Design and Implementation of Efficient Integrity Protection for Open Mobile Platforms

Xinwen Zhang, *Member, IEEE,* Jean-Pierre Seifert, *Member, IEEE,* and Onur Acıiçmez, *Member, IEEE*

Abstract—The security of mobile devices such as cellular phones and smartphones has gained extensive attention due to their increasing usage in people's daily life. The problem is challenging as the computing environments of these devices have become more open and general-purpose while at the same time they have the constraints of performance and user experience. We propose and implement SEIP, a *simple and efficient* but yet *effective* solution for the *integrity protection* of real-world cellular phone platforms, which is motivated by the disadvantages of applying traditional integrity models on these performance and user experience constrained devices. The major security objective of SEIP is to protect trusted services and resources (e.g., those belonging to cellular service providers and device manufacturers) from third party code. We propose a set of simple integrity protection rules based upon open mobile operating system environments and application behaviors. Our design leverages the unique features of mobile devices, such as service convergence and limited permissions of user installed applications, and easily identifies the borderline between trusted and untrusted domains on mobile platforms. Our approach thus significantly simplifies policy specifications while still achieves a high assurance of platform integrity. SEIP is deployed within a commercially available Linux-based smartphone and demonstrates that it can effectively prevent certain malware. The security policy of our implementation is less than 20kB, and a performance study shows that it is lightweight.

Index Terms—integrity protection, open mobile platforms, smartphone security.

1 Introduction

With the increasing computing capability and network connectivity of mobile devices such as cellular phones and smartphones, more applications and services are deployed on these platforms. Thus, their computing environments become more general-purpose and open than ever before. The security issue in these environments has gained considerable attention nowadays. According to McAfee's 2008 Mobile Security Report [8], nearly 14% of global mobile users have been directly infected or have known someone who was infected by a mobile virus. More than 86% of consumers worry about receiving inappropriate or unsolicited content, fraudulent bill increases, or information loss and theft, and more than 70% of users expect mobile operators or device manufacturers to preload mobile security functionality. The number of infected mobile devices increases remarkably according to McAfee's 2009 report [9].

Existing research on mobile device security mainly focuses on porting PC counterpart technologies to mobile devices, such as signature- and anomaly-based analysis [19], [21], [27], [30], [34], [38]. However, there are several reasons that make these infeasible. First of all, mobile devices such as cellular phones are still limited in computing power. This mandates that any security solution must be very efficient and leave only a tiny footprint in the limited memory. Second, in order to save battery energy, an always concurrently running PC-like anti-virus solution is, of course, not acceptable. Third, security functionality should require minimum or zero interactions from a mobile user, e.g., the end user shouldn't be required to configure individual security policies. This demands then that the solution must be simple but general enough so that most users can rely on default configurations — even after new application installations.

On the other side, existing exploits in mobile phones have shown that user downloaded and installed applications are major threats to mobile services. According to F-secure [28], by the end of 2007, more than 370 different malware have been detected on various cell phones, including viruses, Trojans, and spyware. Most existing infections are due to user downloaded applications, such as Dampig [1], Fontal, Locknut, and Skulls. Other major infection mechanisms include Bluetooth and MMS (Multimedia Message Service), such as Cabir, CommWarrior, and Mabir. Many exploits compromise the integrity of a mobile platform by maliciously modifying data or code on the device (cf. Section 2.2 for integrity compromising behaviors on mobile devices). PandaLab reports the same trends [13] in 2008. Based on the observation that user downloaded applications are the major security threats, one objective of securing mobile terminal should be confining the influence of user installed applications. This objective requires restricting the

- *X. Zhang is with Huawei Research Center, Santa Clara, CA, USA. E-mail: xinwen.zhang@huawei.com*
- *J-P. Seifert is with Deutsche Telekom Laboratories and Technical University of Berlin. Email: jean-pierre.seifert@telekom.de*
- *O. Acıiçmez is with Samsung Information Systems America, San Jose, CA, USA. Email: o.aciicmez@samsung.com*

1. Description of all un-referred viruses and malware in this paper can be found at http://www.f-secure.com/en_EMEA/security/security-threats/virus/.

permissions of applications to access sensitive resources and functions of mobile operating system (OS), customers, device manufacturers, and remote service providers, thus maintaining high integrity of a mobile device, which usually indicates its expected behavior. Considering the increasing attacks through Bluetooth and MMS interfaces, an effective integrity protection should confine the interactions between any code or data received from these communication interfaces and system parts.

Towards *simple, efficient, and yet effective solution*, we propose SEIP, a mandatory access control (MAC) based *integrity protection* mechanism of mobile phone terminals. Our mechanism is based upon information flow control between *trusted* (e.g., customer, device manufacturer, and service providers) and *untrusted* (e.g., user downloaded through browser or received through Bluetooth and MMS) domains. By confining untrusted applications' write operations to trusted domains, our solution effectively maintains runtime integrity status of a device. To achieve the design objectives of simplicity and efficiency, several challenges exist. First, we need to efficiently identify the interfaces between trusted and untrusted domains and thus simplify required policy specification. For example, in many SELinux systems for desktop and servers, very fine-grained policy rules are defined to confine process permissions based on a least-privilege principle. However, there is no clear integrity model behind these policies, and it is difficult to have the assurance or to verify if a system is running in a good integrity state. Second, many trusted processes on a mobile device provides functions to both trusted and untrusted applications, mainly the framework services such as telephony server, message service, inter-process communications, and application configuration service. Therefore simply denying the communications between trusted and untrusted process decreases the openness of mobile devices – mobile users enjoy downloading and trying applications from different resources. We address these challenges by efficiently determining boundaries between trusted and untrusted domains from unique filesystem layout on mobile devices, and by classifying trusted subjects (processes) according to their variant interaction behaviors with other processes. We propose a set of integrity protection rules to control the inter-process communications between different types of subjects.

Our contributions are three-fold in this paper.

- We analyze integrity threats on mobile platforms based on infection and compromising mechanisms. We then identify salient requirements for security solutions on mobile terminals and propose our strategies towards these requirements.
- Based on different functional behaviors of subjects in mobile systems, we propose a set of integrity protection rules to control their interactions with others. Our solution focuses on the protection of phone and platform management services from untrusted applications such as those downloaded by users or received from Bluetooth/MMS.
- We present the design and implementation of our solution in a real-world Linux-based mobile phone device, and we leverage SELinux to define a respective policy. Our policy size is less than 20kB in binary form and requires less than 10 domains and types. We demonstrate that our implementation can prevent major types of attacks towards system integrity through mobile malware. Our performance study shows that the incurred overhead is significantly smaller compared to PC counterpart technology.

Outline In the next section we discuss potential integrity assets and threats to mobile platform integrity. Section 3 gives an overview of SEIP. Details of our design and integrity rules are described in Section 4, and implementation and evaluation in Section 5 and Section 6, respectively. We discuss some limitations of our solution in Section 7. Related work on integrity model and mobile platform security are presented in Section 8. We conclude this paper in Section 9.

2 Threat Model

We focus our study on the integrity protection of mobile platforms. Particularly for this purpose, we study the adversary model of mobile malware from two aspects: integrity assets and attack mechanisms. Note that in this paper we consider attacks from application level.

2.1 Assets for Platform Integrity

Besides regular computing and operating resources, there are several unique targets towards a mobile platform's integrity. We identify the following sets of assets.

Operating system resources The operating system resources mainly consist of the trusted computing base of an operating system, including boot components, system binaries and configurations, and services. Attacks to these resources can easily compromise the integrity of the whole platform.

Resources of network service provider A mobile device usually consists of sensitive data from network service provider, such as those stored in SIM card for network and service profiles. Unauthorized access to these data can compromise the running behavior of the device and communications between the device and wireless network.

Device data and status settings Modern mobile devices are employed with many sensors, such as timer, GPS, touchscreen, and webcam. Manipulating these sensors without authorization from the user can cause unexpected behavior of a device. Also, a device provides many status setting functions such as those for 3G, WiFi, Bluetooth, screen brightness level, and battery. A malware can maliciously change these settings to attack the integrity of a platform, e.g., to turn on screen when the device is in idle, or switch between data network channels stealthily.

Resources of mobile user A user stores many personal data on device, such as messages, address book, and online credentials. Many online service providers store user data on

TABLE 1
Example Mobile Malware and Their Behaviors

Malware	Infection/Propagation	Compromising Integrity Assets
DAmpig, Fontal, Locknut,	Bluetooth, MMS, Internet	modify system files and configurations, disable application manager and phone services.
Cabir, CommWarrior, Mabir	Bluetooth, MMS	scan new devices with Bluetooth , sends user data and malicious code to new targets without authorization.
Doomboot	Bluetooth, Internet, MMS	change binaries of system files in c:, make device unable to reboot.
Skulls	Bluetooth	disable phone and message services.
Cardblock	Bluetooth, MMS	block access to memory card, delete system files and installed application files and data.
Redbrowser	download Java application	send SMS messages to premium rate number at a rate of $5 - $6 per message.
Mquito	download game application	sends SMS messages to premium rate number.

device side, such as online bank and entertainment services, which are targets for malware. By sending SMS/MMS messages and making hidden phone calls to premium phone numbers, a malware can generate monetary cost to a mobile user.

We note that this is not intended to be an exhaustive list for integrity assets. Table 1 summarizes some example malware and their infection mechanisms and target integrity assets.

2.2 Attach Mechanisms

Infection Mechanisms With the constraints of computing capability, network bandwidth, and I/O, the major usage of mobile devices is for consuming data and services from remote service providers, instead of providing data and services to others. Based on this feature, the system integrity objective for mobile devices is different from that in desktop and server environments. For typical server platforms, one of the major security objectives is to protect network-faced applications and services such as httpd, smtpd, ftpd, and samba, which accept unverified inputs from others [31]. For mobile phone platforms, on the other side, the major security objective is to protect system integrity threaten by user installed applications. According to mobile security reports from F-Secure [28] and PandaLab [13], most existing malware on cell phones are unintentionally downloaded and installed by user, and so far there are no worms that do not need user interaction for spreading.

Although most phones do not have Internet-faced services, many phones have local and low bandwidth communication services such as file-sharing via Bluetooth. Also, the usage of multimedia messaging service (MMS) has been increasing. Many viruses and Trojans have been found in Symbian phones which spread through Bluetooth and/or MMS such as Cabir, CommWarrior, and Mabir. Therefore, any code or data received via these services should be regarded as untrusted, unless a user explicitly prompts to trust it. The same consideration applies for any received data via web browsers on mobile devices.

Integrity Compromising Mechanisms Many mobile malware compromise a platform's integrity by disabling legal phone or platform functions. For example, once installed, mobile viruses like Dampig, Fontal, Locknut, and Skulls maliciously modify system files and configurations thus disable application manager and other legal applications. Doomboot installs corrupted system binaries into c: drive of a Symbian phone, and when the phone boots these corrupted binaries are loaded instead of the correct ones, and the phone crashes at boot. Similarly Skulls can break phone services like messaging and camera.

As a mobile phone contains lots of sensitive data of its user and network service provider, they can be targets of attacks. For example, Cardblock sets a random password to a phone memory card thus makes it no longer accessible. It also deletes system directories and destroys information about installed applications, MMS and SMS messages, phone numbers stored on the phone, and other critical system data. Other malware such as Pbstealer and Flexispy do not compromise the integrity of a platform, but stealthily copy user contact information and call/message history and send to external hosts.

Monetary loss is an increasing threat on mobile phones. Many viruses and Trojans trick a device to make expensive calls or send messages. For example, Redbrowser infects mobile phones running Java (J2ME) and sends SMSs to a fixed premium rate number at a rate of $5 - $6 per message, which is charged to the user's account. Mquito, which is distributed with a cracked version of game Mosquitos in pirate channels, sends an SMS message to a premium rate number before the game starts normally.

Propagation Mechanisms On an Internet-enabled desktop or server, a malware can propagate itself by building connections to other internet-connected hosts and sending its malicious code. For example, many worms randomly pick IP addresses and try to build connections if there are vulnerable services running on target hosts. On mobile devices, however, a malware can propagate with many other connectivity mechanisms, such as Bluetooth, SMS/MMS, and WiFi. Many malware have been found which use Bluetooth and SMS/MMS to propagate. Also, some malware can leverage multi-connection interfaces on a mobile devices for more complex attacks, such as cross-service attacks [34] and deny-of-service attack to cellular network [24].

3 SEIP Overview

3.1 Requirements

Simplicity and Effectiveness It is very expensive to change any system function on a mobile device once it is massively produced and sold to final customers. Also, a mobile user typically does not have skills and knowledge to configure any security functions except some simple application-level settings. On the other side, due to the complexity of mobile business model, i.e., hardware and software components can be integrated from different vendors, complex security policies and mechanisms can increase the cost of system

integration and deployment. Therefore it is desirable to have a simple but effective security model.

Efficiency One critical performance requirement lies on typical usage behaviors with mobile devices. A mobile user is not so "sticky" as that in desktop environments, where she spends a lot of time using a particular application. While a mobile user is "bouncy", and flies in and back out between different applications [17]. Therefore a trusted subject (e.g., a service daemon) may accept requests from both trusted and untrusted application concurrently (although may not be absolutely concurrent). Traditional integrity models are not efficient and flexible under these scenarios. For example, LOMAC [25] requires a process dynamically downgrade its security level whenever it accesses low integrity objects or receives inputs from low integrity processes. However, the process needs to restart whenever it needs to access high integrity objects later, which is not efficient for mobile devices. Efficiency is a must requirement for user experience of consumer electronic (CE) devices.

3.2 Design Overview

To effectively achieve integrity protection goals while respect the constraints of mobile computing environments, we propose the following tactics for our design.

Simplified boundary of trusted and untrusted domains Instead of considering fine-grained least privileges for individual applications, we focus on integrity of trusted domains. With this principle, we identify domain boundary along with relatively simpler filesystem layout in many Linux-based mobile phones than in PC and server environments. Specifically, based on our investigation, most phone-related services from device manufacture and network provider are deployed on dedicated filesystems, while user downloaded application can only be installed on another dedicated filesystem or directory, or flash memory card. Thus, for example, one policy can specify that by default all applications belonging to the manufacturer or service provider are trusted for integrity purpose, while user installed applications are untrusted. An untrusted application can be upgraded to trusted one only through extra authentication mechanisms or explicit authorization from user.

MAC-based integrity model Many mobile platforms use digital signature to verify whether a downloaded application can have certain permissions, such as Symbian [26], Qtopia [15], MOTOMAGX [10], and J2ME [4]. All the permissions specified by a signed application profile are high level APIs, e.g., making phone call or sending messages. There are several issues with these approaches. First of all, they cannot check the parameters of allowed API calls, thus a malicious application may get sensitive access or functions with allowed APIs, such as call or send SMS messages to premium rate numbers. Second, API invocation control cannot restrict the behaviors of the target application when it makes low level system calls, e.g., invoking system process or changing code and data of trusted programs. This is especially true for platforms that allow installing native applications such as LiMo [5], Maemo [7], GPE [3], Qtopia [14], and JNI-enabled Android. Thirdly and most importantly, most of these "ad-hoc" approaches do not apply any kind of foundational security model. Logically, it is nearly impossible to specify and verify policies for fundamental security properties such as system integrity. In our design, we use MAC-based security model, thus different processes from the same user can be assigned with different permissions. Further, our approach enables deeper security checks than simply allowing/denying API calls. Finally, instead of considering extremely fine-grained permission control thus requiring a complete and formal verified policy, we focus on read- and write-like permissions that affect system integrity status, thus make integrity verification feasible.

Trusted subjects handling both trusted and untrusted data Traditional integrity models either prohibit information flow from low integrity sources to high integrity processes (e.g., BIBA [18]), or degrade the integrity level of high integrity processes once they receive low data (e.g., LOMAC [25]). However, both approaches are not flexible enough for the security and performance requirements of mobile platforms. Specifically, due to function convergence, one important feature of mobile phone devices is that resources are maintained by individual framework services which run as daemons and accept requests from both trusted and untrusted processes. For example, in LiMo platform [5], a message framework controls *all* message channels between the platform and base stations, and implements all message-related functions. Any program that needs to receive/send SMS or MMS messages has to call the interfaces provided by this framework, instead of directly interacting with the wireless modem driver. Other typical frameworks include telephony service serving voice conversation and SIM card accesses, and network manager serving network access such as GPRS, WiFi, and Bluetooth. All these frameworks are implemented as individual daemons with shared libraries, and expose their functions through public interfaces (e.g., telephony APIs). Similar mechanisms are used for platform management functions such as application management (application installation and launch), configuration management, and data storage. Many frameworks need to accept inputs from both trusted and untrusted applications during runtime — to provide, e.g., telephony or message services. However, due to performance reasons they cannot frequently change their security levels. Thus, traditional integrity models such as BIBA and LOMAC are not flexible enough to support such security and performance requirements. Similarly, domain transitions used in SELinux are infeasible for mobile platforms.

Towards this issue, we propose that some particular trusted processes can accept untrusted information while maintaining their integrity level. The critical requirement here is that accepted untrusted information does not affect the behavior of such trusted process and others. We achieve this goal via separating information received from subjects

of different integrity levels. For example, the telephony server accepts requests from both trusted and untrusted applications, while enforces constraints to the types of phone calls that an untrusted application can make. For another example, during the installation and launching of applications on mobile platforms, untrusted applications can be installed and launched through a trusted package management tool and application launcher (which of course operate on both trusted and untrusted domains), while the installer and launcher still maintain their respective integrity. We insert security hooks into these daemons, and define a set of integrity rules to label the data a trusted subject reads and isolate them when these data are processed or handed over to other subjects. We identify the general principles and places to isolate untrusted access requests in typical service frameworks on a mobile terminal including telephony service, message service, platform configuration service, and device status monitoring service. An important feature that distinguishes our approach from traditional information flow–based integrity models is that, we do not sanitize low integrity information and increase its integrity level, as Clark-Wilson–like [22], [29], [37] models do. Instead, we separate the data from different integrity levels within a trusted subject so receiving untrusted data does not affect its behavior.

4 Design of SEIP

This section presents design details and integrity rules of SEIP for mobile platform based on discussed security threats and our strategies. Although we describe within the context of Linux-based mobile systems (specifically, LiMo platform), our approach can be applied to other phone systems such as Symbian, as they have similar internal software architecture. One assumption is that we do not consider attacks in kernel and hardware, such as installing kernel rootkits or re-flashing unauthentic kernel and filesystem images to devices. That is, our goal is to prevent software-based attacks from application level.

4.1 Trusted and Untrusted Domains

To preserve the integrity of a mobile device, we need to identify the integrity level of applications and resources. In mobile platforms, typically trusted applications such as those from device manufacture and wireless network provider are more carefully designed and tested as they provide system and network services to other applications. Thus, in our model we regard them as high integrity applications or subjects. Note that completely verifying the trustworthiness of a high integrity subject, e.g., via static code analysis, is out of the scope of SEIP. As aforementioned, our major objective is to prevent platform integrity compromising from user installed applications. Therefore by default all user installed applications later on the platform are regarded as low integrity. In some cases, a user installed application should be regarded as high integrity, e.g., if it is provided by the network carrier or trusted service provider and requires sensitive operations such as accessing SIM or user data, e.g., for mobile bank and payment applications. For an application belonging to 3rd party service provider, its integrity level may be based on the trust agreement between the service provider and user or manufacturer/network provider. For example, an anti-virus agent on a smartphone from a trusted service provider needs to access many files and data of the user and network provider and should be protected from modification of low integrity software, therefore it is regarded as high integrity. Other high integrity applications can be trusted platform management agents such as device lock, certificate management, and embedded firewall.

Usually, extra authentication mechanism is desired when a user installed application is considered as high integrity or trusted, such as application source authentication via digital signature verification, or explicitly authorized via user interface actions from the user.

Based on the integrity objective of SEIP — to protect system and service components from user installed applications — we specify the boundary in the filesystem of mobile devices. We lay out all Linux system binaries, shared libraries, privileged scripts, and non-mutable configuration files into dedicated file system or system partition. Similar layout can be used for all phone related application binaries, configurations, and framework libraries. All user applications can only be installed in a writable filesystem or a directory and the `/mnt/mmc`, which is mounted when a flash memory card is inserted. However, many phone related files are mutable, including logs, tmp files, database files, application configuration files, and user-customizable configuration files, thus have to be located in writable filesystems. Policies should be carefully designed to distinguish the writing scope of a user application. We have observed similar approaches have been used on Motorola EZX series [12] and Android [1].

We note that filesystem layout is determined by the manufacturer of a device and such a separation can be enforced quite easily. This approach simplifies policy definitions and reduces runtime overhead compared to traditional approaches such as SELinux on PCs, which sets the trust boundaries based on individual files. Note that we assume there is secure re-flashing of the firmware of a device. If arbitrary re-flashing is enabled, an attacker can install untrusted code into the trusted side of the filesystem offline, and re-flash the filesystem image to the device. When the devices boots, the untrusted code is loaded into trusted domain during runtime and can ruin the system integrity.

4.2 Subjects and Objects

Like traditional security models, our design distinguishes subjects and objects in OS. Basically, subjects are *active entities* that can access objects, which are *passive entities* in a system such as files and sockets. Subjects are mainly active processes and daemons, and objects include all possible entities that can be accessed by processes, such as files, directories, filesystems, network objects, program and data files. Note that a subject can also be an object as it

can be accessed by another process, e.g., being launched or killed. In an OS environment, there are many different types of access operations. For example, SELinux pre-defines a set of object classes and their operations. For integrity purposes, we focus on three access operations: create, read, and write. From information flow perspective, all access operations between two existing entities can be mapped to read-like and write-like operations [16].

In many cases, objects are not visible in OS, while only accessible through framework services, i.e., accessed by daemons. Other processes communicate with daemons to create and access objects via APIs. This encapsulated approach can be found in many service-oriented software architectures. For example, telephony server creates and maintains telephone conversation sessions for other applications which request to make phone calls. Also, the telephony server maintains SIM data of a mobile phone device, which are not be accessed via other application directly in OS environment, but can only be accessed by the telephone server. The network manager framework in LiMo creates and maintains all network connections and profiles for different applications, such as Packet Data Protocol (PDP) sessions for GPRS and access point associations for WiFi connections. Another example, Gconf daemon (`gconfd`) stores configuration data for individual phone applications, which can only access their data via GConf APIs, and `gconfd` is the only subject that can physically (in OS point of view) read and write the objects. Not only for those objects in regular OS such as files and sockets, our design protects objects *internally* maintained by these service daemons which affect the integrity of a platform.

4.3 Trusted Subjects

We distinguish three types of trusted subjects [2] on mobile platforms, according to their functionalities and behaviors. Different integrity rules (cf. Section 4.4) are applied to them for integrity protection purpose.

Type I trusted subjects This type includes high integrity system processes and services such as `init` and `busybox`, which are basically the trusted computing base (TCB) of the system. Only high integrity subjects can have information flow to those subjects, while untrusted subjects can only read from them. Type I trusted subjects also include pre-installed applications from device manufacture or service provider, such as dialer, calendar, clock, calculator, contact manager, etc. As they usually only interact with other high integrity subjects and objects, their integrity level is constant during runtime.

Type II trusted subjects These are applications provided by trusted resources, but usually read low integrity data only, such as browser, MMS agent, and media player. They are usually pre-deployed in many smartphones by default, which can be considered as trusted subjects in our strategy. However, they mostly read untrusted Internet content or play downloaded media files in flash memory card. These subjects usually do not communicate with other high integrity subjects in most current smartphone systems, and they do not write to objects which should be read by other trusted subjects. Therefore, in our design we downgrade their integrity level during runtime without affecting their functions and system performance.

Type III trusted subjects These are mainly service daemons such as telephony, message, network manager, inter-process communication (IPC), device status manager (reading and setting hardware status), and application and platform configuration services. Usually these subjects need to interact with both low and high integrity subjects. For integrity purpose, we need to prevent any information flowing from low integrity entities to high integrity entities by using these daemons' functions.

4.4 Integrity Rules

For integrity protection, it is critical to control how information can flow between high and low integrity entities. We propose a set of information flow control rules for different types of subjects, with focus on create, read, and write operations [3]. Table 2 summarizes these integrity rules, where $L(x)$ is the integrity level of subject or object x, and Figure 1 shows allowed information flow between subjects based on these rules.

TABLE 2
Rules for Information Flow Control

Create Object:

Rule 1: $create(s, o) \leftarrow L(o) = L(s)$: when object o is created by a process s, o inherits s's integrity level.

Rule 2: $create(s_1, s_2, o) \leftarrow L(o) = MIN((L(s_1), L(s_2))$: when o is an object created by process s_1 with input from another process s_2, o inherits the lower bound of integrity levels of s_1 and s_2.

Read/Write:

Rule 3: $can\_read(s, o) \leftarrow L(s) \leq L(o)$: a low integrity process s can read from a low or high integrity process or object o.

Rule 4: $can\_write(s, o) \leftarrow L(s) \geq L(o)$: a high integrity process s can write to a low or high integrity process or object o.

Rule 5: $can\_read(s, o_1) \leftarrow L(s) \geq L(o_1) \wedge can\_write(s, o_2) \wedge L(o_1) \geq L(o_2)$: a high integrity process s can receive information from low integrity subject or object o_1, provided that the information will be written to low integrity subject or object o_2 by the high integrity process s.

Rule 6: $change\_level : L'(s) = L(o) \leftarrow read(s, o) \wedge L(s) > L(o)$: when a high integrity process s reads low integrity object o, its integrity level is changed to $L(o)$.

***Rule** 1: $create(s, o) \leftarrow L(o) = L(s)$: when an object is created by a process, it inherits the integrity level of the process.*

2. The concept of trusted subjects in SEIP is different from that in traditional trusted operating system and database [33]. Traditionally, a trusted subject is allowed to bypass MAC and access multiple security levels. Here we use it to distinguish applications from trusted resources and user downloaded.

3. We consider destroying/deleting an object is the same operation as writing an object.

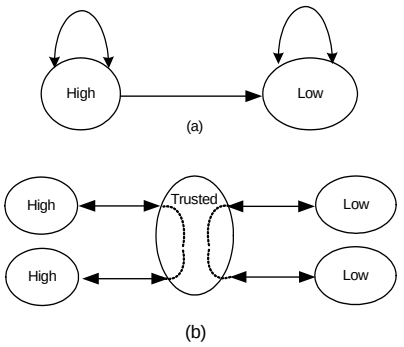

Fig. 1. Information flows are allowed between high and low integrity entities, directly or indirectly via trusted subjects.

***Rule** 2:* $create(s_1, s_2, o) \leftarrow L(o) = MIN((L(s_1), L(s_2))$: *when an object is created by a trusted process s_1 with input/request from another process s_2, the object inherits the integrity level of the lower bound of s_1 and s_2.*

These two rules are *exclusively* applied upon a single object creation. Typically, Rule 1 applies to objects that are *privately* created by a process. For example, an application's logs, intermediate and output files are private data of this process. This rule is particularly applied to Type I trusted subjects and all untrusted subjects. Rule 2 applies to objects that are created by a process upon the request of another process. In one case, s_1 is a server running as a daemon process, the s_2 can be any process that leverages the function of the daemon process to create objects, e.g., to create a GPRS session, or access SIM data. In another case, s_1 is a common tool or facility program that can be used by s_2 to create object. In these cases, the integrity level of the created object is corresponding to the lower of s_1 and s_2. This rule is applied to Type III trusted subjects as aforementioned.

***Rule** 3:* $can\_read(s, o) \leftarrow L(s) \leq L(o)$: *a low integrity process can read from both low and high integrity entities, but a high integrity process can only read from entity of the same level.*

***Rule** 4:* $can\_write(s, o) \leftarrow L(s) \geq L(o)$: *a high integrity process can write to both low and high integrity entities, but a low integrity process can only write to entity of the same level.*

As Figure 1(a) shows, these two rules indicate that there is no restriction on information flow within trusted entities, and within untrusted entities, respectively. However, these also imply that information flow is only allowed from high integrity entities to low integrity entities, which is, fundamentally, BIBA-like integrity policy. Note that a reading or writing operation can happen between a subject and an object, or between two subjects via IPC mechanisms. These two rules are applied to direct communication between Type I trusted subjects and all untrusted entities.

***Rule** 5:* $can\_read(s, o_1) \leftarrow L(s) \geq L(o_1) \wedge write(s, o_2) \wedge L(o_1) \geq L(o_2)$: *a high integrity process s can receive information from low integrity entity o_1, provided that the information is separated from that of other high integrity entities, and it flows to low integrity entity o_2 by the high integrity process s.*

As Figure 1(b) shows, this rule allows a trusted subject to behave as a communication or service channel between untrusted entities. This rule is particularly for the Type III trusted subjects, which can read/receive inputs from low integrity entities while maintaining its integrity level, under the condition that the low integrity data or requests are separated from high integrity data and handled over to a low integrity entity by the trusted subject.

Rule 5 requires that any input from untrusted entities does not affect the runtime behavior of the high integrity subject. Therefore not every subject can be trusted for this purpose. Typically, communications between applications and service daemons can be modelled with this rule. For example, on a mobile phone device, a telephony daemon can create a voice conversation between an application and wireless modem, upon the calling of telephony APIs. A low integrity process cannot modify any information of the connection created by a high integrity process, thus preventing stealthily forwarding the conversation to a malicious host, or making the conversation into a conference call. For another example, data synchronizer from device manufacture or network provider can be trusted to read both high and low integrity data without mixing them.

***Rule** 6:* $change\_level : L'(s) = L(o) \leftarrow read(s, o) \wedge L(s) > L(o)$: *when a trusted subject reads low integrity object, its integrity level is changed to that of the object.*

This rule is dedicated for the Type II trusted subjects, which usually read untrusted data (e.g., Internet content or media files). As these subjects do not communicate with other trusted subjects, downgrading their integrity level does not affect their functions and system performance in our design. This is the only rule that changes a subject's integrity level during runtime.

4.5 Enforcing Integrity Rules

For Type I subjects, since no low integrity information can flow to them, any write access from untrusted subject is blocked. The enforcement is implemented in OS kernel by controlling IPC between processes. Note we have the assumption that the kernel is trusted. For Type II subjects, the enforcement of integrity rules is straightforward: when a subject read low integrity data, its integrity level is downgraded to the low level, which is also enforced by kernel.

Integrity policy enforcement for Type III trusted subjects is more complex. Towards information flow control, we further identify two classes of Type III subjects: those implementing functions to manage critical resources of a platform such that only trusted subjects can write, e.g., the device status manager in our LiMo evaluation platform (cf. Section 5.3) and telephone daemon for SIM data access, and those implementing variant services for both untrusted

and other trusted subjects, e.g., IPC service like D-Bus daemon, telephony server, message server, and configuration server. As policy enforcement is within a trusted Type III subject, we have to hook into its implementation to check access requests from other subjects.

Specifically, a typical service framework provides functions to other applications via APIs defined in library files. We identify all read-, create-, and write-like APIs for a framework, and track their implementing methods in the daemon. For the first class Type III subjects, read-like accesses are allowed to all other subjects, while write-like accesses are only allowed to trusted subjects, e.g., to allow them to set devices status. Our implementation of secure SIM data access (Section 5.3) and secure device status service (Section 5.4) takes this approach.

For the second type of Type III subjects, careful investigation to the internal architecture of a daemon is needed to isolate information from trusted and untrusted subjects following integrity Rule 5, i.e., low integrity information flows to low integrity subjects or objects, and high integrity information can only flow to high integrity subjects or objects via the trusted subject. In general, for a creating method, we insert a hook to label the new object with the integrity level of the requesting subject. For a write-like method, we insert a hook to check the label of target object and the requesting subject, and deny or allow the access according to pre-defined policies according to our integrity rules. Our implementation for secure D-Bus (Section 5.2), secure phone services (Section 5.3), and secure configuration service (Section 5.4) is in this case.

The reminder of this section illustrates the high level design of applying these principles to some critical platform management framework including IPC and application installation. We present the implementation details of secure phone and device services in next section.

4.6 Dealing with IPC

According to Rule 1, most IPC objects inherit the integrity level of the processes that create them, including domain sockets, pipes, fifo, message queues, shared memory, and shared files. Therefore, when a low integrity process creates an IPC object and write to it, a high integrity process cannot read from it, according to our integrity rules. In many mobile Linux platforms such as LiMo, OpenMoko, GPE, Maemo, and Qtopia, D-Bus is the major IPC, which is a message-based communication mechanism between processes. A process builds a connection with a system- or user-wide D-Bus daemon (`dbusd`). When the process wants to communicate to anther process, it sends messages to `dbusd` via its connection. The `dbusd` maintains connections of many processes, and routes messages between them. A D-Bus message is an object in our design, which inherits integrity level from its creating process. According to Rule 5, Type III trusted subject `dbusd` (specified by policy) can receive any D-Bus message (low or high integrity level) and forward to corresponding destination process. Typically, a trusted process can only receive high integrity messages from `dbusd`. Also, according to Rule 5, if a process is a Type III trusted daemon, like telephony or message server daemon, it can receive high and low integrity messages from `dbusd`, and handle them separately within the daemon.

4.7 Program Installation and Launching

An application to be installed is packaged according to particular format (e.g., the .SIS file for Symbian and .ipk for many Linux-based phone systems), and application installer reads the program package and meta-data and copies the program files into different locations in local filesystem. As the application installer is a Type III trusted subject specified by policy, it can read both high and low integrity application packages. Also, according to our integrity Rule 2 and 5, it writes (when installing) to trusted part of the filesystem when reads high integrity software package, and writes to untrusted part of the filesystem when reads low integrity package.

Similar to installation, during the runtime of a mobile system, a process is invoked by a trusted program called program launcher, which is also a Type III trusted subject according to policy. Both high and low integrity processes can be invoked by the program launcher. All processes invoked from trusted program files are in high integrity level, and all processes invoked from untrusted program files are in low integrity level. Compare to traditional POSIX-like approaches, where a process's security context and privileges typically are determined by a calling process, in our design, a process's integrity level is determined by the integrity level of its program files including code and data [4]. On one aspect, this enhances the security as a malicious application cannot be launched to a privileged process, which is a major vulnerability in traditional OS; on the other aspect, this simplifies policy specification in a real system, which can be seen in next section.

4.8 Dealing with Bluetooth/MMS/Browser and Their Received Code/Data

As aforementioned in Section 2, increasing malware infect mobile phones via variant communication channels between phone devices and networks. For example, many malware in Symbian-based phones send malicious codes via Bluetooth channel, or distribute with MMS messages (either in message contents or as attachments). In the mean time more smartphones have been deployed with Internet browsers, which is another interface to access and receive untrusted code and data. SEIP regards MMS agents and mobile browsers as Type II trusted subjects via security policy. According to integrity Rule 6, their integrity level is changed to low whenever they receive data from outside, e.g., reading message or browsing web content. Any code or data received from Bluetooth, MMS, and browser is

4. Superficially this feature is similar to the setuid in Unix/Linux. However setuid is mainly for privilege elevation for low privileged processes to complete sensitive tasks, while here we confine a process's integrity aligning with its program's integrity level.

untrusted by default since during runtime these subjects can only write untrusted system resources such as filesystems. Thus, any process directly invoked from arbitrary code by MMS agent or browser is in low integrity level, according to our integrity Rule 1. Furthermore, any code saved by these subjects is in low integrity level and it cannot be launched to high integrity processes, as the program launcher is Type III trusted subject following integrity Rule 5. Thus it cannot write to trusted resources and services, such as corrupting system binaries or changing platform configurations. More fine-grained policy rules can be defined to restrict phone related functions that an untrusted process can have, such as accessing phone address book, sending messages, and building Bluetooth connections, which prevent further distribution of potentially malicious code from this untrusted subject.

It is possible that some software and data received from Bluetooth, MMS, and browser are trusted. For instance, a user can download a trusted bank application from his PC via Bluetooth or from a trusted financial service provider's website via browser. For another example, many users use Bluetooth to sync calendar and contact list between mobile devices and PC. SEIP does not prevent these types of applications. Usually, with extra authorization mechanism such as prompting via user actions, even an application or data is originally regarded as untrusted, it can be installed or stored to the trusted side, e.g., by the application manager or similar Type III trusted subjects on the phone. Similar mechanism can be used for syncing user data or installing user certificate via browser.

5 IMPLEMENTATION

We have implemented SEIP on a real LiMo platform [5]. Our implementation is built on SELinux, which provides comprehensive security checks via Linux Security Module (LSM) in kernel. Also SELinux provides domain-type and role-based policy specifications, which can be used to define policy rules to implement high level security models. However, existing deployments of SELinux on desktop and servers have very complex security policies and usually involve heavy administrative task. Furthermore, current SELinux does not have an integrity model built-in. On one side, our implementation simplifies SELinux policy for mobile phone devices based on SEIP. On the other side, our implementation augments SELinux policy with built-in integrity consideration.

5.1 Trusted and Untrusted Domains

Figure 2 shows a high-level view of the filesystem and memory space layout in our evaluation platform. All Linux system binaries (e.g., `init`, `busybox`), shared libraries (`/lib`, `/usr/lib`), scripts (e.g., `inetd`, `network`, `portmap`), and non-mutable configuration files (`fstab.conf`, `inetd.conf`, `inittab.conf`, `mdev.conf`) are located in a read-only cramfs filesystem. Also, all phone related application binaries, configurations, and framework libraries are located in another cramfs

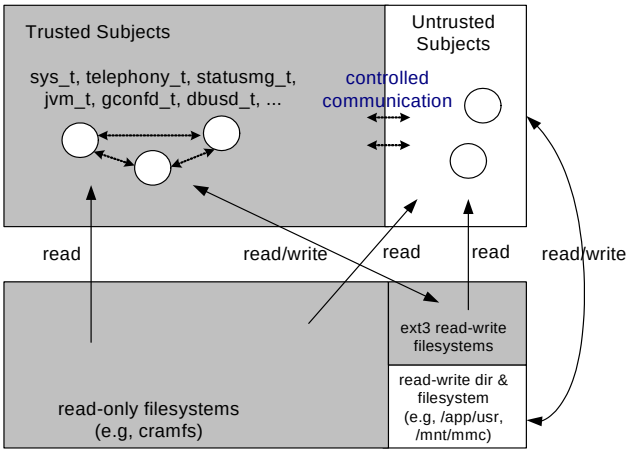

Fig. 2. Trusted and untrusted domains and allowed information flow between them on evaluation platform.

filesystem. All mutable phone related files are located in an ext3 filesystem, including logs, tmp files, database files, application configuration files, and user-customizable configuration files (e.g., application settings and GUI themes). By default, all codes and data downloaded by user, e.g., via USB, Bluetooth, MMS, or browser, are stored and installed under `/app/usr` of the ext3 filesystem and in `/mnt/mmc` (which is mounted when a flash memory card is inserted), unless explicitly prompted by the user to install to the trusted ext3 filesystem.

As Figure 2 shows, all read-only filesystems and part of ext3 filesystem where phone related files are located are regarded as trusted, and user writable filesystems are regarded as untrusted. By default, processes launched from trusted filesystems are trusted subjects, and processes launched from untrusted filesystems are untrusted subjects. Note that our approach does not prevent trusted user application from being installed on the device. For example, a trusted mobile banking application can be installed in the trusted read-write filesystem, and the process launched from it is labelled as trusted. We also label any process invoked by message agent (e.g., MMS or email agent) or browser as untrusted. According to SEIP, trusted subjects can read and write to trusted filesystem objects, and untrusted subjects can read all filesystem objects, but can only write to untrusted objects. Figure 2 also shows the information flow between subjects and filesystem objects. The access controls are enforced via SELinux kernel level security server.

As created by kernel, virtual filesystems like `/sys`, `/proc`, `/dev` and `/selinux` are trusted. Similar to the ext3 filesystem, `/tmp` and `/var` include both trusted and untrusted file and directory objects, depending on which processes create them.

5.2 Securing IPC via D-Bus

D-Bus is the major IPC mechanism for most Linux-based mobile platforms. The current open source D-Bus implementation has built-in SELinux support. Specifically,

a `dbusd` can control if a process can acquire a well-known bus name, and if a message can be sent from one process to another, by checking their security labels. These partially satisfy our integrity requirement: a policy can specify that a process can only send message to another process with the same integrity level. However, as in mobile devices, including our evaluation platform, both trusted and untrusted applications need to communicate with framework daemons via `dbusd`, e.g., to make phone calls or access SIM data, or set up network connections with connectivity service. Therefore, existing D-Bus security mechanism cannot satisfy this requirement.

Following our integrity rules, we extend D-Bus built-in security in two aspects. Firstly, each message is augmented with a header field to specify its integrity level based on the process which sends the message, and the value of this field is set by `dbusd` when it receives the message and before dispatches it. As `dbusd` listens to a socket on connection requests from other processes, it can get the genuine process information from the kernel. Note that as `dbusd` is a Type III trusted process, both high and low integrity processes can send messages to it. Secondly, according to our integrity rules, if a destination bus name is a Type I trusted subject, it can only accept high integrity messages; otherwise, it can accept both high and low integrity messages.

With these, each message is labeled with an integrity level, and security policies can be defined to control communication between processes. When a message is received by a trusted daemon process, its security label is further used by the security mechanism inside the daemon to control which object that the original sending process can access via the method call in the message, which is explained in the following subsections.

Our implementation introduces less than 200 line of code based on the D-Bus framework of our LiMo platform, mainly for adding a security context field of D-Bus message header, setting this field by `dbusd` message dispatcher, and checking security policy before dispatching based on the integrity level of a message and its destination process. Specifically, as each client process (the sender) connects `dbusd` with a dedicated socket, the dispatcher calls `dbus_connection_get_unix_process_id()` to obtain the pid of the sender, then the security context of the sender with `getpidcon()`, and then sets the value into the header field. This field is used later when the message arrives at a destination trusted subject, i.e., to make access control decision of whether the original sender of the message can invoke a particular function or access an object.

5.3 Securing Phone Services

The telephony server provides services to typical phone-related functions such as voice call, data network (GSM or UMTS), SIM access, messages (SMS and MMS), and GPS. An application calls telephony APIs (TAPI) to access services provided by the telephony server, which in turn connects to the wireless modem of the device to build communication channels. In our LiMo platform, message framework and data network framework are dedicated for short message and data network access services. An application first talks to these framework severs which in turn talk to the telephony server. Security controls for those services can be implemented in their daemons.

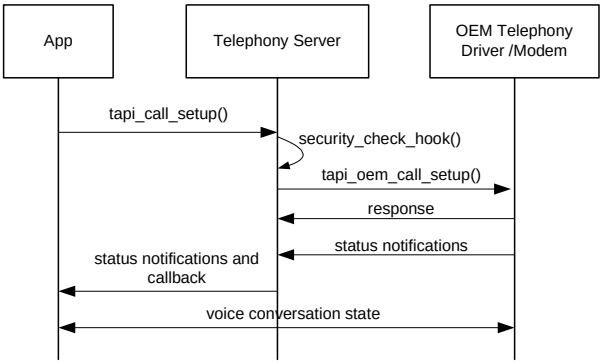

Fig. 3. Secure telephony server. A security hook in telephony daemon checks if a voice call can be build for the client application.

TABLE 3
Telephony and Message Server APIs where Integrity Check is Enforced

Telephony Service		
Call_setup	Call_answer	Call_release
SendDTMF	Call_releaseall	Hold
Retrieve	ECT	Join
Split	Call_forward	Call_wait
Call_bar	SetRestrictCallId	
Short Message Service		
Sms_write	Sms_delete	Sms_send
Sms_setCbconfig	Sms_setPreferredbearer	Sms_setSca
Sms_setSMSParameters	Sms_SendDeliveryReport	
Access SIM Resources		
Sim_changePin	Sim_unblockPin	Sim_EnablePin
Sim_DisablePin	Sim_update	Sim_EnableFDN
Sim_DisableFDN	Sim_UpdatePBRecord	Sim_deletePBRecord

Different levels of protection can be implemented for secure voice call. For example, one policy allows that only trusted applications can make phone calls, while untrusted application cannot make any phone call, which is the case in many feature phones. For another example policy, untrusted applications can make usual phone calls but not those of premium services such as payment-per-minute 900 numbers. Different labels can be defined for telephone numbers or their patterns. In our implementation, we allow untrusted applications to call 800 toll-free numbers only. Similar design is used in message framework. Figure 3 shows the workflow for a typical voice call. A client application calls `tapi_call_setup()` to initialize a phone call with $TelCallSetupParams\_t$, which includes the target phone number and type (voice call, data call, or emergency call), and a callback function to handle possible results. The telephony server provides intermediate notifications including modem and connection status to the client. Once the call is established with the modem, the telephony server sends the connected indication to the TAPI library which in turn notifies the application

via the registered callback function about the status of call (connected or disconnected), and then the application handles the processing. We insert a security hook in the telephony server daemon which checks the integrity level (security context) of calling client from D-Bus message and decides to allow or deny the call request. For simplicity the `dbusd` is ignored in Figure 3.

As mentioned in Section 4.5, security hooks are inserted into the APIs of each trusted daemon subject (Type III). As examples, we list the voice call APIs in telephony server in Table 3. Similar security hooks are inserted to control access to SIM data items, such as contacts and PIN, and access to SMS service. Note that by default any write-like operation to SIM is denied for all untrusted applications in our policy. We also have introduced new SELinux permission class `telephony` and added corresponding permissions, such as `call_setup`, `sms_send`, and `sim_disablepin`, according to individual APIs. Note that for integrity protection we only list create- and write-like functions here.

5.4 Securing Device and System Services

The system framework in our platform maintains all system status such as phone status, device status, memory status for out-of-memory, audio path, and volume status, and provides get/set APIs for accessing them. A status manager (a daemon) in this framework collects global system status from other frameworks, such as phone status (busy/idle/standby) from telephony server, power levels from power manager in kernel, and device status from various kernel device drivers. Although typically every application can get the statuses via get APIs, only trusted applications can set these variables via set APIs. That is, the status manager is a Type III trusted subject. We insert security hooks in corresponding API implementations in the status manager to enforce our integrity protection policy.

GConf is a service to store configuration data for other applications on our evaluation platform and many other Linux-based mobile phone devices including Maemo, OpenMoko, GPE, and ALP. In current implementation of GConf, any application can read and modify any configuration data, and there is no separation among configuration data from different applications. Thus, the major security requirement for GConf is to control access to configuration keys from processes of different integrity levels. Specifically for integrity purpose, we insert security hooks into all write operations to the configuration keys of high integrity applications. The hooked GConf APIs are similar to those in [20]. However we note that a difference between [20] and our implementation is that [20] leverages the GConf client library to pass security context of a calling process to the GConf daemon (`gconfd`), which is an "undesirable" solution [20] since the client library can be bypassed via static linking or malicious implementation. In our implementation, we leverage the trusted D-Bus daemon to pass the security context of a process, which is a securer solution.

TABLE 4
Example Trusted Subjects Specified by Policy

Type I trusted subjects	all Linux TCB processes and services, dialer, calendar, clock, calculator
Type II trusted subjects	browser, email/SMS/MMS clients, media player
Type III trusted subjects	daemon processes of telephony server, message server, data network connection server, D-Bus, GConf, program installer and launcher, device status manager, contact manager
Untrusted subjects	all applications in `/app/usr` and `/mnt/mmc`

5.5 Policy Specification

As we do not consider very fine-grained least privilege principle, and our integrity rules classify different subjects clearly, defining security policy with our design is a much easier task than that in traditional SELinux systems. However, it is still desired to define different domains for some other purposes on a phone. For example, for confidentiality reason, DRM keys are only readable to DRM agent on a mobile phone, although typically other applications or services from device manufacturer and network service provider are trusted by the DRM agent. Therefore we define a specific domain for DRM agent and types for DRM key objects, and corresponding policy rules. Similarly, only Java Virtual Machine (JVM) can access Java related policies and digital certificates, only `gconfd` can write configuration resource files, and only package switch manager (PSMAN) can write to data network profiles.

We define four domain attributes to specify high and low integrity processes: *trustedI*, *trustedII*, *trustedIII*, and *untrusted*, for different types trusted subjects and untrusted subjects, respectively. Table 4 list some subjects of each type. The development of policy rules follows two principles: maximally enable permissions between trusted domains, and minimally enable permissions between trusted and untrusted domains. Specifically, regular permissions are enabled between *trustedI* and *trustedIII* domains such as those defined in permission classes of process, socket, and netlinks. For untrusted domains and objects, by default we enable all read-like permissions and disable all write-like operations from untrusted domains to trusted side (including processes and objects). In most cases, we do not need to define rules to allow a trusted domain to access an untrusted domain, except that some basic domain transition rules are needed to enable the program launcher to invoke untrusted processes from untrusted programs. Note that in mobile phone devices, trusted domains typically do not actively read untrusted data except that framework daemons accept service requests from untrusted processes, which is reflected by our integrity rules. For *trustedII* domain, our current policy regards them as same as untrusted domain, although they are deployed by device manufacture and regarded as trusted subjects on our evaluation platform.

We use generalized filesystem labelling mechanism (`genfscon`) [11] to label both cramfs and ext3 filesystems. This saves space in filesystem, while introduces extra

footprint in RAM as all file labelling information is loaded when the kernel boots. However, with our clearly defined trusted and untrusted filesystem partitions and limited number of types, we only have less than 20 filesystem labelling rules, including those for `proc`, `sysfs`, `selinuxfs`, and `tmpfs`.

6 EVALUATION

6.1 Security Evaluation

It is a challenging task to test a security mechanism completely. Instead, we test three types of attacks: typical attacks towards mobile platform integrity according to our threat model in Section 2, cross-service attacks, and attacks towards access control mechanism. As we do not have the source code of existing mobile malware, we mimic such attacks with our own implementation on evaluation platform and test the effectiveness of our solution.

Malicious access to system files and status data Several Symbian malware have been found which maliciously install files into system directories and disable application manager such that applications cannot be launched and the device cannot be rebooted. These include Dampig, Fontal, Locknut, and Skulls, all of which can be accidentally installed by an unintentional user. In our implementation, a user application can only be installed on untrusted side of the filesystem and it cannot write to trusted side, therefore this type of attacks can be easily prevented. WiFi faker [30] is another malware which rapidly exhausts the battery of a mobile phone. With the power management functions of `DevicePower-Notify()` and `SetDevicePower()` in Symbian system, this malware sets the WiFi radio operating in the highest power mode but shows inactive status icon in system tray. We simulate this attack using similar APIs provided by the system framework in our platform. This attack causes the phone to run out of battery in less than one hour even it is not used at all. By controlling that only trusted processes can set the WiFi status, our implementation prevents this attack successfully.

Cross-service attacks With multiple network interfaces on a single mobile device, there are multiple network connections and services available for mobile applications. Attackers can exploit from one network interface and get unauthorized access to another network service. For example, some existing mobile malware leverage vulnerability in Bluetooth stack to infect devices and thus send SMS/MMS messages via their wireless interface such as Bluebug [2] and commwarrior. One cross-service attack prototype is implemented in Windows CE mobile phone device [34], which compromises a user installed FTP server running with WiFi interface via buffer overflow vulnerability and then makes premium-service phone calls via TAPI interface. We simulate this attack with a dummy `echo` server which accepts inputs from ethernet via USB. With a simple `strcpy` buffer-over-flow vulnerability, a client can execute a shellcode which dials a premium phone number. However, when our security mechanism is enforced, as the echo server is untrusted process, the shellcode's phone call is denied by the telephony server.

Bypassing security mechanism A common way to circumvent an access control mechanism is to exploit privileged applications. This implies two methods in our system: exploit trusted processes and hijack their security level to obtain privileged accesses, or exploit trusted service daemons which enforce access control policies. We first implement a code-injection attack with a user installed application (developed by ourselves). This untrusted application maliciously injects a shellcode to any trusted process with `ptrace()` and changes the program register eip (register 15) to the address of injected shellcode. The shellcode is then executed and the malicious process writes back the original code and register thus the trusted process resumes. The shellcode simply disables the SELinux enforcement in the kernel therefore after this attack any untrusted process can get full privileges of writing to trusted side. Further, we implement a code-replacement attack to Type III trusted subject `gconfd`, which bypasses the security hook in `database_handle _set()` and always returns true (allowed) to any access request. Thus the untrusted process can set new key values to any high integrity object such as phone unlock password and data network connection profile. We implement similar attack to the telephone server by replacing the permission check hook function in `tapi_call_setup()` with a trivial one, thus an untrusted process can make phone calls to any phone number including premium rate ones. All these attacks work as both trusted and untrusted applications run with the same uid in our evaluation thus untrusted process can `ptrace` to trusted process. When our security mechanism is enabled, both attacks failed with denied write operations (`ptrace`) from untrusted processes to trusted processes.

6.2 Performance Evaluation

Our policy size is less than 20KB including `genfscon` rules for filesystem labelling. Comparing to that in typical desktop Linux distributions such as Fedora Core 6 (which has 1.2MB policy file), our policy footprint is tiny. As aforementioned, the small footprint of our security mechanism is result from the simple way to identify the borderline between trusted and untrusted domains, and the efficient way to control their communications.

We study the performance of our SELinux-based implementation with microbenchmark to investigate the overhead for various low-level system operations such as process, file, and socket accesses. Our benchmark tests are performed with the LMbench 3 suites [6]. Table 5 shows the measurements in microseconds and percentage overhead, compared with the NSA SELinux overhead on desktop environment [32]. The results show that our security enforcement has much better performance than the counterpart technology on PC [32].

TABLE 5
Benchmark Results. Smaller is Better.

Benchmark	Base (ms)	SEIP (ms)	Overhead (%)	NSA SELinux Overhead (%)
null I/O	1.89	1.97	4	33
stat	9.7	11.3	16	28
open/close	14.1	16.7	18	27
0KB create	42.2	44.9	6	18
0KB delete	28.5	29.5	4	10
fork	2182	2230	2	1
exec	7539	7801	3	3
sh	121	130	7	10
pipe	398.2	421.4	6	12
AF_UNIX	443	449	1	19
UDP	735	771.7	5	15
TCP	1017	1051	3	9
TCP connect	1742	1761	1	9

7 Limitations

Although we have implemented our design in some major services of our evaluation platform including IPC (D-Bus), telephony, device status manager, and system configuration service, obviously this is not a complete list for a whole platform. For example, due to lack of source code, we do not have implementation on data network service and message service. In general, the framework services of a mobile phone device can be provided by many different vendors, such that a complete implementation so far is not feasible in our prototype. One of our design goals is to ease the integration of security between functional frameworks. Typically, a framework provider just needs to identify sensitive functions or APIs that need to be controlled for integrity purpose, declare a set of corresponding permission names, and insert a common security hook function into the API implementations of the service functions, which is implemented in a trusted library based on our integrity rules. We believe this significantly releases the burden of security considerations for system framework developers.

One novelty of our integrity protection mechanism is that a trusted daemon can accept information (i.e., service request) from low integrity processes. Therefore we require that low integrity information cannot affect the behavior of a trusted daemon. This is viable for many daemons which are appropriately implemented to separate received low and high integrity information by inserting security hooks. However, as a daemon provides interfaces to other applications, vulnerabilities such as buffer-over-flow can be used by malicious processes to get the high privileges of the trusted daemon, thus bypassing security enforcement in the daemon as well as the whole system. By defining least privileges for the daemon we can reduce the damages when this kind of attacks occurs. However, fundamentally, our approach is not to prevent such implementation vulnerabilities. That is, the high integrity level of a subject does not mean no vulnerability of its implementation, although we put more trust on its provider's effort to eliminate vulnerabilities. Overall, SEIP takes a user-space object manager like approach to control interactions between Type III trusted subjects and the rest, which we believe is a cost-effective solution for large legacy codebase.

8 Related Work

Information flow-based integrity models have been proposed and implemented in many different systems, including the well-known Biba [18], Clark-Wilson [22], and LOMAC [25]. Biba integrity property restricts that a high integrity process cannot read lower integrity data, execute lower integrity programs, or obtain lower-integrity data in any other manner. In practices, there are many cases that a high integrity process needs to read low integrity data or receive messages from low level integrity processes. LOMAC supports high integrity process's reading low integrity data, while downgrading the process's integrity level to the lowest integrity level it has ever read. PRIMA [29], [35] and UMIP [31] dynamically downgrade a process's integrity level when it reads untrusted data. As a program may need to read and write to high integrity data or communicate to high integrity subjects after it reads low integrity data, it needs to be re-launched by a privileged subject or user to switch to high level. Although these approaches can achieve a platform's integrity status, they are not efficient for always-running service daemons on mobile devices.

Clark-Wilson [22] provides a different view of integrity dependencies, which states that through certain programs so-called transaction procedures (TP), information can flow from low integrity objects to high integrity objects. CW-lite [29], [37] leverages the concept of TP where low integrity data can flow to high integrity processes via filters such as firewall, authentication processes, or program interfaces. Different from the concept of filter, UMIP [31] uses exceptions to state the situations that require low integrity data to flow to high integrity processes. A significant difference between these and our solution is that, we do not sanitize low integrity information and increase its integrity level. Instead, our design allows a trusted process to accept low integrity data if it is separated from high integrity data thus does not affect the behavior of the process. This fits the requirements of framework services which provide functions to both high and low integrity processes on open mobile platforms.

Mulliner et al. [34] develop a labelling mechanism to distinguish data received from different network interfaces of a mobile device. However, there is no integrity model behind this mechanism, and this approach does not protect applications accessing data from multiple interfaces. Also, the monitoring and enforcing points are not complete, which only include hooks in execve(2), socket(2) and open(2) system calls, and cannot capture program launching via IPC such as D-Bus.

Android classifies application permissions into four protection levels, namely Normal, Dangerous, Signature, and SignatureOrSystem [1], [23]. The first two are available for general applications, while the last two are only available for applications belonging to those signed by the same application provider. Most permissions that can change a system's configurations belongs to the last one, which usually only allows Google or trusted party to have. This is similar to SEIP: sensitive permissions that alter platform in-

tegrity are only available to trusted domain. However, SEIP does not have a complete solution to distinguish permission sets for general third-party applications as Android does, since SIP focuses on platform integrity protection only.

Shabtai et al. [36] deploy SELinux to confine high-privileged processes on Android devices, such as those run in root and system uids. The main motivation of this confinement is to limit attacking interfaces by exploiting any vulnerability of these privileged processes; that is, it tries to offer least privileges to root and system processes. Different from this, SEIP aims to protect system integrity by confining risky operations from user downloaded application to system resources. Therefore these two approaches complement each other.

9 Conclusion

In this paper we present a simple but yet effective and efficient security solution for integrity protection on mobile phone devices. Our design captures the major threats from user downloaded or unintentionally installed applications, including codes and data received from Bluetooth, MMS and browser. We propose a set of integrity rules to control information flows according to different types of subjects in typical mobile systems. Based on easy ways to distinguish trusted and untrusted data and codes, our solution enables very simple security policy development. We have implemented our design on a LiMo platform and demonstrated its effectiveness by preventing a set of attacks. The performance study shows that our solution is efficient by comparing to the counterpart technology on desktop environments. We plan to port our implementation to other Linux-based platforms and develop an intuitive tool for policy development.

References

[1] Android, http://code.google.com/android/.

[2] Bluebug, http://trifinite.org/trifinite_stuff_bluebug.html.

[3] Gpe phone edition, http://gpephone.linuxtogo.org/.

[4] J2ME CLDC specifications, version 1.0a, http://jcp.org/aboutjava/communityprocess/final/jsr030/index.html.

[5] Limo foundation, https://www.limofoundation.org.

[6] LMbench-tools for performance analysis. http://www.bitmover.com/lmbench.

[7] Maemo, http://www.maemo.org.

[8] Mcafee mobile security report 2008, http://www.mcafee.com /us/research/mobile_security_report_2008.html.

[9] Mcafee mobile security report 2009, http://www.mcafee.com/us/local_content/reports/ mobile_security_report_2009.pdf.

[10] Motomagx security, http://ecosystem.motorola.com/get-inspired/whitepapers/security-whitepaper.pdf.

[11] NSA Security-Enhanced Linux Example Policy. http://www.nsa.gov/selinux/.

[12] OpenEZX, http://wiki.openezx.org/main_page.

[13] Pandalab report, http://pandalabs.pandasecurity.com/blogs/images /pandalabs/2008/04/01/quarterly_report_pandalabs_q1_2008.pdf.

[14] Qtopia phone edition, http://doc.trolltech.com.

[15] Security in qtopia phones, http://www.linuxjournal.com/article/9896.

[16] Setools–policy analysis tools for selinux, http://oss.tresys.com/projects/setools.

[17] The six secrets to mobile computing success, http://news.cnet.com/8301-13579_3-9929210-37.html.

[18] K. J. Biba. Integrity consideration for secure computer system. Technical report, Mitre Corp. Report TR-3153, Bedford, Mass., 1977.

[19] A. Bose and K. Shin. Proactive security for mobile messaging networks. In *Proc. of ACM Workshop on Wireless Security*, 2006.

[20] J. Carter. Using gconf as an example of how to create a userspace object manager. In *Proc. of Security Enhanced Linux Symposium*, 2007.

[21] J. Cheng, S. Wong, H. Yang, , and S. Lu. Smartsiren: Virus detection and alert for smartphones. In *Proc. of ACM Conference on Mobile Systems, Applications*, 2007.

[22] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. *Proceedings of the IEEE symposium on security and privacy*, 1987.

[23] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security & Privacy*, 7(1), 2009.

[24] W. Enck, P. Traynor, P. McDaniel, and T. L. Porta. Exploiting open functionality in sms-capable cellular networks. In *Proc. of ACM CCS*, 2005.

[25] T. Fraser. LOMAC: MAC you can live with. In *Proc. of Usenix Annual Technical Conference*, 2001.

[26] C. Heath. Symbian os platform security, symbian press, 2006.

[27] G. Hu and D. Venugopal. A malware signature extraction and detection method applied to mobile networks. In *Proc. of 26th IEEE International Performance, Computing, and Communications Conference*, 2007.

[28] M. Hypponen. State of cell phone malware in 2007, http://www.usenix.org/events/sec07/tech/hypponen.pdf.

[29] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: Policy-reduced integrity measurement architecture. In *Proc. of ACM SACMAT*, 2006.

[30] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proc. of MobiSys*, 2008.

[31] N. Li, Z. Mao, and H. Chen. Usable mandatory integrity protections for operating systems. In *Proc. of IEEE Symposium on Security and Privacy*, 2007.

[32] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of USENIX Annual Technical Conference*, pages 29 – 42, June 25-30 2001.

[33] T. Lunt, D. Denning, R. Schell, M. Heckman, and M. Shockley. The seaview security model. *IEEE Transactions on Software Engineering*, 16(6), 1990.

[34] C. Mulliner, G. Vigna, D. Dagon, , and W. Lee. Using labeling to prevent cross-service attacks against smart phones. In *Proc. of Conference on DIMVA*, 2006.

[35] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring integrity on mobile phone systems. In *Proc. of ACM SACMAT*, 2008.

[36] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using selinux. *IEEE Security & Privacy*, 8(3), 2010.

[37] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proc. of NDSS*, 2006.

[38] D. Venugopal, G. Hu, and N. Roman. Intelligent virus detection on mobile devices. In *Proc. of International Conference on Privacy, Security and Trust*, 2006.