Old, New, Borrowed, Blue – A Perspective on the Evolution of Mobile Platform Security Architectures

Kari Kostiainen
Nokia Research Center
kari.ti.kostiainen@nokia.com

Elena Reshetova
Nokia
elena.reshetova@nokia.com

Jan-Erik Ekberg
Nokia Research Center
jan-erik.ekberg@nokia.com

N. Asokan
Nokia Research Center
n.asokan@nokia.com

ABSTRACT

The recent dramatic increase in the popularity of "smartphones" has led to increased interest in smartphone security research. From the perspective of a security researcher the noteworthy attributes of a modern smartphone are the ability to install new applications, possibility to access Internet and presence of private or sensitive information such as messages or location. These attributes are also present in a large class of more traditional "feature phones." Mobile platform security architectures in these types of devices have seen a much larger scale of deployment compared to platform security architectures designed for PC platforms. In this paper we start by describing the business, regulatory and end-user requirements which paved the way for this widespread deployment of mobile platform security architectures. We briefly describe typical hardware-based security mechanisms that provide the foundation for mobile platform security. We then describe and compare the currently most prominent open mobile platform security architectures and conclude that many features introduced recently are borrowed, or adapted with a twist, from older platform security architectures. Finally, we identify a number of open problems in designing effective mobile platform security.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Security and Protection

General Terms

Security, Design

1. INTRODUCTION

In the past few years, there has been a dramatic increase in the popularity of the category of mobile phones commonly known as "smartphones." Consequently there is increased interest in the security and privacy research community in "smartphone security". What exactly constitutes a "smartphone" is a matter of debate [16]. But from the perspective of a security researcher, some attributes of smartphones stand out. One is the ability to extend the functionality of phone by incorporating new software components. At the moment, this takes the form of installing new applications. Another is the ability to access (and be accessed from) the Internet. A third is the presence of private or sensitive information like personal messages, location etc.

These security-relevant attributes lead us to two important observations. First, these attributes are also present in a large class of mobile phones commonly known as "feature phones". For example, Java Platform Micro Edition (Java ME), which makes it possible for application developers to develop and deploy Java midlets to mobile devices, is reportedly present on over three billion phones [23]. Therefore, we argue that instead of focusing on "smartphone security", security researchers should study mobile platform security more generally.

Second, these attributes are instantly recognizable as characteristics of any personal computer (PC) platform. PC platforms started out as open systems with no platform security schemes in place. Even today, security mechanisms in PC platforms are based primarily on perimeter control, like firewalls, and reactive mechanisms like anti-virus tools. Although various platform security architectures (such as Security-Enhanced Linux [17]) have been designed and implemented, none has seen widespread deployment.

In contrast all significant mobile phone platforms have widely deployed platform security schemes. The primary reason for this is how the business, regulatory, and end-user requirements on mobile phones have shaped the evolution of mobile platform security over the last decade or so.

In this paper, we begin by taking a brief look at the motivation and background for mobile platform security and the requirements they implied. Then we describe the types of hardware-security mechanisms that provide the foundation for mobile platform security. After that we describe and compare the currently most prominent open mobile platform security architectures. Finally, we identify a number of open problems in designing effective mobile platform security.

2. BACKGROUND AND MOTIVATION

In contrast to PC platforms, mobile phones began as closed systems with limited functionality. Right from the beginning different stakeholders had certain clear security require-

CODASPY'11, February 21–23, 2011, San Antonio, Texas, USA.

ments for mobile devices. For example, in many parts of the world, several *mobile network operators* provide subsidized mobile phones to their customers in return to commitment to a specified contract period. They require mobile phones to incorporate mechanisms to enforce *subsidy locks*, to prevent a customer who received a subsidized mobile phone from changing operators before the subsidy contract was ended. *Regulators*, like the Federal Communication Commission in the United States, are interested in aspects affecting the public good. For example, during manufacture a mobile phone undergoes a configuration process where the parameters for radio frequency (RF) operations (like transmission power) are calibrated and stored. Regulators want to ensure *secure storage* of such RF parameters for end-user safety. Additionally, if a malicious user could manipulate these parameters he could gain unfair advantage (bigger communication bandwidth than was intended for him) or disrupt communications for other users. Regulators are also interested in theft-deterrent mechanisms such as the means to track stolen devices.

Some of these requirements made their way into standards specifications. The European Telecommunications Standards Institute (ETSI), the body that originally specified the Global System for Mobile Communications (GSM) standard had representatives from mobile device manufacturers, mobile network operators as well as regulators. In the early 1990s, a version of the ETSI recommendation on "security aspects" recommended protection of the IMEI (International Mobile Equipment Identifier, a unique code for a specific device) and the IMSI (International Mobile Subscriber Identifier, a unique code for each subscription) by specifying that "Both IMSI and IMEI require physical protection. Physical protection means that manufacturers shall take necessary and sufficient measures to ensure the programming and mechanical security of the IMEI" [10]. Immutability of IMEI and IMSI is essential for enforcing subsidy lock. Immutability of IMEI also serves as a theft-deterrent mechanism.

Ten years later, by the time the first "smartphone" (the Nokia 9210) appeared, the importance of these basic requirements grew. A subsequent version of the same ETSI specification re-iterated the requirement that "The IMEI shall not be changed after the ME (mobile equipment) final production process. It shall resist tampering, i.e. manipulation and change, by any means (e.g. physical, electrical and software)" [11]. It also implied that compliance to this requirement would be needed for type approval by stating that "This requirement is valid for new GSM ... MEs type approved after 1st June 2002."

In addition to operator and regulator requirements, it was also evident that *end-users* had come to expect a certain level of predictability and reliability from mobile phones. Unlike in PC platforms, where malfunctioning system software or malicious applications are usually merely a nuisance to the user, in mobile domain such software can cause considerable harm to him, e.g. in the form of monetary losses due to an increased phone bill. To retain that level of end-user trust while opening up mobile phone platforms for application installation and Internet connectivity, the platforms needed to support appropriate platform security schemes.

Mobile platform vendors responded to these operator, regulator and end-user needs by developing "hardware-security mechanisms" and by integrating "platform security architectures" to the mobile operating systems and application platforms. In the next sections of this paper, we describe typical hardware-security mechanisms used in mobile phones and compare widely deployed open platform security architectures.

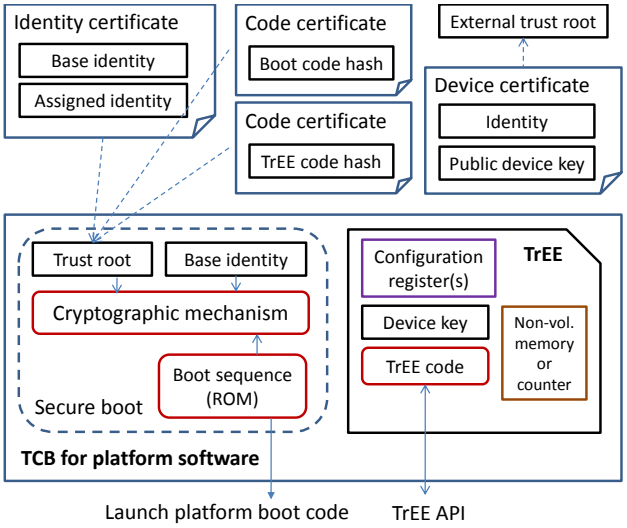

Figure 1: Common hardware-security mechanisms used in mobile devices.

3. HARDWARE-SECURITY MECHANISMS

The augmentation of device hardware to support operating system security is by no means a new invention. As early as in 1971, Lampson examined the concept of protection domains [15] and e.g. the Cambridge CAP computer [37] developed in the 1970s provided hardware support for such a feature. The first standardization efforts for hardware-assisted security took place in late 1990s when the Trusted Computing Platform Alliance (TCPA) [25], a predecessor of current Trusted Computing Group (TCG) [34], defined a hardware-based security element for PCs. Roughly at the same time, the first large scale deployments of hardware-assisted security started when mobile device manufacturers added hardware-based integrity protection of booted system software. The mechanism was originally designed as a safety feature rather than a security one, but in the context of mobile devices its primary purpose was to prevent software replacement in the field.

In this section we describe common hardware-security mechanisms used in mobile devices today (see Figure 1), and explain the rationales of mobile device manufacturers for introducing such security mechanisms. Then we briefly list some existing and standardized hardware-security architectures in the mobile domain that implement some or all of the explored mechanisms.

3.1 Base identity and trust root

For typical mobile devices, two pieces of immutable information are needed in hardware. First, virtually all devices need at least one *base identity* that uniquely identifies the device hardware. The base identity can be the IMEI of the device or any other (statistically) unique identifier. Immutability of the device identifier can be achieved e.g. by storing the value in a read-only memory (ROM) during the

manufacturing of the application-specific integrated circuit (ASIC).

Second, mobile devices need to authenticate external information (see e.g. assigned identities and secure boot in Sections 3.2 and 3.3). A pre-condition for external information authentication is that another piece of immutable information, a *trust root*, is stored on the device hardware during manufacturing. A typical trust root is a hash of the device manufacturer public key [32, 33], and thus the trust root is usually shared within a family of manufactured devices, unlike the device-specific immutable identity which is typically unique.

3.2 Assigned identities

Mobile devices usually need more than one identity, e.g. for MAC addresses of all supported radio interfaces. The ability to assign additional device identities after the ASIC manufacturing, rather than to fixing them into ROM at the time of manufacturing, gives more flexibility to the device manufacturers. The combination of a trust root and at least one base identity gives the device manufacturer or integrator the possibility to assign arbitrarily many other identities to a chosen device.

One way to achieve this is to issue an *identity certificate* for the device. This certificate is signed with respect to the device trust root and it binds the an *assigned identity* to the base identity. Additionally, the device hardware must be enhanced with a *cryptographic mechanism* that can verify the identity certificate (or a certification chain). The operational integrity of the cryptographic mechanism implementation must be trustworthy, and thus such algorithms, say RSA implementation, are often part of the ROM on the ASIC, where they can be deployed during the boot sequence at a stage where no untrusted code has yet been run.

3.3 Secure boot

Due to regulatory, operator and end-user requirements some mobile device manufacturers assert the integrity of system software during device boot-up and stop the boot process if system software has been modified. This process is called *secure boot*. To implement secure boot, a typical approach is to make the beginning of the *boot sequence* immutable, e.g. by the virtue that it resides in ROM, and that the processor unconditionally starts executing from that memory area. The device manufacturer can issue a set of *code certificates* that contain *boot code hashes* and are signed with respect to the device trust root. Again, the hardware must be enhanced with a cryptographic mechanism that validates the first executed system component (e.g. boot loader) before it further validates and executes the next one (e.g. OS kernel). If any of the validation steps fail, the boot process aborts.

This "measure-before-execute" principle can be iterated as many times as necessary to include not just the initially launched OS kernel (or hypervisor) binary, but also any other code or configuration data whose integrity needs to be validated. Secure boot based on code signing does not necessarily imply that only a single code set-up can be booted on a given device. The launched image may be one of several certified alternatives, depending on user selection (see Section 5) or other external or internal context.

3.4 Trusted Execution Environment

All the hardware-security mechanisms listed so far have been integrity-related, and implementation of these mechanisms does not require storage of or operation on secrets. There are, however, security services and use cases (see Section 3.7) that require secure storage and isolated execution.

A way to construct an isolated execution environment is to validate the Trusted Computing Base (TCB) using secure boot, since TCB by definition is isolated from the rest of the system. This approach may suffice, especially in cases where the TCB is small, like a hypervisor. For complete operating systems (kernels) this is not a viable approach, since these are so large that implementation bugs that enable software-attacks against TCB are inevitable.

When dealing with confidential information, the attack model also changes. When attacking integrity features, isolation must be broken at every boot, whereas a breach against confidentiality needs to succeed only once and the secret falls into the hands of the attacker. Thus, where confidentiality is concerned, at least simple hardware attacks like memory-bus monitoring or side-channel attacks must be considered as plausible attack vectors in addition to software-based attacks.

To overcome the security problems of TCB that consists of entire OS kernel, mobile device manufacturers have enhanced their ASICs with support for *Trusted Execution Environment* (TrEE). A typical TrEE includes secure storage for a (statistically) unique *device key* and an execution environment in which small pieces of code (*TrEE code*) can be executed isolated from the rest of the system. The isolated execution environment is typically based on on-chip memory to alleviate memory-bus attacks. Combined with the secure boot features, trust roots and identities described above, this set effectively can become a minimal *TCB for platform software* to be leveraged by the booted operating system or platform software on the device.

Like with any TCB, since the TrEE will contain and provide access to secrets, there must be some assurance that code that is run inside it will not, as part of its operation, reveal the secrets to outside, either accidentally or by malice. This assurance can be achieved either by code-signing (code certificates that contain TrEE code hashes) or by constructing the TrEE such that any code run in it gets only indirect access to the confidential data (e.g. a secret key can be applied to a cryptographic algorithm, but it is never directly accessible to TrEE code). The same applies to runtime protected data that may be stored in TrEE memory. TrEE typically provides an API for loading executed code.

3.5 Configuration registers

Additionally, TrEEs often support *configuration registers*. These registers can be used to store measurements from the (possibly validated) booted software or an aggregate of the non-validated code the TCB executes over time. Also hardware configuration options, configuration file contents or user inputs can be stored on these registers. The integrity of configuration registers must be guaranteed, but it is not persistent across boot cycles. The information contained in such registers can be used in two ways. First, code run inside the TrEE can adjust their logic based on the current system state, or e.g. user input received at boot when no untrusted software still was running. Secondly, the state of the system can be attested to a remote party (see Section 3.7).

3.6 Authenticated boot

A boot process in which (1) the measure-before-execute principle is followed during boot without certificate validation nor possible boot termination, and (2) all intermediary measurement results are added to configuration registers (without the possibility of data rollback), is often called *authenticated boot*. The aggregate information from the registers will uniquely identify the system state up to the instance where the first piece of untrusted code is launched. Put another way, if no untrusted code was launched, this fact can later be attested to external parties without the need for certificate checking or halting during boot.

3.7 Sealing and attestation

A persistent device key initialized during ASIC manufacturing and only accessible within the TrEE, can be used for various security mechanisms. First, with this key, or any derivation from it, the TrEE can locally encrypt data, and then give the result to an unprotected domain, such as external memory, for persistent storage. This mechanism is often called *sealing*.

If the device key is usable for public key cryptography, then an external certification authority (CA, typically run by the device manufacturer) may issue a *device certificate* that binds the public part of the device key to any of the device identities (base or assigned). The certification process can take place e.g. during device manufacturing, when the CA can still trust the integrity of the public part of the device key.

The *public device key* together with the device certificate can be used to set up an authenticated and confidential communication channel from an external party into the TrEE, e.g. for provisioning or data migration purposes. Also, the TrEE can by means of a signed statement to a third party remotely attest any state information inside it, e.g. the measurements stored to configuration registers from an authenticated boot sequence.

3.8 Statefulness

Many services in which the user can be considered an adversary, such as digital rights management, device lock PIN retry control or micropayment protocols, need reliable rollback protection of system state. During one device boot cycle, rollback protection is implementable using the configuration registers described in Section 3.5. To implement *off-line* rollback protection, stored data needs to be bound to reliable time information within the TrEE. Straightforward ways to implement this is to include either a monotonic *counter* or *non-volatile memory* in the TrEE.[1]

3.9 Hardware-security architectures

Most of the hardware-security mechanisms described above have seen widespread deployment in proprietary mobile *hardware-security architectures*, such as ARM TrustZone [3, 4] or M-Shield [32]. TrustZone architecture augments the processing core to provide a new set of processing (register) contexts, as well as memory management unit (MMU) and direct memory access (DMA) security integration. ASICs with TrustZone architecture support secure boot, and can also be populated with isolated RAM and ROM residing within the ASIC package to provide memories resistant to simple hardware attacks. For confidential information, trust roots and identifiers some amount of chip-specific "write once, read many times" persistent memory (typically implemented with E-fuses) is also available. Thus TrustZone provides secure boot, integrity protected trust roots and device identifiers, confidential device keys and isolated execution for TrEE code, but typically lacks secure non-volatile memory and counters, and thus rollback protection must be set up with external means.

On the PC side, the most widely deployed hardware-security architecture is the Trusted Platform Module (TPM) [35] defined by Trusted Computing Group (TCG). TPM by definition is a self-contained, stand-alone secure element. TPM does not provide secure boot, but combined with an associated processor feature, the Dynamic Root of Trust for Measurement (DRTM), it can be used to set up a limited TrEE [19] within the logic confines of an Intel VTx or AMD processors. In this context TPM has trust roots, device secrets, device certificate capability, counters, i.e. all features listed above.

TCG has also defined a standard called Mobile Trusted Module (MTM) [9] for mobile devices. Unlike TPM, MTM is an interface specification that can be implemented using various means, e.g. using TrustZone. MTM specification supports most TPM features, with a few mobile-domain specific additions.

4. OPEN MOBILE PLATFORMS

In this section we briefly describe the prominent open mobile operating systems and application platforms: Symbian, Java ME, Android and MeeGo. We focus on mobile platforms for which reference implementations and security architectures are publicly available, and exclude other popular mobile platforms, such as iPhone, Windows Phone 7 and Blackberry, that are open to third-party application development but the internals of which are not public.

4.1 Symbian

Symbian is an evolution from EPOC operating system used in Psion devices in 1990s. In Symbian most of the operating system services, such as file system and networking, are implemented as user-space system servers. Communication between system servers and user applications takes place via built-in interprocess communication (IPC) framework [28]. Symbian platform security architecture was added in 2005 and at the time Symbian was the first smartphone OS to incorporate a platform security architecture. Symbian is currently the most used smartphone OS [12] and primarily used in Nokia devices. Application development in Symbian is done in C++ and using Qt framework. Symbian operating system supports three types of executables: UI applications, background servers and libraries.

Symbian developers define two configuration files for each application: a project definition file (MMP file) defines project settings, such as the identifier of the application and source code files and libraries used, and a packaging file (PKG file) controls how an installation package in constructed.

In the Symbian platform security architecture, access to

[1]Many existing mobile devices support neither non-volatile secure memory nor secure counters (see e.g. [30] for rationale). In some cases the need for *local* rollback protection can be mitigated; e.g. when on-line server communication is an unconditional requirement of the use case at hand, the rollback-protection can be server-assisted, i.e. the server provides authenticated time information.

protected APIs is controlled using a finite set of permissions which are called "capabilities". Applications that access protected APIs must be signed by a central trusted authority (SymbianSigned). During the signing process the trusted authority checks that the application conforms to publication criteria and assigns a globally unique application identifier (Secure Identifier, SID) from a *protected range* of application identifiers. The authority maintains a mapping between the issued identifiers and identities of the software issuers. For most applications the developer identity verification is based on simple online registration (nominal fee of 1 euro). For applications that require restricted capabilities, the application developer must purchase a publisher identity certificate (200 USD per year). Nokia Ovi Store is the primary distribution channel for Symbian applications.

Symbian applications that do not require access to protected APIs can be self-signed and distributed via other channels. In such a case, the developer picks the application identifier from an *unprotected range*.

4.2 Java ME

Java Micro Edition (Java ME) is an application platform supported by various devices from embedded devices to mobile phones and set-top boxes. Java ME platform consists of device "configurations" that define the used Java virtual machine and the core APIs, and "profiles" that define additional APIs for building complete applications. Mobile phones typically support Connected, Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP). Java ME is the most widely supported third-party application development platform for feature phones with over 3 billion devices deployed [23]. However, many latest high-end smartphone platforms, including Android, iPhone and MeeGo, do not support Java ME.

The current MIDP standard [21] supports only standalone UI applications, or *midlets*.[2] Midlets are packaged into JAR files before deployment. Applications specific attributes, such as name, version, vendor and requested permissions, are shipped in a Java Desriptor (JAD) file or manifest file.

In Java ME platform security architecture access to protected APIs is controlled with permissions. Application signing binds the application to a *protection domain* according to a local security policy. The policy defines the permissions that applications of each protection domain can have. Typically mobile phones have four predefined protection domains: device manufacturer domain, network operator domain, and domains for identified and unidentified third-party applications.

4.3 Android

Android is a Linux-based smartphone OS developed by Google. Android was released in 2008 and currently it is the second most used smartphone operating system [12]. Application development in Android is primarily done in Java, although applications can include native components as well. Each Android application runs in a separate Dalvik virtual machine in its own process context. Android provides an IPC framework for communication between Java applications.

[2]Next MIDP version [22] adds support for background midlets, shared libraries and inter-midlet communication. We exclude analysis of these features from our discussion, because this version is not yet widely supported.

In Android platform security architecture access to protected APIs is controlled with permissions. Android applications are distributed as Android packages. A manifest file inside the package defines the permissions requested by the application and permissions required to use the services (IPC APIs) offered by the application. Android applications must be signed before installation to the device. Most third-party Android applications can be self-signed (applications accessing system APIs must be signed by Google). Android Market is the primary distribution channel for Android applications. Publishing an application requires a registration fee of 25 USD.

4.4 MeeGo

MeeGo is an upcoming Linux-based mobile platform developed jointly by Intel and Nokia. MeeGo is an evolution from Nokia's Maemo platform and Intel's Moblin OS. Compared to Android, MeeGo is much closer to a standard Linux distribution. Application development is primary done in native C/C++ and using Qt framework. MeeGo supports IPC between applications via standard Unix sockets and with Desktop Bus (D-Bus) framework [6].

MeeGo provides a new platform security architecture called Mobile Simplified Security Framework (MSSF) [14, 13]. MSSF is an evolution from Maemo 6 platform security solution, which was initially developed by Nokia. In this paper we will concentrate on the latest design of MSSF framework [20]. In MSSF access to sensitive APIs and files on the device can be controlled using both traditional Linux access control rules and permissions that are called "resource tokens".

MeeGo applications may be installed from various software sources. The notion of a "software source" is abstract and can represent a different range of entities starting from central software repository, such as Nokia Ovi Store, to single developers. Individual software sources are part of a tree-like structure, and a *trust level* is associated with each software source in hierarchical manner.

MeeGo applications are distributed as RPM packages. Each package must be signed by the software source. MeeGo devices have a local list of known software sources and their public keys. A local security policy on a MeeGo device defines *trust levels* for known software sources and permissions that each software source is allowed to grant. The local security policy can either be defined by a manufacturer, operator or even a device user, when a device booted in "developer mode" (see Section 5).

5. OPERATING SYSTEM BOOTSTRAPPING

In this section we start a comparative security analysis of the open mobile platforms described in the previous section. We begin our analysis be comparing different operating system validation and bootstrapping approaches that are used in these platforms.

Symbian. Most Symbian devices support secure boot, with hardware-security architectures like ARM TrustZone, as described in Section 3. Thus, with Symbian devices developers cannot boot their own custom kernels.

Android. In Android, the bootstrap issue is up to the device manufacturer. There is little information available about the different bootstrapping schemes chosen by different Android device manufacturers, but at least in principle developers can update devices with custom kernels [36].

MeeGo. Also in MeeGo different device manufacturers

may implement different OS bootstrapping strategies. Nokia MeeGo devices can support a dual boot approach in which the device can be booted to *normal mode* with official OS kernel image provided by the device manufacturer or to *developer mode* with custom kernel provided by any developer [26]. Integrity of each component of the boot sequence, starting from the bootloader, is verified using boot certificates (see Section 3.3 for more details). However, unlike in usual secure boot, if the integrity verification of operating system image fails, the boot process is not halted, but the user is notified and asked permission to continue the boot. If the user decides to continue, this information is stored in a configuration register inside TrEE.

Later, when the OS requests access to certain device secrets, such as digital rights management keys, this access is prevented (inside TrEE) if the device was booted to developer mode. Both modes allow user space applications to utilize cryptographic services provided by the TrEE. The keys for these services are derived from the device key, and the derivation process includes information about the device mode. This guarantees that the content, encrypted in the normal mode, can not be decrypted in the developer mode.

6. PLATFORM SECURITY ARCHITECTURE COMPARISON

Next, we compare the platform security architectures of mobile platforms described in Section 4. The key differences and similarities are summarized in Table 1.

6.1 Application identification

Symbian. During application installation and at runtime each executable (application or background server) has two identifiers: Secure Identifier (SID) uniquely identifies the executable and Vendor Identifier (VID) identifies the software vendor. These identifiers are typically assigned by the central trusted authority; for self-signed applications the developer may pick any SID from the unprotected range. Libraries inherit SID and VID from the executable that loads them.

Java ME. During installation, a midlet is identified based on installation package signature key and midlet attributes, such as package name. Midlets are standalone applications that do not communicate with each other, and thus runtime code identification is not applicable to Java ME.

Android. During installation Android applications are identified based on signing key and package name. The installation process assigns a locally unique Linux user identifier (UID) to each installed application and at runtime Android applications can be identified either based on UID or package name (one should note that package names are not globally unique, developers may freely use any package name). Only applications that are signed with the same key can be assigned the same UID.

MSSF. During installation applications are identified by software source (signing key) and package name. At runtime applications can be identified by a globally unique application identifier that consists of three parts: software source, package name and package-specific application identifier.

6.2 Application update

Symbian. Centrally signed Symbian application can only by updated by an installation package that has been assigned the same SID by the central trusted authority. Self-signed applications can be updated by any installation package that has the same SID from the unprotected range.

Java ME. Update of signed midlets is allowed only if the new midlet package is signed by the same key as the previous version was. This approach is often called *same-origin policy*. For signed midlet update the application persistent storage is retained. In unsigned application update the user is asked whether the new application should have access to the persistent storage of the old application.

Android. In Android, application update is always based on same-origin policy, i.e. an application can be updated only from an installation package that is signed with the same developer key as the currently installed application. Updated application gets access to the same data storages as its previous version.

MSSF. Application can be updated if the installation package is signed by the same software source from which it was originally installed or by another software source that has higher trust level. Also in MSSF, the updated application automatically gets access to data of previous application version.

6.3 Permission granularity

Symbian. Symbian platform security architecture defines a fixed number (21) of permissions (capabilities). The capabilities are divided into four categories: User Capabilities can be granted by the user during application installation. System Capabilities require application signing by the central trusted authority. Restricted Capabilities require application signing with stronger application developer identity checking (publisher identity certificate). Manufacturer Capabilities are reserved for device manufacturer.

Java ME. Java ME provides more fine-grained permission set. System API developers can define their own permissions. The permissions are mapped into coarse-grained "function groups". The purpose of the function groups is to present permission requests to the user in human understandable format. The MIDP specification recommends 15 function groups for mobile devices.

Android. The default permission set defined by Google includes 112 permissions. The permission names are intended to be user understandable, but in practice this is not the case with all permissions (e.g. a permission named "BROADCAST_STICKY"). Android developers may define their own permissions, and thus the number of permissions is unlimited. Permissions are categorized into four protection levels: Normal, Dangerous, Signature and SystemOrSignature.

MSSF. In MSSF architecture access control can be defined using traditional Linux access control mechanisms and using permissions (resource tokens). Resource tokens names should be understandable for users. MSSF provides a standard set of global resource tokens. Additionally, applications can define their own local resource tokens. The number of permissions in MSSF is unlimited.

6.4 Permission assignment

Symbian. A developer declares the requested permissions in MMP file. For most applications the permission assignment is done by the central trusted authority during application signing. During application installation the Symbian installer validates application signature if System,

	Symbian	Java ME	Android	MSSF
application identification at runtime	application and vendor identifier assigned by central authority	not applicable	local UID and package name	sotfware source, package name and package-specific application identifier
application update	application identifier assigned by central authority	same-origin policy and user approval	same-origin policy	trust level of software source
permission granularity	coarse-grained	fine-grained and coarse-grained groups	unlimited permissions	unlimited permissions
permission assignment	signature by central authority and user at installation	signature by protection domain owner and user at runtime	user at installation	signature by software source
runtime application integrity	dedicated directory (manufacturer permissions)	Java sandboxing	Linux access control and Java sandboxing	Linux access control and permissions and IMA [27]
offline application integrity	not supported	not supported	not supported	hardware-assisted EVM [31]
access control policy declaration	permissions and application and vendor identifier	permissions	permissions	permissions and Linux access control and application identifier
access control policy scope	system APIs and application IPC	system APIs	system APIs and application IPC	system APIs and application IPC and file access
runtime application data protection	dedicated directory (manufacturer permissions)	Java sandboxing	dedicated directory	fine-grained permission-based policies
offline application data protection	hardware-assisted with restricted API	not supported	not supported	hardware-assisted with file system integration

Table 1: Summary of key features in mobile platform security architectures.

Restricted or Manufacturer Capabilities are requested. For requested User Capabilities a user prompt is generated during application installation.

The permission set that an application gets during installation remains the same throughout the application lifetime. When an executable loads a library, the operating system checks the capabilities of the library. The library must have all capabilities of the executable, otherwise loading fails. (Because of this Symbian system libraries typically have almost all capabilities.)

Java ME. The developer declares the permissions that his application needs in JAD or manifest file. When an application is installed, the signature is checked against the protection domains on the device. If the protection domain does not support the requested permissions the installation is denied. At application runtime API calls that require user-grantable permission trigger a prompt to the user. The prompts are presented in terms of function groups. If the user grants the requested permission, the decision applies to all permissions of the same function group. The user can grant permissions to a midlets permanently, for midlet execution lifetime or for one-time access. The permissions of a midlet remain constant during midlet lifetime.

Android. Android developers declare the requested permissions in a manifest file. Normal permissions do not require explicit user granting. Each application that requests such permissions will get them. Dangerous permissions must be granted by the user during application installation. Signature permission can be given to an application, only if it is signed by the same key as the application that declared the permission. SystemOrSignature permissions are additionally granted to OS manufacturer applications. The permissions set that an Android application gets during application installation remains constant during application lifetime.

MSSF. Developer declares requested permissions (resource tokens) in the application manifest file. The manifest file can additionally declare a requested Linux UID, GID and POSIX permissions. When an application is installed, the installer checks the requested permissions against the permissions of the software source from a local policy file. The application is granted the intersection of these two permission sets. A special resource token type, unique application identifier, is generated by the installer for all applications.

At runtime applications can request the kernel to drop some of their permissions. When an application loads a shared library, its set of permissions stays the same, but the library loading may fail if a library comes from a software source that cannot grant all the permissions that the application currently possesses. The permission set of an installed applications can also increase during the lifetime of the application. This happens when a plugin library is installed as an extension to an already installed application.

If the plugin requires permissions currently not possessed by the application, these permissions can be added to the permission set of the application, if software sources of both the application and the plugin are allowed to grant the missing permissions.

6.5 Application integrity

We use term "application integrity" to refer to the protection of installed applications against unauthorized modifications both when the system is running (runtime application integrity) and when the device is powered down (offline application integrity).

Symbian. Executable files are kept in and exclusively loaded from a dedicated directory (/sys/bin) in the device internal memory. Only processes with manufacturer capabilities are allowed to access this directory which provides runtime application integrity.

Symbian supports also application installation to removable memory elements. In such a case, the installer calculates a hash of the executable binary and stores the hash to device internal memory and the executable itself to removable element. When an executable is loaded from the removable memory element, a hash of the executable is calculated again and compared to the one stored on internal memory.

Symbian platform security model does not support offline application integrity protection. Instead, the platform security architecture relies on the assumption that the device internal memory cannot be accessed, and thus the already installed applications cannot be modified, by the attacker when the device is powered down.

Java ME. Midlets are executed in a sandbox of Java virtual machine and do not have direct access to the device file system which prevents them from modifying each other. Java ME platform security architecture does not address offline application integrity.

Android. Applications are stored in directories that are assigned unique Linux UIDs during installation. Standard Linux access control mechanisms prevent applications from modifying each other. Additionally, Android devices do not have root account available and third party applications cannot run with root UID, which preserves integrity of these directories. Java sandboxing prevents applications from modifying their own stored attributes such as permissions.

MSSF. Running processes as root is not explicitly prevented in MeeGo devices, and thus relying on UID-based access control is not enough. Instead a combination of Integrity Measurement Architecture (IMA) [27] and Extended Validation Module (EVM) [31] is used. During application installation a reference hash for executable binaries (and also other executable file, such as scripts) is calculated. The reference hashes are stored in an extended Linux file system attribute called security.ima for each file. IMA/EVM verifies these hashes when applications are executed. The security.ima attribute is automatically recalculated, when an application binary is modified during system run-time. MSSF uses the Smack kernel module in order to enforce the access control permissions of the filesystem in addition to standard Linux filesystem permissions.

The offline integrity of security.ima attribute and other file attributes is preserved using hardware-based TrEE. EVM module calculates a keyed message authentication code using a key that is protected by TrEE. This prevents unnoticed modification of security.ima when the device is powered down.

6.6 Access control policy

We use term "access control policy" to refer to both the declaration and scope of rules that control access to the system APIs and IPC services provided by installed applications.

Symbian. Symbian servers can provide services to other Symbian executables through Symbian IPC framework. Developer of a Symbian server defines access control policy for the server APIs by assigning required permissions and application identifiers (SID and VID) for each API function call. The access control policy declaration is done by writing C++ code. The Symbian IPC framework automatically enforces permission-based access control rules for IPC function calls (SID or VID based access control enforcement must be implemented in code).

Java ME. In Java ME platform system APIs provide access to protected resources. Midlets themselves cannot communicate with each other in current MIDP version. System API developers define access control policies by assigning required permission to appropriate API function calls in Java code. The API developer may either reuse existing permissions or define their own.

Android. Android applications can provide services to each other through Android IPC framework. Developers declare access control policies by defining the set of permissions that are needed to use entire service (defined in manifest file) or to use an individual function from an IPC service interface (implemented in code). Also Android IPC framework automatically enforcement permission based access control rules.

MSSF. Applications can communicate with each other through D-Bus and local socket interfaces. Application developers may declare access control policies for such IPC in terms of permissions (resource tokens) or traditional Linux access control mechanisms (e.g. UID or GID). The access control policy declaration is done in the application manifest file. Developers can use common system wide permissions or declare their own application specific permissions. Unlike in other platforms MSSF access control policies can also be defined for file access (e.g. an application developer may define the resource tokens needed to access any of the files created by the application). Internally, the access control enforcement is done by Smack kernel module [29].

6.7 Application data protection

We use term "application data protection" to refer to the protection of application persistent storage (e.g., files on device file system) against unauthorized modification and eavesdropping both when the device is running (runtime protection) and powered down (offline protection).

Symbian. In Symbian platform security model a dedicated directory is created in the file system for each application. This directory can only be accessed by the owner application or a process with manufacturer capabilities. Application developers may define whether the contents of application private directory should be included to backups that are made from the device data. The default policy is to exclude private directory contents from backups.

Nokia Symbian devices provide a restricted API for sealing (authenticated encryption) data with TrEE-resident device

key (or derivation of it). This feature provides support for offline data protection for certain Symbian applications.

Java ME. In Java ME architecture applications do not have direct access to device file system, instead database system is provided for persistent storage. Databases are either private to the application itself or shared with all other applications on the same device. Java ME platform security model does not address offline application data protection.

Android. Each application has a dedicated directory protected with Linux UID and GID. By default, the files in this directory can be written and read only by the application itself. During file creation, the application may explicitly define that the created file should be readable or writable by other applications as well. Android platform provides an automatic backup feature that creates backups of application data to an online server. These backups are protected by Google account authentication.

MSSF. MSSF architecture provides fine-grained data caging model. Application developers can define in manifest file required permissions for each type of file access (read, write etc.) for each application file. By default applications files can be accessed only by the application itself.

MSSF allows applications to encrypt data using a TrEE-resident device key. The key derivation can be based on application identifier or specified resource token which allows an application to encrypt data only for itself or to a set of applications. This encryption feature is integrated to the device file system which allows legacy applications to utilize offline data protection without changes.

7. DISCUSSION

Modern mobile platform security architectures have generously borrowed from old ideas but with new twists to adapt these ideas to the needs of mobile devices. For example, all of the software security architectures discussed in Section 4 incorporate access control schemes built around the notion of permissions that are granted to the subjects and checked at the time of access control. This is similar to the VAX/VMS notion of "privileges" introduced back in the late 1970s [24]. However, while privileges in VAX/VMS are typically granted to a user of the system, in modern platform security architectures, they are granted to software modules. Similarly, the notion of secure bootstrapping was discussed in the 1990s [2], but saw widespread adaption when smartphones started to be deployed. As described in Section 5 MSSF borrows from the notion of "authenticated boot" introduced by the Trusted Computing Group, with a new twist: allowing any OS software image to be booted, but if the booted image is not authorized by a trusted party (e.g., device manufacturer), the user is alerted to the fact; furthermore access to sensitive resources (such as cellular network access) or data (such as encrypted device-specific keys) are rendered inaccessible by the OS.

Other examples of old techniques adapted, sometimes with new twists, include the use of code signing for code identification and as the basis for permission assignment. Offline data caging (discussed in Section 6.7) makes use of hardware-assisted secure storage similar to the notion of sealing in Trusted Computing Group specifications, but with the addition that sealed data can be bound to application identities or permissions.

Despite the widespread deployment of mobile platform security architectures, a number of open problems remain.

7.1 Permission granularity

Symbian platform security took the approach of using a fixed number of pre-defined permissions. Resource and service providers who need access control attempt to find the most relevant pre-defined permission to protect their resource or service but may not always succeed. This may lead to developer confusion.

Java ME, Android and MSSF on the other hand allow fine-grained permissions and make it possible to define new permissions. But this does not completely address the problem either. Having numerous permissions can be a cause of potential confusion among users and developers [5]. For this reason some of these architectures allow permissions to be grouped together for presentation to the user.

Designing a flexible and sufficiently rich platform security architecture without sacrificing usability remains a challenge. Solutions like Security Enhanced Linux are not widely adapted because of their perceived complexity. This choice may need to be revisited.

7.2 Permission assignment

Perhaps the most serious problem with mobile platform security architectures is the issue of permission assignment. Android (and also partially Java ME and Symbian) take the approach of relying on the user to decide whether an application can be grated dangerous permissions. Two basic approaches are used: the user can be prompted to grant permissions during application installation (Android and Symbian) or during application runtime (Java ME). Both of these approaches have drawbacks. Runtime permission prompts are often considered annoying by the users while permission assignment during application installation suffers from the lack of relevant context (e.g., the user might know only at runtime whether Internet access should be granted to an application). In both cases users are ill-equipped to make permission assignment decisions and may become habituated to click-through access control prompts.

7.3 Software appropriateness

The problems of user-based permission assignment can be avoided by having a central authority that does permission assignment by means of code signing as in the case of iTunes AppStore or SymbianSigned. While centralized permission assignment is preferable from a usability perspective, it is problematic in cases where subjective judgement is involved. This is particularly so in cases where centralized judgement regarding the appropriateness of applications (e.g., classifying applications as offensive or otherwise inappropriate) is made [8]. One alternative is to rely on the wisdom of crowds (e.g., the WhatApp service at `http://whatapp.org`) or the wisdom of smaller and more personalized "cliques" [8, 7].

7.4 Access control policy enforcement

Two basic approaches for platform internal access control policy enforcement exist. First, a *reference monitor* (e.g. the platform security architecture on a mobile device) can enforce access control policies over *subjects* (e.g. caller application and callee application in IPC communication) [1]. Second, the full responsibility of access control enforcement can be left to the callee application, assuming that the underlying platform provides the needed information about the caller to the callee.

Most of the platform security architectures described in

this paper use a hybrid approach. The platform security architecture typically enforces access control policies defined in application manifest files automatically, but the application developers can implement additional access control checks, e.g. based on IPC call input data, on top. In Symbian and MSSF callee application can query attributes of caller, such as application identity and possessed permissions.

Pure reference monitor approach preserves application privacy, i.e. the callee does not learn more about the caller than what is needed to make the access control decision, while providing full information about the caller to the callee enables useful security services, such as attestation of untrusted application attributes to an external party by a trusted callee application. Finding the optimal balance between these two approaches remains an open challenge.

7.5 Colluding applications

All of the mobile platform security architectures discussed above grant and enforce accesss control to individual pieces of software. Two such pieces could collude over overt or covert inter-process communication channels so that they gain access to resources or services that neither was able to acting along [18]. Defending against this problem appears to be very hard. In particular, in cases where the user is responsible for granting permissions, visualizing the different potential collusion scenarios and their implications to the user is a security usability challenge.

8. CONCLUSIONS

"Smartphone security" is becoming a popular research topic. The fundamental security issues with smartphones are also present in a larger class of personal mobile communication devices, as well as in personal computers. However, unlike PC platforms, all dominant mobile platforms incorporate widely deployed mobile platform security architectures. The history of mobile platform security goes back long beyond the current popularity of smartphones. The widespread deployment of mobile platform security architectures is due to specified and perceived business, regulatory and end-user requirements for mobile communication devices.

We surveyed four mobile platform security architectures. In all of these architectures the fundamental concepts are borrowed from older commercial or research systems, but some of them have been adapted with new twists to suit the needs of mobile platforms. We also discussed a number of issues that are insufficiently addressed by the current generation of mobile platform security architectures. They constitute fertile ground for further research on the topic.

9. REFERENCES

[1] James Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Electronic Systems Division, 1972.

[2] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, pages 65–71. IEEE Computer Society, 1997.

[3] ARM. Trustzone-enabled processor. `http://www.arm.com/products/processors/technologies/trustzone.php`.

[4] ARM. *Building a Secure System using TrustZone^{TM} Technology*, 2009. Available from `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`.

[5] David Barrera, Hilmi Günes Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 73–84. ACM, 2010.

[6] Desktop bus project page. website. `http://www.freedesktop.org/wiki/Software/dbus`, 2010.

[7] Pern Hui Chia, Andreas Heiner, and N. Asokan. Use of ratings from personalized community for trustworthy application installation. In *Proceedings of the the 15th Nordic Conference in Secure IT Systems*, 2010.

[8] Pern Hui Chia, Andreas Heiner, and N. Asokan. The wisdom of cliques: Use of personalized social rating for trustworthy application installation. Technical Report NRC-TR-2010-001, Nokia Research Center, July 2010. Available at `http://research.nokia.com/files/tr/NRCTR2010001.pdf`.

[9] Jan-Erik Ekberg and Markku Kylänpää. Mobile trusted module. Technical Report NRC-TR-2007-015, Nokia Research Center, November 2007. Available at: `http://research.nokia.com/files/NRCTR2007015.pdf`.

[10] ETSI. *ETSI GSM 02.09 Security Aspects*. European Telecommunication Standards Institute, April 1993. Version 3.1.0; Available from `http://www.3gpp.org/ftp/Specs/html-info/0209.htm`.

[11] ETSI. *ETSI GSM 02.09 Security Aspects*. European Telecommunication Standards Institute, June 2001. Version 8.0.1 Release 99; Available from `http://www.3gpp.org/ftp/Specs/html-info/0209.htm`.

[12] Gartner. Press release; worldwide mobile phone sales in trhid quarter 2010. `http://www.gartner.com/it/page.jsp?id=1466313`, 2010.

[13] Gitorious. Mssf project source code. `http://meego.gitorious.org/meego-platform-security`, 2010.

[14] Dmitry Kasatkin. Mobile simplified security framework. In *Proceedings of the 12th Linux Symposium*, 2010.

[15] Butler Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and System*, pages 18–24, 1971.

[16] Steve Litchfield. Defining the smartphone. On-line article at AllAboutSymbian.com, July 2010. Available at `http://www.allaboutsymbian.com/features/item/Defining_the_Smartphone.php`.

[17] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In Clem Cole, editor, *USENIX Annual Technical Conference, FREENIX Track*, pages 29–42. USENIX, 2001.

[18] Claudio Marforio, Srdjan Capkun and Aurélien Francillon. Personal communication, November 2010. Paper in submission.

[19] Jonathan M. McCune, Bryan Parno, Adrian Perrig,

Michael K. Reiter, and Arvind Seshadri. Minimal TCB Code Execution (Extended Abstract). In *Proc. IEEE Symposium on Security and Privacy*, May 2007.

[20] MeeGo. Mobile simplified security framework overview. `http://conference2010.meego.com/session/mobile-simplified-security-framework-overview`, 2010.

[21] Sun Microsystems. Mobile information device profile for java 2 micro edition, version 2.1. `http://www.oracle.com/technetwork/java/index-jsp-138820.html`, 2006.

[22] Motorola. Mobile information device profile for java micro edition, version 3.0. `http://opensource.motorola.com/sf/projects/jsr271`, 2009.

[23] Oracle. Java technology. `http://www.java.com/en/about/`, 2010.

[24] Hewlett Packard. Openvms guide to system security. Available from `http://www.hp.com/go/openvms/doc/`, June 2010.

[25] Siani Pearson, editor. *Trusted Computing Platforms: TCPA technology in context*. Prentice Hall, 2003.

[26] Elena Reshetova. Mobile simplified security framework overview. `http://userweb.kernel.org/~jmorris/lss2010_slides/reshetova_LinuxCon_overview_v_final.pdf`, 2010.

[27] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[28] Jane Sales. *Symbian OS Internals*. Wiley, 2005.

[29] Casey Schaufler. Smack in embedded computing. In *Proceedings of the 10th Linux Symposium*, 2008.

[30] Dries Schellekens, Pim Tuyls, and Bart Preneel. Embedded trusted computing with authenticated non-volatile memory. In *Proc. of the 1st International conference on Trusted Computing and Trust in Information Technologies (TRUST 2008)*, 2008.

[31] SourceForge. An overview of the linux integrity subsystem. `http://heanet.dl.sourceforge.net/project/linux-ima/linux-ima/Integrity_overview.pdf`, 2010.

[32] Jay Srage and Jerome Azema. M-Shield mobile security technology, 2005. TI White paper. `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`.

[33] Harini Sundaresan. OMAP platform security features, July 2003. TI White paper. `http://focus.ti.com/pdfs/vf/wireless/platformsecuritywp.pdf`.

[34] Trusted Computing Group. `https://www.trustedcomputinggroup.org/home`.

[35] TCG. Trusted Platform Module (TPM) Specifications. Available at: `https://www.trustedcomputinggroup.org/specs/TPM/`.

[36] Android-DLS wiki. Howto: Unpack, edit, and re-pack boot images. `http://android-dls.com/wiki/index.php?title=HOWTO:_Unpack%2C_Edit%2C_and_Re-Pack_Boot_Images`, 2010.

[37] Maurice Wilkes. *The Cambridge CAP computer and its operating system*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1979.