

The secure storage and reporting of these measurements are typically reliant upon a root of trust in hardware like the Trusted Computing Group's Trusted Platform Module (TPM) [32]. This commodity cryptographic co-processor has facilities storing hash chains in a tamper-evident fashion. It can also securely generate public key pairs that are used to sign attestations and identify itself to remote parties. Samsung demonstrated a phone with a hardware TPM, called the Mobile Trusted Module [31], at the CES conference in Las Vegas in January 2008 [1].

Several architectures exist to gather integrity measurements such as the Linux Integrity Architecture (IMA) [10]. It obtains run-time integrity measurements of all code that is memory-mapped as executable. This facilitates the detection of any malware present on a system.

However, the IMA approach is too simplistic for phone systems for two reasons. First, if any untrusted code is run on the phone system, such as a third-party game, then an IMA verification will result in the entire phone being untrusted. Second, if an attack can modify a data file used by a trusted process, then the remote party may be tricked into thinking that a compromised phone is high integrity because IMA only measures the code and static data files. We aim to enable a phone system to run some untrusted code as long as the MAC policy enables verification that the trusted code is protected from inputs from such untrusted code.

2.5 PRIMA

The Policy-Reduced Integrity Measurement Architecture (PRIMA) [13] addresses the problem of run-time integrity measurements by additionally measuring the implied information flows between processes from the system's security policy. This way, a verifier can prove that trusted components in the system are isolated from untrusted and potentially harmful inputs. Moreover, PRIMA's CW-Lite integrity enforcement model only requires the trusted portions of a system to be measured and thus reduces the number of measurements required to verify a system.

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 in-

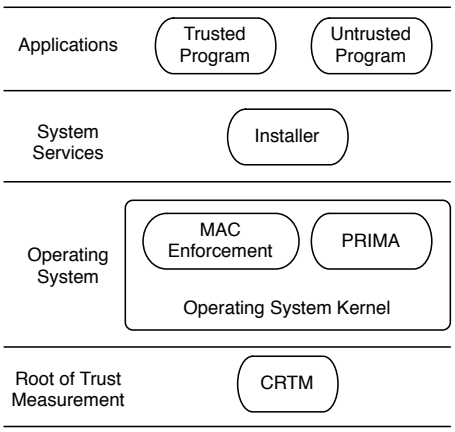

Figure 1: Software architecture for a phone system

tegrity:

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.

By measuring how the code maps to system subjects, PRIMA enables a remote party to verify that the system runs high integrity code, perhaps with acceptable filtering interfaces, in its trusted subjects, and that these subjects are protected from information flows from untrusted subjects by the MAC policy.

3. APPROACH

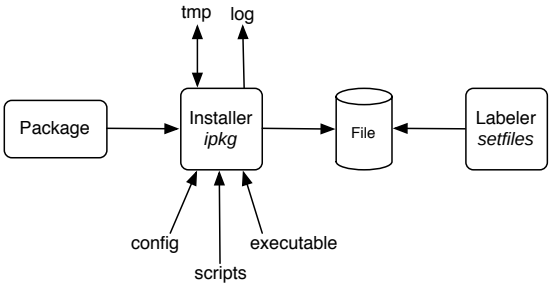

Figure 2: The software installation process

3.1 System Architecture

Figure 1 shows the software architecture for a phone system. First, the phone system contains some root of trust which is the basis for integrity in a phone system. An example is the Mobile Trusted Module [31] proposed by the Trusted Computing Group. We are not aware of phone system with a TPM, but we believe that some satisfactory root of trust for phone systems will emerge.

Second, the phone system has an operating system kernel that supports mandatory access control (MAC) and integrity measurement using the PRIMA approach. The MAC policy of the kernel will be used to define the system's information flows. The MAC policy is enforced by a reference monitor in the kernel that mediates all the security-sensitive operations of all the user-level programs. SELinux is an example of a kernel with such MAC enforcement (see Section 2.2). The PRIMA module measures the information flows in this MAC policy as well as the code that is loaded in the system, as described in Section 2.5, to enable verification that the trusted subjects are protected from untrusted subjects.

Third, the phone system has a software installer for installing both trusted and untrusted software packages. Most phone systems permit the phone users to install new software packages. In many cases, such installations require confirmation from the device user, but that is not always the case. Also, some software packages may include signed hash files that enable verification of the originator of package and the integrity of its contents.

Fourth, the packages loaded on a phone system may include trusted packages, such as a banking client, and untrusted packages, such as a game program. While some phone systems only permit the installation of signed packages from trusted authorities (e.g., Symbian-signed packages [28]), we envision that ultimately phone systems will also have to support the use of arbitrary packages. However, the trusted components of the system, such as the banking client and the installer itself, must be provably protected from such software.

3.2 Software Installation and Execution

Figure 2 shows the process of software installation. A software installer is a program that takes a software package consisting of several files and installs these files into the appropriate location in the phone's file system. Since the software installer may update virtually any program on the phone system, it is entrusted with write access over all software in the system. As a result, the integrity of the system

is dependent on the integrity of the software installer.

The software installation process also determines the labels of the installed files. Typically, this is not done by the software installer, however, but a MAC labeling service, outside the kernel, that labels the files based on a specification in the MAC policy. In SELinux, a program called `setfiles` interprets the MAC policy specification for file labeling to set the correct labeling for the newly-installed package files. The MAC labeling service must also be trusted, but unlike the software installer, it need not interact with any untrusted package files directly.

When the software installer executes, its executable uses information in a variety of other files to implement installation. Such files may include installer configurations, scripts, logs, and temporary files. Installation configurations, scripts, and the installer executable itself are rarely modified (e.g., only on installer upgrades), so these can be assumed not to be written on the loading of untrusted software. Other files, such as logs and temporary files may be updated on each installation. That is certainly the intent of the log file, which is designed to collect information from each installation. Temporary files may or may not be used depending on the installation process. In designing an access control policy, in the next section, we must consider the use of these files in designing policies that protect system integrity properly.

After the software packages are installed, the programs included in these packages may now be executed (i.e., as processes). In order to protect the integrity of the system, trusted processes, such as the banking client, must be protected from untrusted processes, such as the game program. As we identified in the PRIMA background in Section 2.5, a process's integrity depends on its code and the data that it depends on. The banking client should be isolated from untrusted programs, so it should not depend on data that can be modified by untrusted processes. However, the installer clearly receives input from untrusted processes (e.g., the untrusted programs themselves) which is necessary for correct functioning. Thus, integrity must be justified while allowing some access to untrusted data, but we also want to minimize the amount of untrusted data that installers must access.

3.3 System MAC Policy Design

The system MAC policy must enable practical justification of integrity for the software installer and the trusted packages that it installs. Here, we sketch the requirements for the MAC policies for trusted programs and the installer. The actual policies are defined in Section 4.

For trusted packages, such as the banking client, we believe that a conservative model of integrity is practical. Biba integrity [15] (see Section 2.3) can be enforced for the banking client because its files can be isolated from all untrusted programs. Since there is only one user on the phone system, there is no need to have separate principals for different banking client data files. We envision that many trusted programs, such as those used to maintain phone books, service configurations, etc., will be isolated from untrusted programs, and generally one another.

For the installer, isolation from untrusted programs is not possible. As a result, only the more liberal justification of CW-Lite integrity [25] (see Section 2.3) is possible[1]. In addition, in the design of MAC policy for the installer, we also wish to minimize exposure to the *confused deputy problem* [9] as well. As a result, our software installer runs with permissions that permit any program to invoke it and permissions that permit it to access a package file anywhere in the phone's file system, but the installer's permissions are dropped based on the label of the package that it will install.

[1]We find that other system services on the phone, such as the baseband processor daemon that enables phone calls, SMS, etc., can also be invoked by untrusted programs, so these will also achieve CW-Lite integrity at best. A similar approach would be used for securing them, but their examination is outside the scope of this paper.

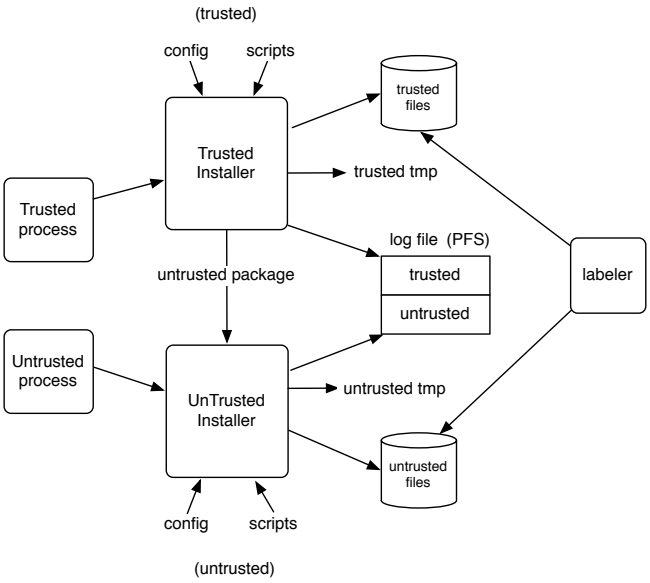

Figure 3: The modified installer process showing filtering interface to handle untrusted input

Figure 3 shows the modified installer process and outlines the installer's MAC policy. The installer must provide a filtering interface that protects it from compromise on invocation. Thus, the installer's integrity will not be compromised by either a malicious invocation by an untrusted process or an invocation that includes a malformed package. The installer immediately determines whether it is installing a trusted or untrusted package, and drops privileges to that label. This prevents the confused deputy problem by not allowing the installer to use its trusted privileges when installing untrusted software.

3.4 System Security

In this section, we show informally that the MAC policy described above will enable verification of the integrity of the trusted programs and the installer and these policies will provide the necessary permissions for trusted programs and the installer to function properly.

Biba Integrity.

For trusted programs, the MAC policy aims to ensure isolation from untrusted programs. Isolation from other trusted programs may be desirable for *least privilege* permissions, but is not required to prove system integrity. All trusted programs must be trusted by the remote party in a

successful verification, so we assign all the same MAC policy label, `trusted`. In addition to the initial verification of the filesystem integrity on boot (see Section 2.5), PRIMA measures the code for each trusted program that is executed, and the information flows in the MAC policy. All trusted programs are isolated from untrusted programs in the MAC policy, so if the trusted program's code is acceptable to the remote party, the integrity of the trusted programs is justified to the remote party.

CW-Lite Integrity.

For the installer, the MAC policy defines two principals that the installer may run under: `trusted`, and `untrusted`. The installer starts in the `trusted` principal and protects itself from its invocation inputs using a filtering interface. If the installer detects that it is installing untrusted software (e.g., by lack of a signature), it then switches to `untrusted` and loads the untrusted package. In this case, the installer cannot modify any files of the `trusted` packages with untrusted data.

Supports Necessary Function.

Clearly, the trusted programs will run correctly, as their configuration permits isolated execution from untrusted processes, but we must be careful to enable the installer to run correctly when it runs as an `untrusted` process. There are two issues: one for files that are only read by the installer and one for files that are read and written by the installer. First, since the `untrusted` installer does not modify configurations and scripts, it can access these as normal. We have found that the installer scripts do not require exceptional privilege, so they can be run as `untrusted`.

Second, the installer may modify log files and temporary files. Since the temporary files apply to the untrusted package being loaded and/or the untrusted files that are being replaced, these can be treated as `trusted` or `untrusted` according to the case. At present, the SELinux MAC policy assigns the same label to all of the temporary files created by an installer, so this will need to be changed. The log files are more difficult. In this case, we want a single log file name, but the untrusted installer should not be allowed to modify trusted logs. Our design choice is to create separate log files for trusted and untrusted installers, and we use a file system implementation to enable these files to share the same name (see Section 4.2).

4. IMPLEMENTATION

4.1 Experimental System

The initial design and development of mobile phones are typically carried out on evaluation boards that have hardware very similar to mobile phone hardware. Our implementation was performed on a Spectrum Digital Evaluation board (OMAP 5912 OSK) with the following features:

- ARM CPU (ARM926EJ-S) operating at 192 Mhz
- 32 MB DDR RAM
- 32 MB Flash ROM

We run the Linux kernel 2.6.18.1 on the evaluation board with the SELinux security module enabled and the PRIMA patches applied (see Section 4.4). The kernel is booted with the `u-boot 1.3.1` bootloader which is commonly-used on embedded devices. We cross-compiled the kernel and bootloader for the ARM platform using a toolchain generated using `buildroot`, a script for generating toolchains and root filesystems.

The root filesystem software originates from the Qtopia Linux distribution [30]. We use the `ipkg` installer from the Linux OpenMoko distribution [23] as our installer. It is the only software we use from outside the Qtopia distribution.

Once the kernel is uploaded to the flash memory on the board, almost all further compilation is performed directly on the board. An exception is the compilation of the SELinux policy which fails on the board due to memory constraints. It was compiled on an x86 machine and copied to the board's root filesystem.

We use Journalling Flash File System 2 (JFFS2) with extended attributes support as the file system format on the evaluation board. JFFS2 is becoming increasingly popular as the filesystem format for mobile devices due to the numerous advantages for flash memory. Due to storage space limitations of the evaluation board, we divided our root filesystem into two parts. The majority of the filesystem was flashed onto the memory on the board while the remaining(`/usr` and `/tmp`) was mounted via NFS from a remote machine. We note that SELinux does not trust NFS by default and gives the NFS-mounted files the same label, `nfs_t`. On the phone, this would not be a problem as the root filesystem would not be mounted remotely.

4.2 Simplifying the Installer Policy

The basic idea behind the reduction in policy complexity is to move from a fine-grained, least privilege policy to a coarse-grained, integrity policy. We make a distinction between the purpose SELinux policies serve in personal computers and what we want it to do in phone systems. Traditionally, SELinux is used to enforce a least privilege MAC which results in a very fine-grained policy. For example, a large number of SELinux types are defined for which there must be many more rules between types. As an example, consider how the policy for installers are defined in a traditional SELinux policy [2]. Installers needs access to different kinds of files, namely, the executable, configuration files, log files, scripts, temporary files, and libraries. Because the aim is to have a least privilege policy, each of these files is given its own SELinux type to make a distinction among them. This results in many types defined just for the `ipkg` installer as shown below.

```
type ipkg_t;                /* ipkg process */
type ipkg_exec_t;           /* ipkg executable */
type ipkg_file_t;           /* ipkg configuration */
type ipkg_log_t;            /* ipkg log */
type ipkg_script_exec_t;    /* ipkg scripts */
type ipkg_script_t;         /* ipkg script process */
type ipkg_script_tmp_t;     /* ipkg script temp files */
type ipkg_script_tmpfs_t;   /* ipkg script tmpfs use */
type ipkg_tmp_t;            /* ipkg's temp files */
type ipkg_tmpfs_t;          /* ipkg's use of tmpfs */
type ipkg_var_lib_t;        /* ipkg files in /var/lib */
```

Our purpose for the SELinux policy in the phone systems is to preserve integrity and hence we do not need to make

[2]The SELinux reference policy defines policy for the `rpm` installer which we adapt for the `ipkg` installer used in this experiment. The architecture and function of these installers is very similar.

a distinction between all these types and to control interactions between them. All we need to know is whether the installer reads or writes trusted or untrusted data. In our goal of achieving integrity protection on the phone systems we move from a fine-grained policy to a coarse-grained by viewing the system as consisting of only three types of integrity entities namely `trusted`, `untrusted` and `kernel`.

There are SELinux rules that assign the installer process access to its files (SELinux `allow` rules) and other process's ability to run the installer with its privileges (SELinux `type_transition` rules). In the SELinux reference policy, there are over 2000 rules of these types, most describing how other processes can invoke the installer (specifically, RPM installer). However, the installer is also given access to the entire file system because we do not know where the new files may be installed.

As an initial approach to providing system integrity, we use the three types in a Biba integrity policy [15] with `kernel` as the highest integrity and `untrusted` as the lowest integrity. In order to obtain all its privileges, the installer runs as `trusted`, and all installer files with the SELinux types shown above are relabeled to `trusted` as well. However, requiring that the installer's policy satisfies Biba integrity is too restrictive for the software installer. There are two cases that would violate Biba integrity:

- When a process labeled `untrusted` calls the installer
- When the package being installed is labeled `untrusted`

In the first case, the invoking process can set the environment and arguments to the software installer, so it can write data to the software installer. This violates the Biba integrity because a low integrity process (the `untrusted` requester) cannot write to a higher integrity process (the installer with type `trusted`). In the second case, the data being written to the installer is low integrity. This violates Biba because a high integrity process (the installer) cannot read lower integrity data (the untrusted package).

We can modify the policy to solve the first problem. If we do not provide a transition rule for `untrusted` processes, then the installer will run as `untrusted` as well. The installer no longer violates Biba integrity because it is allowed to read the high integrity installer files. The second case cannot be solved using policy modifications – we need a different approach to solve this problem.

4.3 Filtering Interface

In the second case, Biba integrity cannot work. Instead, we use the CW-Lite integrity model semantics [25]. CW-Lite requires that any process that receives lower integrity data must supply filtering interfaces that either immediately upgrade or discard this data. For the installer, we use a variant of discarding the data, where the installer includes an interface to safely receive the input, determine its integrity, then the installer can downgrade its own label dynamically if the input is untrusted. The CW-Lite semantics should be modified slightly to include this case.

While this approach is basically implementing LOMAC integrity [8], by automatically downgrading the integrity of the software installer when it is used to install low integrity software packages, we find that CW-Lite is more general. Other critical applications, such as the telephony server, may have to process inputs from untrusted and they must enforce access using the traditional CW-Lite semantics. We believe that both approaches should be supported.

This works as follows. Since the installer is being started by a trusted process it will run with the `trusted` type initially. However, when the input is examined and is determined to be `untrusted`, the installer will transition to `untrusted`. Since this is a dynamic transition decision, it cannot be defined in the policy. In order to enable this transition, we need to make a small addition to the installer code at the point where the input is examined.

```
if(!checkcontext(package_context,trusted_context)
{
 strcpy(str,"system_u:object_r:untrusted_t");
 i = setcon((security_context_t) str);
}
```

The `setcon` function is used to dynamically change the context of a process from within the program. Thus, on examining the input and seeing that an untrusted package is being installed the installer will self-transition to `untrusted`.

We would ideally prefer to reduce the SELinux policy complexity to the bare minimum, and have only three types. 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 include many additional SELinux types. Some of the dependencies occured at compile time; it was possible to create a policy of approximately 100 types compile. However, other SELinux types, particularly types for devices, 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, including our three 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.

Recall from Section 3.4 that the only installer files that are written are log files (`ipkg_log_t`) and temporary files (`ipkg_tmp_t` and `ipkg_tpmfs_t` and the analogues for `ipkg` scripts) that are both now labeled `trusted`. Since the temporary files are specific to an installer run, we can use `untrusted` temporary files for an installation by an `untrusted` installer.

For the log files, we generate separate logs for the trusted and untrusted versions of the installer. While this only permits an untrusted subject or a trusted subject with a filtering interface to read all the logs[3], it does protect system integrity. The problem is that both versions of the installer would use the same log file, and we cannot give a single file two types, even if it made sense. To permit two processes to write to two versions of a file with the same name, we propose using a polyinstantiated [14] file system, which Linux already has for enabling processes of different secrecy levels to write files of the same name without leaking data. In a polyinstantiated file system, multiple files correspond to the same name, but the system chooses which file to access based on the security label of the process. For example, when a high integrity process writes a file, it would write the highest integrity file it can access (i.e., at its own level). When it reads a file, the process reads data from all the files

[3]Separate secrecy requirements may be installed to prevent untrusted processes from reading the trusted process's log.

whose integrity dominates the process (i.e., Biba read-up). This is future work.

4.4 PRIMA Implementation

To create our PRIMA system, we created a custom 2.6.18-1 Linux kernel by altering the SELinux LSM hooks with modified IMA functions. We also changed the evaluation board's `init` program to properly load the PRIMA's trusted subjects policy and process the information flows.

4.4.1 Kernel Modifications

The Linux community has rejected the use of the LSM interface for integrity measurement, so to implement PRIMA, we converted IMA to function like a Linux kernel library that can be called from SELinux. Originally, IMA uses 5 LSM hooks as seen in Table 1, to gather integrity measurements and manage internal data. The first step in integrating integrity measurement functionality with SELinux, was to add the IMA callback functions to each of the SELinux's equivalent hooks. We added each callback after authorization, so they are only triggered if the SELinux hook would have authorized the policy decision.

First, we modified the `file_mmap` hook to gather the SELinux subject of the current context. When calls are made to code memory mapped as executable, PRIMA is invoked to parse the PRIMA list of trusted subjects for the current context. If a match is found, then the calling process is indeed trusted and the code is measured as was done in IMA. PRIMA then performs a subsequent measurement that binds the code hash to the SELinux subject type by concatenating the two and measuring the result. However, if the file being loaded by a trusted subject is a library, the function does not take the second subject measurement since it is unimportant which trusted processes is using it.

The LSM hooks `inode_permission` and `sb_umount` detect when a measured file that is still open is opened for writing or unmounted. If this happens, the measurement list is invalidated as it is no longer clear if the measurement actually represents the loaded code. Since this behavior is the same in IMA as in PRIMA, we simply inserted the function call into the hook.

A trusted subject list specifies the set of SELinux subject types that must run trusted software. This list is created as part of the system's policy so its contents are independent of PRIMA mechanism. Before the subject list is loaded, PRIMA assumes all subjects are trusted as the early boot phase is critical and must all be trusted.

To load the set of trusted subjects for the system and to view this set, we created a `sysfs` file `/selinux/ts_load`. IMA also by default exposes a `sysfs` file `/selinux/measurereq` which accepts a file pointer and performs a PRIMA integrity measurement on arbitrary files. We use this file to measure the SELinux system policy used in this run of the system.

PRIMA also takes integrity measurements in two additional cases. The first is when the aggregate of all pre-kernel integrity measurements is generated during PRIMA's initialization. Here, no subject binding measurement is performed as there is no PRIMA policy available before the OS has loaded. The second case is whenever a kernel module is loaded into the kernel. In this case, the subject binding measurement is performed as for other trusted code as described for `file_mmap` above.

4.4.2 Building the Information Flow Graph

SELinux hook	IMA task
file_mmap	Code measurement
sb_umount	Detecting concurrent write
inode_permission	Detecting concurrent write
inode_free_security	Free PRIMA data structures
file_free_security	Free PRIMA data structures

Table 1: The SELinux LSM hooks and their equivalent IMA purpose.

We have a program to extract the information flows from the SELinux binary policy file. The security policy defines all the types and permission assignments between types. We are interested in the interactions between types, specifically, we are interested in what information flows are allowed to or from a trusted type. We, therefore, parse the policy and extract only the data relevant to building the information flow graph.

The program first extracts the types and their string representations and builds a Type_number:Type_name map. Then, the policy entries are read and a hashtable of *access vectors* (AV) is built. AVs represent the SELinux `allow` rules that define the access rights in the system.

In the second stage, we go through the hashtable and interpret the permissions between the source and target type. Interpreting the permissions amounts to identifying if the operation corresponds to a read or a write or both. This identifies the information flow between the subject type and the object type implied by the AV. The resulting information flow is collected into our information flow graph after mapping the type numbers to the type names (using the type_number:type_name map). An entry in the information flow graph looks like this

```
fsadm_t proc_t both
```

where fsadm_t is the source type, proc_t is the target type and both indicates a bi-directional flow.

4.4.3 `init` Modification

PRIMA requires a list of trusted subjects to operate properly. So we modified the phone's `init` program to load this before loading the SELinux policy. This is done by first writing the contents of the subject list stored in `/etc/selinux/subjects` to `measurereq` for measurement and then to `/selinux/ts_load` to load it into kernel memory. The information flow graph is also generated through a call from `init` and its resulting graph is also passed to the `measurereq` interface for measuring.

4.4.4 Measurements

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`.

Figure 4 is a small sample of what the measurement list looks like. The fields are the PCR which is extended with the hash (if a TPM is present), the hash of the file, the filename and the SELinux subject type. All code measurements lack a subject type since they are just the measurements of the code. One measurement is made per file (if invoked by a trusted subject) unless the file is changed (i.e., the hash of

```
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
```

Figure 4: Example of a PRIMA measurement list. Each line consists of a (a) PCR Location, (b) SHA-1 Hash, (c) filename and (d) subject type.

Measurement of Policy binary	0.33
Generate Information flows	13.47
Measure trusted subjects list	0.06
Measure a generic program (busybox)	0.10
Measure generic file (Size:2.5MB)	2.76
Measure generic file (Size:30MB)	27.4

Table 2: Time taken to perform certain tasks on the evaluation board (in seconds)

the binary has changed) in which case a new measurement is taken.

The measurements with subjects show the mapping between the software and its subject type. Their hashes are different from the code measurements, as they measure the concatenation of the subject and software. Only one measurement is needed per file, so all subsequent loads of the code by different subjects will result in just a measurement of the mapping.

We tested the performance of our system's integrity measurement mechanism in processing requests. Since the evaluation board lacks a TPM and uses NFS for a part of its root filesystem, performance is not what would be expected in an actual mobile phone, but shows the scalability of the mechanism. Table 2 shows time taken to measure typical files. The time grows roughly linearly with the size of the file. Table 3 compares the boot time and the time taken by the ikpg installer in a Vanilla, IMA and PRIMA kernel.

5. RELATED WORK

There have been some efforts to have a policy based access control on mobile phones. A recent paper [20] summarizes the status of MontaVista Software's efforts to implement security solutions based on ARM cores that provide separated computing environment, as well as SELinux to provide MAC for embedded devices. The paper cites memory footprint and performance trade-off as the two most critical constraints for developing security solution for embedded devices. The emphasis is on virtualization in embedded devices with concepts like containment, root of trust and SELinux. Although there are no design details about SELinux policy for phones systems, the paper does address the need for a careful analysis of the policy to ensure that it is not unnecessarily comprehensive.

There has also been a recent effort by the MontaVista Corporation that incorporates SELinux into Mobile phones. A press release of Mobilinux 5.0 says it is the first operating system to include MontaVista MicroSELinux [18], a miniaturized version of SELinux. It claims to be the first Linux release for mobile phones that incorporates SELinux. While our work also emphasizes a small policy, we have shown how such a policy can be designed by using coarse grained specification of integrity and tied it to an integrity measurement architecture.

Zhang et al proposed an isolation technique for resource constrained mobile platforms [35]. They realize this goal by employing the TCG's Trusted Mobile Phone specification by leveraging SELinux. They also integrate IMA for integrity measurement by defining SELinux policy language extensions that adds another attribute to the context. We have implemented SELinux with integrity measurement without modifications to SELinux policy structure.

Motorola also made an attempt to provide a SELinux-like system on the Motorola A1200 mobile phone. The operating system is based on a v2.4 linux kernel and includes a proprietary access control system known as MotoAC. On analysis of this system, MotoAC appears to be disabled by default and does not provide any advantage over their previous phone models.

6. CONCLUSION

In this paper we have shown that it is possible to construct an SELinux policy capable of justifying a phone system's integrity. We have demonstrated that the policy can protect critical applications from untrusted code allowing cell phone users to install and run trusted applications in a safe fashion. Further, we have achieved this with a minimal policy which is 90% smaller than the SELinux reference policy (less than 300KB from a 3MB policy originally). By porting PRIMA onto the phone system we have enabled verifiable integrity on the phones. In future work, we plan to analyze the policy to further reduce its complexity. We also plan on experimenting on a variety of applications to assess our design's robustness.

7. REFERENCES

	Vanilla kernel	IMA Kernel	PRIMA Kernel
Boot time	11.81	14.54	14.33
Execution time of a program (ipkg)	0.24	0.39	0.39

Table 3: Comparison of performance of Vanilla, IMA and PRIMA kernel on the evaluation board (in seconds)

[1] Trusted Platform. http://www.sisa.samsung.com/innovation/tp/index.htm.

[2] Bank of America. Mobile banking. http://www.bankofamerica.com/onlinebanking/index.cfm?template=mobile_banking&statecheck=PA.

[3] W. E. Boebert and R. Y. Kain. A practical alternative to heirarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[4] F-Secure Computer Virus Information Pages: Cabir. http://www.f-secure.com/v-descs/cabir.shtml, 2006.

[5] L. S. Clair, J. Schiffman, T. Jaeger, and P. McDaniel. Establishing and sustaining system integrity via root of trust installation. In *Proceedings of the 2007 Annual Computer Security Applications Conference*, Dec. 2007.

[6] D. D. Clark and D. Wilson. A comparison of military and commercial security policies. In *1987 IEEE Symposium on Security and Privacy*, May 1987.

[7] J. de Haas. Symbian Phone Security. http://www.blackhat.com/presentations/bh-europe-05/BH_EU_05-deHaas.pdf.

[8] T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *2000 IEEE Symposium on Security and Privacy*, May 2000.

[9] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), 1988.

[10] IBM. Integrity Measurement Architecture for Linux. http://www.sourceforge.net/projects/linux-ima.

[11] 236 mln wireless subscribers in the us in 2006. http://www.itfacts.biz/index.php?id=P8421, 2007.

[12] O. W. R. M. J. Marchesini, S.W. Smith. Experimenting with tcpa/tcg hardware, or: How i learned to stop worrying and love the bear. Technical Report TR2003-476, Computer Science Technical Report, Dartmouth College, Dec. 2003.

[13] 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.

[14] C. S. Janak Desai, George Wilson. Extending selinux to meet lspp data import/export requirements, Feb 2006. http://selinux-symposium.org/2006/papers/04-lspp.pdf.

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

[16] F-Secure Computer Virus Information Pages: Mabir.A. http://www.f-secure.com/v-descs/mabir.shtml, 2005.

[17] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetoh, S. Yoshihama, and T. Nakamura. Trusted platform on demand. Technical Report RT0564, IBM, Feb. 2004.

[18] Montavista. Montavista Mobilinux. http://www.mvista.com/product_detail_mob.php.

[19] Motorola. Opensource Motorola. https://opensource.motorola.com.

[20] H. Nahari. Trusted secure embedded Linux. In *Proceedings of the Linux Symposium Proceedings of the Linux Symposium Proceedings of the Linux Symposium*, 2007.

[21] Novell. AppArmor Linux Application Security. http://www.novell.com/linux/security/apparmor/.

[22] Security-Enhanced Linux. http://www.nsa.gov/selinux.

[23] openmoko.com. http://www.openmoko.com/, 2008.

[24] V. Rao. Security in mobile phones - handset and networks perspective. Master's thesis, The Pennsylvania State University, 2007.

[25] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium (NDSS'06)*, Feb. 2006.

[26] E. Shi, A. Perrig, and L. V. Doorn. BIND: A time-of-use attestation service for secure distributed systems. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.

[27] Symbian OS: the open mobile operating system. http://www.symbian.com/, 2008.

[28] Symbian Limited. Symbian signed. http://www.symbiansigned.com.

[29] Trifinite.org – home of the trifinite.group. http://trifinite.org/trifinite_stuff.html, 2008.

[30] Trolltech. Qtopia Open Source. http://trolltech.com/products/qtopia/opensource.

[31] Trusted Computing Group. Trusted computing group: Mobile. https://www.trustedcomputinggroup.org/groups/mobile.

[32] Trusted Computing Group. TCG TPM specification version 1.2 revision 85, Feb 2005. https://www.trustedcomputinggroup.org/groups/tpm/.

[33] Windows mobile: Smartphone and pda software. http://www.microsoft.com/windowsmobile/, 2008.

[34] 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.

[35] X. Zhang, O. Aciicmez, and J.-P. Seifert. A trusted mobile phone reference architecture via secure kernel. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, 2007.