# Market-driven Code Provisioning to Mobile Secure Hardware

Alexandra Dmitrienko[2], Stephan Heuser[1], Thien Duc Nguyen[1], Marcos da Silva Ramos[2], Andre Rein[2], and Ahmad-Reza Sadeghi[1]

[1] CASED/Technische Universität Darmstadt, Germany
[2] CASED/Fraunhofer SIT Darmstadt, Germany
*email*: {alexandra.dmitrienko, andre.rein}@sit.fraunhofer.de, {stephan.heuser, ducthien.nguyen, marcos.dasilvaramos, ahmad.sadeghi}@trust.cased.de

**Abstract.** Today, most smartphones feature different kinds of secure hardware, such as processor-based security extensions (e.g., TrustZone) and dedicated secure co-processors (e.g., SIM-cards or embedded secure elements). Unfortunately, secure hardware is almost never utilized by commercial third party apps, although their usage would drastically improve security of security critical apps. The reasons are diverse: Secure hardware stakeholders such as phone manufacturers and mobile network operators (MNOs) have full control over the corresponding interfaces and expect high financial revenue; and the current code provisioning schemes are inflexible and impractical since they require developers to collaborate with large stakeholders.

In this paper we propose a new code provisioning paradigm for the code intended to run within execution environments established on top of secure hardware. It leverages market-based code distribution model and overcomes disadvantages of existing code provisioning schemes. In particular, it enables access of third party developers to secure hardware; allows secure hardware stakeholders to obtain revenue for usage of hardware they control; and does not require third party developers to collaborate with large stakeholders, such as OS and secure hardware vendors. Our scheme is compatible with Global Platform (GP) specifications and can be easily incorporated into existing standards.

## 1 Introduction

Today, mobile devices have become an integral part of our life. The increasing computing, storage and networking capabilities and the vast number and variety of mobile apps make smart devices convenient replacements for traditional computing platforms such as laptops. As a consequence, mobile devices increasingly collect, store and process various privacy sensitive and security critical information about users such as e-mails and SMS messages, phone call history, location data, photos, authentication credentials (e.g., for online banking), etc. This makes them very attractive attack targets, as the rapid growth of mobile malware shows [40]. Hence,

mobile security has become an important topic for industrial and academic research in recent years.

One of the major approaches to harden mobile platform security is to leverage isolated (secure) environments, where apps can execute security sensitive operations (e.g., encryption, signing, etc.) in sub-routines referred to as trusted applications, applets or trustlets. While generally isolation between the secure environment and the rest of the mobile system can be enforced in software or in hardware, hardware-supported isolation provides stronger security guaranties: It can resist software-based attacks (e.g., compromised OS-level components) and even be resilient, to a certain degree, against physical tampering.

Hardware-based isolated environments, on which we focus in this paper, can be established on top of general purpose secure hardware, such as processor-based security extensions (e.g., TrustZone [11] and M-Shield [14]) and dedicated secure coprocessors, e.g., an embedded secure element available on NFC-enabled devices. However, while such secure hardware has been available for a decade and even widely deployed on mobile platforms [20, 35, 41], it is owned and exclusively used by their respective stakeholders such as phone manufacturers and mobile network operators (MNOs). For instance, processor-based security extensions are normally used by phone manufacturers to securely store radio frequency parameters calibrated during manufacturing process [20], or to ensure secure boot[3], while secure elements owned by MNOs (e.g., SIM-cards) are used to protect authentication credentials of users in mobile networks. Unfortunately, they are almost never utilized by commercial apps developed by third parties. Exceptions are solutions driven by large service/OS providers like NFC payment apps Google Wallet [3] and upcoming Apple Pay [13]. Hence, many security critical apps cannot be satisfactorily implemented as long as the secure hardware interfaces remain inaccessible.

The major obstacle for utilizing hardware-based isolated environments by third party apps is the fact that underlying secure hardware is currently under full control of their stakeholders. Trusted applications, applets or tustlets must first be admitted (e.g., signed) by the respective stakeholder in order to be executed within the isolated environment. Existing code provisioning schemes either rely on a stakeholder, or require a developer to become a Service Provider (SP) and maintain code provisioning services on their own. In either case, a collaboration between developers and stakeholders is required. However, such collaboration is often infeasible in practice, as stakeholders are typically large companies whereas app developers are small or middle-size businesses. Further, regular app developers typically would not become a service provider and maintain online code provisioning services.

To overcome these obstacles, we propose to apply the app market code distribution model (as currently used by mobile platform vendors) for the distribution of code (applets, trustlets and trusted apps) for secure hardware. Mobile app markets have been successfully bridging the gap between app developers and large OS vendors, and thus could also serve in the same way between developers and secure hardware stakeholders. However, there are several challenges to be tackled before

[3] Secure boot means a system terminates the boot process in case the integrity check of a component to be loaded fails [32]

an app market based code submission system can be applied for the distribution of secure hardware code. We need mechanisms that (i) allow the regular app to be coupled with corresponding applets, trustlets or trusted apps, given that the OS vendor and the stakeholder of secure hardware are typically different entities; (ii) provide financial incentives to secure hardware stakeholders in order to motivate them to allow third parties to leverage secure hardware; (iii) make access to secure hardware much more flexible, e.g., configurable by app developers independently from OS vendors; and, (iv) finally, address limitations of resource-constraint secure environments (e.g., Java cards), given that ability to leverage secure hardware by third party developers may result in large variety of applets exceeding resource constrains of respective secure elements.

**Goals and contributions.** In this paper, we aim to tackle the challenges mentioned above and enable third party developers to leverage secure hardware widely available on commodity devices. In particular, we make the following contributions:

*Market-driven code provisioning to secure hardware.* We propose a new paradigm for code provisioning to secure hardware (cf. Section 3). The main idea is to use an app market model for distribution of secure hardware code. Our solution (i) allows developers to distribute security sensitive code (e.g., trusted apps or applets) as a part of the mobile app package. Hence, developers do not need to act as service providers (SPs) and maintain online code provisioning servers; (ii) it supports flexible and dynamic assignment of access rights to secure hardware APIs and applets by mobile app developers independently from an OS vendor and a secure hardware stakeholder; (iii) allows the secure hardware stakeholder to obtain revenue for every provisioned piece of code; (iv) allows for automated and transparent installation and deinstallation of applets on demand in order to permit arbitrary number of applets, e.g., in a constraint Java card environment. Our scheme is compatible with Global Platform (GP) specifications and can be easily incorporated into existing standards [23–25, 29].

*Prototype implementation and evaluation.* We prototyped our solution on Android and a Java-based secure element (SE) (cf. Section 4). For SE prototyping, we ported the open source JCardSim Java Card emulator [4] to Android and enhanced it with our extensions. We will make the code for JCardSim Java Card emulator on Android open source[4]. Our prototype provides a wide range of SE options: (i) Java Card emulator placed on the mobile platform, (ii) Java Card emulator resides on a separate hardware token, for instance a smartwatch, and (iii) Java Card emulator provided by a cloud-based service. We further evaluated our prototype with the NFC-based access control application [15,18] which turns the smartphone into a key ring for electronic door (or car) keys. The security sensitive sub-routines of the app were implemented as a Java applet which was then executed within the JCardSim-based emulated environment deployed either on a smartphone, or on a smartwatch that acts as a trusted token. We then evaluated performance to confirm efficiency of our implementation.

[4] Please visit our project page `jcandroid.org`.

## 2 Background and Problem Description

In this section, we review possible secure hardware alternatives available to developers and discuss existing code provisioning and access control mechanisms for these environments. We discuss their trade-offs and highlight disadvantages that impede use of these environments in practice.

### 2.1 Hardware-based Secure Hardware Alternatives

Generally, Global Platform specifies different implementations of isolated execution environments [27]: Processor-based trusted execution environments (TEEs), and embedded or removable secure elements (SEs).

**Processor-based TEEs** are realized via a secure processor mode. Almost every smartphone and tablet today contains a processor-based TEE, such as TrustZone [11] and M-Shield [14]. However, their use requires third party developers to collaborate with mobile device vendors (such as Samsung and Apple) in order to get the security sensitive code admitted to run in a secure processor mode. Further, collaboration with the operating system vendor would also be required in order to enable communication between a mobile app and TEE-residing code.

**Embedded SEs** are distinct security sub-systems, which are available on many commercial mobile devices. They can be realized either as a standalone chip attached to the motherboard, or integrated into an NFC chip. Secure elements usually use standardized and widely supported JavaCard environment that can run Java applets. However, their interfaces are not usually exposed to third party developers. There are only a few products on the market powered by large companies, such as Google Wallet [3], Visa payWave [37] and MasterCard's PayPass [17] solutions for NFC payments, that utilize an embedded SE.

**Removable SEs** are security co-processors which can be attached to the device via peripheral interfaces, such as Universal Integrated Circuit Cards (UICC) (also known as SIM cards) and plug-in cards for an SD card slot (ASSD cards). UICC cards are controlled by MNOs – yet too large entities for small-size developers. Moreover, collaboration with a single MNO can only reach limited number of users. Hence, more complex business models arose that involve Trusted Service Managers (TSM) – intermediate entities that have agreements with multiple MNOs [22]. In contrast, ASSD cards (e.g, [16,39]) are not controlled by external stakeholders.On the downside, however, they are quite costly[5] and their use is limited to smartphone platforms featuring an SD card slot.

To summarize, all existing options have disadvantages which in fact impeded use of secure hardware by application developers in practice. This resulted in a notable shift in favor of software-based solutions compared to prior years [19], despite of general opinion that hardware-based solutions provide stronger security.

[5] For instance, the retail price for the cgCard [16] is 99 EUR per piece

### 2.2 Secure Hardware APIs and Access Control Mechanisms

Currently, Android does not allow mobile apps to directly access processor-based TEEs[6]. Only embedded and removable SEs are accessible via respective APIs.

Embedded SE on Android can be accessed via an NFC API. Initially, access to this API was limited to system-level components signed with platform keys [21]. This has been changed in 4.0.4 version, which introduced a more flexible approach based on an access control list (ACL) stored in a system file. Although potentially the ACL could be updated by system apps or by the OS vendor through the over-the-air (OTA) system update, these mechanisms do not seem to be used in practice – once deployed, ACLs typically remain unmodified.

Access to removable SEs on Android is provided via a SmartCard API implemented within seek-for-Android project[7]. Access control to the SmartCard API is compliant to the Global Platform (GP) specification [29]. In particular, it uses an SE internal access control list (ACL) of which a read-only copy is fetched on system boot and enforced by an access control enforcer (ACE) – an OS side system component. ACLs on the SE can only be updated by an SE stakeholder or by a trusted (by the SE stakeholder) party.

To summarize, all existing approaches are inflexible in performing ACL updates. In either case, involvement of a trusted party is required – for instance, OS vendors are responsible for ACL updates for embedded SEs, while MNOs manage ACLs on UICC-based secure elements.

### 2.3 Code Provisioning

Currently, code provisioning specifications for processor-based TEEs are under development by the Global Platform Device Specification Working Group. Hence, we will discuss specifications of the code provisioning mechanisms for NFC-based and UICC-based secure elements which are already published [23, 26].

Generally, there are three options for code provisioning specified: (i) Simple mode, (ii) delegated mode, and (iii) authorized mode. In a simple mode, the service provider (SP) delegates full management of its applet to an SE stakeholder. In the delegated mode, each operation for code provisioning is performed by a Trusted Service Manager (TSM) and requires a pre-authorization from the SE stakeholder for each operation. In the authorized mode, however, the SE stakeholder authorizes either TSM or SP to perform code provisioning on their own. In any mode code provisioning is performed either by the SE stakeholder or by authority the SE stakeholder trusts, via over-the-air (OTA) secure channel. In this way, third party developers must either become service providers or delegate code provisioning tasks to the SE stakeholder, which raises the bar for entering market of SE-supported applications.

[6] Indirect access is available for certain crypto operations provided by Android's *KeyStore* `https://developer.android.com/about/versions/android-4.3.html`

[7] https://code.google.com/p/seek-for-android/

## 3 Market-driven Code Provisioning to Secure Hardware

In this section we present our market-driven code provisioning mechanism which enables access of third party developers to secure hardware. Generally, our solution can be applied for code provisioning to secure hardware of different types. However, in the following we will concentrate on secure elements (SEs) and mechanisms for Java applet provisioning for brevity.

Our solution enables application developers to distribute applets via the app market place, e.g., packaged together with the mobile app or pulbished on a dedicated market place for applets. It relies on a developer to couple apps with corresponding applets – an approach which does not require interaction between OS vendors and SE stakeholders. Further, the scheme allows developers to define access control rules for accessing applets that are deployed during applet provisioning and independently from OS vendor. Moreover, our solution makes use of applet installation tokens issued by SE stakeholders to end users, which effectively allows SE stakeholders to enforce per-installation license fees (e.g., if obtaining the installation token requires payment). Finally, our mechanism makes use of an SE internal access control list (ACL) as defined by Global Platform (GP) Access Control Specification [29] for access control to SE APIs and is, hence, compatible with the established Global Platform mechanisms.

### 3.1 System Model and Assumptions

**System Model.** As shown in Figure 1, our system model involves the following entities: (i) app market $\mathsf{M}$, (ii) SE stakeholder $\mathsf{S}$, (iii) mobile host $\mathsf{H}$, (iv) secure element $\mathsf{E}$, and (v) developer $\mathsf{D}$. Here, $\mathsf{D}$ develops the mobile app $A$ and a corresponding applet $a$ which includes security-sensitive sub-routines. Further, $\mathsf{H}$ is a mobile device of the user (e.g., a smartphone or a tablet) for which the mobile app was developed, while $\mathsf{E}$ is a secure element of $\mathsf{H}$, which is trusted to securely execute the applet $a$. Moreover, $\mathsf{M}$ is a regular market place for mobile apps managed by the (OS or device) vendor, while $\mathsf{S}$ is an online service managed by the SE stakeholder.

**Assumptions.** We assume that the SE Stakeholder $\mathsf{S}$ shares a symmetric key $K_E$ with every secure element $\mathsf{E}$ it controls. This assumption is reasonable, as similar keys are already used by SE stakeholders to perform code management on deployed SEs[8]. Further, the mobile host $\mathsf{H}$ is aware of the SE identifier $id_{\mathsf{E}}$ which uniquely identifies its secure element. We also assume that all interactions between $\mathsf{D}$ and $\mathsf{S}$, $\mathsf{D}$ and $\mathsf{M}$, $\mathsf{M}$ and $\mathsf{H}$ are performed over secure (authentic and confidential) channels. For instance, the Global Platform specifications describe various standards for secure channel protocols (e.g., AES-based SCP03 [25], SCP10 [23] based on asymmetric crypto-system, and SCP81 [24] based on SSL/TLS), which can be used for secure channel establishment and communication.

[8] For instance, GP specifies [23] that Java Cards share with the card issuer (i.e., a stakeholder) the symmetric Data Encryption Key (DEK)

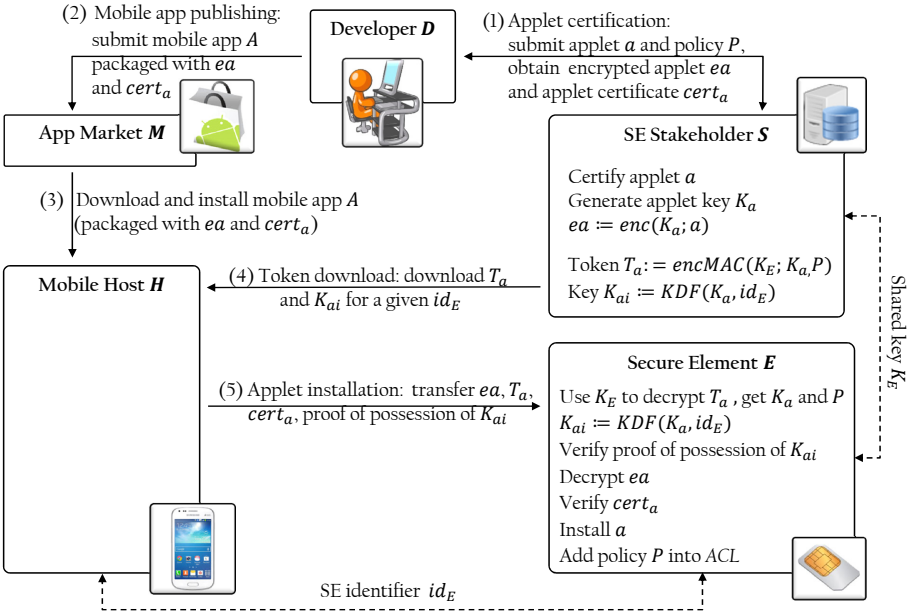


**Fig. 1.** Market-driven code provisioning

### 3.2 Code Provisioning Scheme

The general architecture of our market-driven code provisioning scheme is depicted in Figure 1. It shows the involved parties and their interactions in the following use cases: (1) applet certification, (2) mobile app publishing, (3) mobile app download and installation, (4) token download, and (5) applet installation. In the following we describe use cases in more details.

**Applet Certification.** As a first step, the developer D submits an applet $a$ via a code submission system to SE stakeholder S for certification. The applet is accompanied with the access control policy $P$ defining which mobile apps will be allowed to communicate with this applet (e.g., the app $A$). Upon receipt, S performs applet verification according to its security policy. In particular, it can perform code vetting process (as typically done by OS vendors for mobile apps). If this check passes, it creates an applet certificate $cert_a$, generates an applet-specific key $K_a$ and encrypts the applet $a$ under $K_a$. The encrypted applet $ea$ is then returned to D together with its certificate $cert_a$.

Note, that for better efficiency one could replace applet certificate $cert_a$ with the message authentication code (MAC) of the applet generated under the key $K_a$. We opted for certificates in our system design due to legacy reasons, as in current systems applets are certified by means of certificates.

**Mobile App Publishing.** To publish a mobile app $A$ at the app market M, the developer D includes the (encrypted) applet $ea$ and its certificate $cert_a$ into an in-

stallation package of $A$ and submits it to $\mathsf{M}$. This step is common for a regular app development process. Whenever the mobile app $A$ is verified by $\mathsf{M}$, it will appear at the app market and will be ready for download by mobile users.

Alternatively to packaging the applet $a$ and its certificate $cert_a$ together with a mobile app $A$, the developer $\mathsf{D}$ may opt to publish an applet on a dedicated applet market (e.g., maintained by $\mathsf{S}$) and include a download link referencing the required applet, e.g., into app $A$'s manifest (not shown in Figure 1 for brevity).

**Mobile App Download and Installation.** Our solution relies on standard mechanisms for mobile app downloading and extends the app installation process with routines to detect applet-related dependencies and, if detected, to trigger a token download procedure.

**Token Download.** Whenever the mobile host $\mathsf{H}$ detects that the mobile app $A$ requires an applet $a$, it connects to the SE stakeholder $\mathsf{S}$ and requests an applet installation token for the secure element with the identity $id_{\mathsf{E}}$. Hence, $\mathsf{S}$ generates the token $T_a$ for a given $id_{\mathsf{E}}$ and $a$, where $T_a$ is an authenticated encryption (which we denote as $encMAC$) under the key $K_E$ over the key $K_a$ and the policy $P$. Further, $\mathsf{S}$ derives an applet installation key $K_{ai}$ by applying a one-way key derivation function ($KDF$) to the key $K_a$ and the identity $id_{\mathsf{E}}$. The resulting token $T_a$ and the key $K_{ai}$ are returned to $\mathsf{H}$.

As one can notice, our token download procedure requires interaction between a mobile host $\mathsf{H}$ and an SE stakeholder $\mathsf{S}$. While potentially such a communication could be avoided using cryptographic techniques such as key derivation and oblivious hash functions, we aim to keep the SE stakeholder on the installation path in order to enable it to enforce license fees. In particular, if a license fee is required, the token download procedure can be preceded by a payment procedure, which can be realized in the same way as a mobile app purchase.

**Applet Installation.** To install the applet, the mobile host $\mathsf{H}$ sends the (encrypted) applet $ea$, the token $T_a$ and the proof of possession of $K_{ai}$ to the secure element $\mathsf{E}$. Then, $\mathsf{E}$ extracts $K_a$ and $P$ from the token $T_a$ and derives the key $K_{ai}$ by applying a one-way key derivation function to values $K_a$ and $id_{\mathsf{E}}$. Next, it verifies if $K_{ai}$ is known to the host (e.g., by means of challenge-response authentication). Further, it decrypts $ea$ with the key $K_a$ in order to obtain the applet $a$, verifies $cert_a$, installs $a$ and adds the policy $P$ for the applet $a$ into $ACL$.

### 3.3 Applet Invocation

As soon as the applet $a$ is installed, it can be invoked by user space apps residing on the mobile host $\mathsf{H}$. We realized the communication between the apps and applets as defined by GP Access Control Specification [29]. In particular, a communication channel is mediated by the OS-level component access control enforcer (ACE), which fetches the access control list (ACL) from the SE-internal access rule application master (ARA-M) component. The ACL consists of data objects (DO) which

contain access rules for SE access and application protocol data unit (APDU) filtering. Rules are identified by the identifier AID-REF-DO of the applet to be accessed and the hash of the application's certificate Hash-REF-DO. Further, it may include an APDU filter consisting of an APDU header and an APDU filter mask.

When the app $A$ requests access to an applet $a$ identified by AID-REF-DO, ACE identifies Hash-REF-DO of the app and reads the ACL rule for the specific {AID-REF-DO, Hash-REF-DO} pair. Access is granted, if such access is permitted by the ACL rule, or denied, if access is prohibited by an ACL rule or no rule is found. Further, the application can communicate with the SE applet if the command APDUs match the filter list (if given) checked by ACE.

### 3.4 On-demand Applet Installation

Although a secure element (SE) may host multiple applets at once, generally the space on SE is limited. As soon as a limit is reached, it may not be longer possible to install further applets. Currently, this is not yet a concern for SE environments due to the lack of available applets. Further, SE stakeholders may specify resource quota for every applet and ensure that SE resource limits are not exceeded. However, resource quota mechanisms might not be effective for our market-based code provisioning scheme, as it is not under full control of SE stakeholders. Further, our scheme is likely to stipulate development of new applets, so that a space limitation may become a concern.

To address this issue, we extended applet invocation mechanism with an SE applet manager (SEAM) component which allows for on-demand installation and deinstallation of applets in order to dynamically re-use available resources. In particular, SEAM maintains applet usage statistics in order to identify more frequently accessed applets. Whenever a currently deinstalled applet is invoked, SEAM performs dynamic applet installation and then allows the access control enforcer (ACE) to establish a communication channel between the applet and the app. Whenever the applet installation requires more resources than currently available, SEAM deinstalls a suitable (i.e., rarely used and sufficiently large) applet in order to release additional resources. Our on-demand applet installation extension is compatible to current GP specifications, i.e., it is transparent to ACE component and to mobile apps. Further, our prototype implementation and performance measurements indicate that our extension imposes low performance overhead which is well acceptable for runtime environments (cf. Section 4.2).

### 3.5 Platform Architecture

In Figure 2 we depict the mobile platform architecture. It includes components we introduced to support our extensions, as well as standard components defined by Global Platform Reference Specification [29] (which we show in the figure in the dark gray color). The architecture separates the execution environment of the mobile platform into two independent worlds: Mobile host (H) and secure element (E). Apps are deployed on H via the modified app installer AI, which interacts with the untrusted service manager USM for applet deployment and token management. At

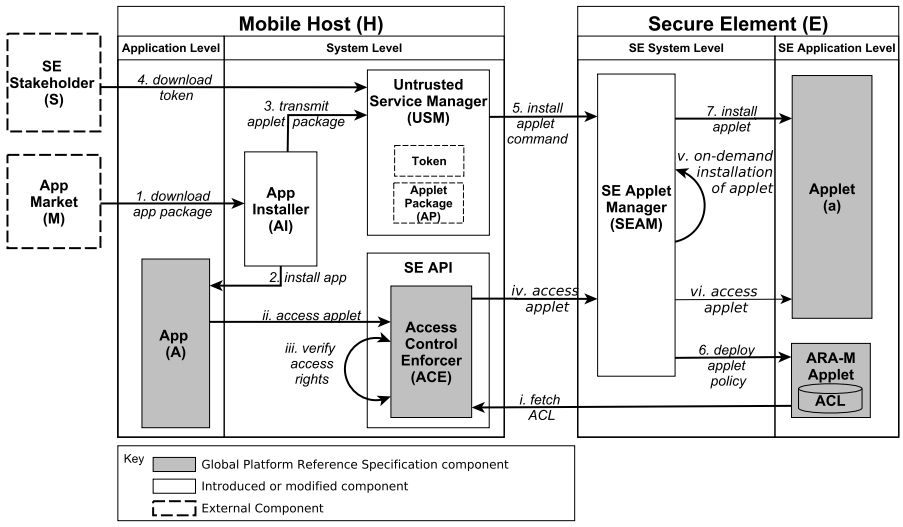


**Fig. 2.** Instantiated Platform Architecture

runtime, apps interact with their applets via the secure element API (SE API) that embeds the GP-defined access control enforcer (ACE).

In the following, we describe component interactions for two major use cases: (i) download and installation of applet-dependent apps, and (ii) execution of applet-dependent apps.

**Download and installation of applet-dependent apps.** When the applet installer AI receives a request to install an applet-dependent app $A$ (step 1), $A$ is first extracted from the app package and installed on the mobile host H (step 2). Next, AI extracts the applet package AP (consisting of the encrypted applet $ea$ and its certificate $cert_a$) and sends it to the untrusted service manager USM (step 3). USM stores AP in its internal storage and requests a token $T_a$ via a secure communication channel from S using the remote application management over HTTP protocol (SCP81) [24] (step 4). Next, USM triggers applet installation process by sending the the applet package AP and token $T_a$ to the SE applet manager SEAM (step 5). In turn, SEAM proceeds to verify the integrity and authenticity of $T_a$ using the secret key $K_E$ shared by the secure element E and the SE stakeholder S and decrypts $T_a$ using $K_E$. Further, the encrypted applet $ea$ is decrypted using $K_a$ (which is embedded in $T_a$), and $cert_a$ is verified. If the verification process is successful, the applet $a$ is ready for installation. Finally, SEAM deploys the new ACL rules to the ARA-M Applet (step 6) and installs the applet $a$ on the secure element E (step 7).

**Execution of applet-dependent apps.** Either on system boot, or just before the access rules are verified, the SE access control enforcer (ACE) fetches and caches all current ACL-rules from the ARA-M Applet (step i). When an app $A$ requests access to an applet $a$ (step ii), AID-REF-DO and Hash-REF-DO are retrieved

by ACE and access rights for applet access are verified (step iii). If the verification was successful, ACE grants access and forwards applet access request to the SE applet manager (SEAM) (step iv), which verifies the installation status of the applet. If the applet is not yet installed, SEAM triggers the applet installation process (step v). Finally, a communication request is forwarded to the applet $a$ (step vi) and the communication channel is successfully established between the app $A$ and the applet $a$.

## 4 Implementation and Evaluation

In this section we briefly describe our prototype implementation and provide evaluation results.

### 4.1 Implementation

Our implementation is based on Java and currently targets Android devices. To prototype the secure element environment, we used the open-source JavaCard simulator jCardSim [4], which we ported on Android. As summarized in Table 1, our implementation consists of 7 software modules and includes 9558 Lines of Code (LoC) in total, of which 5651 LoC consist of ports of third party open source projects.

The JavaCard emulator is realized in the jCardSim4Android and SmartCardIO modules, consisting of 4923 and 728 LoC, respectively. The main functionality is included into a jCardSim4Android Android library – a modified version of *jCardSim* which we adapted to run on Android. The emulator provides an environment to run third-party applets, as well as the previously described ARA-M applet which stores access control rules. *jCardSim* has a dependency on the Java Remote Method Invocation (RMI) API and javax.smartcardio classes, which are not available on Android. Hence, we removed RMI functionality (which is not used in our project) and extracted the required javax.smartcardio classes from the source code of OpenJDK v7 into a SmartCardIO library.

Our secure element environment is implemented within an Android SE app which holds an instance of the JavaCard emulator and implements the functionality of the SE Applet Manager (SEAM) component. It consists of 791 LoC and depends on the jCardSim4Android module and the SpongyCastle [6] API for crypto support. The functionality of the USM component is implemented within the Android Host app consisting of 1124 LoC. It depends on the SpongyCastle Crypto API and Communication API modules.

The Communication API module is responsible for the communication between different entities. In particular, it provides a unified communication framework which can be instantiated for different types of interfaces (Bluetooth, SSL/TLS, or local socket connections). It implements requests and responses and includes helper classes for data serialization, deserialization and transfer between parties. It is implemented in Java 6 and consists of 545 LoC. Further, it was ported to Java/Android to be compatible with the Android Host app.

**Table 1.** Software Modules

| Module | Size (LoC) | Language | Codebase | Dependencies |
|---|---|---|---|---|
| Android SE | 791 | Java/Android | - | SpongyCastle Crypto API, jCardSim4Android |
| jCardSim4Android | 4923 | Java/Android | jCardSim [4] | SmartCardIO |
| SmartCardIO | 728 | Java/Android | OpenJDK | - |
| Android Host | 1124 | Java/Android | - | SpongyCastle Crypto API, Communication API |
| Communication API | 545 | Java 6, Java/Android | - | - |
| Developer | 656 | Java 6 | - | BouncyCastle Crypto API, Communication API |
| SE Stakeholder | 1390 | Java 6 | - | BouncyCastle Crypto API, Communication API |

The SE stakeholder S and developer D modules are implemented in Java 6 and consist of 1390 LoC and 656 LoC, respectively. Both modules have dependencies on the BouncyCastle [1] Crypto API and the Communication API.

**External SE Deployment.** As our solution strictly separates the mobile host H and the secure element E, we are able to deploy E not only on the same device as H, but also on external devices like smartwatches or even in a cloud environment. To implement such a scenario we chose to use a smartwatch emulating the secure element E and a smartphone acting as the mobile host H. The communication between the smartphone and the smartwatch is based on Bluetooth with security mode 3 (Link-Level Enforced Security) with enabled data encryption. This allows us to make use of our socket based client-server communication between H and E by simply establishing RFCOMM channels [31] and to perform the previously described applet installation and execution without any further modification to the existing code.

### 4.2 Evaluation

To evaluate the performance of our prototype we deployed it on an Samsung Galaxy S3 smartphone running Android 4.4 and a Samsung Galaxy Gear SM-V700 smartwatch running Android 4.2. The components that do not rely on a mobile platform (app market and SE stakeholder) were executed on a server machine (Intel i7-2600 CPU, 8 GB RAM) running Ubuntu 12.04.4 and OpenJDK 6b31. The smartphone and the server were connected via a a 802.11abgn Wi-Fi network. In our evaluation we used the emulated and hardware-based secure element. Hardware-based secure element was represented by a Mobile Security Card [39] which is a representative of an ASSD card. Further, we utilized a Java applet developed for a SmartToken access control solution [15,18] (10953 Bytes). All experiments were performed 1000 times, and we present the average values and standard deviation for selected operations.

We first measured time required for the execution of the applet certification and token download protocols. Applet certification requires 173.975 ms (± 40.517 ms) on average, while token download needs 144.235 ms (± 26.729 ms).

The most performance-critical operations are applet installation, deinstallation and applet execution, as they are performed at runtime during execution of the mobile app. We measured their performance in two different use-cases: (i) mobile host H and secure element E deployed on the same smartphone and (ii) mobile host H deployed on smartphone while the secure element E is deployed on a smartwatch. Further, for the sake of comparison we measured the applet execution for the hardware-based SE. However, we could not measure applet installation and deinstallation operations for the hardware-based SE, as standard JavaCard environments do not support on-demand installation and deinstallation of applets.

**Table 2.** Applet installation, deinstallation and execution. Average values and standard deviation

| | Applet installation, ms | Applet deinstallation, ms | Applet execution, ms |
|---|---|---|---|
| Mobile host H and secure element E on the same smartphone | 46.265 ± 19.188 | 15.763 ± 7.851 | 38.430 ± 18.464 |
| Mobile host H on the smartphone, secure element E on the smartwatch | 415.266 ± 77.998 | 205.356 ± 72.539 | 150.335 ± 50.069 |
| Hardware-based secure element E | - | - | 24.471 ± 1.863 |

Table 2 shows the average time (and standard deviation) required for the applet installation and deinstallation process for use-cases (i) and (ii). Furthermore, it shows how long it took to execute a simple operation (receiving 4 Bytes and sending 10 Bytes) in the applet from an app residing on H. The process starts within the app, requests the execution and ends after the result of the operation is successfully received by the app.

Overall, we deem these results reasonable and promising for a real-life deployment of our architecture, especially when considering that our implementation has not been optimized for performance yet.

## 5 Related Work

The most relevant work to ours is the On-board credentials (ObC) framework developed by Nokia researchers [34]. In particular, incentives behind ObC are similar to ours – to open secure hardware to third party developers. ObC enables developers to implement security sensitive subroutines of their applications in the form of ObC scripts, which can be loaded into and executed within an isolated execution environment. Put forward by Nokia, the framework is deployed on commercial Nokia de-

vices (e.g., Nokia Lumia), on top of ARM TrustZone and TI M-Shield TEEs. However, the ObC framework does not address access control aspects to ObC APIs from mobile apps – in fact, such access is still controlled by the OS vendor. Hence, third party developers need to collaborate with the OS vendor in order to execute their ObC scripts within the isolated execution environment. Further, the framework is an intellectual property of Nokia and is limited to Nokia platforms. Moreover, ObC is primarily tailored for processor-based TEEs, while we focus on secure co-processors – Java-cards, and address Java-card specific challenges (e.g., on-demand applet installation).

Akram et al. [7–10] aim to solve similar problem by different means – they propose a new paradigm to hand over the control and management of smartcard applications to the end-user. Similarly, Global Platform specifies a consumer-centric provisioning model where the user has more control over their isolated execution environments [28]. In contrast to these works, we aim to remain compliant with the traditional SE ownership model and expose secure hardware to third party developers by means of more flexible SE code provisioning mechanisms and providing financial incentives to SE stakeholders.

Vasudevan et al. [41] proposed a challenge to the research community to present sound technical evidence that application developers and users can benefit from hardware security features. Our work aims to address challenges related to utilizing secure hardware by application developers.

Ekberg et al. [20] discussed reasons for limited use of secure hardware on mobile devices, such as security requirements and concerns of different stakeholders and absence of standardized APIs for accessing secure hardware. We believe, that our work can help to satisfy security requirements of different stakeholders.

Masti et al. [38] proposed an architecture that can provide an isolated execution environment as a cloud service. The authors focus on light-weight processor extensions (like Intel TXT) and virtualized trusted platform modules (TPMs) in order to provide concurrent dynamic root of trusts to multiple cloud-based virtual machines. Generally, this work aims to solve an orthogonal problem. However, our cloud-based architecture instantiation can largely benefit from proposed hardware-based security anchors in the cloud.

González, et al. [30] proposed an open big data platform for sensors that leverages the Open Virtualization framework – an open source implementation of the Global Platform's TEE specifications [2] for ARM TrustZone [5]. Their efforts are directed towards building an open source community around Open Virtualization, while our primary goal is to enable access to secure hardware for third parties.

Marforio et al. [36] concentrated on secure and practical bootstrapping techniques for security services on mobile devices. They particularly discussed the importance of binding user identities to underlying mobile platforms and proposed an architecture to provide secure user enrollment and migration from one platform to another in the context of mobile TEEs.

The white paper [33] describes <tBase, a commercial trusted OS by Trustonic and highlights provisioning mechanisms for trusted apps. Similarly to our approach, provisioning mechanisms of <tBase leverage symmetric keys shared between the

TEE stakeholder and the TEE. However, similarly to provisioning solutions specified by Global Platform (cf. Section 2.3), they require a trusted third party (in a form of TSM) to manage code provisioning process, while our solution relies on untrusted service manager (USM) which can reside on untrusted mobile host.

Anwar et al. [12] proposed a new access control to secure element APIs by mobile apps on Android devices. Their concern is the fact that Access Control Enforcer (ACE) that mediates access to the secure element is an OS-level component which can be manipulated in case OS gets compromised. Authors propose to utilize trusted computing concepts in order to establish trust into OS-level components. In particular, they leverage processor-based TEE in order to ensure integrity of ACE component and lock access to SE if integrity is not preserved. On a down side, this solution requires significant modifications to system level software, as well as additional support in hardware, which is hard to achieve in practice. Hence, we opt for approach specified by Global Platform for sake of compatibility. Nevertheless, our code provisioning scheme can be combined with the access control solution proposed in this paper.

## 6 Conclusion

Currently, there is no flexible model for third party app developers to access and use the available secure hardware on smartphones. This is an unfortunate situation since secure hardware provides an isolated execution environment that would drastically improve the security of mobile apps. We propose a new model for flexible distribution and provisioning of secure hardware code (applets, trustlets, or trusted applications) for third party app developers. Our solution is compatible to specifications of Global Platform (GP) and allows developers to use existing app markets and couple their secure hardware code (e.g., applets in case of Java card) to mobile apps that require security critical operations to be executed in an isolated environment. The proposed ecosystem will allow the secure hardware stakeholders to generate revenue by enforcing per-installation fees for secure hardware code. We developed a prototype based on Java card and applied it to a smartphone (and a smartwatch) for an access control application that uses smartphone to open doors with NFC locks. We are planning to open source our port of JCardSim Java Card emulator to Android which will help industry and other researchers to build upon our work and deploy applet-dependent apps on smartphone platforms or even use a smartwatch as an isolated execution environment.

**Acknowledgements.** We thank N. Asokan for several fruitful discussions and feedback to the paper draft. Further, we thank anonymous reviewers for their helpful comments. This work was partially supported by the German ministry of education and research (Bundesministerium fr Bildung und Forschung, BMBF) within the Software Campus initiative.

## References


1. BouncyCastle crypto API. https://www.bouncycastle.org/.
2. GlobalPlatform - device specifications. http://www.globalplatform.org/specificationsdevice.asp.
3. Google Wallet: Shop. Save. Pay. With your phone. http://www.google.com/wallet/.
4. jCardSim Java card runtime environment simulator. http://jcardsim.org/.
5. Sierraware. http://www.sierraware.com.
6. SpongyCastle crypto API. http://rtyley.github.io/spongycastle/.
7. R. N. Akram and K. Markantonakis. Rethinking the smart card technology. In *the Second International Conference on Human Aspects of Information Security, Privacy, and Trust*, pages 221–232, 2014.
8. R. N. Akram, K. Markantonakis, and K. Mayes. A paradigm shift in smart card ownership model. In *International Conference on Computational Science and its Applications (ICCSA '10)*, pages 191–200, Washington, DC, USA, 2010. IEEE Computer Society.
9. R. N. Akram, K. Markantonakis, and K. Mayes. User centric security model for tamper-resistant devices. In *IEEE International Conference on e-Business Engineering (ICEBE'11)*, pages 168–177, 2011.
10. R. N. Akram, K. Markantonakis, and K. Mayes. Trusted Platform Module for smart cards. In *6th International Conference on New Technologies, Mobility and Security*, NTMS '14, pages 1–5. IEEE, 2014.
11. T. Alves and D. Felton. TrustZone: Integrated hardware and software security. *Information Quaterly*, 3(4), 2004.
12. W. Anwar, D. Lindskog, P. Zavarsky, and R. Ruhl. Redesigning secure element access control for NFC enabled Android smartphones using mobile trusted computing. In *International Conference on Information Society (i-Society)*, June 2013.
13. Apple Press. Apple Announces Apple Pay: Transforming Mobile Payments with an Easy, Secure & Private Way to Pay, Sep 2014. https://www.apple.com/pr/library/2014/09/09Apple-Announces-Apple-Pay.html.
14. J. Azema and G. Fayad. M-Shield mobile security technology: Making wireless secure. Texas Instruments white paper, 2008. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.
15. C. Busold, A. Dmitrienko, H. Seudi, A. Taha, M. Sobhani, C. Wachsmann, and A.-R. Sadeghi. Smart keys for cyber-cars: Secure smartphone-based NFC-enabled car immobilizer. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, Feb. 2013.
16. Certgate. Certgate products. cgCard, 2012. http://www.certgate.com/wp-content/uploads/2012/09/20131113_cgCard_Datasheet_EN.pdf.
17. S. Clark. MasterCard and Samsung introduce embedded NFC payments. http://www.nfcworld.com/2013/12/13/327343/mastercard-samsung-introduce-embedded-nfc-payments/, 2013.
18. A. Dmitrienko, A.-R. Sadeghi, S. Tamrakar, and C. Wachsmann. SmartTokens: Delegable access control with NFC-enabled smartphones. In *5th International Conference on Trust and Trustworthy Computing (TRUST)*. Springer, June 2012.
19. Edgar Dunn & Company. Advanced payments report. http://www.paymentscardsandmobile.com/wp-content/uploads/2014/02/PCM_EDC_Advanced_Payments_Report_2014_MWC.pdf, 2014.

20. J.-E. Ekberg, K. Kostiainen, and N. Asokan. The untapped potential of trusted execution environments on mobile devices. *IEEE Security & Privacy*, 99(PrePrints):1, 2014.
21. N. Elenkov. Accessing the embedded secure element in Android 4.x, 2012. http://nelenkov.blogspot.de/2012/08/accessing-embedded-secure-element-in.html.
22. European Payments Council - GSMA. Trusted Service Manager. Service management requirements and specifications. EPC 220-08. Version 1.0. http://www.europeanpaymentscouncil.eu/index.cfm/knowledge-bank/epc-documents/epc-gsma-tsm-service-management-requirements-and-specifications/epc220-08-epc-gsma-tsm-wp-v1pdf/, 2010.
23. Global Platform. Card specification. Version 2.2, 2006.
24. Global Platform. Remote application management over HTTP protocol, Sep 2006.
25. Global Platform. Global Platform card technology: Secure channel protocol 03, Sep 2009.
26. Global Platform. GlobalPlatforms proposition for NFC mobile: Secure element management and messaging. White paper. http://www.sicherungssysteme.net/fileadmin/GlobalPlatform_NFC_Mobile_White_Paper.pdf, 2009.
27. GlobalPlatform. GlobalPlatform Device Technology. TEE System Architecture. Version 1.0. http://globalplatform.org/specificationsdevice.asp, 2011.
28. GlobalPlatform. A new model: The consumer-centric model and how it applies to the mobile ecosystem. http://www.globalplatform.org/documents/Consumer_Centric_Model_White_PaperMar2012.pdf, 2012.
29. GlobalPlatform. Secure element access control. http://www.globalplatform.org/specificationsdevice.asp, 2012.
30. J. González and P. Bonnet. Towards an open framework leveraging a trusted execution environment. In *Cyberspace Safety and Security*, volume 8300 of *Lecture Notes in Computer Science*, pages 458–467. Springer International Publishing, 2013.
31. Google. Android API guide - Bluetooth. http://developer.android.com/guide/topics/connectivity/bluetooth.html, 2010.
32. N. Itoi, W. A. Arbaugh, S. J. Pollack, and D. M. Reeves. Personal secure booting. In *6th Australasian Conference on Information Security and Privacy (ACISP)*, pages 130–144, Jul 2001.
33. Jan-Erik Ekberg. Trustonic. <t-base - a trusted execution environment. White paper, 2014.
34. K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala. On-board credentials with open provisioning. In *ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pages 104–115. ACM, 2009.
35. K. Kostiainen, E. Reshetova, J.-E. Ekberg, and N. Asokan. Old, new, borrowed, blue – a perspective on the evolution of mobile platform security architectures. In *First ACM Conference on Data and Application Security and Privacy*, pages 13–24, 2011.
36. C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Čapkun. Secure enrollment and practical migration for mobile trusted execution environments. In *the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, pages 93–98, New York, NY, USA, 2013. ACM.
37. C. Marlowe. Intel and Visa join forces to boost mobile payments. http://www.dmwmedia.com/news/2012/02/28/intel-and-visa-join-forces-to-boost-mobile-payments, 2012.

38. R. J. Masti, C. Marforio, and S. Čapkun. An architecture for concurrent execution of secure environments in clouds. In *The ACM Cloud Computing Security Workshop (CCSW)*, pages 11–22, 2013.
39. Press Release, Giesecke & Devrient. G&D makes mobile terminal devices even more secure with new version of smart card in microSD format. `http://www.gi-de.com/en/about_g_d/press/press_releases/G%26D-Makes-Mobile-Terminal-Devices-Secure-with-New-MicroSD%E2%84%A2-Card-g3592.jsp`.
40. TrendLabs. 3Q 2012 security roundup. Android under siege: Popularity comes at a price. `http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-3q-2012-security-roundup-android-under-siege-popularity-comes-at-a-price.pdf`, 2012.
41. A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune. Trustworthy execution on mobile devices: What security properties can my mobile platform give me? In *5th International Conference on Trust and Trustworthy Computing (TRUST)*, pages 159–178, Berlin, Heidelberg, 2012. Springer-Verlag.