PEDECIBA INFORMÁTICA

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

TESIS DE MAESTRÍA
EN INFORMÁTICA

Offloading cryptographic services to the SIM card in smartphones

Daniel Pedraja
dpedraja@fing.edu.uy

February 2019

Thesis advisors: Gustavo Betarte[1] and Javier Baliosian[2]

[1] InCo, Facultad de Ingeniería, Universidad de la República
gustun@fing.edu.uy

[2] InCo, Facultad de Ingeniería, Universidad de la República
javierba@fing.edu.uy

Offloading cryptographic services to the SIM card in smartphones

Abstract

Smartphones have achieved ubiquitous presence in people's everyday life as communication, entertainment and work tools. Touch screens and a variety of sensors offer a rich experience and make applications increasingly diverse, complex and resource demanding. Despite their continuous evolution and enhancements, mobile devices are still limited in terms of battery life, processing power, storage capacity and network bandwidth. Computation offloading stands out among the efforts to extend device capabilities and face the growing gap between demand and availability of resources. As most popular technologies, mobile devices are attractive targets for malicious attackers. They usually store sensitive private data of their owners and are increasingly used for security sensitive activities such as online banking or mobile payments. While computation offloading introduces new challenges to the protection of those assets, it is very uncommon to take security and privacy into account as the main optimization objectives of this technique.

Mobile OS security relies heavily on cryptography. Available hardware and software cryptographic providers are usually designed to resist software attacks. This kind of protection is not enough when physical control over the device is lost. Secure elements, on the other hand, include a set of protections that make them physically tamper-resistant devices.

This work proposes a computation offloading technique that prioritizes enhancing security capabilities in mobile phones by offloading cryptographic operations to the SIM card, the only universally present secure element in those devices. Our contributions include an architecture for this technique, a proof-of-concept prototype developed under Android OS and the results of a performance evaluation that was conducted to study its execution times and battery consumption. Despite some limitations, our approach proves to be a valid alternative to enhance security on any smartphone.

Keywords: Security Offloading, Cryptographic services, SIM card, Secure computation

Offloading cryptographic services to the SIM card in smartphones

Resumen

Los smartphones están omnipresentes en la vida cotidiana de las personas como herramientas de comunicación, entretenimiento y trabajo. Las pantallas táctiles y una variedad de sensores ofrecen una experiencia superior y hacen que las aplicaciones sean cada vez más diversas, complejas y demanden más recursos. A pesar de su continua evolución y mejoras, los dispositivos móviles aún están limitados en duración de batería, poder de procesamiento, capacidad de almacenamiento y ancho de banda de red. Computation offloading se destaca entre los esfuerzos para ampliar las capacidades del dispositivo y combatir la creciente brecha entre demanda y disponibilidad de recursos. Como toda tecnología popular, los smartphones son blancos atractivos para atacantes maliciosos. Generalmente almacenan datos privados y se utilizan cada vez más para actividades sensibles como banca en línea o pagos móviles. Si bien computation offloading presenta nuevos desafíos al proteger esos activos, es muy poco común tomar seguridad y privacidad como los principales objetivos de optimización de dicha técnica.

La seguridad del SO móvil depende fuertemente de la criptografía. Los servicios criptográficos por hardware y software disponibles suelen estar diseñados para resistir ataques de software, protección insuficiente cuando se pierde el control físico sobre el dispositivo. Los elementos seguros, en cambio, incluyen un conjunto de protecciones que los hacen físicamente resistentes a la manipulación.

Este trabajo propone una técnica de computation offloading que prioriza mejorar las capacidades de seguridad de los teléfonos móviles descargando operaciones criptográficas a la SIM, único elemento seguro universalmente presente en los mismos. Nuestras contribuciones incluyen una arquitectura para esta técnica, un prototipo de prueba de concepto desarrollado bajo Android y los resultados de una evaluación de desempeño que estudia tiempos de ejecución y consumo de batería. A pesar de algunas limitaciones, nuestro enfoque demuestra ser una alternativa válida para mejorar la seguridad en cualquier smartphone.

Palabras clave: Offloading de seguridad, servicios criptográficos, tarjeta SIM, computación segura

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Smartphones have become one of the most popular computers. Mobile devices in general are increasingly present in people's everyday life as communication and entertainment tools. They are used for personal as well as professional reasons and even work environments are progressively moving towards the Bring Your Own Device (BYOD) model, which allows employees to bring their personal devices such as smartphones, laptops, and tablets to the workplace and access the company's resources and information. The diversity of applications available in mobile platforms is comparable to the existent for traditional PCs. On top of this, touch screens and a variety of sensors offer an even richer experience and making the software that runs inside them increasingly complex and resource demanding (such as video processing, augmented reality, games and all sorts of content based services). Despite their continuous evolution and enhancements, mobile devices are still considered limited when compared to a traditional PC. Their main limitations are observed in battery life, processing power, storage capacity and network bandwidth. The increasing gap between demand and resources availability has drawn the attention of both academy and industry, who are dedicating efforts to extend the capacity of mobile devices and networks. Computation offloading [62] is one of the solutions drawing most attention. Different approaches exist to apply this optimization technique, but essentially all of them are based on the idea of enhancing the device capabilities by migrating computation to a more resourceful computer (such as servers on the network or peripherals). One of the main objectives is to enhance application performance perceived by users and at the same time reduce energy consumption on the device.

As most popular technologies, mobile devices are attractive targets for malicious attackers. They suffer by definition from a broader attack surface than traditional computers because of their strong connectivity. On top of this, they usually store sensitive private data of their owners and are increasingly used for security sensitive activities such as online banking or mobile payments. Unfortunately, as often happens with technology, security aspects are relegated in favor of enhancements in functional or operational aspects. While the introduction of computation offloading presents new challenges in terms of security and privacy, which are being addressed in most research works in the area ([62] [60] [81] [68] [57] [42] [64]), it is very uncommon to apply this technique with the purpose of addressing security related requirements.

Many mobile OS security features are highly dependent on cryptography, especially those countermeasures aimed at protecting sensitive data and credentials from an attacker who has gained physical access to a lost, stolen, decommissioned or unattended device [72] [22] [41]. File or device encryption and secure keystores are typical examples of the former. Ironically, these rely on cryptographic services that cannot claim to be secure at a hardware level. This kind of protection is instrumental when an adversary with limitless access can compromise the device by inspecting or tampering with its hardware or the software it contains (Man-At-The-End or MATE attacks [40]). Services currently found on most mobile devices are based on software implementations or even hardware-backed ones that are designed to protect keys by introducing some form of Trusted Execution Environment (TEE) [51] such as the one provided by ARM TrustZone Technology [70]. The protection profile defined for a TEE [51] targets threats to its assets that arise during the end-usage phase and can be achieved by software means. Attackers have remote or physical (local) access to the device embedding the TEE and may use tools to jailbreak/root/reflash the device in order to get privileged access to the main mobile OS or Rich Execution Environment (REE) allowing the execution of a software exploit. The attacker may have some level of expertise but the attacks do not require any specific equipment. The exploits come from an identification phase attacker that discovers some vulnerability, conceives malicious software and distributes it. This attacker may have software and/or hardware expertise and access to low-budget equipment such as protocol analyzers or JTAG debuggers. Although in TEE based services keys cannot be extracted by an attacker with REE root privileges, he may be able to use them depending on how access controls (such as application binding or user authentication) are enforced by the specific implementation (see Chapter 4). Neither TEE specifications nor TrustZone's cover how the interfaces provided by applications (such as a keystore service) to the main OS should be protected. Also, since the design of those interfaces is application specific too, it might be vulnerable to API-level attacks (see [45]) that attempt to compromise or use secret keys.

Secure Elements [50] such as smart cards include a set of protections that make them physically tamper resistant devices. They are usually designed to prevent keys from being extracted even by advanced attackers with access to laboratory equipment, such as electron microscopes, that can perform almost unlimited reverse engineering of the device. They have the ability to securely store key material and to perform encryption and signature operations without secret keys ever leaving the device's isolated execution environment. Furthermore, user verification methods provided (such as PINs) include countermeasures against brute-force attacks (such as try counters) and can prevent key usage when their owner is not present. These capabilities suggest that their use in the implementation of the above mentioned features can lead to more secure mobile systems. Unfortunately, despite some very recent encouraging signs from Google by the introduction of StrongBox Keystores in their latest Android version [2] and the Titan M chip in their Pixel 3 phones [10], the presence of cryptographic services based on Secure Elements is far from being a standard in smartphones from most manufacturers and even the availability of TEE based solutions is completely optional. According to the background information gathered (see Chapter 2), while there are other alternative manufacturer-dependent form factors that may be present on some devices, it is widely acknowledged that a SIM card can act as a universally present Secure Element in mobile phones.

This work studies the state of the art of computation offloading in smartphones and how it relates to security in those devices. Our main focus was in proposing an offloading technique that prioritizes enhancing security capabilities in mobile phones. We present a rather original approach where the computation load to be optimized are cryptographic operations and the 'more resourceful computer' to which we offload them is the SIM card, a richer environment in terms of security. We were also interested in learning about power consumption and performance for this scenario as secondary optimization variables.

We chose Android Operating System developed by Google as the environment for prototypes and experiments, since it is the most widely deployed mobile OS. Also because it is open source and the ecosystem surrounding it provides a wide variety of tools and information not available in other environments. Android is in constant development and evolution. As of starting our research, 6.0.1 was the latest stable version and Android 7.0 was being released. A lot has changed since, including a re-architecture of the Hardware Abstraction Layer in Android 8.0 and new Secure Element support in version 9.0. Although our prototype and initial background study were based on Android 6.0.1, we provide notes about new features that are relevant for the discussion of our results.

1.2 Objectives

The general goal of this work was to study how computation offloading approaches relate to security and especially to find scenarios where it is applied with security purposes. The first objective was to design a computation offloading technique that uses security as its main optimization objective. After studying the state of the art on the subject and refining the research domain the following more specific objectives were defined:

- Study the approach of offloading cryptographic operations from the main processor of smart phones to the SIM card as an unconventional offloading technique with the purpose of enhancing mobile security.
- Define an architecture to apply this approach in smart phones.
- Provide a proof of concept prototype for a widely used mobile OS.
- Learn about its performance in terms of response time and energy consumption, aspects that reflect typical user concerns around mobile experience and are usually part of the offloading optimization objectives.

1.3 Contributions

The main contributions of this work are:

- An architecture for offloading cryptographic operations to the SIM card in smart phones.
- A detailed exposition of how to implement its components under Android OS including a proof of concept prototype.
- Performance benchmarking results comparing response times and energy consumption of executing the most widely used cryptographic algorithms in both the SIM and the smartphone main chip. Results of this type are a valuable input when conducting a profiling analysis of a program's components as a step of an offloading decision process.

These contributions were validated by the publication of a full paper in the proceedings of an international conference [78].

Secondary contributions of this work include a discussion of possible applications of this technique that might have an immediate impact on Android's Security and the proposal of future research lines and implementation work in that realm. These contributions were also validated by a publication, in this case a short paper [77] was presented in the Student Forum of the same conference, getting feedback from the community.

1.4 Document outline

The rest of this document is structured as follows: Chapter 2 introduces background needed to motivate and contextualize our work. Chapter 3 presents our proposed technique, architecture, provides a detailed description of the developed prototype and the results of performing some experiments and conducting a performance evaluation on a real phone. In Chapter 4 we compare and link our work with similar research on the area, discuss potential applications and point out some limitations of the followed approach. Finally, Chapter 5 concludes.

Chapter 2

Background

In this chapter we summarize the required background knowledge to help us put across the motivation behind our work and discuss its potential. We will start by presenting the concept of computation offloading, its relation with security and provide a summary of the state of the art on the subject. We will later focus on some relevant threats to mobile security where our work may prove to make an impact. We also go over some preexistent conditions (mostly common features in state-of-the-art SIM cards and smartphones) that form the basis for our proposed architecture and hint at some of the challenges identified, such as baseband modem's exclusive access to the UICC.

2.1 Computation Offloading

In this section we focus on the concept of computation offloading in mobile devices [62]. Different approaches exist to apply this optimization technique, but essentially all of them are based on the idea of enhancing the device capabilities by migrating computation to a more resourceful computer. This involves making a decision regarding whether and what computation to migrate in order to obtain benefits. One of the main objectives is to enhance application performance perceived by users and at the same time reduce energy consumption on the device. It is very uncommon to take security into account as one of the optimization objectives.

The rest of this section provides a summary of the state of the art of computation offloading strategies in mobile devices and their relationship with security.

2.1.1 Motivation

In spite of the fast development of key components such as CPU, GPU, memory, wireless networks and operating systems, mobile devices still present great disadvantages compared to a PC. The main reasons for this are their limited processing power and that the battery, only energy source in these terminals, has shown relatively slow progress in the last decade. Lithium-ion batteries, currently the most used kind, have increased their energy density an average of 51% annually since their emergence [66] and due to material limitations it is unreal to expect greater increases. Moreover, at current rates projections show that density theoretical limits will be reached within the next decade. In this scenario, if a new generation of batteries does not come into play, significant increases in capacity would be tied to increases in volume, something that goes against the demands of the smartphone and tablet market where hardware is expected to be lighter and thiner every year. The emergence of cloud computing has changed completely the service model of modern applications. It is possible to access infrastructure, platform, and software as a service through the Internet, commercialized by cloud vendors (such as Amazon, Google, Microsoft, etc.) in convenient ways following pay-as-you-go models. This offers resource provisioning on demand in a reliable and flexible manner and at a low cost, allowing companies to grow fast without the extra efforts of mounting and maintaining their own infrastructure or the extra costs of over-provisioning. Mobile cloud computing can benefit from these advantages to expand device capabilities.

Computation offloading is a mechanism that, by migrating computation tasks to a more resourceful computer (such as servers on the network or peripherals), increases mobile device capabilities over its physical limits and may extend battery charge intervals by saving energy. This is different from a pure client-server model where a thin client always delegates computation to a server. In computation offloading a decision process is in charge of determining if and which part of the load is to be migrated. The process may determine that benefits are obtained by migrating all or part of the load, but it also may conclude that it is more efficient to execute it locally. This decision usually depends on several parameters including, bandwidth, server speed, memory available or the amount of data exchanged between the terminal and the server through the network.

Some of the main challenges presented by computation offloading systems include:

- Interoperability: a wide variety of mobile devices with different resource levels, bandwidth, hardware and software may offload computation to one or more servers.
- Mobility and fault tolerance: the use of wireless networks may produce instability due to congestion and mobility problems. The system must be fault tolerant and continue executing tasks in the event of these problems.
- Security and privacy: these aspects are a concern, since programs and user data may be sent to servers outside the mobile environment at risk of being accessed or tampered with by third parties.

2.1.2 Models

We will start by introducing the offloading decision problem. Since not every component of an application is suitable for executing remotely (outside of the main processing environment of the device), as a first step towards offloading we need to distinguish between offloadable and unoffloadable ones. Unoffloadable components normally include those that handle access to local I/O (including peripherals such as the device's camera) or user interface and interaction. Second, a profiling analysis of the components is required to determine their computation costs in terms of the optimization variables (i.e. execution time, energy consumption, etc.). An estimation of intermediate data exchanged between components is also required. Next, we need to apply an optimization algorithm to decide, based on these inputs, which of the offloadable components are actually going to execute remotely. That is the result of the offloading decision and is referred to as *partitioning*. Offloading decisions are usually represented as a graph [53]. Consider an application consisting of an unoffloadable component and N offloadable ones. Even if more than one unoffloadable component exists, we still can merge them without losing generality in this representation. We will use a weighted directed graph $G = (V,E)$ with $|V| = N + 1$ to represent the relationship between components, see Fig. 2.1. Application execution is characterized by the direction of edges. Each vertex $v \in V$ denotes a component and the weight of a directed edge (u, v) represents the size of data migration from vertex u to v. For example, in Fig. 2.1, component with index 2, before it is executed, requires the data from components with indexes 0 and 1. The component with index 0 is unoffloadable and the other N components are offloadable. The application execution completes when the component with index N returns the result to component 0. Although vertices 1 to N represent possible offloading candidates, it is not convenient to apply a decision algorithm independently on each of them. Intermediate data exchange (edge weights) must be taken into account. For example, if e_{12} is too large components with index 1 and 2 should execute on the same system to guarantee local communication. An optimal decision implies finding a balance between variables such as response times, energy consumption and communication, which requires considering every vertex simultaneously.

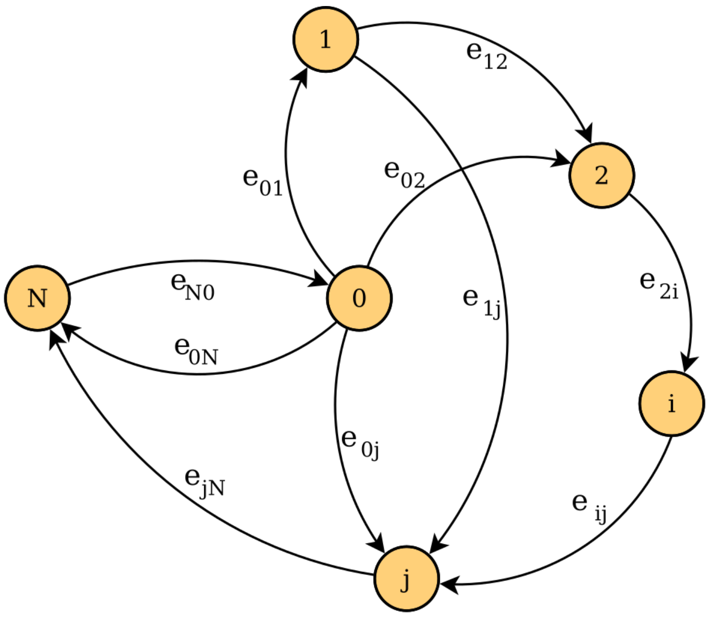


Figure 2.1: Offloading decision graph model.

2.1.3 Categories

A variety of approaches exist to make offloading decisions. They can be classified according to different factors. We will start by focusing on the optimization objective. As explained in [62], the two alternatives that receive most attention in research works are enhancing performance and saving energy. These objectives are not opposites, actually they depend on some common variables. Performance optimization is generally measured in terms of response or execution time of tasks. An alternative objective might be complying with real time restrictions. For example in a navigation system where information from different sources must be analyzed simultaneously to give real time feedback, restrictions on response time can be necessary so that feedback does not lose its value. The other important categorization factor for offloading solutions is related to 'when' the decision is taken. The alternatives include doing it statically during application development or dynamically during its execution. When the decision is static the program is partitioned during development reducing overhead introduced during execution. However, this approach is only valid when parameters can be precisely predicted beforehand. For example, execution time and energy consumption can usually be predicted for a specific system or a minimum execution speed guaranteed by a cloud provider, while network bandwidth available may vary at execution time in most scenarios. Different prediction techniques can be employed in order to statically decide how to partition the application. We will see some of them in the following section. On the other hand, dynamic decisions can adapt to changes in execution conditions such as variable bandwidth. These approaches may also use prediction techniques to make decisions. In this case parameter monitoring is required to later predict their behavior. Monitoring execution conditions causes a higher overhead in this type of decisions. Although the decision is dynamic, the process of identifying offloadable tasks is usually conducted during development, precisely to reduce the overhead of program analysis. According to the survey and categorization presented on [62], since 2005 most works in the area use dynamic decisions.

2.1.4 Techniques

In this section we review the challenges faced when performing offloading decisions and the type of techniques used to solve them.

The offloading decision problem, as many graph partitioning problems where vertices are divided into sets with specific properties such that few edges cross between sets, is known to be NP-complete even if all the parameters are known in advance [62]. This can be relevant since graph sizes vary depending on the program and the decomposition strategy chosen. For example, class [76] or function [62] level decompositions may lead to large graphs and so to hard problems. There is a huge amount of research about partitioning graphs in polynomial time using heuristics, but these techniques must be adapted to be applied for offloading decisions, since they can be expensive and produce an amount of overhead incompatible with the optimization requirements. For example [76] adapts a multilevel heuristic algorithm so that is light enough to make dynamic offloading decisions based on metrics obtained on-the-fly.

There is a wide variety of options to implement dynamic decision algorithms. Alternatives may vary from repeating partitioning in every execution taking metrics on-the-fly, to calculating partitions that guarantee certain probability to remain optimal in the event of changes in order to repeat the process less frequently or even monitoring execution context to trigger partitioning only when a change in conditions dictates so. One of the most variable metrics in mobile systems, which heavily affects offloading decisions, is available bandwidth. Solutions that assume it as a constant value present great disadvantages. Some solutions monitor this type of variables and dynamically react to changes using rules. Others, such as [83], model bandwidth as a random variable and present the partition problem as an integer programming one with probability in search of change resistant partitions. Here service time and energy are introduced as service quality restrictions met by solutions with certain probabilities.

A set of frameworks and middlewares are available that simplify offloading implementation in applications following certain patterns so that developing solutions to the most typical problems is not required. Some examples of these are:

- MACS [61], a framework for Android that solves partitioning with resource monitoring and adaptability.
- Cuckoo [59], a framework that transparently integrates offloading features into Android.

In [39] a recent work studies all available frameworks and provides a classification.

2.1.5 Offloading and security

Mobile devices are attractive targets for malicious attackers. The introduction of computation offloading tends to bring even greater challenges in terms of security and privacy, since it usually implies a broader attack surface including servers in the cloud and insecure networks. This causes that security is at least presented as a concern in most research works in the area ([62] [60] [81] [68]) and that novel techniques are being employed to protect security [57] [42] and privacy [64]. Besides the challenges it may present, offloading can also have a positive impact on these aspects. Many security solutions are available in the form of applications directly installed on the device. Although these on-device strategies may offer some advantages, they generally imply significant resource consumption on the device leading to a reduction in battery charge duration. As mentioned, it is possible to use computation offloading to migrate part of this load to remote servers. Some examples of this can be found in [73], [55] and [54]. It is rare to find research works where offloading decisions take security or privacy as one of the optimization objectives. They typically appear as statically imposed conditions or restrictions, but final decisions are based on resource usage and response times.

2.2 Mobile Device Security

As their name suggests, one of the key differences between this type of devices and other computation environments is mobility. In terms of security, it is the first reason why they have always been significantly more vulnerable to attacks. As shown in the following section, physical access is one of the main threat categories for this type of devices. Many of the typical countermeasures to mitigate these threats (such as file or device encryption and secure keystores) are highly dependent on cryptography, especially those aimed at protecting sensitive data and credentials from an attacker who has gained physical access to a lost, stolen, decommissioned or unattended device. The level of protection provided by such features is as reliable as the cryptographic services behind them. In these scenarios cryptography modules are exposed to attacks focused on their hardware support. The use of tamper resistant devices is an instrumental countermeasure for this exposure, yet their presence has not been standardized by chip or phone manufacturers.

2.2.1 Threats and countermeasures

In this section some relevant mobile threats catalogs and taxonomies are reviewed. We will start with OWASP Mobile Security Project [21]. Table 2.1 provides an extract of the latest Top 10 Mobile Risks released in 2016 [22]. We can observe that in ‘Insecure Data Storage’ and ‘Insufficient Cryptography’, two of the top five risks, an adversary with physical access to the device is a key threat agent. We also observe that cryptography plays an important role in the countermeasures for risks for M3, M4 and M5. Another concern raised is that encryption protections are vulnerable to rooting.

The National Institute of Standards and Technology (NIST) has been working on a Mobile Threat Catalog [72] in recent years. This catalog includes physical access as one of the twelve categories selected to arrange the threats identified, table 2.2 shows some extracts. Two of the main threats inside this category are ‘Device loss or theft’ and ‘Unauthorized access to device data resulting from poor lifecycle management’ (for example improper disposal). The countermeasures presented for these threats include enforcing data encryption and device lock policies such that the recovery of data becomes highly improbable. The effectiveness of other countermeasures not based on cryptography like remote wipe depends on device connectivity which might be disabled by the attacker, for example simply turning the device off or removing its SIM card.

In [41] the authors present a recent review on Android data storage security. They propose taxonomies of threats and countermeasures where the two main categories are Software Threats and Physical Threats. The main physical threats presented (such as evil maid and cold boot attacks) are viable because of the use of software based encryption systems and extraction of keys from RAM and internal memory (see table 2.3). They explain how Android Full disk encryption feature is vulnerable to some of these attacks. The countermeasures presented for this category (see table 2.4) of attacks include enhanced encryption systems (such as Sentry, Armored or Droidvault) where keys are stored in CPU registers, TrustZone [70] or other architecture provided mechanisms. These types of protections are said to make attacks more difficult and expensive, while leading to lower performance or turning the system more impractical for end-users.

Table 2.1: OWASP Mobile Security Project TOP 10 Risks extracts.

	Threat Agent	**Vectors**	**Weaknesses**
M2: Insecure Data Storage	An adversary that has attained a lost/stolen mobile device.	In the event that an adversary physically attains the mobile device, the adversary hooks up the mobile device to a computer with freely available software.	Rooting or jailbreaking a mobile device circumvents any encryption protections.
M3: Insecure Communication	An adversary that shares your local network (compromised or monitored Wi-Fi).	The exploitability factor of monitoring a network for insecure communications ranges.	Mobile applications frequently do not protect network traffic. They may use SSL/TLS during authentication but not elsewhere.
M5: Insufficient Cryptography	Anyone with physical access to data that has been encrypted improperly.	Decryption of data via physical access to the device.	Weak encryption algorithms or flaws within the encryption process.

2.3 Secure Computation

In this section we introduce and compare two forms of secure computation environments. Understanding these two alternatives and how they differ is instrumental to appreciate how our work's proposal can have an impact on mobile security against the types of threats covered in the previous section.

2.3.1 Trusted Execution Environments

GlobalPlatform defines specifications to standardize the concept of Trusted Execution Environment (TEE) [51]. A TEE is an execution environment that runs alongside but isolated from a Rich Execution Environment, has security capabilities and meets certain security related requirements: It protects TEE assets from general software attacks, defines rigid safeguards as to data and functions that a program can access, and resists a set of defined threats. There are multiple technologies that can be used to implement a TEE, and the level of security achieved varies accordingly. The TEE consists of three parts: hardware-based isolation technology (such as Arm TrustZone [70]), trusted boot and a small trusted OS. The TEE can be used to run multiple isolated trusted applications usually called trustlets which may be provisioned over the air. In GlobalPlatform TEE documents, trusted storage indicates storage protected either by the hardware of the TEE, or cryptographically by keys held in the TEE. A GlobalPlatform TEE Trusted Storage is not considered hardware tamper resistant to the levels achieved by Secure Elements [50].

Table 2.2: NIST Mobile Threat Catalog extracts.

Category	**Threat**	**Countermeasures**
Physical access	Device loss or theft.	To mitigate the impact of a lost or stolen device in the possession of an attacker, use remote lock, activation lock, locate, or wipe capabilities as deemed appropriate based on the sensitivity of data stored on or capabilities of the device.
Physical access	Unauthorized access to device data resulting from poor lifecycle management.	Use EMM or MDM solutions in combination with devices that successfully enforce data encryption and device lock policies (unlock code set, unlock code strength requirements, auto-locking enabled, and auto-wipe enabled) such that the recovery of data from an improperly retired device becomes highly improbable.

Table 2.3: Android data storage security review extracts: Threats.

Category	**Threat**	**Weakness**
Physical threat	Cold boot attack.	RAM can be re-plugged to another computer to get its content such as encryption keys. This attack can be made against all software-based encryption technologies. Researchers determined this type of attack can break Android Full disk encryption (FDE).
Physical threat	Evil maid attack.	Any software-based encryption needs a part of the disk unencrypted; in Android, this is the entire system partition.

Table 2.4: Android data storage security review extracts: Countermeasures.

Category	Countermeasure	Strategy
Physical threat solution	Sentry	Avoid storing encryption/decryption keys in RAM and use ARM system-on-chip (SoC) low capacity storage next to the CPU instead.
Physical threat solution	Armored	Store the encryption keys and intermediate values of AES inside the CPU registers of the ARM microprocessor.
Physical threat solution	Droidvault	Utilize TrustZone to manipulate the unencrypted data

ARM TrustZone

Most mobile phones and tablets are based on an ARM processor. ARM does not produce processors itself but rather licenses architecture designs to chip manufacturers, who add their own features and produce the actual chips. As explained in [70], ARM TrustZone Technology is an optional hardware security extension of the ARM processor architecture, that creates an isolated secure world which can be used to provide confidentiality and integrity to the system. This is achieved by partitioning all of the SoC's hardware and software resources so that they exist in one of two worlds. The secure world for the security subsystem and the non-secure world for everything else. ARM has implemented this split-environment processor with various hardware additions to enforce security restrictions while preserving the low power consumption and other advantages of their designs. These hardware additions enable a single physical processor core to safely execute code from both worlds in a time-sliced fashion, removing the need for a dedicated security processor core. Each physical processor core provides two virtual cores, one considered non-secure and the other secure, plus a mechanism to context switch between them. The security state is encoded on the system bus and this enables trivial integration of the virtual processors into the system security mechanism; the non-secure virtual processor can only access non-secure system resources, but the secure virtual processor can see all resources. The main bus contains a non-secure (NS) bit that indicates whether a read/write operation is directed to secure or non-secure memory. A bridge between this bus and the peripheral bus allows for secure communication between a CPU and peripherals by checking for appropriate permissions and blocking unauthorized requests. The cache controller also looks for an NS bit. Since both worlds share the same physical cache, the same location may have two distinct addresses, requiring a controller to look up the correct location. This also includes L2 cache and other smaller locations. The Direct Memory Access (DMA) Controller is used to transfer data to physical memory locations instead of devoting processor cycles to this task. This controller can handle secure and non-secure events simultaneously, with full support for interrupts and peripherals. It prevents non-secure access of secure memory. Additions to several controllers prevent non-secure access of secure memory, allow dynamic classification of memory-mapped devices as secure or non-secure and prevent non-secure interrupts from unauthorized access, among other controls. These extensions to enforce a separation are presented to be more resource efficient than actually using two separate processors.

The normal world is usually a general OS such as Linux or Android, while the secure world can be anything between a full OS and a small library. As pointed out by [47], the hardware features described do not implement or ensure a secure environment. Some functionalities (such as context switching between the two worlds) are left to the software running in the secure world. The communication of data between the two worlds is left to the software running in both worlds to implement. To complete a secure execution environment and to allow multiple applications to be run in the secure world a second OS is required. The secure world OS should schedule resources and context switches between both the applications running in the secure world and the operating system running in the normal world so that no data is leaked. Securely booting both parts of this system is a key concern. Without proper verification of both images, the device may inadvertently boot a malicious version, giving attackers an entry route. Therefore, ARM has designed TrustZone-enabled systems to use a Secure Boot Sequence. To build a chain of trust, each step can be cryptographically verified.

2.3.2 Secure Elements

GlobalPlatform defines a Secure Element (SE) [50] as a tamper resistant component which is used in a device to provide the security, confidentiality, and multiple application environment required to support various business models. It may exist in any form factor such as UICC, embedded SE, smart SD, smart microSD, etc. A variety of Secure Elements may be present in mobile devices, Fig. 2.2 shows the alternatives and their compatibility. The SIM card is the only universally present SE in every smartphone. The embedded SE typically appears as part of an NFC controller (when present). Recently the embedded SE has been deprecated in favor of host-based card emulation in these controllers. The smartSD is basically an SD card with an embedded SE chip. These memory cards comply with both the widely deployed SD architecture and the Advanced Security SD (ASSD) standard [24] by the SD Association (global ecosystem of companies setting industry-leading memory card standards). Their presence is completely optional.

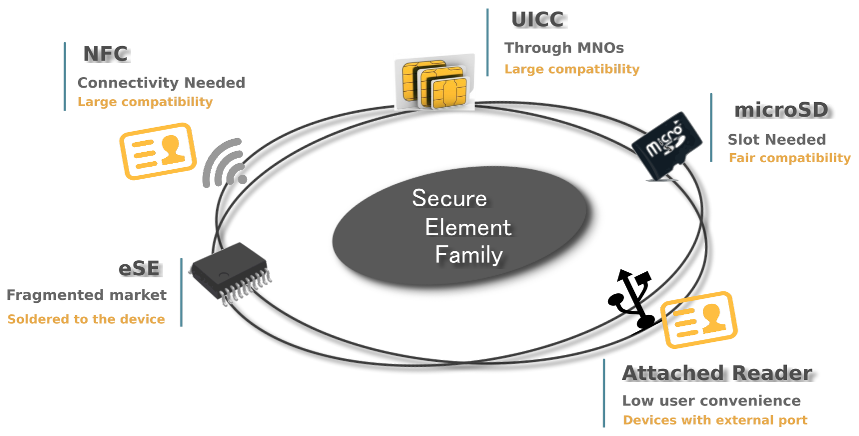


Figure 2.2: Secure Element form factors in mobile devices.

There are a few key differences between any Secure Element present in a mobile device and a TEE solution like TrustZone [46].

- A Secure Element does not provide a trusted path. The secure world in TrustZone has access to the display controller and peripherals providing a trusted path with the user.

A Secure Element usually can only communicate with the main processor over a serial channel and cannot control the display.

- Secure Elements are physically tamper resistant. In contrast, a TrustZone-enabled processor does not necessarily have any physical tamper protection [43].

- Secure Elements are usually platforms with power constraints. As a result the computing power of a Secure Element may be limited compared to the full computing power of all cores on an ARM chip available to the TrustZone-based TEE.

In summary, while both Secure Element and TEE provide secure computation, a SE has more physical security features while the TEE is safer for interaction with the user and provides higher computing power.

2.4 The SIM card

The SIM (Subscriber Identity Module) card is a smart card used in mobile phones starting from 2G cellular networks. It is issued by mobile network operators and securely stores subscriber's service key (among other parameters) allowing consumers to establish secure and trusted voice and data connections, as well as storing contact and SMS information. SIM cards are smart cards, therefore they comply with physical and electrical specifications in ISO-7816 [12]. However, not every smart card is necessarily capable of working as a SIM card. For this to be true, it must comply with one of 3GPP (3rd Generation Partnership Project) [1] Technical Specifications (TS) and be certified by SIMalliance [28] (SIMalliance members represent 90 % of the global SIM card market). The use of this element became mandatory in GSM networks where it was named SIM card and no distinctions were made between hardware and software that composed it (see [32]). Its equivalent since 3rd generation UMTS networks is called USIM (Universal Subscriber Identity Module) card or UICC (Universal Integrated Circuit Card). The UICC is unequivocally defined as universal hardware that can execute multiple applications for different mobile network technologies simultaneously (see [37]), an important advantage over original SIM cards. An application always present, the USIM, is in charge of identifying the subscriber to a service provider of one of the following standards: Universal Mobile Telecommunications System (UMTS), High Speed Packet Access (HSPA) or Long Term Evolution (LTE). A separate contacts application is also always present. Other optional applications include IP Multimedia Services Identity Module (ISIM) for secure mobile access to multimedia services and payment related applications. ETSI TS 102 266 [30] features an application called UICC Security Service Module (USSM), a general security module, which offers services to other applications on the UICC through an API. This short specification that has not evolved since its introduction in 2006 and its stage 2 in 2007 [31]. UICC standard applications do not provide general purpose cryptographic services to mobile devices.

Over The Air (OTA), defined for UICCs in [34], is a mechanism that permits sending commands to the card over binary SMS or other services, allowing operators to maintain control over the card's configuration after its issuance (includes installing and activating applications). OTA requires command wrapping according to [33], which provides integrity check and encryption. One of the main uses of OTA has been maintaining SIM Toolkit (STK) applications.

As it is commonly accepted, in this document the terms 'SIM card' and 'UICC' will be used interchangeably. Although originally presented as Full-size (FF) smart cards, today most popular SIM card sizes are Mini-SIM(2FF), Micro-SIM(3FF) and Nano-SIM(4FF).

2.4.1 Current UICC

In terms of hardware and platform, first SIM cards (and first smart cards in general) were based on a file system model, where elementary files (EF) and dedicated files (DF) were named with 2 byte identifiers. In those cards, defining an application consisted of describing its file structure in terms of DFs and EFs. For example, GSM application's DF identifier is '7F20' and under USIM application DF you can find an IMSI EF and a keys EF. Today most UICCs are based on the Java Card technology (JC) [20] and comply with GlobalPlatform (GP) [8] specifications. In this platform, mobile network applications are implemented as Java Card Applets and only emulate a file system for backwards compatibility. An Applet in Java Card Framework is the application unit and where the commands exposed are implemented. The Issuer Security Domain (ISD), a GP component inside the card that represents its issuer, gives network operators exclusive full control over card life cycle and content management. This is enforced by requiring the establishment of a secure channel based on symmetric and asymmetric cryptography.

Since current SIM cards are multi-application UICCs, higher end smart cards are being employed for this task. Examples of these are Gemalto's Multi-tenant SIM [7] which includes hardware support for RSA and ECC or Kona I LTE-NFC USIM [15] which includes hardware RSA support.

2.4.2 Performance and energy consumption

Smart card processor designs are based on proven components [79]. Lower end processors include 16 bit CISC and RISC architectures which may feature symmetric cryptography co-processors for algorithms such as Triple DES or AES. In fact, that has historically been very common because of their standard use in payment or telecommunication applications. Also because they can be efficiently implemented in hardware and execution times decrease is notorious. At the upper end of the performance scale for smart card microcontrollers, 32-bit types manage larger memories above the 64-KB limit and are suitable for the requirements of interpreter-based platforms such as Java Card. The key selection criteria for processors include code density, power consumption, and resistance to attacks. ARM has achieved great success in the smart card world with their SecurCore 32 bit processor family, a RISC architecture based on ARM Cortex-M series. SecurCore SC300 processors [5], designed specifically for high-performance smart cards, currently occupy between 0.028 mm^2 and 0.40 mm^2 and present power consumptions between 13 μW/MHz and 162 μW/MHz. For calculations in the realm of public key algorithms, there are specially designed calculation units that are located on the silicon along with the usual functional components of the microcontroller. They are limited to performing some basic calculations that are necessary for these types of algorithms: exponentiation and modulo on large numbers. Both operations are essential elements of public-key encryption algorithms, such as RSA and ECC. The speed of these components results from their very broad architectures and high clock rates. According to [79], in their specific application area, some of them can even outperform a high-performance PC.

2.4.3 Phone Integration

A fact sometimes overlooked is that smartphones as telecommunication devices compliant with the already mentioned standards, general purpose application platforms and sensor rich hardware are not composed from a single processor or controlled by a single operating system [49].

Complexity and diversity of mobile network protocols that must be supported by these devices, require that their implementation is delegated to a dedicated hardware component, the baseband processor or baseband modem. Although this hardware module may communicate with the main processor or main memory and there has been a general trend towards tighter integrated hardware components (System on a Chip or SoC), the baseband is a separate processor that runs its own operating system and is in charge of all radio related operations. This type of functionality requires real time OS and architecture. ARM Cortex R architectures are commonly chosen, as they are designed for this kind of task. OS is usually delivered as closed source proprietary firmware provided by chip manufacturers.

Our main interest in the baseband modem comes because the UICC is directly connected to it and the only way to access the card is through the interface provided by the modem. As this is a proprietary interface, manufacturers must provide proper wrappers or drivers for mobile operating systems such as Android or iOS. Although these wrappers always expose access to the SIM for network authentication, contact management and STK execution, general APDU commands exchange support is not mandatory. The only standard way to achieve this is using extended AT commands as defined by [35]. Examples of these commands are *AT+CSIM* (Generic SIM access) and *AT+CGLA* (Generic UICC Logical Channel Access).

Mobile operating systems do not usually expose SIM card low level access through the application framework.

2.5 Android OS Security

Android provides an open source platform and application environment for mobile devices [3]. As Fig. 2.3 illustrates, the main Android platform building blocks are:

- Device hardware: Android runs on a wide range of hardware configurations including smartphones, tablets, watches, automobiles, smart TVs, OTT gaming boxes, and set-top-boxes. Android is processor-agnostic.
- Android operating system: The core operating system is built on top of the Linux kernel. All device resources, like camera functions, telephony functions, network connections, etc. are accessed through the OS.
- Hardware Abstraction Layer (HAL): A HAL defines a standard interface for hardware vendors to implement, which enables Android to be agnostic about lower-level driver implementations.
- Android Application Runtime: Android applications are most often written in the Java programming language and run in the Android runtime (ART). However, many applications, including core Android services and applications, are native applications or include native libraries. Both ART and native applications run within the same security environment, contained within the Application Sandbox. Applications get a dedicated part of the filesystem in which they can write private data. There are two primary sources for applications:
  - Pre-installed applications: These function both as user applications and to provide key device capabilities accessed by other applications. Pre-installed applications may be part of the open source Android platform (phone, contacts, etc) or developed by device manufacturers.
  - User-installed applications: Android provides an open development environment that supports any third-party application. Google Play offers users hundreds of thousands of applications.

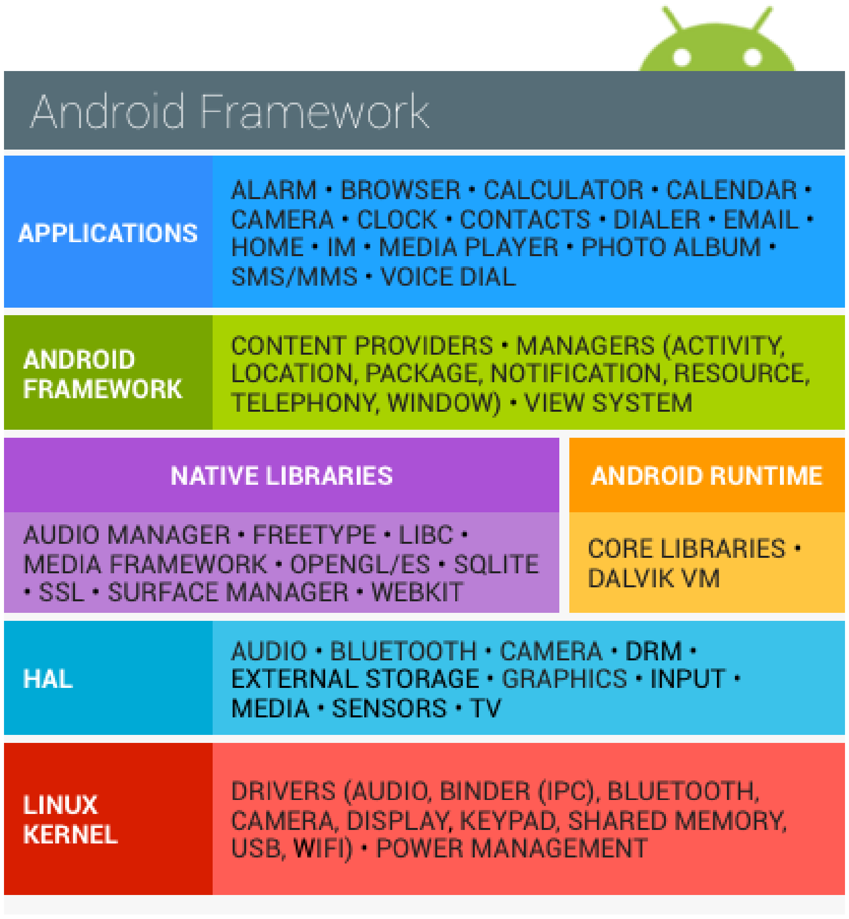

Android Framework
APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

Figure 2.3: Android Software Stack [3].

2.5.1 Overview

Android states in its official security documentation [3] that it seeks to be a secure and usable operating system for mobile platforms by providing these key security features:

- Robust security at the OS level through the Linux kernel
- Mandatory application sandbox for all applications
- Secure interprocess communication
- Application signing
- Application-defined and user-granted permissions

As the base for a mobile computing environment, the Linux kernel provides Android with several key security features, including:

- A user-based permissions model
- Process isolation
- Extensible mechanism for secure IPC
- The ability to remove unnecessary and potentially insecure parts of the kernel

Android 6.0 and later supports verified boot and device-mapper-verity. Verified boot guarantees the integrity of the device software starting from a hardware root of trust up to the system partition. During boot, each stage cryptographically verifies the integrity and authenticity of the next stage before executing it.

Android 7.0 and later supports strictly enforced verified boot, which means compromised devices cannot boot.

2.5.2 Rooting of Devices

By default, on Android only the kernel and a small subset of the core applications run with root permissions. Android does not prevent a user or application with root permissions from modifying the operating system, kernel, or any other application.

As Android documentation states [3], the ability to modify an Android device they own is important to developers working with the platform. On many Android devices users have the ability to unlock the bootloader in order to allow installation of an alternate OS, allowing them to gain root access for purposes of debugging applications and system components or to access features not presented to applications by Android APIs.

On some devices, a person with physical control of a device and a USB cable is able to install a new operating system that provides root privileges to the user. To protect any existing user data from compromise the bootloader unlock mechanism requires that the bootloader erase any existing user data as part of the unlock step. Root access gained via exploiting a kernel bug or security hole can bypass this protection.

Encrypting data with a key stored on-device does not protect the application data from root users. Applications can add a layer of data protection using encryption with a key stored off-device, such as on a server or a user password. This approach can provide temporary protection while the key is not present, but at some point the key must be provided to the application and it then becomes accessible to root users. In the case of a lost or stolen device, full filesystem encryption on Android devices uses the device password to protect the encryption key, so modifying the bootloader or operating system is not sufficient to access user data without the user's device password. As [3] states, a more robust approach to protecting data from root users is through the use of hardware solutions.

2.5.3 User Security Features

Android can be configured to verify a user-supplied password prior to providing access to a device. Use of a password and/or password complexity rules can be required by a device administrator.

Android 3.0 and later provides full filesystem encryption, so all user data can be encrypted in the kernel. Android 5.0 and later supports full-disk encryption, which uses a single key—protected with the user's device password—to protect the whole of a device's userdata partition. Upon boot, users must provide their credentials before any part of the disk is accessible. Android 7.0 and later supports file-based encryption, which allows different files to be encrypted with different keys that can be unlocked independently.

2.5.4 Cryptography

Android provides a set of cryptographic APIs for use by applications [3]. These include implementations of standard and commonly used cryptographic primitives such as AES, RSA, DSA, and SHA. Additionally, APIs are provided for higher level protocols such as SSL and HTTPS. These programming interfaces are based on a provider model, where different implementations may coexist as separate providers. Android usually defaults to a software provider called ‘AndroidOpenSSL’ and based on the OpenSSL Library [19].

The availability of a TEE in a SoC offers an opportunity for Android devices to provide hardware-backed security services to the OS, platform services, and even to third-party applications. The API to access those services is called Keystore. It is provided by the Keymaster HAL and made also available through standard application APIs by a specific cryptography provider. If no hardware based implementation of the Keymaster is provided, either because a TEE is not present or because drivers to take advantage of it are no provided, Android defaults to a software based Keymaster, again using the OpenSSL Library. In Android 6.0, Keystore added access control system for hardware-backed keys. Keys can be restricted to be usable only after the user has authenticated, and only for specified purposes or with specified cryptographic parameters. In Android 7.0, Keymaster 2 added support for key attestation and version binding (binds keys to operating system and patch level version, preventing rollback attacks). In Android 9, updates include support for embedded Secure Elements [2].

2.5.5 SE support

Although the different types of Secure Elements mentioned in section 2.3 may be present in Android phones, the application framework has historically lacked of a standard API to interact with them. SIMalliance defined the *Open Mobile API Specification* [65] (transfered to GlobalPlatform at the end of 2016), with the goal to give applications access to the different form factors of Secure Elements present in mobile devices. These can be integrated as a UICC, embedded in the handset, connected to a SD card slot, accessed through a USB reader or even through a wireless interface. An open source implementation (for compatible devices) of the API is available from the SEEK for Android project [25] maintained by Giesecke & Devrient. SEEK provides Android patches that implement a Service Manager (*SmartCardService*) that can connect to any SE available (including the UICC) and extensions for Android’s Telephony Framework that enable transparent APDU exchange with the card. From API level 21 (Android 5.0) onwards, Android’s framework already provides primitives to specifically interact with the UICC as part of the Telephony Manager [4]. Both APIs require extended AT commands support by the baseband modem and its wrappers in order to make the UICC available.

Most recently, since Android 9 release, the OS integrates an implementation of the Open Mobile API in its core source tree [18]. Device manufacturers must provide SE HAL implementations [2] to make GlobalPlatform-supported Secure Elements available through the Secure Element Service.

2.5.6 TEE support

Android platform includes Trusty, which is a set of software components supporting a Trusted Execution Environment (TEE). According to [3] Trusty consists of an operating system (the Trusty OS) that runs on a processor intended to provide a TEE (such as TrustZone enabled ones), drivers for the Android kernel (Linux) to facilitate communication with applications running under the Trusty OS and a set of libraries for Android system software to facilitate communication with trusted applications executed within the Trusty OS using the kernel drivers. Trusted applications are written as event-driven servers (much like Java Card applets) waiting for commands from other applications running on the TEE or main processor. Currently all Trusty applications are developed by a single party (usually manufacturers) and packaged with the Trusty kernel image. The entire image is signed and verified by the bootloader during boot.

According to [46] chip manufacturers like Qualcomm and Texas Instruments provide proprietary Keymaster trustlets and drivers to make available hardware-backed keystore services in Android phones based on their chips. Their implementations of access restrictions such as application binding can be bypassed by a root attacker.

Chapter 3

Offloading cryptographic services to the SIM card

Physical access to the device is one of the main threat categories in mobile security [72] [22] [41]. Cryptography plays a key role among the available countermeasures (see section 2.2). It is no surprise that mobile OS security features aimed at protecting user data from this type of threats are heavily based on it. Unfortunately, they often rely on software or TEE based cryptographic services, which resist software attacks, but are not physically tamper resistant (see section 2.5). Secure Elements on the other hand, as their main advantage over a TEE (see section 2.3), provide that type of protections which are instrumental in this scenario. Because of this, their use in the implementation of the above mentioned features can lead to more secure mobile systems.

It is widely acknowledged that the SIM card can act as a universally present Secure Element in mobile phones [46] [49]. A further question to address is whether it is also possible that offloading cryptographic operations from the main processor to the UICC results in lower power consumption while achieving at least similar performance.

This chapter puts forward the main body of our work. In the following sections we present a computation offloading method based on the UICC, an architecture for it, a proof-of-concept prototype under Android OS and the results of conducting a performance evaluation in a real phone in order to address our research questions and specific objectives.

3.1 Proposal

We propose a computation offloading method where applications or programs are decomposed into cryptographic functions, i.e. tasks based on a sequence of cryptographic operations (e.g. digest, cipher, signature, random number generation, etc.). These functions are the offloadable parts of the application. Offloading a function here means that the required calculations are delegated to the UICC, rather than being executed in the main processor. The main variables to take into account in an offloading decision process for this method should be security, that is enhanced by executing operations in the UICC and can be the main objective, alongside execution time and energy consumption. These last variables affect user experience and should comply with service quality restrictions or be secondary optimization objectives. This offloading method does not present the typical problems of cloud computing based ones:

- It does not have the negative concerns in terms of security and privacy of data and code leaving the device for remote servers.

- It does not have the instability factors of wireless connectivity and variable bandwidth.

3.2 Architecture

We propose an architecture for offloading cryptographic services to the UICC in mobile devices. Fig. 3.1 depicts its main composition, where black represents the standard hardware and software of a smartphone's computation environment and orange identifies components and communication flows introduced by our work. At card level our architecture features a general purpose cryptographic service application deployed by mobile network operators. To make these services available at OS and application level we present a software service module inside the phone that may expose different high level interfaces to access cryptographic primitives. This module (represented as 'Cryptographic Service') encapsulates and makes transparent the fact that the primitives are based on smart card commands (APDU commands) sent to the UICC application and hide any low level communication channels involved. The architecture also defines how this communication works and the hardware and software requirements the device and its OS must meet to be compatible (shown in blue in Fig. 3.1).

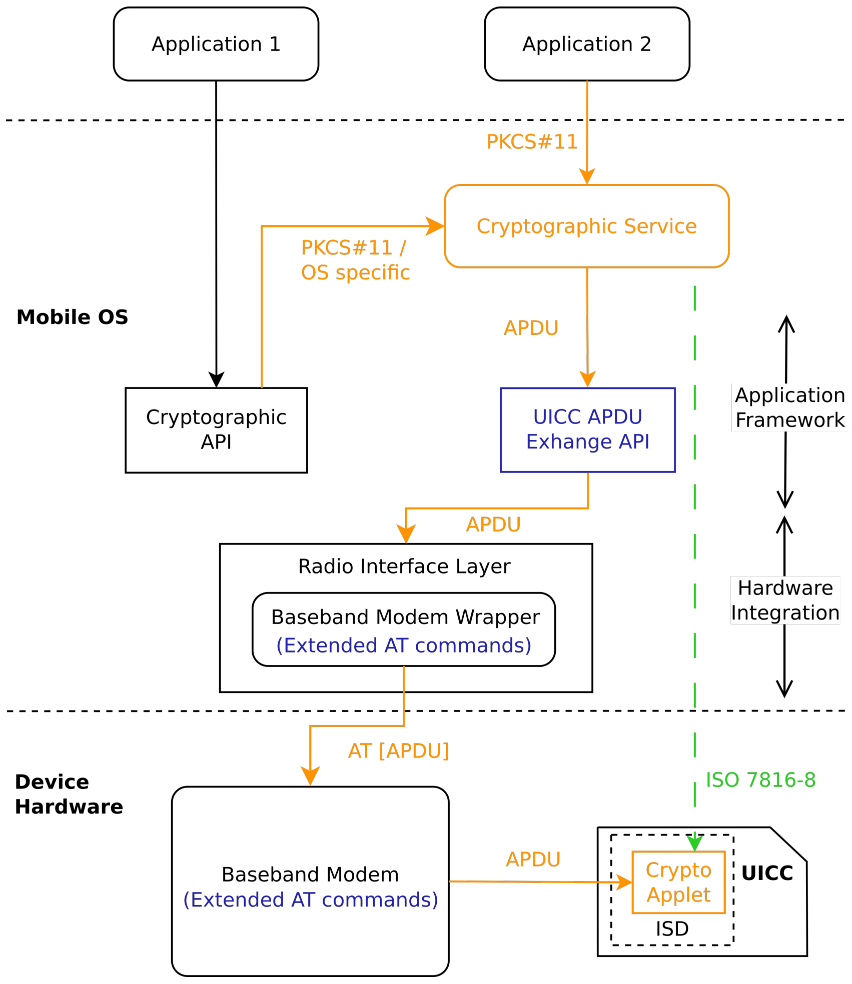


Figure 3.1: Cryptography offloading architecture.

3.2.1 Crypto applet

In order to execute hash and encryption operations in the UICC a cryptographic application that implements the proper commands, already specified at ISO 7816-8 [12], should be installed on the card. The multi-application nature of the smart cards currently used as UICC (see section 2.4.1) is consistent with this idea. ISO 7816-8 provides a basic standard interface that only includes key generation, signature, encryption and decryption, as well as hash calculations. Since it does not support exporting secrets even in encrypted form or any high level operation that combines multiple keys, we believe its design is not vulnerable to the most common API-level attacks, such as the ones presented in [45]. Sensitive key operations, for example encryption, should always be protected by user verification methods (such as PINs).

Section 2.4.1 explains how mobile network operators have exclusive full control over card life cycle and content management of the UICC through the ISD, since they are by definition the issuer. This makes it possible for them to install the intended application on the card. The UICC representation in Fig. 3.1 shows our cryptographic application as a Java Card Applet called 'Crypto Applet' installed under the ISD. Installation could be performed during the initialization and personalization phase, before the card enters `SECURED` state and is ready to be issued to the customer. In the case of production cards, those already issued, the application could still be installed using the OTA interface. Notice that we are making an important assumption here, namely, that operators agree to host that application on their cards as a benefit to their clients. On the other hand, mobile network standards already foresee the presence of a set of applications related to multimedia services and payments so it is not unlikely that a security application could became part of them, perhaps as an evolution of the USSM module mentioned in section 2.4.

3.2.2 Communication channel

At mobile device level we identified hardware as well as OS requirements. These are highlighted in blue in Fig. 3.1. The diagram shows that communication with the UICC is achieved through the baseband modem which is part of the hardware of a phone (the only possible alternative, see section 2.4.3). Our first requirement is that this component's hardware, drivers and OS integration wrappers should support extended AT commands (see section 2.4.3), so that it is possible to send custom APDUs to the card embedded through that channel. On top of this, since direct Radio Interface Layer access is not usually available for applications, our second requirement is that mobile OS and its application framework should provide access to the functionality enabled by those AT commands through a higher level API. As covered in section 2.5.5 a theoretically cross platform alternative for this interface is found in SIMAlliance/GlobalPlatform Open Mobile API [65]. However, since it is only present in select Android models from some manufacturers, OS specific APIs (such as Android's Telephony Manager primitives) prove to be more cross device compliant and are our preferred alternative. Since Android 9 [2] released a few months ago, an Open Mobile API implementation is part of the OS and is probably the best choice from now on, as it meets both criteria.

The two requirements explained above guarantee that a channel to send APDU commands to the UICC application (represented by a green arrow in Fig. 3.1) can be established seamlessly from application space, hiding the complexity of the hardware and software that compose the phone.

3.2.3 Smartphone module

The goal of this module (presented as 'Cryptographic Service' in Fig. 3.1) is making cryptographic services based on the Crypto Applet operations available to applications through high level interfaces detaching them from smart card and communication channel specifics. This component takes advantage of the communication channel explained in the previous section (green arrow in the diagram) to send ISO 7816-8 cryptography related commands to the applet upon request of cryptographic services by applications through the APIs exposed. The module may expose its services to applications directly through its own API (see application 2 in Fig. 3.1) or integrate with the OS application framework so that applications may access its services through the framework's cryptographic API (see application 2 in Fig. 3.1). A good alternative for the high level interface exposed might be PKCS#11 standard, a platform-independent API. Mobile operating systems typically define a provider based cryptography architecture and standard application framework APIs for cryptographic operations, where specific implementations must comply with predefined requirements to achieve integration. PKCS#11 is usually one of the integration options in most platforms, so it covers both OS integration as well as direct API exposure scenarios. An alternative is achieving integration by implementing OS framework specific provider interfaces. Note that despite the interface chosen, the actual operations provided should be limited to acting as proxys to ISO 7816-8 basic commands. For example, PKCS#11 provides a wide interface aimed at supporting a variety of cryptographic devices including smart cards and Hardware Security Modules (HSMs). While PKCS#11 may be vulnerable to API-level attacks in HSMs that provide key wrapping commands (see [48]) it is safe for standard ISO 7816 cryptographic smart cards that do not support such instructions in their PKCS#11 drivers.

Lower level integration at purely OS level might also prove to be a good choice. For example in Android, integration as a Keymaster device also exposes a keystore as a cryptographic provider in the cryptography framework. Such implementation would be in the HAL, a couple layers below the application framework (see section 2.5). To achieve a communication channel through AT commands, the Keymaster implementation would have to access the RIL directly through its HAL interfaces where primitives to send APDU commands to the UICC are also available. We will discuss Keymaster integration in further detail in Chapter 4.

3.3 Prototype

In order to run experiments on a real phone and carry out a performance evaluation we developed a prototype under Android OS, the most widely used mobile operating system, whose open source ecosystem and relative manufacturer detachment gives us the flexibility needed for this scenario. The main objective of the experiments is both to verify that it is possible to use the UICC as a Secure Element (SE) in a smartphone and to learn about its performance compared to OS default implementations.

3.3.1 Architecture

This prototype includes an implementation of all the components of our proposed architecture. However, it is not considered a full one because the smartphone module does not expose a high level interface to detach itself from applications and in turn is integrated as part of the application that uses its services. We understood that achieving compliance with an interface such as PKCS#11 was no relevant to the objectives of the prototype and the cost implied was not justified. Another simplification is that each component only implements the operations and algorithms required for the experimentation scenario. Fig. 3.2 depicts the prototype's general architecture, where black represents standard hardware and software components of an Android smartphone and orange identifies the implemented modules and communication flows introduced by our customizations. Inside our test SIM Card a reference implementation of the Crypto Applet defined by our architecture is provided. The smartphone used for the experiments fulfills the requirements defined to establish a communication channel to the UICC from application space. An Android system application called 'Crypto Test' implements the functionality of the smartphone module and also integrates with Android cryptography APIs to provide testing features for both our solution and the system's default providers. The following sections provide further details of each hardware and software component.

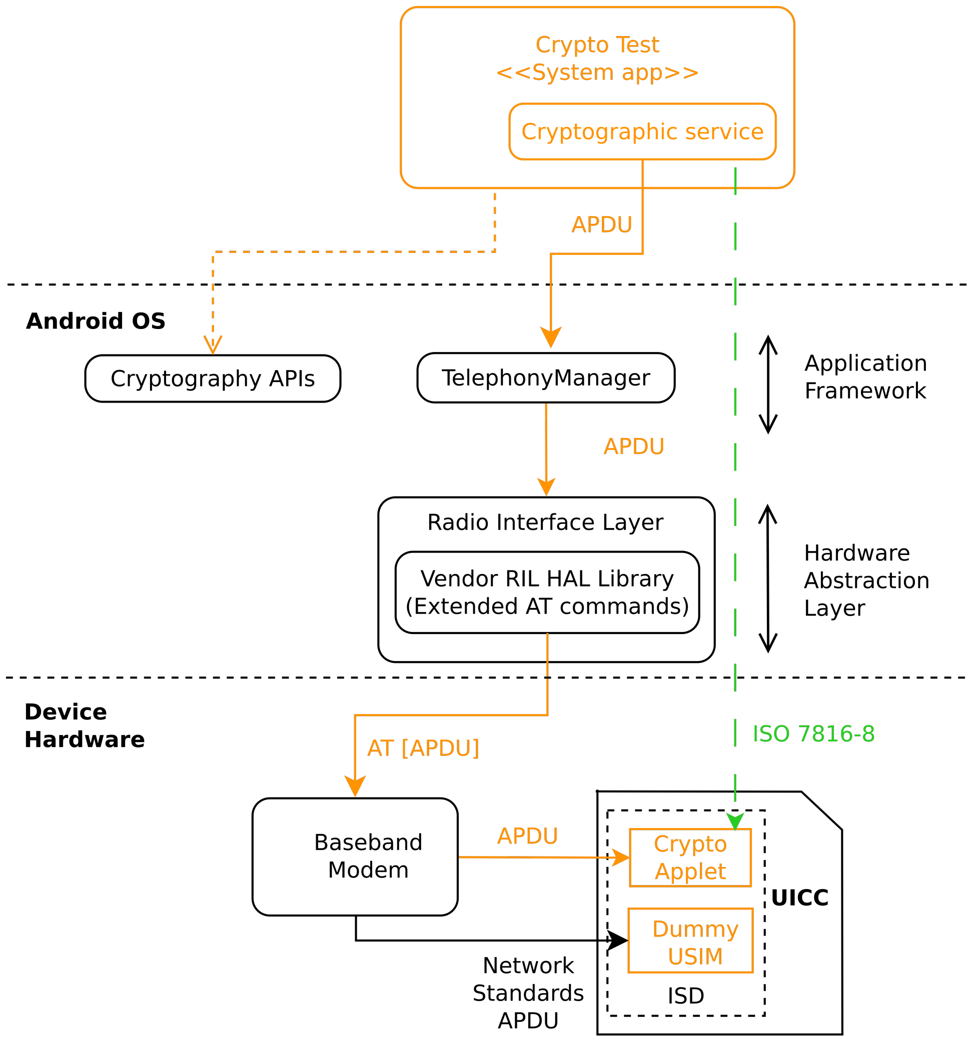


Figure 3.2: Cryptography offloading prototype under Android.

3.3.2 UICC

The first step in developing our prototype was to create a UICC capable of exposing general purpose cryptographic services. As standard USIM cards do not include that functionality, our architecture features a Java Card(JC) application that provides those services. This means the reuse or development of such application was required. Also, since UICCs made available on the market by network operators are closed cards which would not allow the installation of such application, an alternative was required. It is important to note that, as expected, development cards are not available from operators either as they are usually not willing to reveal details about their network implementation. We also searched for development USIM cards offered by manufacturers or suppliers, but found no options there either. Our best alternative was to get a generic development JC card and turn it into a SIM card by also installing a USIM application compliant enough with network standards to be acknowledged as such by a smartphone. In the following subsections we provide details about hardware selection and the development of the two required applications.

Hardware

Choosing the right smart cards among the samples available in the market was instrumental to the success of the prototype and experiments. We understand there are three key factors to take into account in this type of decision: compatibility, functionality and performance. Since the cards were to be used as SIM cards in a standard phone they must be compatible with electrical specifications and other physical characteristics defined for the UICC in [36]. At an OS and software level we needed multi-application smart cards based on GlobalPlatform and Java Card. On top of this, according to the design of our experiments (see section 3.4) the cards also needed to provide hardware support for the most popular symmetric and asymmetric cryptography ciphers, as well as digest algorithms in its Java Card implementation. All of these aspects allowed us to narrow our search by checking card specifications.

The final sorting criteria between the compliant candidates was performance. For this task we took advantage of the results published by the JCAlgTest tool [13], which is devoted to automatic testing of cryptographic smart cards running the Java Card platform. This tool gathers and visualizes information about card's hardware, supported cryptographic algorithms, and performance in various settings. The results are contributed by the tool designers and the community, creating the largest publicly available database with more than 50 different smart cards. After comparing the compatible card samples available on the market we sorted them as listed in Table 3.1, where the ones that presented the best combinations of response times for the algorithms required by the experiments are at the top. Although we were able to acquire samples for three of the first four alternatives, only two of them showed practical compatibility with our experimentation scenario. The other was not accepted by our test smartphone at a hardware level without providing much feedback due to scarce debug information available from proprietary baseband modem firmware and drivers.

Table 3.1: Cryptography performance comparison of Java Cards. Extract from [13].

	AES 256 (256B)	**SHA-2 (256B)**		**RSA 2048**	
		SHA-256	SHA-512	Pub	Priv
	(ms)	(ms)	(ms)	(ms)	(ms)
JavaCOS A40	3.63	35.13	118.37	7.98	147.04
JC30M48CR	3.60	27.0	146.66	8.49	155.55
NXP J2D081	7.66	21.18	n/a	17.65	594.93
Sm@rtCafé E. 6.0 80K	22.42	39.07	57.14	13.08	469.21
NXP J3A080	23.2	69.32	n/a	31.5	645.58
Sm@rtCafé 3.2 72K	17.1	114.26	n/a	28.53	2659.69

The first card used for our cryptographic SIM implementation was a Sm@rtCafé Expert 6.0 80K by Giesecke & Devrient. These are ISO 7816, Java Card 3.0.1 Classic and GlobalPlatform 2.1.1 compliant smart cards based on NXP P5CC081 [17] chips running Sm@rtCafé Expert 6.0 Operating System. They provide implementations for the most common cryptographic algorithms, including up to 2048 bit RSA, AES 256 bit, DSA up to 1024 bit, Triple-DES 3-key, ECDSA up to 256 bit, ECDH up to 256 bit, SHA-512 and Random Number Generator according to NIST SP 800-90 [44]. P5CC081 chips include in their design Triple DES, AES and Public Key (RSA, ECC) hardware co-processors to enhance the performance of these operations (Triple

DES in less than 40μs) with reduced power consumption in all supported voltage classes. They support 1.62V to 5.5V extended operating voltage range for class C, B and A and optional extended class B operation mode (2.2V to 5.5V targeted for battery supplied applications).

The second card model tested was a JC30M48CR by JavaCardOS [14] with 48k EEPROM. These are ISO 7816, Java Card 3.0.4 Classic and GlobalPlatform 2.1.1 compliant smart cards based on Infineon chips running JavaCardOS Operating System. They also provide implementations for the most common cryptographic algorithms, including MD5, SHA-1/SHA-224/SHA-256/SHA-384/SHA-512, DES, AES, RSA and ECDSA. No more details are available for this model from their manufacturer. Although not stated, we are confident they use hardware co-processors in their cryptography implementations, because of their great performance. We also tested them to verify they supported the algorithms selected in section 3.4.

The original full sized cards were cut to a Micro-SIM (3FF) size to fit the smartphone in both cases. The development environment chosen for coding, testing and delivering the JC Applets to the cards consists of Netbeans IDE with JC specific plugins, Oracle JC SDK tools and emulators, GlobalPlatformPro shell [9] to manage card contents, a PC/SC compatible contact smart card reader and a Micro-SIM to full size (FF) adapter.

These cards are compatible with electrical specifications and other physical characteristics defined for the UICC in [36]. We used GP APIs to change the original historical bytes to announce typical SIM capabilities. We were also required to disable EMV key diversification, which was active by default in the Sm@rtCafé Expert 6.0 80K, in order to make it more easily compatible with GP applet installation tools using default keys. We accomplished this using the 'unlock' feature of the GlobalPlatformPro tool.

Since neither card support an ISO 7816-4 file system natively (not a common feature in development cards), we decided to implement file related commands (such as `SELECT FILE`, `READ BINARY` or `READ RECORD`) as part of the Dummy applet discussed later. As the baseband modem expects native support and normally starts sessions by reading the card's MF (the root directory) right after receiving the ATR (Answer To Reset) and continues to read other files later, our applet was required to process every command sent to the card during the session. In GP that is accomplished by setting it as the default applet in the basic communication channel upon install. To guarantee that file system support is operative during the experiments and simplify the prototype we decided to merge both, SIM and Cryptography related commands in a single application for this particular implementation.

Dummy USIM

In order for the card to be accepted by the baseband modem and rendered usable by Android's framework, it needed to behave as a SIM card as specified at [36]. Since open USIM applet implementations were not found, we decided to develop what we call a *Dummy USIM* applet, which consists of an application capable of giving valid answers to a minimal set of file system queries and other standard commands (such as PIN verification). That minimal set can be initially determined by general mandatory specifications, but it can also be deduced by using reverse engineering to observe the behavior of the smartphone selected for the prototype. A real production UICC from a local operator was used in order to help us quickly assemble suitable values for standard records and binary files. All network operator and client specific identifiers (such as SPN, MSISDN or IMSI) were replaced by proper random values, as we did not intend to clone the card or get any form of access to the network. For our purposes it is convenient that the smartphone determines that the UICC is operative but no compatible network is found, therefore a connection negotiation is never initiated.

We coded our applet in an incremental fashion. We started by comparing the answers of its initial version and the production UICC to the commands sent by two desktop SIM card management tools called *SIM Manager* [26] and *SIM Card Manager* [27]. These tools are meant to explore and edit user information stored on the card, for example address-book entries or SMS archive. They usually check if the card is valid and read its standard file system entries and records. By making use of a PC/SC APDU sniffer called *Smartcard Sniffer* [29] we were able to analyze bit by bit the commands sent to the card and the responses produced in each case. After several iterations and fixes our *Dummy USIM* application proved to be compliant with those applications.

The next step was to test the applet on the phone. The Android Debug Bridge tool allows us to access mobile OS logging information through a USB connection to a PC. Since baseband modem manufacturers save debug capabilities for themselves, we were not able to obtain details whenever the modem stopped communicating with the card and just marked it invalid with a generic error code. Instead, we implemented a set of debug commands in the USIM applet which helped us record sessions between phone and UICC. After each failure we dumped the session trace saved on the card to a PC, replayed the commands sent by the modem on the production card, analyzed both outputs and installed a new version of the applet. Several iterations later, we were able to confirm that our dummy applet accomplished its purpose by verifying that Android's Telephony Manager API reported the card in `READY` state.

The final version of our applet supports PIN verification, file selection, content retrieval and general status commands. It serves over fifty GSM and UMTS standard elementary files including their binary contents, records contents and protection level.

Crypto Applet

This component of the prototype implements a subset of the standard cryptographic service commands defined in ISO 7816-8. Since one of the goals is to run experiments on the performance of digest as well as symmetric an asymmetric cipher algorithms, the subset of commands was chosen accordingly. We implemented the `HASH` and `ENCIPHERMENT` variations of the `PERFORM SECURITY OPERATION` command to execute the calculations mentioned earlier and the `MANAGE SECURITY ENVIRONMENT` to select keys, algorithms and their parameters. The specific algorithms supported are limited to those chosen for the experiments, namely AES-256, RSA-2048 and SHA-512, which are discussed in section 3.4.

As explained, we preferred to merge both JC applets into one for their installation on the card and phone interaction. Nevertheless, the implementations of the commands mentioned in this section were encapsulated in a separate module. This module's code makes heavy use of JC's Security and Cryptography Framework API which in its turn relies on card specific providers that are based on the device's cryptography optimized hardware components and architecture. This allows us to take advantage of the selected cards' full potential in terms of performance.

3.3.3 Smartphone

The other half of our prototype can be divided in two main steps. First, finding an Android smartphone whose base hardware and software fulfilled the two requirements established by our architecture. Second, developing the Cryptographic Service module under Android's application framework and implementing a test application that allowed us to run the experiments explained in section 3.4. The following sections explain each step in detail.

Base hardware and software

Smartphone selection was another key element to the success of the prototype. The selection criteria included the following two requirements:

- The phone's baseband modem's hardware and drivers are required to support extended AT commands.
- OS is required to provide high level APIs to send APDU commands to the UICC embedded inside AT commands.

In Android's radio architecture (see Figure 3.3) communication with the baseband modem is only available for applications and system services through the application framework, which in turn can only interact with the modem indirectly through the Radio Interface Layer (RIL) daemon (rild). This daemon is the one that actually can access the hardware using a RIL Hardware Abstraction Layer(HAL) library provided by the manufacturer, that serves as a wrapper for the proprietary interface provided by the baseband. To fulfill the first requirement, the phone's modem must expose extended AT commands through this HAL library. Of course, it is very hard to find this type of information in the public specifications of smartphones, which focus on higher level features. Fortunately, the SEEK for Android project documentation provides a (today unmaintained) list of devices [16] that were tested successfully with Open Mobile API (see section 2.5.5) support and thus also expose extended AT commands via the RIL HAL library, meeting the first requirement. The report found in [80] provides a similar list of more recent devices with the same properties.

As for the second requirement, according to [49], [16] and [80] some Android builds from major vendors (including Samsung and Sony) provide an implementation of Open Mobile API (see section 2.5.5) in their factory images, and as discussed in section 2.5.5, Android can also be patched to add support for it with an open source implementation. As an alternative, from API level 21 (Android 5.0) onwards, Android's framework provides primitives to send custom APDU commands to the UICC through the TelephonyManager [4]. These primitives require that the calling application has carrier privileges, which means it must be signed by a key that matches the carrier certificate in the UICC. A second option is declaring the `MODIFY_PHONE_STATE` permission which is only available for System applications. In other words only operators or phone manufacturers can use this API in their code. As stated in our architecture description, this type of cross device standard OS API was our preferred option for the Android versions available at implementation time.

Most of the devices on SEEK's list, including all the development ones, only supported older Android versions and could not be upgraded to API level 21. Only two commercial models remained safe candidates that met our requirements, namely Samsung Galaxy S5 (SM-G900M) and Samsung Galaxy S6 (SM-G920I). The list provided in [80], however, includes several other valid commercial candidates from different manufacturers. Table 3.2 shows all the smartphone models found to be compatible.

The smartphone used for the Android prototype is a Samsung Galaxy S5 (SM-G900M) with Android 6.0.1 OS (API level 23). It is based on the System on a Chip (SoC) Qualcomm MSM8974AC Snapdragon 801, which includes 4 Qualcomm® Krait^{TM} 400 CPU cores, a Qualcomm® Gobi^{TM} 4G LTE Advanced modem and several other common features (a GPU, WiFi, Bluetooth, GPS, NFC, Camera,etc.). It is equipped with 2GB of RAM, 16GB of internal memory, a 5.1 inches display and a removable Li-Ion 2800 mAh battery.

Cryptographic service and Crypto Test application

Once we had a phone that met the requirements to establish a communication channel from application space to the UICC, the next step was to provide an implementation of the smartphone module to access the services provided by the crypto applet from an Android application. Since the phone used for the prototype supported the required Android version, our preferred API

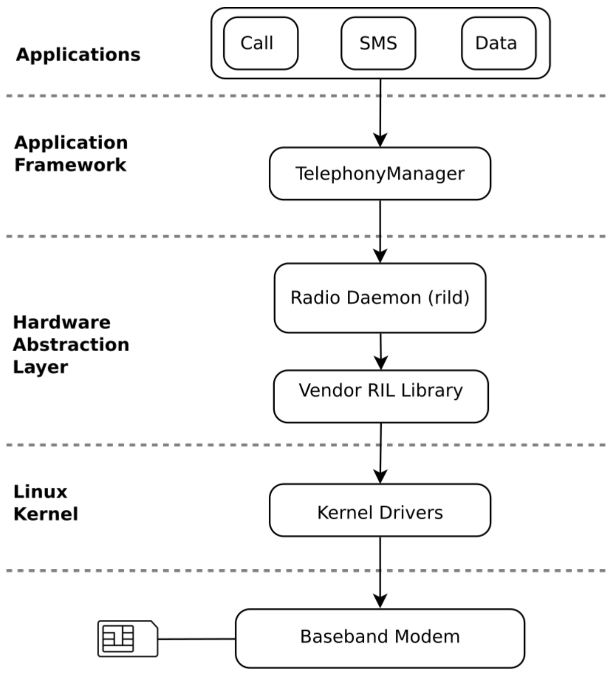


Figure 3.3: Android radio architecture [49].

Table 3.2: Smartphones with support for Open Mobile API and Android API level 21+.

Smartphone Model	**Supported API level**
Samsung Galaxy S4 (GT-i9500)	22
Samsung Galaxy S5 (SM-G900M)	23
Samsung Galaxy S6 (SM-G920I)	24
HTC One (M8)	23
Huawei Ascend P7	22
Huawei P8 Lite	22
LG nexus 4	22
LG nexus 5	23
Motorola Nexus 6	22
Sony Xperia M2 Aqua	22
Sony Xperia Z3 compact	23

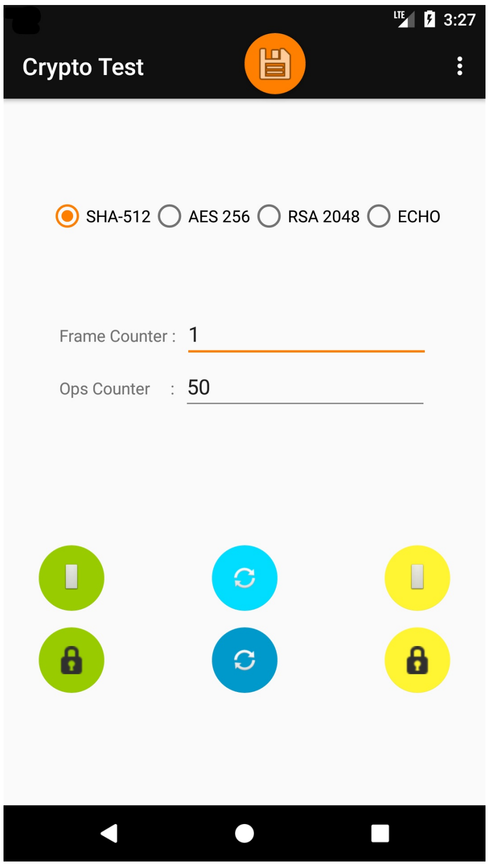
choice to send APDU commands to the UICC was the one provided by the TelephonyManager class. As explained earlier, these primitives require special privileges and one of the alternatives to obtain them is declaring the `MODIFY_PHONE_STATE` permission which is only available for System applications. Applications of this type reside under the *system* partition, which is the OS read-only image distributed by the manufacturer, but are otherwise regular Android applications. The *system* partition is not placed on actual read-only internal memory, its file system is just mounted in read-only mode and of course remounting it with write access requires root permissions. In order to deploy an apk file as a System application, rooting the smartphone was required. We preferred this alternative because it was easier to configure than an application signature that could be validated by making further customizations to our test UICC. We chose a minimally invasive rooting procedure [6] that installed a *su* command, but preserved the rest of the factory system partition, as well as the boot and recovery ones.

Since the Cryptographic service was already required to be coded inside a System application, we implemented it as part of the Crypto test application. This simplified the development phase avoiding the need to comply with another standard interface and the extra technical challenges of inter-application communication, while still meeting the proof-of-concept and experimentation purposes of the prototype. Interaction with the UICC through the TelephonyManager was encapsulated in a series of primitives of an internal controller. Cryptographic operations in the main processor were implemented using the framework's standard APIs with its default provider (*AndroidOpenSSL*) and the *AndroidKeystore* provider, inside an analogous set of primitives. The Crypto Test application itself, as can be observed in Fig. 3.4, provides the required functionality for a user to run custom experiments on both the card and the main processor by letting him choose the target, algorithm, operation count, as well as other parameters. After the tests conclude the application reports execution time and battery consumption statistics. In order to obtain the latter the application registers to receive battery level change events from the framework's *BatteryManager* interface.

The development environment chosen for coding, testing and delivering the application consists of Android Developer Studio, including SDKs and virtual devices compatible with API level 23 and the Android Debug Bridge (adb) shell to modify file systems in physical and virtual devices. UICC related features were tested directly in the phone environment since the vanilla Android emulator only supports primitive UICC simulation which does not include the customization required. We evaluated patching it with SEEK's Emulator Extension [25] to provide full SIM access through the host PC/SC system, but it required downloading and compiling Android's source tree which we found is a complex and resource demanding task.

3.4 Performance Evaluation

We have argued at the beginning of this chapter that our work has the potential to help improve mobile security. However, other important sides of mobile experience, the ones which usually receive most attention from average users, are performance and battery life. In order to find out how does our architecture impact some aspects of user experience, we have run a series of tests to measure performance (in terms of response times) and battery consumption of the execution of the most common digest and encryption algorithms (including symmetric and asymmetric cryptography). Our goal is to compare the default alternative of executing cryptographic operations in the main chip of the phone with our SIM based cryptographic services. We repeated the same test cases executing cryptographic operations in both scenarios. We use Android's standard cryptography APIs without choosing a provider, so that the default cryptography provider (*AndroidOpenSSL* for Samsung devices) supplied by the manufacturer is used. We also test Android's hardware backed cryptography provider (*AndroidKeystore*). A discussion of the



[✓ SHA-512] AES 256 RSA 2048 ECHO

Frame Counter : 1

Ops Counter : 50

Figure 3.4: Screenshot of Crypto Test application.

results let us draw some conclusions.

3.4.1 Design

In [52] a performance evaluation of cryptographic primitives on smart cards and smartphones is presented. They evaluate the fundamental cryptographic primitives frequently used in advanced cryptographic constructions, such as user-centric attribute-based protocols and anonymous credential systems and conclude that only random number generation, hash functions (namely SHA-1 and SHA-2) and big-integer modular arithmetic operations (especially exponentiation) are needed for all studied protocols. Taking this and their wide deployment into account, we selected SHA-512 as the digest representative and RSA-2048 as modular exponentiation representative for our evaluation. For similar reasons, we also considered appropriate to add a symmetric cryptography representative in AES-256. Key length and block sizes were chosen as the longest supported by the smart cards used in the experiments. Asymmetric cryptography tests were run using private key encryption which is usually more expensive than using the public key and benefits the most from a Secure Element's protections. Input data and key values are randomly generated by each provider before executing the operation. For RSA-2048, we chose the Chinese Remainder Theorem (CRT) optimized implementation since it makes modular exponentiation about four times faster and Android's default provider is based on this alternative.

For each algorithm we coded in our Android application the required functionality to take multiple samples of response times and battery consumption of its execution in the scenarios to be compared. From the different cryptography providers usually available in Android, we included the default provider since it is probably the option used by most applications and also the provider based on Android's Keymaster (*AndroidKeystore*) since it is theoretically the most secure alternative provided in every Android phone and is potentially hardware backed or protected by a TEE. Samsung does not provide details about Keymaster implementations in its phones, but existing studies [47] [63] on some of their devices and other models using Qualcomm processors suggest that, in the architectures that support it, they use a Keymaster based on a TrustZone TEE. Debugging information from our test Samsung device confirms that it supports that feature.

Response time measurements only include the actual execution of the algorithm and the communication channel overhead for the SIM card. Input data lengths for SHA and AES are specified in terms of 240 bytes chunks. We decided to use this unit because it is the maximum common block aligned length that fits into an APDU payload. For each algorithm and provider we run response time tests for input sizes of 1 and 10 chunks(about 2.4KB). RSA input data is always 245 bytes, the maximum allowed for the key size used. Based on referenced works' experience and our own, as the standard deviation and variance were small proportionally to the order of the measurements, we decided not to take more than 30 samples of battery consumption for each operation.

3.4.2 Measurement methods

As [58] and [71] explain, there are different strategies to measure the remaining battery capacity of the device. This includes internal measurements based on battery information provided by the OS (such as Android's *BatteryManager* interface). Other strategies estimate hardware components power consumption using software. Finally, more complex external hardware based methods claim to provide the highest precision. Android's *BatteryManager* interface method was compared to external hardware techniques in [58] and [71]. Their results show that Android's interface provides acceptable and in some cases almost identical accuracy. For that reason we chose this interface despite its lack of precision, as it only informs changes of exactly 1 battery unit, which in our device (as in most) is equal to 1% of the full capacity. To determine the fraction consumed by single operations, we count how many operations can be executed by consuming 1% and infer from that. Also to avoid imprecision in initial battery measurements and provide equal conditions for all scenarios, we start all our test runs with full battery charge.

Another important consideration, especially when measuring energy consumption is keeping the device awake. Measurements of this type require longer tasks and their execution must be as continuous as possible to minimize distortions that affect the precision of results (for example idle battery consumption or device internal state changes). To avoid draining the battery, an Android device that is left idle (no user interaction) quickly falls asleep. However, there are times when an application (such as ours) needs to keep the screen or the CPU awake to complete some work. In our case we only require the CPU to keep running and we actually prefer to turn the screen off as soon as a test execution starts since it is a non-negligible power consumer and our goal is to achieve maximum isolation for our tasks consumption. To keep the CPU running in order to complete a task before the device goes to sleep, Android's *PowerManager* System service provides a feature called wake locks. By grabbing a 'partial wake lock' the application can control the power state of the device and keep the CPU running while the screen is off. Once the task is finished the application should release the lock and its claim to the CPU. We took advantage of this feature in our test application. However, we observed in Samsung SM-G900M that after approximately 90 minutes the device entered complete sleep mode despite of our application's lock. For this reason any samples recorded after that period were discarded.

3.4.3 Results

Here we present and analyze the evaluation results for each metric (response times and battery consumption). Response time penalization while executing operations in the UICC is detected for every algorithm in both cards in comparison with both Android providers. Battery consumption results, however, vary according to the combination of providers compared and offloading operations to the SIM card proves to save energy in some scenarios. Results for RSA 2048 algorithm are in general encouraging and make it the best candidate to gain benefits from our approach.

Response Times

Table 3.3 presents the average results for response time tests. In tests for 240 bytes data, average response times for smartphone operations in the *AndroidOpenSSL* provider take less than 1 ms for both AES-256 and SHA-512, while they are more than a hundred times slower in the SIM cards. Android's secure keystore based provider is slower than *AndroidOpenSSL* for AES encryption taking in average about 8 ms for each operation, but is still much faster than operations in the SIM cards. The voltage class negotiated by the phone which affects transfer rates plus overall communication channel overhead through the baseband modem are probably negative factors here. We tried to get feedback from the modem or enforce a higher voltage class without success. In tests for 2400 bytes all providers show non linear increase in response times, but smartphone operations take just about 60% longer in the worst case, while SIM penalization can be as high as 700%. This reinforces the theory of a communication overhead, as 10 operations are required for these calculations.

Table 3.3: Average response times for Samsung SM-G900M experiments.

	OpenSSL (ms)	**TEE Keystore** (ms)	**Sm@rtCafé E. 6.0** (ms)	**JC30M48CR** (ms)
SHA-512 (240B)	0.780	n/a	131.638	242.689
SHA-512 (2400B)	0.854	n/a	1060.348	1563.568
AES-256 (240B)	0.830	8.139	182.408	168.104
AES-256 (2400B)	1.408	26.604	1130.442	923.933
RSA-2048 (245B)	54.390	181.754	641.292	315.669

Although penalization due to communication channel overhead was one of the challenges early identified for the proposed cryptographic service, we did not expect response times of this order. We had theoretical [79] and empirical [13] data that pointed at much smaller response times (in most cases under 60 milliseconds) for every operation. Table 3.4 shows our own results of running tests on the Sm@rtCafé Expert 6.0 and JC30M48CR cards from a Java Standard Edition Desktop application while connected to a PC through a PC/SC [23] compatible reader. Response times for cryptographic operations in this scenario are slower than the ones reported by the JCAlgTest Tool [13], but still much faster than the ones observed in the smartphone. We also took samples in the PC scenario for 'NOP' (returns without doing anything) and 'ECHO' (responds the same payload received) instructions we made available in the Crypto Applet for the purpose of this test. The response time difference observed for the *ECHO* and *NOP* operations suggests that data transfer from and to the card consumes a considerable part of the time taken. We repeated the samples of the ECHO instruction in the smartphone scenario and it needs an average of 107.7 ms to complete execution for a 256 bytes payload in Sm@rtCafé Expert 6.0. This is more than two times what we observed on a PC.

RSA-2048 calculations, as expected, take longer than digests and symmetric cryptography in every provider. In this case, operations also take longer in the SIM cards, as observed for the other algorithms. However, the difference is proportionally smaller for this type of calculations. For example response times observed for JC30M48CR card are of a very similar order to results for *AndroidKeystore*. For these operations, the fixed overhead is much smaller than the actual calculation time, so it has a smaller weight in response times.

Table 3.4: Test UICCs average response times in PC environment.

	Sm@rtCafé E. 6.0 80K (ms)	**JC30M48CR** (ms)
SHA-512 (240B)	79.605	191.638
AES-256 (240B)	78.214	63.900
RSA-2048 (245B)	523.599	219.845
NOP (0B)	4.489	4.279
ECHO (255B)	52.894	63.545

Battery Consumption

Table 3.5 presents initial average battery consumption test results. However, since SIM operations are slower and tests take more time, there is a higher component of idle phone battery consumption. Each SIM sample takes between 18 and 30 minutes to complete, while phone samples take approximately between 2 and 7. We took measurements of idle battery consumption, which point that 0,00466% is used per minute, without executing any calculations. That means in every battery consumption sample from SIM card operations, up to 0,13932% was due to idle phone power usage. Table 3.6 shows adjusted values. The reduction observed is as expected proportionally higher for the UICC cryptographic service.

Phone calculations prove to be more energy efficient in both providers for AES 256 and SHA-512. The difference in consumption can be as high as 40 times more for these algorithms. Things are very different for RSA calculations which show consumption of a very similar order in the four scenarios. While operations are most efficient in *AndroidOpenSSL*, the difference with JC30M48CR is quite small. Moreover, both UICCs prove to consume less energy for this particular algorithm when compared to the TEE based Keystore provider.

Table 3.5: Average battery consumption for
Samsung SM-G900M experiments.

	OpenSSL (%)	**TEE Keystore** (%)	**Sm@rtCafé E. 6.0** (%)	**JC30M48CR** (%)
SHA-512	5.792E-06	n/a	104.481E-06	242.267E-06
AES-256	6.221E-06	20.849E-06	209.385E-06	230.542E-06
RSA-2048	1.959E-04	7.211E-04	6.224E-04	3.458E-04

3.4.4 Conclusions

We observed a proportionally high penalization in response time and battery consumption for digest calculations and symmetric encryption in the UICC. This penalization grows proportionally

Table 3.6: Adjusted average battery consumption
for Samsung SM-G900M experiments.

	OpenSSL (%)	**TEE Keystore** (%)	**Sm@rtCafé E. 6.0** (%)	**JC30M48CR** (%)
SHA-512	5.648E-06	n/a	89.924E-06	221.551E-06
AES-256	6.083E-06	20.177E-06	189.153E-06	214.033E-06
RSA-2048	1.938E-04	7.089E-04	5.644E-04	3.171E-04

to input data size, making it perceivable, having an impact on battery cycles for large data streams and thus input size becoming a key variable in any tradeoff evaluation. For example, a single text piece encrypted using AES-256 by a messaging application (such as WhatsApp) could fit in the payload of an operation and thus present response times and energy consumption according to tables 3.3 and 3.6, which would not show a perceivable difference for the end user. However, a 1MB multimedia file transferred through the same application would take more than 6 minutes (consuming about 1% of battery) to be encrypted in the UICC, about 10 seconds in the TEE (consuming 0,0881% of battery), and less than 1 second in OpenSSL (consuming 0,0266% of battery). Here, response time and battery penalization are widely perceived by the user who would probably not be willing to accept it. The same applies to larger files. In contrast, encrypting a 20KB secure vault for a password manager application would take only about 8 seconds (consuming 0,018 % battery) in the UICC, more than the TEE and OpenSSL which take a few milliseconds and also perceivable, but a price the user might be willing to pay.

Asymmetric encryption on the other hand can be more energy efficient in the UICC when security requirements are higher and, although slower in the SIM card, presents response time results of a similar order in both scenarios. Also, since it is by design applied over smaller data, penalization is unlikely to be perceivable by users or cause an impact in battery cycles. We believe that the tradeoff between security and overall performance for this algorithm might be seen as positive in most scenarios. Some typical use cases are signing documents (encryption of a hash) and ciphering small arrays of random bytes for challenge-response or key-exchange protocols in hybrid cryptosystems (SSH, TLS/SSL among others). These tasks require a single operation and thus present response times and energy consumption according to tables 3.3 and 3.6, which would not be perceivable by the user and even imply small energy savings in some cases.

Chapter 4

Discussion

We have proposed an architecture to deliver a cryptographic service based on the SIM card in smartphones, developed an Android prototype to verify it works and estimated its expected performance and power consumption. This chapter is devoted to discussing the potential and limitations of our contributions, how they compare with related work sharing the same objectives, strategies or techniques and how they can be applied to further help enhance security in Android.

4.1 Related Work

In this section we summarize the few related works found during our research and highlight the main differences with our proposal. We already explained in section 2.1.5 how research works using computation offloading with the purpose of enhancing security [73] [55] [54] or privacy [64] are scarce. They typically appear as statically imposed conditions or restrictions, but final decisions are based on resource usage and response times. Most of the few examples found present indirect approaches and actually focus on enhancing the performance of security solutions. Our technique on the other hand, is meant to be applied with security as its main objective and performance aspects as secondary ones or as restrictions.

There is a considerable amount of research work and specifications concerning smart cards and their relation to mobile network technologies, as well as cryptographic operations and key storage in mobile OS. Less common is to come across work that combines both, since network related functionality in smartphones is usually isolated even at a hardware level and custom access to the UICC was officially added to Android APIs only three years ago.

Elenkov discusses in a number of blog posts and his book [49] the architecture and some details of secure key storage on Android, including the possibility to use the UICC, as well as other optional hardware components, as a SE. He covers the need for extended AT commands support, but states Android does not provide a standard API to use the SIM card as a SE (which was true for available Android versions at his book's publishing date), mentioning SEEK patches as a possible alternative. The only application explained in detail in his posts is a SIM based password manager. While his work gave us a starting point, ours goes much further by finding such standard API, defining an architecture for a SIM cryptographic service, prototyping its components and measuring its performance.

As of writing this document, we found perhaps the most similar precedent in [74]. The authors propose an architecture that, just as the one presented here, uses the SIM card to provide cryptographic services. In their case the scope of the functions provided by the SIM card is not the entire mobile system, they concentrate on mobile financial service applications. They do propose an application inside the card to expose cryptographic services and a software module on the phone to expose them to mobile applications, but they do not explain how communication between both components works. Although they mention having implemented their designed architecture, they provide no details about such implementation (for example, it is unclear which mobile OS, hardware or emulators might have been used). Unlike our work, they did not perform any evaluations nor provide any experimental results.

Motivated by the security benefits brought by hardware based encryption of mobile data, [69] also intended to use the SIM card to offload cryptographic operations. In order to achieve this, they provide a smart card design where cryptographic processors for AES and RSA are added as well as secure storage areas for keys, in a way that it is not required for them to leave the device. Since such hardware components were already available and accessible through standard Java Card APIs on high end smart cards from multiple vendors long before the paper's publication, we find this level of customization unjustified. They also feature an original alternative to OS-card communication where the interface used is based on the standard commands to manage contact information. This approach has the advantage of avoiding to send APDU commands to the UICC that are not usually supported by devices (phones or modems) compliant with mobile network specifications and thus eliminates the need for extended AT commands. However, since it missuses a standard interface for an unrelated purpose it may be seen as a hack and its implementation in a SIM would imply the substitution of the actual contacts application by a proxy one. The architecture we propose is based on standard hardware and interfaces, providing only an encapsulated extension to them and making it more likely to be accepted by mobile network operators or even standardization organizations.

In [82] we found a study focused on energy consumption of cryptographic operations in mobile devices, that just like ours explores the convenience of offloading them to one of the possible form factors of a SE. By using specialized hardware to take voltage measurements and software analysis tools, they compare the performance (response time) and energy consumption in the execution of digest and encryption algorithms for different clock frequencies on a smart card connected through the microSD slot. The smart microSD, unlike the universally present UICC, is an optional hardware add-on for smart phones. They run experiments for AES, RSA, MD5 and SHA using different key lengths and operation modes. As expected, they conclude that at lower clock frequencies response times are increased (worse performance) and energy consumption is reduced. Their main result is they learn that as frequency reduces, performance penalty grows proportionally faster than energy savings and these tend to zero when execution time increases. It is worth mentioning that they used for the experiments a Java Card chip with an ARM SecurCore SC300 processor for which CPU clock reaches 25 MHz. The card includes cryptographic co-processors for DES, RSA, ECC, etc. This smart card is of a very similar type to the UICCs targeted by our work. Although their alternative reports similar response times for RSA 2048, our prototype performs better for hash and symmetric cryptography algorithms. It is also important to note that while their experimentation setup is composed with measurement tools connected to a PC, ours is completely based on a real battery-powered smartphone.

Finally, [38] presents a security architecture for a mobile payment solution integrating a TEE and the UICC in a TEE-enabled Mobile Phone. The goal of their work is not providing general purpose cryptographic services for the system, but some specific applications. They do provide a proof of concept prototype under Android 4.2.2 which includes a Java Card application on the card and a phone application which directly sends APDU commands (no abstraction modules). They resolve communication using Open Mobile API, the only alternative available in that OS version. No performance results are presented. The greatest advantage of this work is that they provide an architecture and implementation of integration between TEE and UICC. Although TEEs have evolved since, it presents a valid alternative for secure communication between a trusted application and a JC applet reusing GlobalPlatforms's secure channel capabilities.

4.2 Applications

In this section we discuss how Android's security could benefit the most from the proposed architecture. We focus on specific services or features where current OS provided alternatives are optional, vulnerable or present limitations when compared to iOS security.

4.2.1 Keystore Service

The easiest to achieve, yet still an important enhancement, would be a physically tamper resistant hardware-backed keystore service. Android's framework already includes a Keystore service API which provides a basic but adequate set of cryptographic primitives [3]. Its architecture separates the API from the underlying implementation. If no device specific support is provided, the OS ships with a default software implementation [47] based on OpenSSL. The architecture anticipates the availability of different types of backends, including hardware based ones through the HAL. The Keymaster HAL is an OEM-provided, dynamically-loadable library used by the Keystore service to provide hardware-backed cryptographic services. To keep things secure, HAL implementations do not perform any sensitive operations in user space, or even in kernel space. Sensitive operations are delegated to a secure processor reached through some kernel interface. The availability of a TEE in a SoC, such as those based on ARM TrustZone, offers an opportunity for many Android smartphones to provide a hardware-backed implementation. In this scenario a trusted application inside the TEE would be in charge of key related operations. As pointed out by [47] and [63], even those options can present some security shortcomings by design. In [47] they find that the hardware-backed version of the Android Keystore service does offer device binding (i.e. keys cannot be exported from the device), but that would not prevent for them to be used by any attacker with root access. They state that this is not surprising, as it is a fundamental limitation of any secure storage service offered from the TrustZone's secure world to the insecure world. Some details are also provided which reveal that actual keystore files are stored inside encrypted blobs in application internal memory outside the TEE and provide background on possible attacks detected for specific TrustZone implementations that could compromise the master encryption key used to protect them. In [63] a cache side-channel attack technique against AES-256 in Samsung's TrustZone architecture is presented.

As Fig. 4.1 illustrates, by providing a Keymaster module implementation based on the communication channel that allows us to access the Crypto Applet, we could make available a solution with similar characteristics to the one provided by a TEE, but adding resistance to hardware attacks through physical protections and making it highly difficult to even use secret keys for an attacker in possession of the device. Our solution would also present higher execution isolation from the insecure world avoiding most common attack strategies since it uses a separate chip that does not share processor, cache, memory or storage when executing key related operations. Keys never actually leave the SE, not even in encrypted forms. Android 9, the OS most recent version released a few months ago, has introduced StrongBox [2]. Supported devices running API level 28 or higher installed can optionally have a StrongBox Keymaster, an implementation of the Keymaster HAL that resides in a hardware security module (such as an embedded secure element) containing its own CPU, secure storage, a true random-number generator and additional mechanisms to resist package tampering and unauthorized sideloading of applications. To support low-power StrongBox implementations, a subset of algorithms and key sizes are supported, including RSA 2048 and AES 256. While keys inside a TEE, the standard implementation of a Keymaster, were marked as 'hardware backed' in Android's Framework, keys inside a StrongBox are distinguished with a separate tag. This new feature not only shows that Android Project acknowledges the previous lack of Secure Element keystore solutions and hints at making their presence a standard, but also provides us with the ideal type of Keymaster to expose our cryptographic services.

The thesis work of one of [47]'s authors [46] makes a comparison between TEE and SE keystore solutions, where he points out the advantages already mentioned for the SE and an important advantage of the TEE in its ability to access peripherals and sensors through a trusted path (for example to securely obtain user consent). A combination of both would probably lead to a solution without any of the mentioned disadvantages and, as we will see shortly, has the potential to provide more advanced security features.

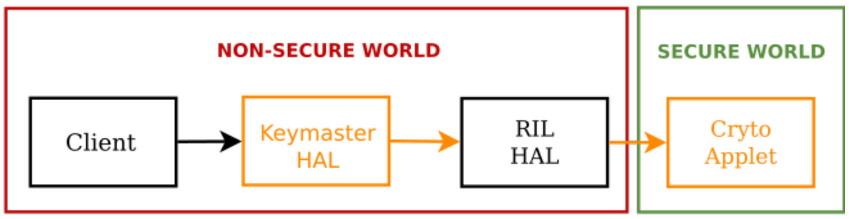


Figure 4.1: Android Keymaster based on UICC cryptographic services.

4.2.2 Secure Enclave

The other possible application for our architecture would be going a step further in the scenario presented for keystores and aim to a more complex component such as Apple's Secure Enclave Processor [56]. The Secure Enclave Processor (SEP) is a co-processor in the Apple A-series processors, which provides all cryptographic operations for Data Protection key management and maintains the integrity of Data Protection even if the kernel has been compromised. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers. The SEP runs a separate microkernel signed by Apple, verified as part of the iOS secure boot chain, and updated through a personalized software update process. When the device starts up, an ephemeral key is created, entangled with the device's UID, and used to encrypt and authenticate the Secure Enclave's portion of the device's memory space. Additionally, data saved to the file system by the Secure Enclave is encrypted with a key entangled with the UID and an antireplay counter. The device's unique ID (UID) and a device group ID (GID) are AES 256-bit keys fused (UID) or compiled (GID) into the Secure Enclave during manufacturing. No software or firmware can read them directly; they can see only the results of encryption or decryption operations performed by a cryptographic engine dedicated to the Secure Enclave. The SEP includes a true hardware random number generator as well.

The Secure Enclave is also responsible for processing fingerprint and face data from the Touch ID and Face ID sensors, determining if there is a match, and then enabling access or purchases on behalf of the user. The fingerprint sensor sends scans to the Secure Enclave through the application processor, which cannot read it because it is encrypted and authenticated with a session key that is negotiated using a shared key provisioned for each Touch ID sensor and its corresponding Secure Enclave at the factory. The shared key is strong, random, and different for every Touch ID sensor. Communication between the SEP and NFC controller's SE that holds payment applications works in the same way, but using a different factory provisioned shared key.

Apple has not revealed many low level details about their SEP implementation but, as their A-series SoCs are based on ARM architectures, many suggest it is likely based on an enhanced implementation of TrustZone. SEP is a TEE with shared memory and a trusted path to peripherals, which can be achieved using TrustZone hardware features. Secure world dedicated cryptographic units are also compliant with ARM architectures. UID and shared keys can be seen as manufacturer specific fuses with exclusive access by the secure world. Finally, secure boot chain is a basic feature of TrustZone. Secure world OS would be in charge of implementing and securing the mailbox communication system using hardware controllers isolation features. Using this strategy an Android full implementation of the Secure Enclave functionality might be provided for TrustZone enabled smart phones with the proper software modifications to the Trusty software as well. In that scenario UID and shared keys could be stored by trustlets or fused depending on each manufacturers choice.

A reverse engineering study presented at 2016 Black Hat Briefings [67] affirms the SEP is a dedicated hardware core with dedicated: crypto engine, random number generator, fuses, ROM, internal memory and IO lines to peripherals. It also suggests it has hardware anti-tamper and isolation features, especially in newer versions. Those kinds of hardware features are beyond the requirements of TrustZone or similar TEE hardware support.

Aside from dedicated IO lines our SE could provide the OS with the required dedicated processor, memory, and calculation units inside of a tamper resistant hardware component to achieve a similar security level. As stated before, a combination with the already mentioned TrustZone features would be required to fully emulate SEP's functionality. Figure 4.2 illustrates the components and interactions proposed to accomplish this. Integration would be achieved through a trustlet with exclusive access to operations with some of the keys stored in the SE. Following the approach in [38], communication between trustlet and applet might not be direct to sort any architecture access restrictions to the UICC form the trusted OS. The software module at mobile OS level (library or application) forwards traffic between both using the Trusty APIs. To protect this communication through the non secure world a secure channel is required. GlobalPlatform secure channel protocols enable an on-card entity such as an applet and an off-card entity such as a trustlet to mutually authenticate and establish cryptographic keys to protect integrity and confidentiality of subsequent communications across untrusted communication channels. With this architecture both secure worlds, TEE and SE, conform a larger one where Secure Enclave features could be implemented selectively taking advantage of the protections provided by each execution environment. For example, in this scenario user input for authentication (such as a PIN or a fingerprint capture) could travel through a secure path from the screen or sensor to the UICC, making it impossible for malicious software in the REE (even with root privileges) to intercept the provided proof. Subsequent cryptographic operations could only be initiated from inside the TEE through the secure channel.

Further elaboration on this possibility is left for future work.

4.3 Limitations

In this section we discuss the main disadvantages and limitations of our work.

Our architecture is highly dependent on mobile network operators and standardization organizations willingness to include the proposed general purpose cryptographic application in their card definitions. Since network standards already foresee optional applications to extend the multi-application environment of the UICC, our application exposes already standardized smart card commands and the services it provides could be beneficial system wide, we find this possibility viable. Mobile device manufacturers are also required to take a role in standardizing the use of extended commands so that our architecture could be more widely deployed.

Our cryptographic service does not provide device binding, because keys are actually only bound to the UICC. Although this may not be a problem in most use cases, since keys are securely stored and require user consent (in the form of a PIN), it should be considered in security assessments for specific applications. Also, every security feature based on our SE (keystore service, File or system encryption and even security enclave like modules) would create

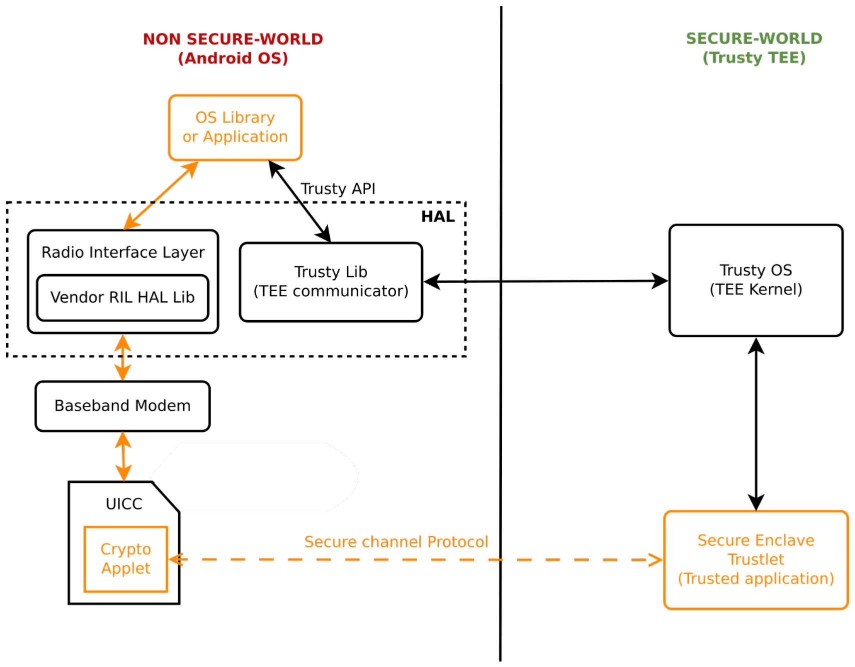


Figure 4.2: Secure Enclave in Android combining TEE and SE.

a new attachment between the phone and the current UICC. Changing the card (legitimate owner change, network technology upgrade and other scenarios) would require a consented pre-detachment procedure with the appropriate security controls. Definition of such procedure is left for future work.

Our solution relies on user verification methods (such as PINs) to prevent an attacker in possession of the device from using the keys stored in the SE. A poor implementation of the smartphone module presented in section 3.2 (or even authentication code running inside a TrustZone TEE) that handles PINs or patterns improperly, storing their values in memory or other hardware components of the main chip of the phone, might make it possible for them to be retrieved by an attacker with the appropriate hardware analysis equipment. Further specification of requirements for security testing of this module is left for future work.

Only system applications or regular ones signed by network operators can send commands to the UICC. If a malicious application from a remote attacker gains root privileges on the smartphone while it is in possession of its owner, it may replicate itself in the system partition, send commands to the UICC and use keys for which the user provides consent. Our basic architecture does not specify protections against unauthorized use of the UICC commands by this application. This is a hard problem to solve since the UICC has no access to the REE and cannot perform any checks over calling applications. Client authentication strategies would require presenting some kind of secret kept inside the smartphone module. Obfuscation techniques could be applied to make extraction of this secret harder but cannot certainly prevent it. The Secure Enclave architecture discussed earlier in this chapter averts this type of attack with an authenticated secure channel from the TEE, where the malicious application has no access.

Results presented in section 3.4 show some drawbacks of our prototype in terms of performance and power consumption. We believe these issues are likely related to communication through the baseband modem, but our efforts to further diagnose the cause or find an optimization have not paid off so far. Bandwidth and transfer rates are usually key variables, often more relevant than the target environment's processing power, which present limitations in offloading decisions. An advantage of our solution is that by not requiring network communication, transfer rates (although limited) are not variable and so they can be accurately predicted. Our next steps will include repeating the experiments with different smart cards and phone models to learn more about these problems and confirm this is purely a limitation of our architecture. Nevertheless, we are most likely facing a tradeoff between security and user experience that requires further assessment. For example, we found a study [75] that identifies cryptographic functionality in Android applications and concludes that for a set of security related ones only about 6% of the methods that compose application code are dedicated to cryptographic operations. Since SIM operations still take less than 200ms in some cases and 641ms at worst, we believe there are many scenarios where penalty might not be perceivable or would represent a small price users may be willing to pay in exchange for enhanced security. The same principle applies to battery consumption in SHA-512 and AES 256. Our future research will continue studying the actual impact of this penalization and how it can be reduced.

Chapter 5

Conclusions

We propose offloading cryptographic services to the SIM card in smartphones as an alternative computation offloading strategy that can have security as one of its main objectives and does not present the usual side effects of cloud computing based strategies. An architecture to apply this technique in smartphones is presented. Our approach makes available a tamper resistant hardware based cryptographic provider on every smartphone that allows custom interaction with the UICC, without further manufacturer hardware requirements. A prototype developed under Android OS demonstrates the feasibility of our solution. An evaluation of experimental results comparing our prototype with the default manufacturer provided software and hardware backed implementations of a real smartphone, shows some execution time limitations and a small battery consumption impact for most algorithms. For RSA-2048 we found that our approach can at the same time enhance security and save energy with a minor impact in response time. The type of performance benchmarking information provided is instrumental when profiling parts of a program in computation offloading.

5.1 Future Work

The research presented here covers our main objectives, however there is still much work to do at different levels on this subject. That includes extending our implementation, running further experiments, providing proposals for the remaining steps to complete our offloading technique and exploring other security offloading alternatives.

5.1.1 Prototype

An interesting future work line would be extending our prototype's components to make them a full reference implementation of our solution that may be adopted by mobile network operators. While the prototype developed includes implementations of all the components of our architecture, we stated that it is not a full one. To achieve that, the Android application must expose a standard high level interface such as PKCS#11. Also, to avoid the requirement of being pre-installed on the phone as a system application and so that it can be published through standard distribution channels (such as Google Play), it should be signed with keys stored on the SIM card according to Android's mechanism to grant special privileges for UICC owner's applications. Since Android now includes Open Mobile API out of the box, extending our code to also support that API to access the UICC might also be an enhancement. Updating our prototype to take advantage of features in newer versions of the OS without losing backwards compatibility would be in general convenient.

Another extension to complete a reference implementation would be to support a wider range of algorithms in our Java Card Applet.

5.1.2 Experiments

Other future steps will also include repeating the experiments with more smart cards and phone models to keep learning about our architecture's performance and further confirm its limitations. Providing a wider database of benchmarking results for SIM cards and smartphone hardware-backed keystore implementations would also be an even better input for offloading decisions.

5.1.3 SIM Offloading

We cover the actual offloading mechanism and provide useful profiling inputs. In order to completely define how to apply offloading based on this strategy, complementary objectives to continue this work should include defining an offloading decision algorithm. This algorithm would take as inputs security assessment and profiling results of each offloadable component of a program. It would be applied statically since our offloading strategy does not depend on variable metrics such as bandwidth and dynamic decisions are more resource expensive. The output would be, as usual, the partition of the program. The algorithm could be tested with a set of case study applications. Experiments would help us learn about its accuracy and how to iteratively improve it. Case studies would help us understand the actual impact of the performance limitations detected and which type of applications, in terms of offloadable functions, benefit the most from our approach. As an example, secure channel protocol implementations that combine symmetric and asymmetric cryptography might be interesting candidates.

Another pending task already discussed in Chapter 4 is defining a consented pre-detachment procedure of the UICC. Since in our current solution keys are bound to the card, another challenging research thread would be finding a scheme to provide device binding.

5.1.4 Applications

We proposed two possible applications of our architecture that we believe would help enhance Android security. Studying in more detail the implementation of a StrongBox Keymaster or a Secure Enclave that combines the UICC with a TEE and developing proof-of-concept prototypes would help us learn more about these alternatives and is left for future work.

5.1.5 Security Offloading

Finally, we believe that the study of how computation offloading can be applied with security purposes, especially in the realm of mobile cloud computing based offloading, is still an open research thread with a wide domain to explore.

5.2 Final words

We learned in Chapter 2 about the risks of losing physical control over mobile devices and the role of hardware backed cryptography among countermeasures. We are convinced that, despite the form factor used, Secure Elements are instrumental to enhance mobile security. In an ideal scenario, following Google's example with Pixel 3 or the eSIM [11] standard, manufacturers should standardize the inclusion of an embedded Secure Element in every SoC. Even if they head in that direction, it might take a long time for new devices to replace most older models.

In the current landscape, the traditional UICC remains the only universal Secure Element. We believe that the opportunity to deliver a solution based on the approach proposed as a service to their customers, lies in the hands of mobile network operators. Since they control SIM card's contents and have the ability to grant the required permissions to an application by means of a signature. Such solution could be deployed over the air to the card and through application markets to the phone, so that high rates of deployment for the devices that allow it could be achieved considerably fast.

Bibliography

[1] 3rd Generation Partnership Project (3GPP). http://www.3gpp.org. Last access: Feb. 2019.

[2] Android 9 release notes. https://source.android.com/setup/start/p-release-notes. Last access: Feb. 2019.

[3] Android Open Source Project - Security. https://source.android.com/security/. Last access: Feb. 2019.

[4] Android TelephonyManager Interface. https://developer.android.com/reference/android/telephony/TelephonyManager. Last access: Feb. 2019.

[5] Arm SecurCore SC300. https://developer.arm.com/products/processors/cortex-m/sc300-processor. Last access: Feb. 2019.

[6] CF-Auto-Root for Android phones. https://desktop.firmware.mobi/. Last access: Feb. 2019.

[7] Gemalto - Multi-tenant SIM. http://www.gemalto.com/mobile/secure-elements/nfc-sim/. Last access: Feb. 2019.

[8] GlobalPlatform. https://globalplatform.org. Last access: Feb. 2019.

[9] GlobalPlatformPro. https://github.com/martinpaljak/GlobalPlatformPro. Last access: Feb. 2019.

[10] Google Pixel 3 security. https://www.blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/. Last access: Feb. 2019.

[11] GSMA eSIM. https://www.gsma.com/esim/. Last access: Feb. 2019.

[12] Identification cards - Integrated circuit cards . Standard 7816, International Organization for Standardization (ISO).

[13] JavaCard Algorithm Test (JCAlgTest). https://www.fi.muni.cz/~xsvenda/jcalgtest/run_time/execution-time.html. Last access: Feb. 2019.

[14] JavaCardOS. https://www.javacardos.com/javacardforum/. Last access: Feb. 2019.

[15] Kona I - LTE and NFC USIM. https://konai.com/business/cards/communication. Last access: Feb. 2019.

[16] List of Smartphones providing support for SIMalliance Open Mobile API by SEEK for Android Project. https://github.com/seek-for-android/pool/wiki/[UNMAINTAINED]-Devices. Last access: Feb. 2019.

[17] NXP P5CC081 PKI smart card controller Data sheet. `https://cache.nxp.com/docs/en/data-sheet/P5CD016_021_041_Cx081_FAM_SDS.pdf`. Last access: Feb. 2019.

[18] Open Mobile API Android 9. `https://developer.android.com/reference/android/se/omapi/package-summary`. Last access: Feb. 2019.

[19] OpenSSL Library. `https://www.openssl.org/`. Last access: Feb. 2019.

[20] Oracle Java Card Technology. `http://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html`. Last access: Feb. 2019.

[21] OWASP Mobile Security Project. `https://www.owasp.org/index.php/OWASP_Mobile_Security_Project`. Last access: Feb. 2019.

[22] OWASP Mobile Security Project - Mobile Top 10 2016. `https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10`. Last access: Feb. 2019.

[23] PC/SC specifications. `https://www.pcscworkgroup.com/`. Last access: Feb. 2019.

[24] SD Standards. `https://www.sdcard.org/developers/overview/index.html`. Last access: Feb. 2019.

[25] Secure element evaluation kit (seek) for the android platform. `http://seek-for-android.github.io/`. Last access: Feb. 2019.

[26] SIM Card Manager: Windows tool to read SIM card content. `https://sourceforge.net/projects/simcardmanager/`. Last access: Feb. 2019.

[27] SIM Manager: SIM Card management tool. `https://sourceforge.net/projects/tfsimmanager/`. Last access: Feb. 2019.

[28] SIMalliance. `http://simalliance.org/`. Last access: Feb. 2019.

[29] Smartcard Sniffer Tool. `https://github.com/ea/smartcard-sniffer/`. Last access: Feb. 2019.

[30] Smart Cards; USSM: UICC Security Service Module; Stage 1. Technical Specification (TS) 102 266, European Telecommunications Standards Institute (ETSI), Jan. 2006. Version 7.1.0.

[31] Smart Cards; USSM: UICC Security Service Module; Stage 2. Technical Specification (TS) 102 569, European Telecommunications Standards Institute (ETSI), July 2007. Version 7.0.0.

[32] Specification of the Subscriber Identity Module - Mobile Equipment (SIM-ME) Interface. Technical Specification (TS) 11.11, 3rd Generation Partnership Project (3GPP), June 2007. Version 8.14.0.

[33] Smart Cards; Secured packet structure for UICC based applications. Technical Specification (TS) 102 225, European Telecommunications Standards Institute (ETSI), Oct. 2014. Version 12.1.0.

[34] Smart Cards; Remote APDU structure for UICC based applications. Technical Specification (TS) 102 226, European Telecommunications Standards Institute (ETSI), May. 2016. Version 13.0.0.

[35] AT command set for User Equipment (UE). Technical Specification (TS) 27.007, 3rd Generation Partnership Project (3GPP), Mar. 2018. Version 15.1.0.

[36] UICC-terminal interface; Physical and logical characteristics. Technical Specification (TS) 31.101, 3rd Generation Partnership Project (3GPP), July 2018. Version 15.0.0.

[37] USIM and IC card requirements. Technical Specification (TS) 21.111, 3rd Generation Partnership Project (3GPP), Apr. 2018. Version 15.0.0.

[38] Z. Ahmad, L. Francis, T. Ahmed, C. Lobodzinski, D. Audsin, and P. Jiang. Enhancing the security of mobile applications by using tee and (u)sim. In *2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing*, pages 575–582, Dec 2013.

[39] K. Akherfi, M. Gerndt, and H. Harroud. Mobile cloud computing for computation offloading: Issues and challenges. *Applied Computing and Informatics*, 14(1):1 – 16, 2018.

[40] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. K. Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *J. Network and Computer Applications*, 48:44–57, 2015.

[41] H. Altuwaijri and S. Ghouzali. Android data storage security: A review. *Journal of King Saud University - Computer and Information Sciences*, 2018.

[42] P. Angin, B. Bhargava, and R. Ranchal. Tamper-resistant autonomous agents-based mobile-cloud computing. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 843–847, April 2016.

[43] ARM. ARM Security Technology (White Paper). `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2009. Last access: Feb. 2019.

[44] E. Barker and J. Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical Report SP 800-90, NIST, 2015.

[45] M. Bond and R. Anderson. Api-level attacks on embedded systems. *Computer*, 34(10):67–75, Oct 2001.

[46] T. Cooijmans. Secure key storage and secure computation in android. Master's thesis, Radboud University, Nijmegen, 2014.

[47] T. Cooijmans, J. de Ruiter, and E. Poll. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 11–20, NY, USA, 2014. ACM.

[48] S. Delaune, S. Kremer, and G. Steel. Formal analysis of pkcs#11. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 331–344, June 2008.

[49] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture.* No Starch Press, 2014.

[50] GlobalPlatform. Secure Element Configuration, Oct. 2012. (Version 1).

[51] GlobalPlatform. TEE Protection Profile, Nov. 2016. (Version 1.2.1).

[52] J. Hajny, L. Malina, Z. Martinasek, and O. Tethal. Performance evaluation of primitives for privacy-enhancing cryptography on current smart-cards and smart-phones. In J. Garcia-Alfaro, G. Lioudakis, N. Cuppens-Boulahia, S. Foley, and W. M. Fitzgerald, editors, *Data Privacy Management and Autonomous Spontaneous Security*, pages 17–33, Berlin, Heidelberg, 2014. Springer.

[53] D. Huang, P. Wang, and D. Niyato. A dynamic offloading algorithm for mobile computing. *IEEE Transactions on Wireless Communications*, 11(6):1991–1995, June 2012.

[54] G. Hurel, R. Badonnel, O. Festor, and A. Lahmadi. Towards cloud-based compositions of security functions for mobile devices. In *IFIP/IEEE International Symposium on Integrated Network Management (IM 2015)*, pages 578–584, Ontario, Canada, 2015.

[55] G. Hurel, R. Badonnel, A. Lahmadi, and O. Festor. Behavioral and dynamic security functions chaining for android devices. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 57–63, Nov 2015.

[56] A. Inc. iOS Security Guide 11th ed. (White Paper). `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`, 2018. Last access: Feb. 2019.

[57] W. Itani, A. Kayssi, and A. Chehab. Energy-efficient incremental integrity for securing storage in mobile cloud computing. In *2010 International Conference on Energy Aware Computing*, pages 1–2, Dec 2010.

[58] K. Kapetanakis and S. Panagiotakis. Efficient energy consumption's measurement on android devices. In *2012 16th Panhellenic Conference on Informatics*, pages 351–356, Oct. 2012.

[59] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: A computation offloading framework for smartphones. In M. Gris and G. Yang, editors, *Mobile Computing, Applications, and Services*, pages 59–79, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[60] A. N. Khan, M. M. Kiah, S. U. Khan, and S. A. Madani. Towards secure mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(5):1278 – 1299, 2013. Special section: Hybrid Cloud Computing.

[61] D. Kovachev and R. Klamma. Framework for computation offloading in mobile cloud computing. *International Journal of Interactive Multimedia and Artificial Intelligence*, 1(7):6–15, 12/2012 2012.

[62] K. Kumar, J. Liu, Y. Lu, and B. Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, February 2013.

[63] B. Lapid and A. Wool. *Navigating the Samsung TrustZone and Cache-Attacks on the Keymaster Trustlet: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, pages 175–196. 08 2018.

[64] J. Liu, K. Kumar, and Y. Lu. Tradeoff between energy savings and privacy protection in computation offloading. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 213–218, Aug 2010.

[65] S. Ltd. Open Mobile API Specification, 2016. (Version 3.2).

[66] X. Ma, Y. Zhao, L. Zhang, H. Wang, and L. Peng. When mobile terminals meet the cloud: computation offloading as the bridge. *IEEE Network*, 27(5):28–33, September 2013.

[67] T. Mandt, M. Solnik, and D. Wang. Demystifying the Secure Enclave Processor. https://www.youtube.com/watch?v=7UNeUT_sRos, Aug. 2016. Black Hat Briefings, Mandalay Bay, Las Vegas. Last access: Feb. 2019.

[68] T. Meng, K. Wolter, and Q. Wang. Security and performance tradeoff analysis of mobile offloading systems under timing attacks. In M. Beltrán, W. Knottenbelt, and J. Bradley, editors, *Computer Performance Engineering*, pages 32–46, Cham, 2015. Springer International Publishing.

[69] N. Nassar, R. Newhook, and G. Miller. Enhanced mobile security using sim encryption. In *2014 International Conference on Collaboration Technologies and Systems (CTS)*, pages 189–196, May. 2014.

[70] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, Nov. 2016.

[71] Q. H. Nguyen, J. Blobel, and F. Dressler. Energy consumption measurements as a basis for computational offloading for android smartphones. In *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, pages 24–31, Aug. 2016.

[72] NIST. Mobile Threat Catalogue. https://pages.nist.gov/mobile-threat-catalogue/. Last access: Feb. 2019.

[73] D. Oh, I. Kim, K. Kim, S.-M. Lee, and W. W. Ro. Highly secure mobile devices assisted with trusted cloud computing environments. *ETRI Journal*, 37(2):348–358, 2015.

[74] K. Ok, S. Senturk, S. Aktas, and C. Cevikbas. Secure cryptographic operations on sim card for mobile financial services. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 10(9):1603 – 1607, 2016.

[75] A. Oprisnik, D. Hein, and P. Teufl. Identifying cryptographic functionality in android applications. In *2014 11th International Conference on Security and Cryptography (SECRYPT)*, pages 1–12, Aug. 2014.

[76] S. Ou, K. Yang, and A. Liotta. An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In *Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06)*, pages 10 pp.–125, March 2006.

[77] D. Pedraja. Turning the sim card into a cryptographic provider. In *2018 Eighth Latin-American Symposium on Dependable Computing (LADC), Student Forum*, Foz do Iguaçu, Brazil, Oct 2018.

[78] D. Pedraja, J. Baliosian, and G. Betarte. Offloading cryptographic services to the sim card. In *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*, Foz do Iguaçu, Brazil, Oct 2018.

[79] W. Rankl and W. Effing. *Smart Card Handbook*. Wiley Publishing, 4th edition, 2010.

[80] M. Roland and M. Hölzl. Open Mobile API: Accessing the UICC on Android Devices. Technical report, University of Applied Sciences Upper Austria, JR-Center u'smile, Jan. 2016.

[81] S. A. Saab, F. Saab, A. Kayssi, A. Chehab, and I. H. Elhajj. Partial mobile application offloading to the cloud for energy-efficiency with security measures. *Sustainable Computing: Informatics and Systems*, 8:38 – 46, 2015. Special Issue on Computing for a Greener Water/Energy/Emissions Nexus; edited by Carol J. Miller.andSpecial Issue on Green Mobile Cloud Computing (Green MCC); edited by Danielo G. Gomes, Rafael Tolosana-Calasanz, and Nazim Agoulmine.

[82] S. Sinha and R. M. Gaudar. Energy consumption for cryptographic algorithms with different clocks on smart cards in mobile devices. *International Journal of Computer Applications*, 66:1–4, 2013.

[83] F. Wu, J. Niu, and Y. Gao. Bandwidth aware application partitioning for computation offloading on mobile devices. *Green Communications and Networking*, 51:63–72, 2012.