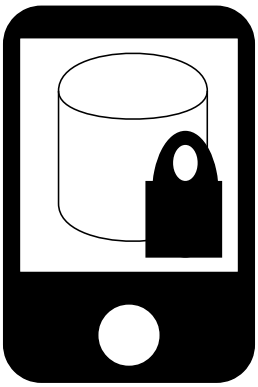
TÉCNICO LISBOA



HCE Mobile Ticketing

Providing Security for Mobile Ticketing Applications Backed by TrustZone

Pedro Guilherme Simões Ribeiro

Thesis to obtain the Master of Science Degree in

Engineering Systems and Computer Engineering

Supervisor: Prof. Nuno Miguel Carvalho Santos

Examination Committee

Chairperson: Prof. José Carlos Alves Pereira Monteiro
Supervisor: Prof. Nuno Miguel Carvalho Santos
Member of the Committee: Prof. Henrique João Lopes Domingos

October 2018

Dedicated to all that have tolerated my occasional bad humor provoked by these five years at IST.

Acknowledgments

I want to start by thanking the people that have directly helped in this work. First, my supervisor, professor Nuno Santos, who was always ready to help me with any problem that came up and was very present during this last year to guide me in the right direction, as well as Nuno Duarte, who was paramount in getting me set up to start developing the work on TrustZone and provided me with the many resources I needed, and who was also very supportive in helping me with the problems I faced during development. I also want to think the OP-TEE community, which was extremely helpful in helping solve some problems that I faced. Lastly, Saidgani, who helped me overcome some really mind-breaking problems.

I also want to thank my parents for the amount of support they have given me through all my life and by giving me the chance of pursuing my dreams. They always believed in my capabilities and raised me to be the person I am today.

To my girlfriend, Ana, who was the person that most had to deal with the stress induced by this college. Thank you for taking me out to dinner or movies when I was stressed the most and for always trying to make me think positively.

To my awesome friends, both those that come from my childhood and those I made over these IST years. Thank you for all the support, laughs and memories over all these years.

Lastly, I also want to thank all the game developers that provided me with many worlds for me to delve to when this one was being too stressful. Specifically, I want to thank World of Warcraft for providing me with an alternate reality where I could forget that IST existed. Thank you for all the fun you've given me over the years.

To everyone, thank you so much!

Resumo

Dispositivos móveis são ubíquos e uma parte central do dia-a-dia. Esta ubiquidade levou a que Operadores de Transportes Públicos desenvolvessem soluções de bilhética que tirassem proveito das capacidades destes dispositivos, alinhado com o crescimento em popularidade e normalização da tecnologia de NFC. Contudo, soluções de bilhética móvel estão sujeitas a serem comprometidas se existir um atacante capaz de controlar o sistema operativo. Embora existam soluções baseadas numa ligação à rede permanente, em certos cenários, é necessário desenvolver uma solução que permita que os cartões virtuais possam estar protegidos no dispositivo enquanto (1) é usada a interface NFC apenas para comunicar com os validadores de bilhetes, e (2) necessitando de um nível de confiança mínimo no sistema operativo. Esta tese pretende fornecer tal solução ao aproveitar os benefícios de ARM TrustZone para o armazenamento de cartões virtuais e execução das transações críticas de aplicações de bilhética móvel. Propomos o DBStore, um sistema de gestão de bases de dados baseado em SQL que permite o armazenamento seguro de cartões virtuais em bases de dados apoiadas por ARM TrustZone. Embora o DBStore tenha sido desenolvido com o foco numa aplicação real de *HCE Mobile Ticketing* e as vulnerabilidades que esta apresenta, a sua aplicabilidade é mais vasta e pode ser utilizado por qualquer aplicação que necessite de armazenamento seguro de dados sensíveis. Implementámos protótipos com o objectivo de serem executados em hardware TrustZone real e a nossa avaliação mostra que o desempenho da solução induz em penalizações reduzidas sobre a aplicação.

Palavras-Chave: Ambientes de Execução Segura, ARM TrustZone, Segurança, Aplicações Android, Bilhética Móvel

Abstract

Mobile devices are ubiquitous and are a central part of everyday's life. This ubiquity led Public Transport Operators to develop ticketing solutions that could take advantage of the capabilities of such devices, aligned with the growth in popularity and standardization of the NFC technology. However, mobile ticketing solutions are prone to be exploited if there is a malicious agent that can compromise the operating system. Although there are security solutions based upon permanent network connectivity, in certain settings, it is necessary to develop a solution that allows for virtual cards to be secured on the mobile device while (1) using the NFC interface only to communicate with ticket validators, and (2) requiring minimal trust in the operating system. This thesis aims to deliver such solution by leveraging the benefits of ARM TrustZone for the storage of virtual cards and execution of the critical transactions of mobile ticketing applications. We propose DBStore, an SQL-based management system that allows for the secure storage of virtual cards in databases protected by ARM TrustZone. Although DBStore was developed with the focus on a real application, to be named as *HCE Mobile Ticketing*, and the vulnerabilities it presents, its applicability is vaster and can be used by any application that requires secure storage of sensitive data. We implemented prototypes aimed at running on real TrustZone hardware and our evaluation shows that the performance of the solution induces small overheads to the application.

Keywords: Trusted Execution Environments, ARM TrustZone, Security, Android Applications, Mobile Ticketing

Contents

List of Tables

List of Figures

Nomenclature

AID Application ID

$APDU$ Application Protocol Data Unit

HCE Host-Card Emulation

MNO Mobile Network Operator

NFC Near Field Communication

NW Non-Secure World

OS Operating System

PTO Public Transport Operator

SE Secure Element

SEI Secure Element Issuer

SMC Secure Monitor Call

SoC System on Chip

$SRPC$ SQL Remote Procedure Call

SW Secure World

TA Trusted Application

TCB Trusted Computing Base

TEE Trusted Execution Environment

TSM Trusted Service Manager

$UICC$ Universal Integrated Circuit Card

$UUID$ Universally Unique Identifier

Chapter 1

Introduction

With the rise in popularity of mobile devices, ticketing solutions that leverage the capabilities of these devices are attracting the interest from Public Transport Operators. As such, mobile ticketing applications are appearing, many of which require an online connection for the validation of cards. It is then interesting to conceive a solution that can work offline securely; this is challenging, however, because many of the current security solutions rely on the correct behavior of the Operating System. This work aims to contribute with a solution based on the ARM TrustZone technology that provides applications with a secure and isolated environment for storage and operations over sensitive data, and we present it in the context of an existing mobile ticketing operation.

1.1 Motivation

Mobile devices are present in everyone's daily life, and the great majority supports high-quality video playback, games, web browsing and has made communication between people more practical than ever. This changed the paradigm and led to mobile devices dominating digital media time over Personal Computers, as a recent study shows [1]. Through online stores like Google's PlayStore, on Android, and Apple's AppStore, on iOS, it is easier than ever to reach millions of potential customers, offering them a new way to access existing services with added convenience.

Public Transport Operators (PTOs) looked at the proliferation of mobile devices as an incredible business opportunity to not only attract new customers but to also deliver existing customers with an updated way to use the service they are used to, but more conveniently. Mobile ticketing applications would allow customers to buy tickets anytime and anywhere, without the hassle of carrying those tickets in their wallets and needing exchange to purchase them. Thus, mobile ticketing solutions became a key point for PTOs, but they are dependent on the technology offered by manufacturers, as the devices had to emulate the behavior of physical contactless cards.

This dependency was very troubling, as there were not any clear standards that would guarantee interoperability and consistency. To that end, the Near Field Communication (NFC) technology was created as a standardized mode of communication between electronic devices. Among the things it defines,

this standard specifies how an NFC enabled device can read or write data and how it can emulate cards. Early solutions required a Secure Element (SE) for card emulation. This SE was usually a SIM card provided by a Mobile Network Operator (MNO), which contained a trustlet application responsible for the card emulation. NFC communications were routed directly to the trustlet on the SIM card, which would conduct the operations independently from the operating system. These solutions, however, were not very simple, as they required negotiation between the service provider and the MNO for the installation of the trustlet on the SIM card. Additionally, clients would need to change their SIM in order to be able to use the service via their mobile devices. Version 4.4 of Android introduced the Host-Card Emulation (HCE) technology [2], which allows an application running on the device's CPU to emulate a card and the Operating System (OS) would route the NFC communication straight into the corresponding application. This was a simpler solution, as it did not require any negotiation with MNOs.

Although HCE is a simpler solution, it provides weaker security guarantees when compared with SE. This is worrying, as these HCE-based applications rely heavily on the OS for assuring the protection of sensitive data. Given that the OS can be compromised by a rootkit malware or by the user rooting the device, it cannot be trusted by either client or provider. Some solutions circumvent this problem by not storing the actual cards, but instead a token of the card data which is itself stored on a remote cloud server. This requires, however, online connection for validating cards. In this thesis, we are analyzing the security requirements of a real-world application named *HCE Mobile Ticketing*, which uses HCE for card emulation, to be able to design a solution that does not require relying on the integrity of the OS for the protection of its sensitive data.

1.2 Goals and Requirements

The main drive behind this thesis is to develop a security mechanism for mobile devices that allows secure storage of sensitive data and operations over it, leveraging the benefits of ARM TrustZone technology, and focusing on the HCE Mobile Ticketing application. As such, the final solution must be able to deliver the following requirements:

No Alterations to the Front-End

As the HCE Mobile Ticketing application will need to store the card data locally on the device, card validations will be done in an offline fashion. This implies that no changes can be done on the validators and that the application will need to fully emulate the behavior and the security guarantees of contactless cards.

Small Trusted Computing Base and Attack Surface

It is important to guarantee that the proposed solution has a small TCB and attack surface. If both of these properties are assured, applications using our solution can be sure that the sensitive data they are trusting our solution to manage is done in a secure way and is not prone to vulnerabilities.

Support for General Applications

Although we are presenting this work in the scope of mobile ticketing applications, we aim to provide a

solution that can be used in a wide range of applications that require secure sensitive data management, by allowing manufacturers to easily deploy it in their products for developers to use.

Developer Friendly

Our solution needs to have strict security protocols but provide an easy-to-use interface for developers. That way, it is easy to integrate it with existing applications and benefit from the security guarantees it provides.

Performance

It is important that the solution has good performance, namely validations, as this operation must be conducted in a timely fashion. In a situation where the user is about to miss the transport, the performance of the solution may be the difference between being able to board it or not.

1.3 Contributions

We thesis presents DBStore, a TrustZone-backed database management system for mobile applications which provides an SQL interface to application-custom databases. A simple-to-use API masks the security protocols designed to defend against rollback and replay attacks, and provides confidentiality and integrity protection of both the databases and SQL commands, including inputs and outputs. The popularity of SQL and the fact that Android provides native support for SQLite make the programming paradigm familiar among developers that require the storage and processing of structured data. As such, the secure databases offered by DBStore, alongside with the respective security protocols, allow implementing a secure ticketing application without requiring altering the validators. Furthermore, the proposed mechanisms are generic and thus allow other applications that require secure storage within the mobile device to take advantage of them, with the isolation being guaranteed by the TEE, backed by the hardware protection offered by ARM TrustZone, and without the need of running arbitrary code on the SW. This thesis presents DBStore as a solution for the real-world problem of *HCE Mobile Ticketing* and the security requirements it has.

In summary, this work makes the following contributions:

- Vulnerability analysis of the HCE Mobile Ticketing Application;
- DBStore, a novel trusted service which operates over databases it manages, backed by TrustZone;
- Application of DBStore as a mechanism to improve the security of the HCE Mobile Ticketing Application;
- Implementation and evaluation of DBStore prototypes in the context of mobile ticketing.

The main results of this work have resulted in two publications, on SECRYPT'18 [3] and INFORUM'18 [4], detailing the architecture of DBStore, the challenges and vulnerabilities of the HCE Mobile Ticketing application, and how DBStore could be used to tackle them.

1.4 Thesis Outline

The rest of this document is organized as follows. Chapter 2 discusses the Background and Related work, by first presenting the context of mobile ticketing upon which our work will be developed and methods for protecting mobile platforms, followed by an analysis of the related work on Trusted Execution Environments, which is the security mechanism we have chosen to implement our solution. Chapter 3 presents the architecture of DBStore, and how it can be used to protect applications that require managing sensitive data, namely an application like HCE Mobile Ticketing. Chapter 4 details the two DBStore prototypes we developed, including every component and the steps took in development. Chapter 5 presents the evaluation performed on the implemented prototypes and the conclusions we can reach. The dissertation concludes with Chapter 6, where we draw the main conclusions over this work.

Chapter 2

Background and Related Work

This section provides some background on the state-of-the-art of ticketing technologies and the specific real-world case we'll be working on, the *HCE Mobile Ticketing* application, including the vulnerabilities it presents. It ends with a presentation on the state-of-the-art of Trusted Execution Environments (TEE) and alternative mobile security solutions.

2.1 Overview of Ticketing Technologies

This section presents the state of the art in ticketing technology. We start by providing a brief roadmap of the evolution of ticketing mechanisms. Then, we focus in more detail on the most technologically advanced solutions of today: smartcard-based ticketing and mobile ticketing.

The first card solutions were based on embossed cards. Then, advancements in technology led to the first cards that were able to store information: magnetic stripe cards. These cards had a band of magnetic material which was able to store data by modifying its magnetism. For banking, the next evolutionary step was the appearance of contact smartcards based on the EMV standard, which is still widely used nowadays for payments. These cards were more advanced than the magnetic stripes cards as they had an embedded integrated circuit, which allowed for the storage of more data with increased complexity. The next step was the appearance of contactless smartcards, which are used in many public transport ticketing systems and are becoming more popular for payments as well. PTOs are now moving mobile-based ticketing solutions.

2.1.1 Smartcard-based Ticketing

Contactless cards [5] are the most widespread ticketing method available currently. These cards contain an integrated embedded circuit that, when in contact with a terminal, exchange data via radio waves. The adoption of this technology provided both customers and PTOs with a number of advantages, as summarized in Table 1.

The fact that contactless cards are able to hold information brought many new business opportunities. For one, the same card can be used for various transport methods, such as train, bus, and subway.

Clients	*Operators*
Convenience and ease of access	Better protection against fraud
Improves cost-efficiency	Less maintenance costs
Improves the protection of the fare token	Facilitates faster boarding
Works better in harsh conditions	Enables new business models for fares

Table 2.1: Contactless card advantages for clients and operators.

Figure 2.1: A Suica card on the left [6] and a terminal on the right [8].

Additionally, it can also be used for different services, like paying parking or buying a coffee. Suica, a contactless card system used in Japan, is one of the first implementations of this technology.

The contactless card environment can be split into two main different type of cards: smartcards and memory cards. The first, smartcards, are cards that have an embedded integrated circuit, which works as a secure element with cryptographic capacities. It is the most widely used type of card, both in payment and ticketing. It is usually complemented with security hardware on the readers. The second type, memory cards, are cards that possess a non-volatile memory storage component, with the device's security being dependent on each type of the proposed supports. Furthermore, the security is specific to the type of implementation, which may leave it vulnerable to attacks, such as replay attacks and cloning. Cards of this type will be the focus of the work being discussed on this report.

Case studies: Suica [6] is a prepaid, FeliCa-based [7] e-money card that can be used for PT and shopping, being one of the first implementations of contactless cards for these systems. It relieves the client from the need of buying a ticket from a vending machine. Purchase of the card is done at dedicated machines in Japan Railway East stations, with recharging being done at machines that display the Suica mark. It can be used not only in Japan Railway East trains but in buses and subways as well. Additionally, as long as it has credit, it can be used for a variety of other services, such as buying a newspaper or lunch, without the need of physical currency, such as coins. Overall, by having a single card, traveling becomes easier and more enjoyable.

The process goes as follows. First, the client passes the card through the reader at the entrance's ticket gate. It is not necessary to buy a ticket (see Figure 2.1). Then, at the exit's ticket gate, the fare is automatically calculated and discounted off the card. Finally, if the balance is insufficient, the client will need to add more credit to it. Note, however, that the card cannot be used for continuous travel between areas. To do so, the client's travel will have to end in the area it began. The client will then have to re-enter to be able to cross to another area.

Clients	*Operators*
Easier and more convenient access to Public Transports	Investment protection in fare management infrastructures
Easier ticket reloading	Better implementation flexibility
Travel experiences richer in information and more accessible	The opportunity to add value with new applications
More enjoyable journeys	Opportunity to build regional nets of PTs

Table 2.2: Mobile ticketing advantages for clients and operators.

There are some other examples that can be referred, such as Oyster [9], in London, which is a contactless card which can hold pay-as-you-go credit, as well as travelcards and bus and train passes and supports automatic top-up. EasyCard [10], in Taiwan, is another example of a contactless card ticketing system, more specifically for the Taipei metro, bus, designated parking lots, among others.

2.1.2 Mobile Ticketing

Smartphones are now ubiquitous items, present in the pockets or purses of almost every individual. This presents itself as a great opportunity for operators of public transports to evolve their ticketing systems, especially when considering some disadvantages of contactless cards. For example, while a client may need several different contactless cards in his/her wallet, he/she may use the same smartphone to emulate all those cards. Given that smartphones accompany the clients, moving the cards to those devices seems like a logical step. Table 2.2 summarizes the main advantages of mobile ticketing to both clients and PTOs. This section provides an overview of a fundamental enabler technology (NFC) and presents a few relevant case studies.

Mobile ticketing [5] allows for mobile devices to emulate contactless cards. To do so, the mobile device must support Near-Field Communication (NFC) and have an application that allows to store and access the card. As the mobile device will emulate the contactless card, no changes in the front office of the PTO will be necessary; this leads to a seamless and simple integration of this new technology. However, this implementation has been severely delayed due to the lack of proper standards and specifications that the manufacturers of mobile devices had to follow to guarantee interoperability. This has changed recently due to the efforts of GSMA and the NFC Forum and mobile devices must now be interoperable with defined standards and specifications (*ISO/IEC 14443*, *ISO/IEC 18092*, NFC Forum Analog) to guarantee interoperability.

Near Field Communication (NFC): NFC is a technology that allows for simple and bidirectional interactions between electronic devices. With a low range and a one-touch user interface, it guarantees that only intended transactions are conducted. The NFC Forum was founded in 2004 as a platform to harmonize and standardize interoperability. NFC complements other popular wireless technologies by using key elements of existing standards for contactless cards (*ISO/IEC 14443*), can be compatible with existing contactless card infrastructures and allows for a consumer to use a single device for different systems. It operates at a range of 13.56 MHz, with a maximum working distance of 10 centimeters and

speeds of up to 424 kbit/s [11].

NFC communicates via radio frequency and uses a protocol named Near Field Communication Interface and Protocol (NFCIP-1). This protocol has two communication modes: active mode and passive mode. In the Active Mode, both Initiator and Target use their own RF field to communicate. The Initiator starts the NFCIP-1 transaction and the Target responds to an Initiator command in the Active communication mode by modulating its own RF field. In the Passive Mode, the initiator generates the RF field and starts the transaction and the Target to an Initiator command in the Passive Mode by modulating the Initiator's RF field. This is known as load modulation. The General Protocol flow between NFCIP-1 devices goes as follows:

1. Any NFCIP-1 device must be in Target Mode initially, not generating any RF field. It stays in that mode waiting for a command from an Initiator;

2. The NFCIP-1 device may change to Initiator Mode and decide to use either Active or Passive communication mode and transfer speed;

3. Initiators shall test for external RF field presence and must not activate their RF field if an external RF field is detected;

4. Otherwise, if an external RF field is not detected, the Initiator may active its RF field for the activation of the Target;

5. Exchange commands and responses in the same communication mode and transfer speed.

The messages traded between devices are called Application Protocol Data Units (APDUs). APDU commands are divided into two categories: command APDUs and response APDUs. Command APDUs are sent by the reader to the card, which contains an obligatory 4-byte header and command-specific data. Cards respond by sending a response APDU to the reader, containing an obligatory 2-byte header and response-specific data. Applications supporting NFC are identified by an Application ID (AID). APDUs and AIDs are specified in parts 4 and 5 of *ISO/IEC 7816*, respectively. The most relevant standards related to mobile ticketing are shown in A.1.

The many advantages of mobile ticketing, motivated many companies to develop their own NFC applications. Early implementations used a Secure Element (SE) to store the cards. This SE would contain a trustlet application responsible for processing the APDUs it receives from a validator and for answering back. Google's adoption of Host-based Card Emulation (HCE) [12] in version 4.4 of Android OS led to many new opportunities for both issuers and providers to implement and deploy NFC solutions without the need for an SE and Trusted Service Manager (TSM) infrastructure, as well as the need for agreements with Secure Element Issuers (SEIs).

SE-based card emulation: The first card emulation solutions that were available required an SE, which contained both the card application and its credentials. A mobile application will also reside in the mobile device, outside of the SE, containing the interface needed for communicating with the card application

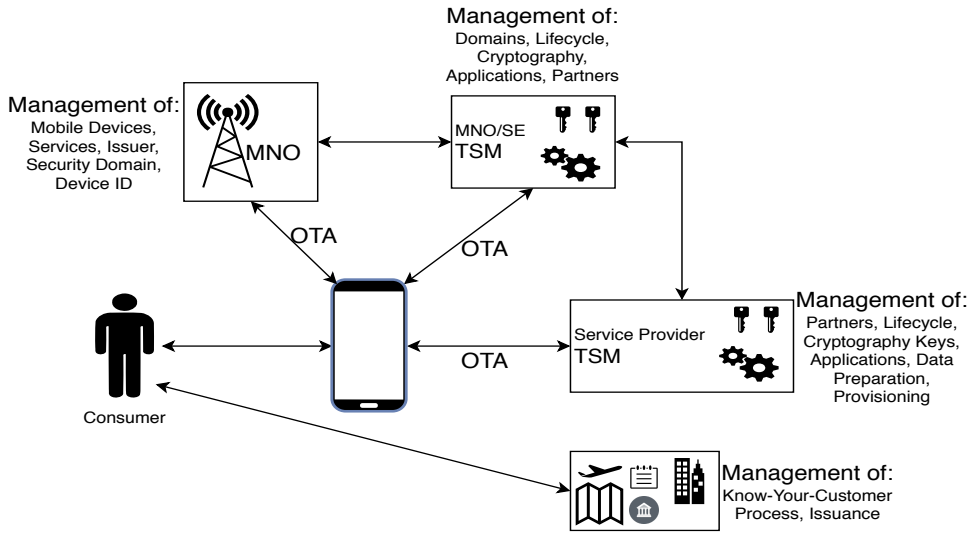


Figure 2.2: TSM used to provision credentials (adapted from [12]).

stored in the SE. When in contact with a reader, the NFC Antenna looks for any ”SELECT AID” APDU and, when it detects one, routes it to the active SE.

The SE can be in an embedded secure smart card chip on the handset, on the SIM, on a Universal Integrated Circuit Card (UICC), or on an SD card. Both SIM and UICC are issued by Mobile Network Operators (MNOs), embedded SE are issued by mobile device manufacturers, and SEs on SD cards are issued without any application provider. Credentials are provisioned via a Trusted Services Manager (TSM), which also keeps the application itself. This requires a good amount of cooperation between issuers, wallet providers, MNOs, among others. More specifically, credentials in the SE are provisioned in domains that follow the GlobalPlatform's specifications, and each provider or issuer is assigned to a specific domain, which is protected by cryptographic keys only known by the intervenients as to protect from unauthorized access. A wallet application is required to be able to communicate with the application residing in the SE. As such, the wallet application authenticates itself to the SE (usually via a PIN or password) as to be able to access data. Figure 2.2 illustrates this architecture.

HCE-based card emulation: Host Card Emulation (HCE) [12] allows a mobile application to emulate a contactless card and communicate with another NFC tag, such as a reader. Previous solutions required a Secure Element, and all communication between mobile device and reader was routed to it without any intervention by the OS. In Android, the HCE architecture uses service components known as HCE Services, which can run in the background without any user interface and are protected by the OS against user application attacks. APDUs are associated with services by an explicit declaration in the manifest of the Android application.

Given the possible coexistence of NFC Card Emulation on an SE and using HCE, a procedure named *AID Routing* is required, where the NFC Controller is able to associate an AID with an application running in the Host CPU or on the SE and route requests to it. The NFC Controller implements a routing table, which is populated by the OS, containing the AIDs of applications residing either in the Host CPU or

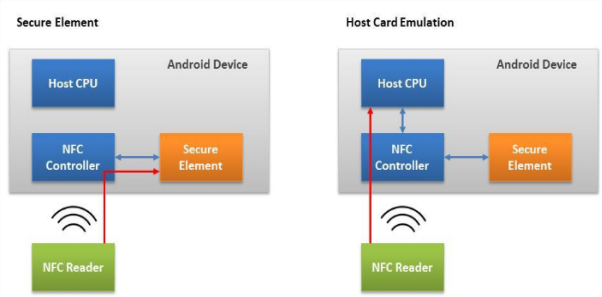


Figure 2.3: NFC communication with an SE (left) and with HCE Services running on the Host CPU (right) [12].

in the SE, if there is one. This routing table consists of a list of routing rules, associating an AID with a destination, which can be either on the Host CPU or on a connected SE. When the NFC Controller receives a "SELECT AID" APDU, it procures the AID in the routing table and if a match is found it routes the APDU and the ones following to the corresponding destination associated with that AID until another "SELECT AID" APDU is received or the NFC link is broken. Both HCE and SE can exist in the same mobile device, but the advantage of HCE is that it does not explicitly require an SE – instead, the SE can be seen as complementary (i.e. to store credentials), if chosen to be used (see Figure 2.3). As such, an application using HCE can be simply downloaded and installed from the Play Store, whereas with SE the user usually needs to request one (i.e. a special SIM from its MNO). The storage of the credentials associated with this HCE-based application is left to the developer's choice. They can be stored in the application itself, on a Trusted Execution Environment (TEE) or in an SE.

Most mobile card emulation solutions nowadays are still SE-based, but solutions based on HCE are becoming more popular, mostly using token-based solutions, where only a token is stored in the device, with the actual data stored remotely. One such example is Rambus HCE Ticketing [13], which uses tokenization and a connection to a cloud-storage server to guarantee that cards are not exploited locally on the device and to authenticate operations. It also uses this online connection to validate cards. It also provides PTOs with tools for travelers data and analytics which may be used for optimizing prices, routes and ticket offers.

SE vs HCE: HCE-based ticketing presents advantages when compared with SE-based ticketing, but it also comes with disadvantages and new vulnerabilities [14]. In terms of *provisioning*, HCE provides more advantages, the biggest one being the removed necessity of an SE, which in turn removes the need for MNO involvement. This leads to a simplification of the NFC ecosystem and makes it much easier to develop applications, as they are independent of the SE issuer and don't require integration with the mobile operator TSM. Additionally, the same HCE Services infrastructure can be reused for multiple different applications.

Considering *usability*, it makes no difference for the consumer. It requires a newer version of Android but, in contrast, requires no new SIM. Apart from this, using an SE-based or HCE-based ticketing solution is essentially identical.

Lastly, the *security* aspect is a considerable problem with HCE. With the removal of the SE, there

are no a priori requirements of where the credentials are to be stored. Some solutions use an ”SE in the Cloud” approach (such as CloudSE [15]), where the credentials are safely stored in a secure data center and the device receives single-use credentials to be used on a single transaction. This requires, however, some form of connection, which is not tolerable in systems that want to work purely offline and brings the problem of how the secure data center can authenticate a legit device and user. These credentials can also be stored in the application data, which leads to the second major security problem, rooted devices. With a rooted device, a user or malware can freely access any information stored on the device, including application data. As such, the stored credentials can be attacked. Additionally, malware can potentially be able to root the device (see RootSmart [16]). Denial of service is also a possible attack if the malware is able to change the routing table. These problems will be discussed in greater detail in the following sections.

Revisiting Suica: Smartphone usage growth in Japan motivated the leaders of JR East to find a mobile-based solution [17]. Since 2006, mobile wallet-enabled devices were able to be used to travel in Japan, only requiring a FeliCa SIM to store data on its secure element. Mobile Suica was the application that made it possible, and it offered new features that a contactless card could not, such as the ability to add and check balance.

Figure 2.4: Mobile Suica operating via NFC on an iPhone [18].

However, this solution was rather limited, as it required a dedicated SIM card. With the rise in popularity of HCE, JR East decided to take the next step and expand Mobile Suica to also support NFC-enabled Android devices, which has been available since 2011. It had a number of advantages for both customers and operator, like the ability to have multiple services on the same device and, given that NFC is a standard, it paves the way to interoperability across Japan, enlarging JR East’s client base. Additionally, in 2016, with the release of Apple Pay in Japan, Mobile Suica also became compatible with iPhones, further enlarging its potential (see Figure 2.4).

2.2 HCE Mobile Ticketing

The many advantages of mobile ticketing discussed above have motivated companies to develop HCE mobile ticketing solutions. It is important that these solutions deliver the same security as the physical contactless cards they emulate, like the isolation of data, its integrity, and confidentiality. In this work,

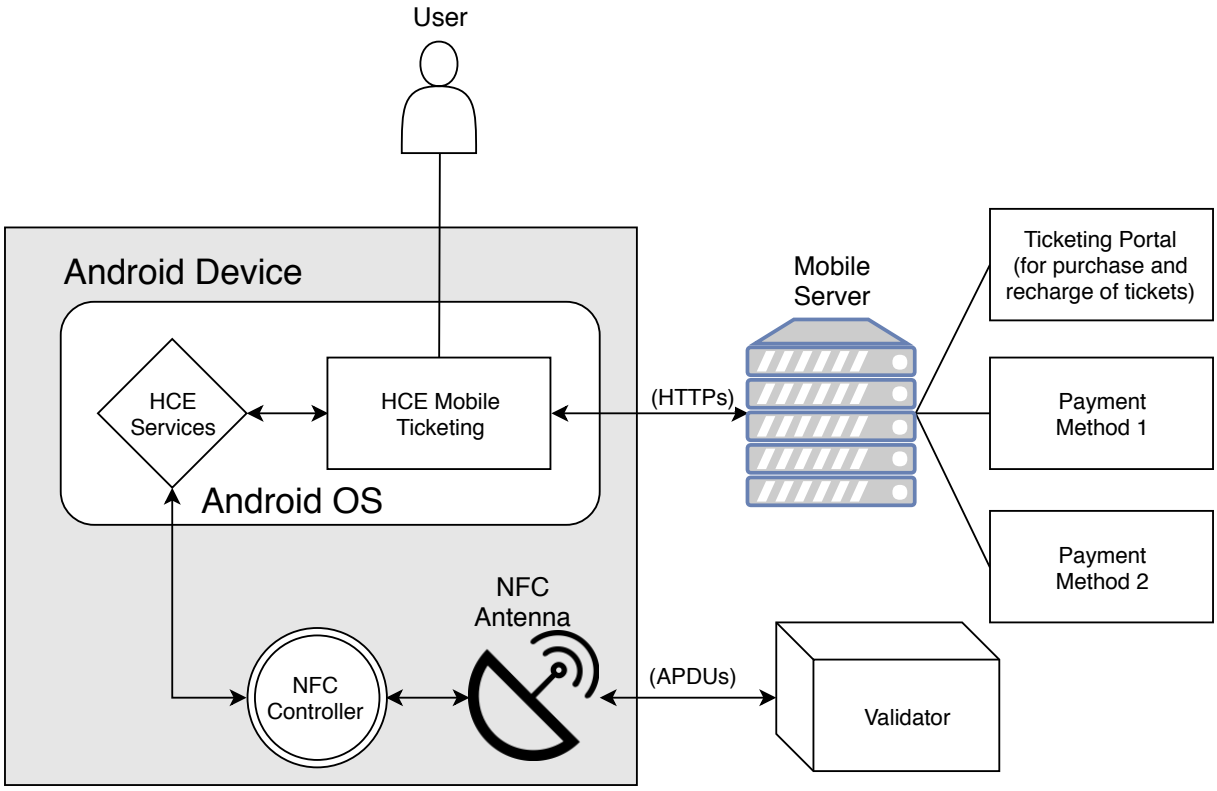


Figure 2.5: Architecture of HCE mobile ticketing system.

we consider protection against hardware tampering provided by these cards as the software protecting mechanisms that prevent an attacker from changing the integrity of the application and its data. Next, we present the architecture and user operations of a typical HCE mobile ticketing system, and then the security mechanisms that existing HCE mobile ticketing systems employ at the mobile endpoints (Section 2.2.3).

2.2.1 Architecture

HCE mobile ticketing solutions tend to be developed with specific requirements according to their contractor and deployment restrictions. In this work, we contemplate the cases in which two specific requirements must be satisfied. The first one is that the system must nor require changes to the already implemented front-end infrastructure, namely to the validators. Additionally, the mobile devices that emulate the ticketing cards must be able to operate without any type of Internet connection, only needing it when the client wants to buy or recharge a ticket.

The general architecture of such an HCE mobile ticketing is shown in Figure 2.5. The system consists of three main components: a *mobile application* through which the user can buy tickets and validate the ticket, a system of *validator* validators in front of which the user must wave the mobile device in order to get access to the transportation system, and a *backend* consisting of multiple servers and subsystems responsible for keeping track of existing tickets, supporting payments, and detecting fraud.

The user can download the mobile application through Google Play Store and, after installing and

opening it, the user is prompted with the login screen. The credentials used to log-in are the same to be used to log-in on a system's website. If the user does not have an account, he is able to create one through the application. After logging in, the user reaches the main hub of the application, where he is able to check information about the cards he owns, purchase new cards or recharge existing ones. Operations are completed via a connection to the Mobile Server, which also works as a middleman for contacts with the Ticketing Portal, which is used for the generation of the cards, and for contacts with the supported payment methods: Payment Method 1 and Payment Method 2. Data exchanges between the application and the Mobile Server are secured via HTTPS.

The application can emulate contactless cards, through a Ticketing Library, which is integrated with Android's HCE. The contactless cards, whose credentials (e.g. serial number and balance) are kept in a file stored in the application data, is encrypted with a symmetric key K that is changed after every time the data is altered and then used to encrypt again the file. The key K is stored in a file, encrypted with the public key $PKey$ of a keypair which is stored on AndroidKeyStore type keystore [19]. This file is based on EN 1545 [20] [21], which specifies the coding of data elements used in public transports, such as date, time, validation event, transport contract, balance, among others. The Ticketing Library runs on top of HCE, receiving the APDUs routed from the NFC Controller and is responsible for managing the emulated cards and the data they contain. Additionally, it is capable of authenticating readers through the commands it receives; the readers use a modified version of *ISO/IEC 7816:4*, as they send an additional signature of the APDU alongside it. Ticketing Library is able to verify this signature (through hardcoded keys) and authenticate a valid reader. However, readers do not authenticate the application, only the credentials of the cards they read. For the communication with validators, the NFC Antenna listens for APDUs, which it sends to the NFC Controller that routes them to the corresponding HCE Service in the application. According to the type of APDU, this results in writes or reads of card information. The validators log every validation they perform, and these logs are analyzed once every week. Serial numbers are cross-referenced with the data stored by the ticketing operator relative to the balance of each card, and cards that present more validations that they were capable of have they serial number blocked, and thus they stopped being accepted by the validators.

In comparison with a solution such as the one by Rambus, it is clear that HCE Mobile Ticketing has a number of challenges and advantages. In terms of challenges, ensuring that cards are not modified locally and that attackers cannot exploit the application are the main concern. This is analyzed with greater detail in Section 3.1. Given that Rambus HCE Ticketing uses tokenization and a connection to a cloud-based storage server, the card data isn't actually stored locally on the device. This means that the application is protected against integrity attacks, but has the downside of requiring an online connection for the validation of operations, and this is where HCE Mobile Ticketing shines. It is able to conduct operations offline and the convenience of allowing clients to conduct card validations without the hassle of requiring an online connection is what makes this application differ from the typical token-based approaches.

2.2.2 User Operations

Typically, users perform three main operations: card purchase, chard recharge, and card validation.

Card Purchase: Card purchase is done by contacting the Ticketing Portal, via the Mobile Server. The user chooses the title to buy and how much credit if must contain. The application sends that information to the Mobile Server, that generates a secret serial number for the new card, which it then signs cryptographically. It sends that information back to the application, and the signature is verified. If it is accepted, the new card is cached. The application then proceeds to the payment screen, which can be done via Payment Method 1 or Payment Method 2. When the payment is concluded with success, the Mobile Server sends an acknowledge back to the application, which then commits the cached card and encrypts it.

Card Recharge: The process is very similar to the purchase scenario; in this case, the card already exists. The user chooses the card and the value to recharge it with. Then, proceeds to payment, as in the purchase case, the Mobile Server verifies the success of the payment and sends an acknowledge back to the application. The application generates a new symmetric key K, decrypts the card file with the old key, adds credit to the card, encrypts the card file with the new key, saves that new key to a file which is encrypted with the $PKey$ of the keypair stored in the keystore.

Card Validations: Validations are done when a reader verifies the authenticity of a card and credit is discounted from it. The mobile device must have NFC enabled and the desired card chosen. Then, it taps the device on the validator. The validator sends a “SELECT AID” APDU that the NFC antenna of the mobile device detects. It is sent to the NFC controller, which then routes it to the corresponding HCE service. The application sends an APDU that permits the reader to authenticate the card and know that it has enough balance. After the reader authenticates the card, it sends back to the application an APDU that instructs for a write to be done on the card (the new credit). The NFC antenna picks this APDU, sends it to the NFC controller that routes it to the corresponding HCE service in the application. Then, the process of altering the file is equal to the purchase scenario. The operation finishes with the device sending an ADPU to the validator with the result of the operation.

2.2.3 Security Mechanisms

In this section, we present in more detail the internals of the HCE mobile ticketing application with particular emphasis on its security mechanisms employed for cryptographic key protection. Figure 2.6 shows a more detailed look at HCE mobile ticketing client endpoint. Cards are stored on the application’s data, encrypted with a key K, which is kept in the file encrypted with the public key $PKey$ of a keypair stored and managed by the AndroidKeyStore. This key K is changed every time the data is altered to ensure that cards cannot be reused; imagine that a malicious client Mallory purchases a card C_1, which is then encrypted with key K_1, $\{C_1\}K_1$, with balance B_1 and copies it for later reuse when C_1 runs out of balance. After the first validation, K_1 is replaced with K_2 as the current key, which is used to encrypt

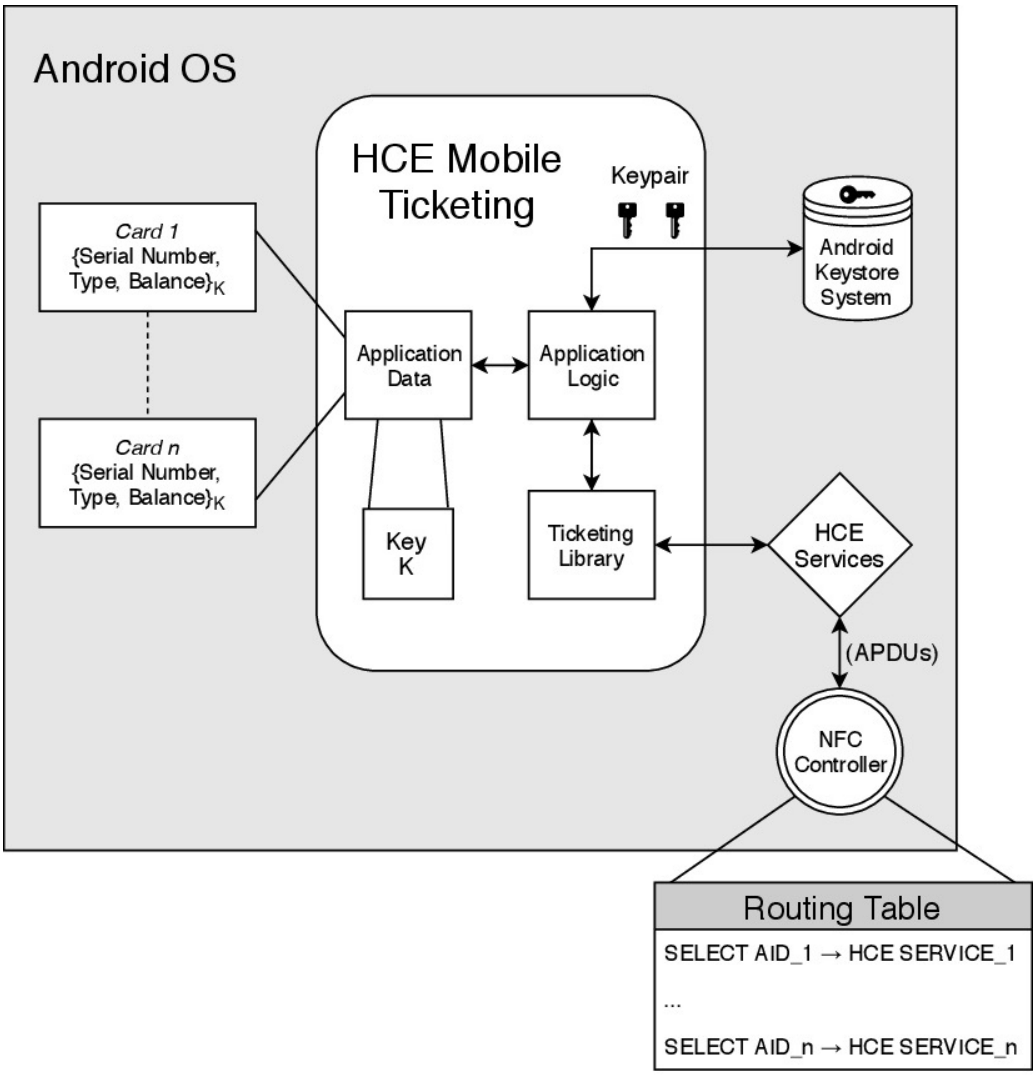


Figure 2.6: In-depth look at the HCE mobile ticketing client endpoint.

again C_1, $\{C_1\}K_2$. After n validations, C_1 will be encrypted with K_n, $\{C_1\}K_n$. If Mallory attempts to reuse $\{C_1\}K_1$, the application will not be able to decrypt it using key Kn and as such fraudulent use is avoided.

As mentioned in Section 2.2.1, the mobile application uses AndroidKeyStore for the storage of keys. Joeri de Ruiter et al. [22] discuss different implementations of key storage in different devices running different Android versions, which will serve as a basis for HCE mobile ticketing. These implementations can either be software or hardware based. In a software-based implementation, applications and internal services are assigned different user IDs, which differs from a standard Linux system where each user as a single user ID and the applications he runs are assigned that same ID. As such, this allows for an initial line of the defense at the OS level: as each application runs with a different user ID, only an application can access its own data. An application is blocked by the OS from accessing the data from another application as the user IDs will mismatch. There are more security mechanisms for key storage on an Android device. As for a hardware-based implementation, a viable option is to use a Trusted Execution Environment (TEE). The application responsible for key management can either reside on the secure

world, or in the insecure world and it requests operations that use the keys from a service running on the secure world, which is responsible for storing and using the keys. For this work, the latter is analyzed with more detail.

Independently of these two mechanisms, keys can be further protected by a password. Passwords can either be stored in the file system or code, which is susceptible to be found by a patient attacker, or can be provided by the user, which prevents the attacker from reading it from the file system or code but leaves the option for a brute-force attack. The greater the entropy for the password the harder it is for an attacker to crack it. The attacker can also use phishing or a keylogger to try and discover the password.

Regarding API and library support, in Android, there are two popular keystore types, Bouncy Castle and AndroidKeyStore. Bouncy Castle [23] is a cryptographic library for Java and it widely present across Android (although in a more limited version than the regular Bouncy Castle library). It contains a keystore type, BKS, which is the default keystore type returned by the Android API in all the analyzed devices. Keys are stored via file-based keystores. Since API Level 18, Android supports an additional keystore type named AndroidKeyStore. It communicates with a KeyStore process that is started when the device boots, through Inter-Process Communication. Manufacturers can develop drivers for their hardware that are able to communicate with this service as to obtain hardware-based secure storage. When there are no drivers available, Android uses a software implementation. Android KeyStore provides no API for a user-provided password to protect the keys. With TEE, AndroidKeyStore encrypts the keystores using a device-specific key (from now on referred as master key) that is stored on the TEE.

With Bouncy Castle, the keystore file is usually stored in the application data, which prevents it to be accessed by other applications. In the case where a stored password is used, it is also saved in the application data, whereas with a user-provided password the entries are only accessible when the user introduces the password. With AndroidKeyStore, for each key pair that is generated, two files are created and stored on */data/misc/keystore/user_0*:

- *USRPKEY*: Stores the key pair parameters. In Qualcomm devices, it also contains the private key, whereas in TI devices the private key is stored on *data/smc/user.bin*. In both cases, the file is encrypted with the master key. It follows the naming format *(App's User ID)_USRPKEY_(Key Entry Alias)*

- *USRCERT*: Stores the certificate for the public key. It has a very similar naming format, it being *(App's User ID)_USRCERT_(Key Entry Alias)*

The *App's User ID* is attributed by Android, being unique to each application and it is essentially what dictates the assignment of keys to applications. The *Key Entry Alias* is left to the programmer.

A more in-depth security analysis will be conducted on Section 3.1, detailing the current flaws of this system and how they can be exploited by a malicious attacker. We will present the Threat Model and examples of three attacks that can be conducted.

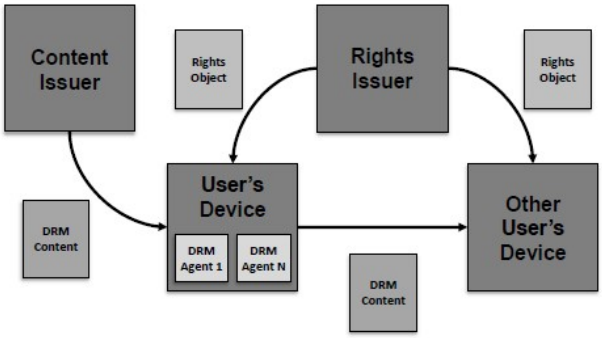


Figure 2.7: DRM ecosystem [24].

2.3 Alternative Mobile Security Systems

While TrustZone is the approach we will be using to protect the *HCE Mobile Ticketing* application, it is interesting to present other alternative approaches to the protection of mobile applications. As Nuno O. Duarte details on his thesis [24], they can be summed up in seven groups: (i) digital rights management, (ii) access control mechanisms, (iii) permission refinement, (iv) application API security, (v) privacy enhancement systems, (vi) access control hook APIs, and (vii) memory instrumentation. We will now present each of these groups and the reason why they are not the best solution to our problem.

2.3.1 Digital Rights Management

Data owners require some assurances about how they can control their data, copies of it and how transferring it to other devices is handled. This is the field of Digital Rights Management (DRM), and it governs the field of music distribution, such as Spotify [25], or movie distribution, such as Netflix [26].

Figure 2.7 illustrates the main actors in a typical DRM scenario. The user is the one that wants to access the digital content and is represented by his device. The DRM Agent is a trusted entity that enforces permissions and constraints related to the DRM content, while the Content Issuer delivers it. Rights Objects are metadata documents that exist alongside the DRM content and contain information regarding permissions and constraints of such content, being generated by the Rights Issuer. In the examples given above, the Content Issuer would be either Spotify or Netflix, which would use a Rights Issuer so that music or movies, the DRM Content, could be distributed along a set of usage restrictions, the Rights Objects, when the content is downloaded to the user's devices.

Some examples of DRM Frameworks include the Android DRM Framework [27], which developers can use to enable in their applications means to control how their right-protected content can be managed and protected. This mechanism is supported by Android natively, but it does not give regular users means to control their personal data. Another example is Porscha [28], which aims to fill this gap in Android DRM Framework by providing mechanisms for users to define policies regarding how their data can be transferred and to whom.

Analysis: DRM solutions are particularly useful in handling the distribution of digital content owned by some entity. However, DRM Content is associated with time constraints, as it typically can only be used for a certain time period until it expires, and is not subject to modifications. For these reasons alone, DRM is not a viable solution to secure the card data in our HCE Mobile Ticketing Application. Furthermore, these solutions are dependent on the integrity of the OS, which makes them vulnerable in the case it is compromised.

2.3.2 Access Control Mechanisms

Access Control Mechanisms define the way objects or system resources, such as files, can be accessed by processes or threads, for instance. Discretionary Access Control (DAC) is a type of access control and the one adopted in Linux, and therefore Android, albeit with a slight difference. While in the Linux kernel there is an User ID (UID) per user, on Android there is an UID per application. This means that when an application creates a file, it is stored in the filesystem using its UID to mark it as the owner. Being the owner, the application may also allow other applications to access the file. This logic applies not only to files, but to all system resources, such as sockets.

The remaining of Android's resources follow the Mandatory Access Control (MAC), which mainly serves to block users from changing permissions, with the access control evaluation being conducted at the kernel level when an access request is triggered. Applications express the permissions that they require in the manifest file, which the kernel will use at runtime to decide whether the access to some resource is allowed or not. For instance, if an application defines in its manifest file that it only requires accessing the microphone, the kernel will permit accessing it at runtime if it tries to do so. If the same application tries to access the camera, the kernel will reject the access because it has not been specified in the manifest file.

Extensions to the access control model have been proposed over the years. SEAndroid [29] is an example, with its goal being to extend the MAC to also be enforced when an application is installed. This means that when an user tries to install a new application, it is checked against the system's policies, with the kernel deciding whether or not the application complies. The main idea is to reduce the chance of a flawed or malicious application to potentially leak access to data. To be able to implement SEAndroid, SELinux [30] was ported as to have MAC at the kernel level, while also adapting it to fit the software stack of Android's userspace.

Another extension is TrustDroid [31], which aims to provide isolation between applications and data, by extending the current MAC mechanism to all plataform's resources, namely the filesystem using DAC, thus isolating the sensitive information present in different domains. As with SEAndroid, it was required having MAC at the kernel level and, to do so, they ported TOMOYO Linux [32]. Additionally, some modifications to the middleware were required to support this new isolation mechanism.

Analysis: Both of these systems enforce a MAC policy, which can help to isolate HCE Mobile Ticketing's data. However, both extensions work at the kernel level, meaning that for them to function correctly it is

implied that the OS is trustworthy, leading to one of the original problems. For this reason, they are not a viable solution to our problem.

2.3.3 Permission Refinement

Android's inflexibility is a problem; when a user wants to install some application, he's presented with a list of permissions that are specified in the manifest file and is forced to either grant them all forever, or quit the installation altogether. Given that the user's intention of installing the application often overcomes his consideration for security or privacy, this may lead to applications being able to access more resources than they actually need, and possibly without the user's knowlege.

This inflexibility led to the development of extensions that allow Android's permission model to be less *extreme* – accept *all* permissions and install the application or accept *none* and don't install it. APEX [33] is such an extension, as it provides users with the ability to control permissions both at installation and runtime, while also being able to define constraints over them. Another example is Compac [34], which considers applications as a group of components and aims to prevent privilege escalation. On example of this problem is an application for tuning a guitar, which requires accessing the microphone. This application is free, but it shows ads to the user. Given that the ad component is part of the application, it has access to the same level of permissions, i.e., can access the microphone as well, which it shouldn't. As such, if the ad component is malicious, it may be able to leak audio without the user knowing.

Other extensions were developed with the intent of using contextual information to alter policies. CRePE [35] relies on trusted third parties that have previously defined security policies on a user's device, which will be enforced depending on the context, which can be time or location. An example may be a company, which enforces a security policy that forbids the employees from accessing the Facebook application when inside the premises during working hours.

MOSES [36] is another context-aware mechanism that enforces domain isolation by defining security profiles. Like CRePE, it changes security profiles according to pre-defined contextual conditions, such as location, and is able to terminate applications that don't obey the currently enforced profile. It also leverages TaintDroid, which will be detailed shortly, for determining which profile owns a certain piece of data, thus preventing an unauthorized profile from accessing data it is not allowed to.

Analysis: While these mechanisms are interesting extensions to Android's permission model, they are not helpful to protect our HCE Mobile Ticketing, because they also rely on the integrity of the OS to function properly. For this reason it is not applicable to the HCE Mobile Ticketing Application.

2.3.4 Application Communication Monitoring

Another interesting group of security extensions is the one responsible for monitoring how applications interact on Android, and how it can be exploited by malicious developers. Figure 2.8 shows two possible attacks that leverage this communication mechanism. The first attack is called the confused deputy attack and is a case of privilege escalation. It consists of an unprivileged application that lacks certain

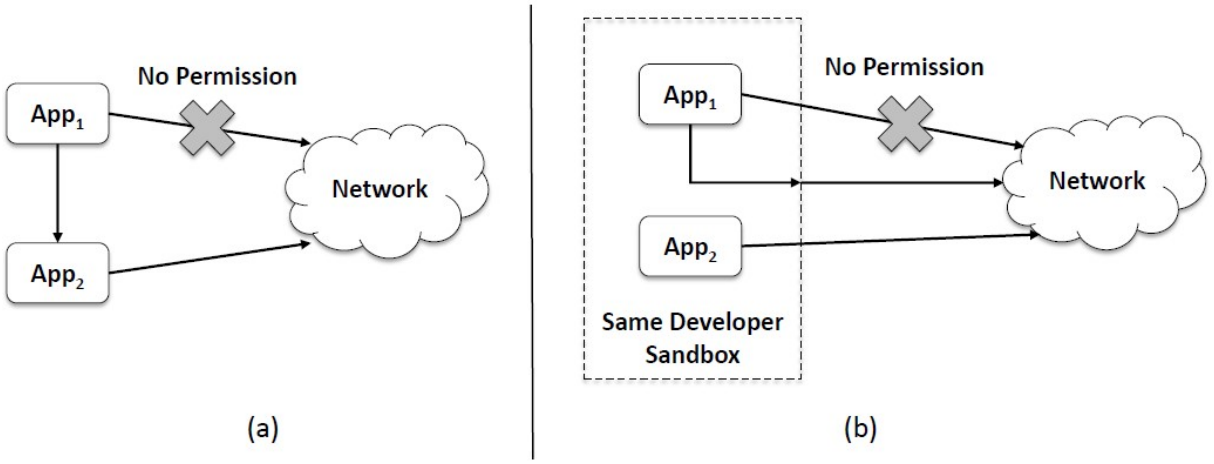


Figure 2.8: Android application communication attacks (confused deputy attack (a), collusion attack (b)) [24].

permissions using a publicly accessible API from another application to access those resources. As Figure 2.8 (a) shows, App1 does not have permissions to access the network, so instead it uses App2's publicly accessible API to do so. The second attack is called the collusion attack. Even with DAC, an application that does not have permissions to perform some operation may still be able to if there exists another application from the same developer installed on the device that has those permissions. Figure 2.8 (b) illustrates this: while App1 does not have permissions to access to the network, it was still able to do so because App2, belonging to the same developer, was installed on the device and has permission to access the network.

Confused Deputy Attacks typically happen if an application exposes in its API a service that allows a second application to receive photos directly from the camera. even if this second application does not have camera permissions, it is able to access the photos. Saint [37] enforces policies both at install and runtime, based on application inter-communication. It essentially allows developers to assign security policies to their APIs in their manifest file, specifying which applications can access them. Any alterations to these policies requires modifying the manifest file and reinstallation.

An example of a Collusion Attack is a malware developer that creates a guitar tuner application, which requires microphone access permissions. He can then persuade the user to install an add-on application alongside the tuner. If this add-on application has Internet access permission, it may collude with the original tuner application and leak sensitive audio data. XManDroid [38] aims to regulate how applications interact with each other by building a graph at runtime, representing interactions between applications, which is then put against the system's policies to regulate inter-communication.

Analysis: These works give a good insight to application inter-communication problems, and how to mitigate them. However, they don't contribute to developing secure applications. For this reason, they are not of particular interest to secure HCE Mobile Ticketing.

2.3.5 Privacy Enhancement Systems

Much work has been done on the management of personal user data. This area includes extensions [39, 40] that allow users to manually specify how applications are able to access data. Unauthorized applications that try to access data get empty or incorrect information instead on the correct one, which is known as data shadowing. myTunes [41] aims to protect not only sensitive data, but Android data structures as well, by establishing semantic links which represent relationships between Content Providers and the system's services that manage such data. The goal is to prevent an application that does not have permissions to access some data to be able to retrieve partial information if it has access to some other application that has access to that data. By linking both applications, if the application does not have permissions to access one, it is also incapable of accessing the other.

Another group of systems apply Dynamic Taint Analysis in order to prevent data leakage. TaintDroid [42] is such a system, using dynamic taint analysis to taint sensitive data, which is then monitored as it goes through the system. If the tainted data reaches a taint sink (such as a network interface, meaning that is about to leave the system), it alerts the user. TaintDroid has some disadvantages, as it can't track many information sources and may trigger false negatives. It may also be too cumbersome for some systems and is not able to detect attacks that make use of covert channels in order to leak data.

Analysis: These systems aim to protect user's private data and thus their goal is mainly to prevent data leakage. However, they still suffer the same problem has the previous presented systems, relying on the integrity of the OS to functional properly. For that reason, these systems are not particularly useful to protect HCE Mobile Ticketing.

2.3.6 Access Control Hook APIs

Many of the extensions we have presented so far require modifying and adding components to the middleware or kernel layers. It is then helpful to develop frameworks whose goal is to ease the development of these security extensions, possible without requiring alterations at the middleware or kernel layers. One way to do this is to develop regular applications that leverage a callback system that alerts when some access is triggered. Figure 2.9 shows the general architecture of an Access Control Hook API, consisting of a set of hooks in the kernel and middleware layers, where this last one controls the applications' lifecycle, system services, accesses to sensitive, and system resources, like camera. A secure application can register for a hook that, whenever is activated, triggers a callback to the Hook API Module, that in turn forwards it to the application, which will then decide if the operation that triggered the hook can proceed or not. This works very similarly to DAC or MAC models, except at an applicational level.

Two examples of these extensions include Android Security Modules (ASM) [44] and Android Security Framework (ASF) [43]. With ASF, developers can create reference monitors, which in turn can be embedded on applications as a programmable API. However, it suffers from limitations ASM does not, as this one enforces restrictions to maintain security guarantees, such as sandboxing integrity, by

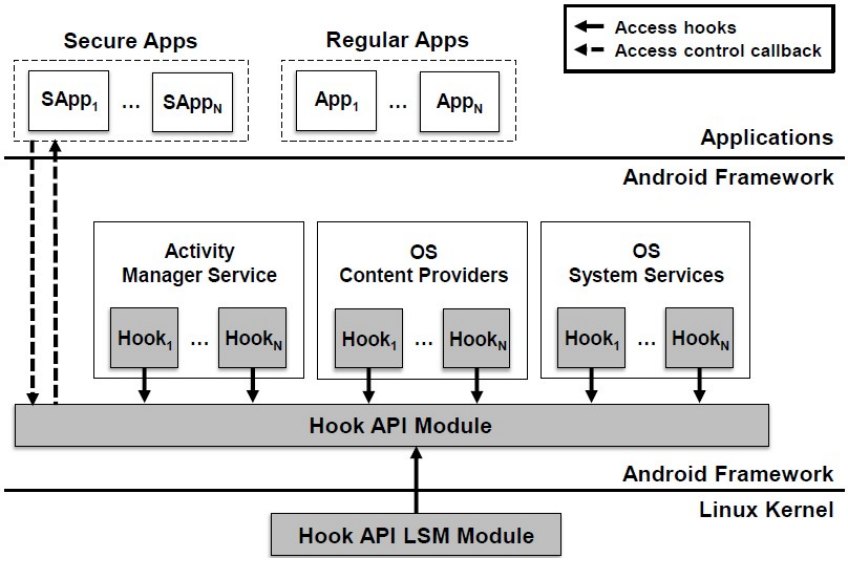


Figure 2.9: General access hook API architecture (adapted from [43]) [24].

providing a reference monitor interface hooks that restricting the system more than what's currently in place. This can mean, for instance, not allowing as many file accesses. Given that ASM restricts the systems further, it contrasts with ASF as this last one allows to load kernel modules, for instance, by trusting the module writer.

Analysis: These systems at helpful at restricting accesses to some resources in a flexible fashion. However, they don't provide any means to secure data, making them an unfeasible solution to HCE Mobile Ticketing's problems.

2.3.7 Memory Instrumentation

Memory Instrumentation systems allow to extend Android's security without requiring alterations in the source code. One type of these systems are the ones that employ static memory instrumentation, which essentially consists in developing an application that can monitor itself by embedding a reference monitor, which has access to the entire internal state, being then able to increase the application's security according to the information it processes by rewriting the vulnerable parts of the application's bytecode. Aurasium [45] is such a system, being able to run these rewriting services through the cloud, while also supporting white/black-listing of IP addresses the application is allowed to access and being able to detect runtime privilege escalation.

Another approach is using Dynamic Memory Instrumentation, consists in correcting processes residing in memory. An example of such implementation is PatchDroid [46], which essentially aims at providing support to devices that have stopped receiving updates, or that receive them late. Considering that the time to update devices is many times dependent on the manufacturer, with PatchDroid a security analyst is able to provide a patch to some vulnerability, which he then submits. The patch would later be deployed on users' devices.

Analysis: These systems extensions that do not require altering Android's source code. However, they work mostly at the middleware layer, thus relaying on the integrity of the underlying OS, and don't help developers building secure applications. For these reasons, they are not particularly useful for protecting HCE Mobile Ticketing.

2.4 Trusted Execution Environments

In the last section, we showcased several mechanisms that aim to improve the security of mobile devices. In this section, we will present another security mechanism, which is singled-out because it is the one we will be focusing on to protect the HCE Mobile Ticketing application. To overcome the security vulnerabilities of this application, we propose to leverage Trusted Execution Environments (TEE). This section gives an overview of TEEs, starting in Sections 2.4.1 and 2.4.2 by describing what TEEs are and providing a specific harware for TEE implementation, ARM TrustZone. Next, in Section 2.4.3, we present existing mobile systems based on TrustZone. The chapter ends in Section 2.3 by providing a list of alternatives for protecting mobile applications.

2.4.1 What are Trusted Execution Environments

TEE is a secure protection domain in the processor or coprocessor of a mobile device where data can be securely stored and processed. It allows for authorized security software (trusted applications) to be safely executed in a trusted environment. TEE technology appeared to address the emerging security needs driven by the trends in mobile computing.

In fact, mobile devices are increasingly more present in people's lives and are used for a multitude of tasks, such as social interactions, business, browsing the internet, among many others. This generates much data, which is susceptible to be attacked or maliciously used. Applications need to be protected against various types of concerns, like attacks originating at the operating system level, correct authentication of users, privacy and protection of data. To protect from these security challenges, it is necessary to be able to provide a small, isolated execution environment that allows developers to increase user experience while also protecting them and themselves from fraudulent use. The Trusted Execution Environment is the solution for these concerns.

In particular, TEE enabled by TrustZone hardware consists of a secure, isolated processor execution mode available in modern ARM-based smartphones. By having data stored in this trusted and isolated environment, it assures that it can be processed securely and remain protected. The possibility to execute sensitive software, known as "Trusted Applications", in a secure and isolated environment shines as the main advantage of TEE, as it enables end-to-end security by enforcing protected execution of authenticated code, confidentiality, authenticity, privacy, system integrity, and data access rights. It is also better in comparison with other security environments as it offers high processing speed and a considerable amount of accessible memory.

There are two main reasons why a solution such as TEE is required: (1) an increasing number of

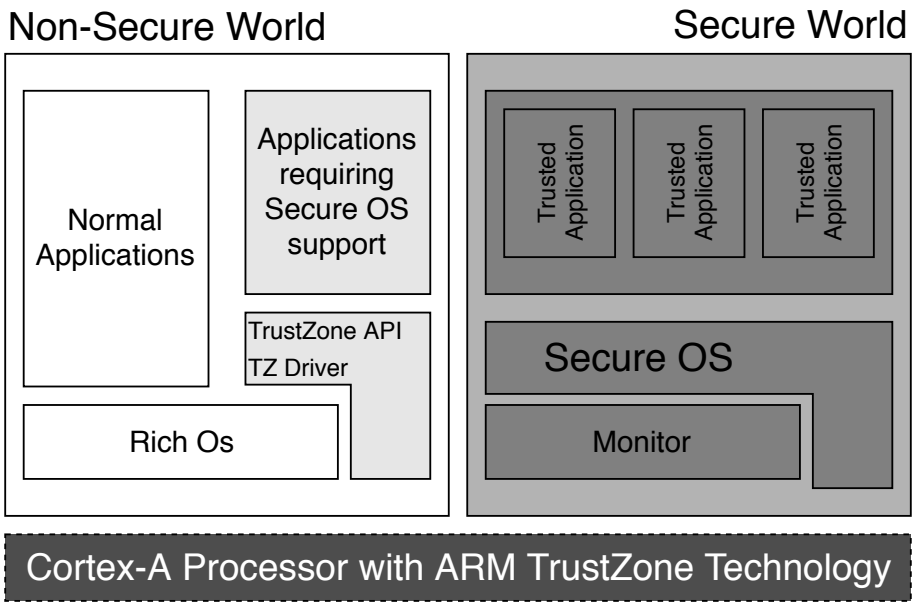


Figure 2.10: ARM TrustZone's stack.

emerging mobile applications, which require a continuously greater level of security, and (2) a greater need for protection against software attacks. Regarding application areas for TEE, we highlight the three major topics:

1. Digital content such as films, television, music and other types of multimedia formats. Download of encrypted movies and episodes from a streaming service such as Netflix, using TEE to decrypt and store them.

2. mCommerce and mPayments credentials and transactions. To store cryptographic keys while a transaction is being authorized.

3. Enterprise and government data. Protection of corporate or government sensitive data, as Bring Your Device is becoming ever more popular.

We propose to leverage TEE to mitigate the security vulnerabilities which will be discussed in Section 3.1.

2.4.2 ARM TrustZone

TEEs depend on specific underlying technology for providing its security features. One of the most popular such technologies is ARM TrustZone. ARM TrustZone technology is a System on Chip (SoC) and CPU system-wide approach to security. TrustZone is hardware-based security built into SoCs by semiconductor chip designers who want to provide secure endpoints and a device root of trust [47]. The main idea behind this technology is the division between a non-secure non-secure world (NW) and a secure world (SW) that are separated via hardware. The non-secure world is where the OS and most software will be running, whereas the secure world is a zone where secure software will be running without having their resources directly accessed by non-secure software. This division is executed by

the processor, and a change between both worlds is achieved via a secure monitor (Cortex-A) or core logic (Cortex-M). This notion of secure world goes beyond the processor, with it also englobing memory, software, interruptions, among others, which leads to the creation of a trusted platform where software is able to execute without being vulnerable to most software and hardware attacks.

TrustZone technology is described in detail in a whitepaper published by ARM [48], and is illustrated in Figure 2.10. It can run inside Cortex-A based application processors to run trusted boot and trusted OS to implement a Trusted Execution Environment (TEE), whose objective is to provide the security mechanisms to protect against attacks described in Section 3.1. Applications running in the secure world are called Trusted Apps. The division between non-secure and secure world is done via hardware logic. Each physical processor has two virtual cores, one for each world, and a security mechanism is provided to context switch between them (Secure Monitor exception). The entry to this monitor can be triggered by software that is executing a dedicated Secure Monitor Call (SMC), or by any number of exception mechanisms. The state of the current world is typically saved by the monitor, which then restores the state of the world it's changing to. Trusted software needs to be developed to use the protected assets in order to implement a secure world on the SoC. This software usually implements trusted boot, the secure world switch monitor, a small trusted OS, and trusted apps. The join between the hardware isolation, trusted boot and trusted OS result in a TEE, which offers confidentiality and integrity to the multiple trusted applications that run on it.

2.4.3 Mobile Security Systems Based on TrustZone

Given the ubiquity of mobile devices, it is paramount that manufacturers develop architectures that are enhanced with trusted hardware and that allow for the creation of a Trusted Execution Environment (TEE). We focus on ARM TrustZone, which we already described, that allows for new security mechanisms to be developed based on it, like it is the case of Samsung Knox [49] and DroidVault [50]. However, given that manufacturers are able to develop their own proprietary versions of such TEE-allowing technology, there is a dire need of standardization. For that, GlobalPlatform [51] has been publishing specifications that permit interoperability and easy porting of applications based on this technology. To further promote confidence in this ecosystem, GlobalPlatform also launched a TEE compliance program that gives assurance to developers and manufacturers that a given TEE product is compliant with their defined specifications.

As Brito [52] details on his thesis work, TrustZone-based systems can be divided into two main groups: Trusted Kernels and Trusted Services.

Trusted Kernels: While the Rich OS controls the non-secure world of mobile devices, it is necessary to have a kernel running in the secure world of TrustZone-based processors to allow for execution of code on their isolated environment. These kernels usually have very similar architectures with a small Trusted Computing Base (TCB) to reduce to risk of vulnerabilities in their code, and present a non-secure world user space client API as well as a TEE device driver that allows for communication between worlds. Genode and OP-TEE are examples of such solutions:

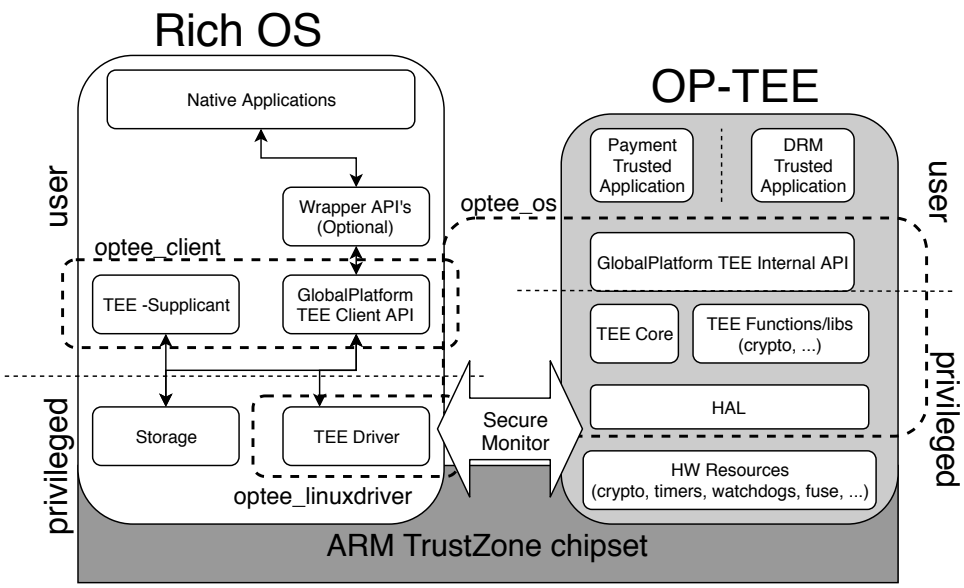


Figure 2.11: OP-TEE's architecture (adapted from [55]).

- **Genode**: Genode [53] is a toolkit that allows for building special-purpose operating systems based on a recursive system structure, where each program runs in a dedicated sandbox having only the access rights and resources it needs for its specific purpose and being able to manage sub-sandboxes out of their own resources. It follows the microkernel abstraction and is one of the few open-source OS that does not derive from Unix. By offering several small building blocks, it allows one to compose a custom system.

  Genode offers a TrustZone aware microkernel *base-hw* [54] that provides a client application that manages a rich OS running on the NW and an SW manager. Because it resembles a Virtual Machine Monitor (VMM), we can call this client application TrustZone VMM (TZ VMM). World-Switching is done by defining a special System Call that will be handled by the VMM. CPU state is saved upon exiting either world and is restored by the monitor. Inter-world communication is possible because the SW has more privileges than the NW: given that it can access CPU registers, stack pointer and a pointer to the start of the NW's memory, it is possible to create a mechanism that allows to read directly from the NWs memory and write to it.

- **OP-TEE**: OP-TEE is an open-source project, owned by Linaro [56], that allows for the full implementation of a Trusted Execution Environment, possible on ARM Cortex-A cores using the TrustZone technology. By providing both the TEE Internal Core API v1.1 [57] as well as the TEE Client API v1.0 [58], defined by the GlobalPlatform TEE specifications, OP-TEE aims to deliver a standardized implementation of a TEE, which makes it easier for developers to create and test their own solutions. The readability of the available documentation is also a great argument in favor of OP-TEE, as it allows for a newcomer to easily learn the basics and get support through the repositories.

The main goals of OP-TEE [59] are then as follows:

- *Isolation* - By providing a TEE that is isolated from the non-secure Rich OS. Trusted Applications are protected from each other through underlying HW support;
- *Small Footprint* - The TEE must have a small TCB to allow for it to reside reasonable amount of on-chip memory as found on ARM based systems;
- *Portability* - OP-TEE, following the Global Platform's specifications, intends to be easily pluggable to different architectures and available HW. Additionally, it supports setups having multiple client OSes or multiple TEEs.

OP-TEE's source code is available on GitHub and is divided into two essential parts: optee_os [60], containing the source code of the TEE OS itself, which specifies the secure privileged layer that hosts the Trusted Applications; and optee_client [61], which contains the source code for the TEE Client Library, allowing for the Rich OS (Linux) to access the TEE OS via the GlobalPlatform TEE Client API Specification v1.0. Its architecture is shown in Figure 2.11

Additionally, the Linux Kernel repository [62] contains the source code for the OP-TEE Linux driver. OP-TEE also offers the source code for test materials [63], a build repository [64] containing premade build instructions for certain platforms and an examples repository [65] which provides a number of TEE Client and Trusted Application examples, as well as accompanying documentation, to help with application development.

Trusted Services: Contrary to the Trusted Kernel approach, where executing general-purpose application code on the SW is allowed, Trusted Services implement specific applications on the SW natively. One such example is Android KeyStore [19], which was already described in Section 2.2. The keystore file that is stored in the application data, ciphered with a key managed by the Android KeyStore service and which is kept in the TEE. Cipher and decipher of this file is completely managed by this service, backed by the TrustZone. Another example is DroidVault [50]. DroidVault offers an isolated data protection manager that runs on the TEE for secure file management in Android, by adopting the memory manager and interrupt handler from Open Virtualization's SierraTEE [66]. It offers a data protection manager, an encryption library and a port of PolarSSL [67]. The user can then download a sensitive file and store it securely in the device through DroidVault, which ciphers and signs it via its data protection manager. DroidVault is very similar to the solution that we'll present in Chapter 3, but differs in one key aspect: while DroidVault is based on file management, our solution will be based on a SQL database and will allow operations over said database via queries that can be arbitrarily chosen. As such, no actual code will ever run in the TEE besides the SQL queries. This flexibility and novelty is what distinguishes our solution from related work in this field. TEA [68], is another Trusted Service, but is more focused on conducting NFC transactions directly on the TEE and protected from the untrusted Rich OS. Both these solutions are interesting but can be problematic, as they allow for arbitrary code execution in the TEE, thus enlarging the TCB.

Additional work done in the field of TEE and, more specifically, on TrustZone, was done and can be put in the perspective of the work we're doing and the solution we'll present. There are systems that provide a one-time password solutions [69], some that provide secure authentication mechanisms [70] and others that aim to establish trusted I/O channels between user and TrustZone-based services [71]. These solutions are interesting, but they don't solve directly the problems we're facing. They can, however, be used as additional mechanisms to make our solution stronger.

2.4.4 TrustZone Security Issues and Vulnerabilities

While TrustZone is a technology that aims to enhance the security of applications by providing an isolated environment that is protected even in the presence of a compromised Rich OS, it may itself be victim of potential attacks. While attacks to TrustZone are not in the scope of this work, we consider interesting to detail just how this technology can be compromised. To do so, we divide this section into the two types of vulnerabilities that can be exploited: TEE-related vulnerabilities and hardware-related vulnerabilities.

TEE-related Vulnerabilities

There are a number of reported vulnerabilities related to TrustZone and TrustZone-based TEEs, some of which can be found on the National Vulnerability Database (NVD), as well as several security bulletins, with most being due to existing bugs in the implementations of the TEE kernel or TEE drivers of some providers, leading to lack of input validation, buffer overflows and race conditions. One such vulnerability [72] could be found on Qualcomm's Secure Execution Environment (QSEE), which failed to properly verify bounds of coming SMC requests. As such, an attacker was able to create a SMC that would cause an overflow, forcing QSEE to write part of that SMC to a secure memory region. With this attack, the attacker could run code inside the TEE and compromise the entire internal interface. Another attack [73], but this time on Huawei devices, exploited privilege escalation to gain root privileges in the NW, allowing an attacker to compromise the TEE by being able to retrieve data from it or even load non-trusted applications to it.

Hardware-related Vulnerabilities

Another type of attacks lie on the hardware supporting the TEE; more especifically, vulnerabilities in the components constituting the platform's root of trust, caches and power management mechanisms. Regarding the root of trust, there is a lack of standards defining how trust can be established in TrustZone-based systems. One proposed solution is based on Physical Unclonable Functions [74], which are essentially fingerprints based on physical variations during the manufacturing of semiconductors and that can be used to create a unique key, which serves as the root of trust. ARMageddon [75] proposes a number of attacks to caches of ARM devices. Although they were not focused on TrustZone-enabled caches, they concluded that it was possible for an attacker to monitor the SW's activity from the NW by inspecting their content. Another range of attacks exploit the energy management mechanisms of

processors, as they typically do not have security in consideration. The CLKSCREW attack [76] is able to break TrustZone-enabled devices, and thus allow the retrieval of cryptographic material or even loading of non-trusted applications, by exploiting Dynamic Voltage Frequency Scaling (DVFS), in conjunction with a malicious kernel driver. If the attacker is able to identify the frequency voltage limits of some system, he can then push the processor beyond those values, thus enabling fault attacks conducted merely via software.

This section illustrates some of the vulnerabilities TEEs may be subject to. It is then easy to conclude that although the security design of a TEE may be correct, the implementation may contain bugs that can be exploited by an attacker. It is then important to develop tools that can verify the source code of TEE implementations against some of the vulnerabilities we described, while also providing means for the TEE to protect itself dynamically and possibly recover from attacks [77]. In this work, we assume that the TEE is correct, and as such it is not our goal to provide solutions to these issues. However, much research has been conducted in the past years, namely in the light of some of the attacks we mentioned, which are leading to more secure TEE implementations.

2.5 Summary

We began this chapter by providing the background of existing ticketing solutions based on contactless smartcards, and provide the example of Suica. We then analyzed the rise in popularity on solutions that leverage the capabilities of mobile devices and the advantages that they bring for both client and operator, such as easier and more convenient access to Public Transports, and investment protection in fare management infrastructures, respectively. We focused on a technology available on Android named Host-Card Emulation, which allows a mobile device to emulate a contactless card without requiring an SE.

Next, we presented an example of such an application, which we named HCE Mobile Ticketing, to illustrate how these type of applications are typically built and the services they require. The idea is to contextualize the work we will be doing by presenting the subject and the problem. We detail the architecture of this system, including the application itself and the operations it supports – Purchase, Recharge and Validation –, the data structures it keeps and the security mechanisms put in place to secure the data against a malicious user.

After we presented the background of this work, we show additional solutions to overcome some of Android's flaws. These solutions fall into seven categories: digital rights management, (ii) access control mechanisms, (iii) permission refinement, (iv) application API security, (v) privacy enhancement systems, (vi) access control hook APIs, and (vii) memory instrumentation. The goal is to illustrate that although these are solutions to some of Android's problems, they are not a reasonable solution to our problems with HCE Mobile Ticketing and why we've chosen TrustZone over them.

This chapter ends with an analysis of Trusted Execution Environments, which is another security

mechanism for protecting mobile applications and the one we will be focusing on to protect HCE Mobile Ticketing. TEEs essentially divide the device into two parts: a non-secure world, where the Rich OS and traditional applications run, and a Secure World, where a Secure OS and trusted services run. This strict isolation guarantees that even in the presence of a compromised Rich OS, the trusted services and their data being kept on the SW remains secure. TEEs can be of two types, Trusted Kernels or Trusted Services. The first is the kernel running on the SW and is responsible for monitoring everything that occurs inside it. Typically they have a small TCB as to reduce the risk of vulnerabilities in their code and present a non-secure world user space client API as well as a TEE device driver that allows for communication between worlds. Genode and OP-TEE are examples of Trusted Kernels and were used to develop both of our prototypes. The second, the Trusted Services, Trusted Services implement specific applications on the SW natively, such as the case of Android KeyStore. We also present some possible attacks on TEEs, either in the implementation itself or on the supporting underlying hardware.

All these systems that aim to enhance the security of mobile devices have limitations, and as such motivate the design of a new solution that can overcome some of them. This is the focus of our next chapter.

Chapter 3

Design

This chapter focuses on the design DBStore, a database management system backed by TrustZone, and how it can tackle the vulnerabilities of the *HCE Mobile Ticketing* application. We start by presenting a study of the vulnerabilities in Section 3.1 and possible attacks that are possible because of them. Next, in Section 3.2, we present the architecture of DBStore. Finally, in Section 3.3, we analyze how DBStore can be integrated with the HCE Mobile Ticketing application to improve its security.

3.1 Vulnerability Analysis

In Section 2.2, we provided the description of a typical HCE Mobile Ticketing application, with particular emphasis on the security mechanisms adopted at the mobile endpoint to secure the ticket-related data. This section aims to perform a systematic analysis of potential security vulnerabilities in the light of the currently implemented mechanisms. We start by describing the threat model that we consider in our analysis. Then, we present a preliminary list of attacks that an adversary can launch on the mobile endpoint. For each attack, we provide a brief description, while also analyzing its potential impact on the normal system behavior. We also sketch how to replicate each attack experimentally. However, it is part of future work to test these attacks in order to demonstrate their feasibility and assess their difficulty.

3.1.1 Threat Model

As proposed by Joeri de Ruiter et al. [22], three relevant attack models can be defined in order to properly analyze the security properties of HCE-based solutions:

- **Malicious app attacker:** An attacker tries to compromise the keystore used by an application through another application that he/she has installed on the device. This application can be downloaded from an official app store, such as Google Play Store or Amazon's store.
- **Root attacker:** The attacker has root credentials and can run applications with root permissions. He's also able to inspect the file system. He can have gained root permissions on the device through an exploit or he's just able to run an application with root permissions.

- **Intercepting root attacker:** The attacker has the same capacities of a *root attacker*, but can also capture user input.

For the scope of this work, we consider an adversary with the capabilities of a *intercepting root attacker*, which consists of the strongest of these profiles. More concretely, the adversary of an HCE Mobile Ticketing application can have total control of the Android OS running on the mobile device through a root exploit. This allows him to have access to the application's data and modify its contents. This is particularly worrisome because the validation process is done solely offline. If the attacker is able to modify the contents of the card he can use it indefinitely.

We also assume that the attacker has the ability to eavesdrop or tamper with the communications between the client endpoint and the server. Since the current system architecture establishes a secure channel between both endpoints using HTTPS, as of now, we dismiss potential attacks involving network communications. However, we plan to study this potential attack vector with greater detail in the future.

Some necessary assumptions need to be made. We consider that the hardware of the mobile device is correct, as well as the cryptographic libraries and algorithms it uses. Additionally, our analysis and solution will not contemplate side-channels or attacks that involve tampering with the hardware.

3.1.2 Attack 1: Rollback Keystore Files and Card Data

In an HCE mobile ticketing system, the card data structure is encrypted with a key K. This key is itself encrypted with the public part of a keypair, which in turn is persisted in an Android keystore file located in /data/misc/keystore/user_0. The keypair files are encrypted with a master key that is maintained inside the TEE and that cannot be retrieved by typical means. Both encrypted items, ticket data and key K are stored on the application data folder. Thus, an adversary with root privileges can backup all these files, and replace them with an older copy, so as to bring the card back to a previous state where the credits were positive, which enables an attacker to travel using a single card for as long as its serial number is not blocked by the system. In terms of the impact to the application security, device-binding is guaranteed, as the master key that encrypts the key pair is device-specific. This means that while data can be reused in the same device, it cannot be used in another device as the key will be different. The reuse of this data allows an attacker to travel using a single card for as long as its serial is not blocked by the system.

This attack can be replicated as follows, assuming the user already obtained root privileges:

1. Purchase a ticket normally through the application.
2. This ticket will be encrypted through a key pair that will be stored as a AndroidKeyStore type. As such, this keystore is encrypted with a master key that resides in a trustlet in the TEE. The attacker then needs to go to */data/misc/keystore/user_0*, discover which files are associated with the HCE mobile ticketing app and copy them. He has to do the same regarding the card information. To do so, he needs to access the application data and copy the card.
3. The attacker can then freely use the card until it runs out of balance.

4. At this point, the attacker only needs to replace the keystore and the card with the ones that he copied before. The keystore will contain the initial key pair that encrypted the card and the card will initially be encrypted with that key pair (see Section 2.2.3). While the attacker never knows the master key, he does not need to as the decryption of the copied keystore will be done automatically by the OS.

3.1.3 Attack 2: Obtain the Keypair from Another Application

This attack leverages the fact that Android keystore preserves the keypairs of a given application in two files: "(App's User ID)_USRPKEY_(Key Entry Alias)" and "(App's User ID)_USRCERT_(Key Entry Alias)". The owner of the keypair is identified by the (App's User ID) bit. The attack is then relatively straightforward. First, the adversary creates a KeyStealer application aimed to retrieve the keypair of the mobile ticket app from the keystore file and write it elsewhere unencrypted; with that keypair, K can be obtained and hence the raw card data. Then, it is only necessary to change the User ID of files generated for the HCE mobile ticketing app to match KeyStealer's. Android keystore will be tricked into believing that the KeyStealer app owns that keypair and will decrypt it using the master key. In terms of impact, the hardest part is to create the KeyStealer app. For the keystore files, a simple *cp* command is necessary. This attack then becomes relatively simple, and with very damaging consequences. If the attacker can easily *steal* the keys from the keystore, he can freely decrypt the cards and modify them. It is then possible to share cards as long as they are encrypted again with their corresponding keys.

This attack can be replicated in the following manner, assuming the user obtained root privileges:

1. Purchase a ticket normally through the application.
2. The attacker then needs to develop a KeyStealer app to steal the keypair from the keystore.
3. This app needs to access its keystore via AndroidKeyStore. However, its keystore will be the one from the HCE mobile ticketing system. To do so, two simple cp commands would suffice [22]:

   cp 10102_USRCERT_TKP 10101_USRCERT_TKP

   cp 10102_USRPKEY_TKP 10101_USRPKEY_TKP
4. Then, with the app having access to the keypair, it only needs to output them, e.g., to a file.
5. With the keypair, the attacker can decrypt the key K and thus the cards. He can change the contents or simply replace it with another card and encrypt them.

3.1.4 Attack 3: Install a Modified Mobile Application

Reverse engineering is always a threat to software. By using software like dex2jar [78], Apktool [79] and JD-GUI [80], an attacker with enough skill and patience could modify the application in various ways, such as not discounting balance upon each validation. By publishing this modified version online, and knowing that the validation is done offline, validators would not be able to distinguish between a

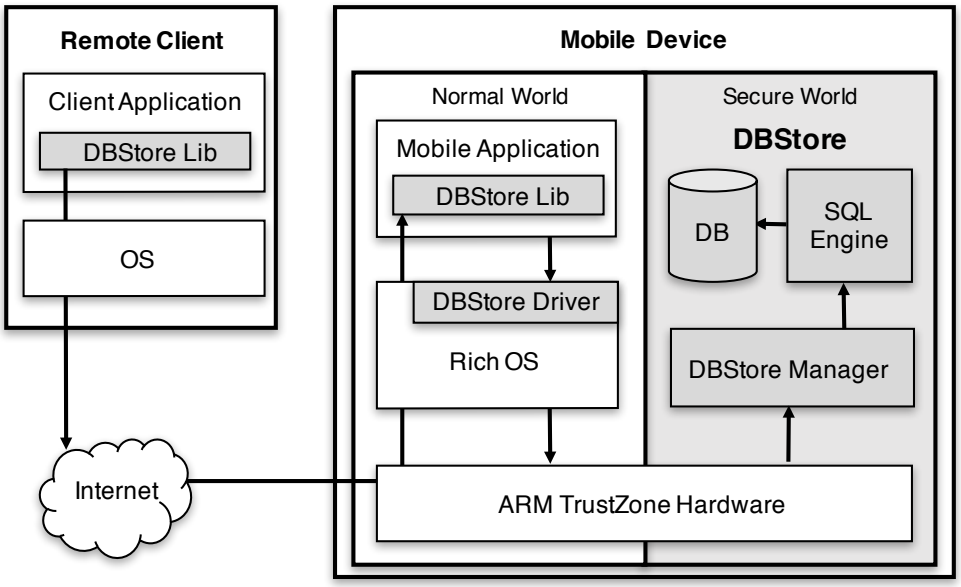


Figure 3.1: DBStore architecture: arrows represent invocation flow of an SRPC.

correct and a modified application. With regards to the potential impact we note that, although it would be difficult to reverse engineer the application, it is not impossible. With the tools referred above, the attacker could pretty much recreate the original project. Even with the code obfuscated with ProGuard [81], a particularly skilled and patient attacker could find the segments of code responsible for key operations, remove them and recompile the APK. This modified APK could be put online for anyone to download and install.

This attack can be replicated in the following manner:

1. The attacker would download the HCE mobile ticketing app's APK.
2. Using dex2jar and Apktool, he would be able to get the corresponding .jar and Manifest files, respectively.
3. Using JD-GUI, the attacker could convert the .class files in the .jar to .java files.
4. Using an IDE, like Eclipse or Android Studio, the attacker could recreate the original project with all these obtained files.
5. After an unknown amount of time, the attacker finds some important segment of code, removes/-modifies it and recompiles the application.
6. He then published the APK online.

3.2 DBStore Architecture

The attacks we previously presented are possible because HCE Mobile Ticketing, as with most applications that need to manage sensitive data locally, rely on the integrity of the OS to secure their data. Then, to protect the security-sensitive state of mobile ticketing applications and defend against such

attacks, we have developed DBStore, leveraging the isolation provided by TEEs. Figure 3.1 presents a high-level architecture of DBStore. In the scenarios we envision, distributed applications consist of two parts: a *client side* which is kept under the control of the application provider, and a counterpart *mobile side* which runs on users' devices. Depending on the specific application, the remote client can be implemented, e.g., by a remote cloud server, a nearby embedded device, etc. The mobile application can solicit storage space at the DBStore for storing dababases. The client application interacts with the mobile application by submitting SQL commands to the remote databases hosted inside the DBStore. The databases are securely maintained by the DBStore, which has exclusive access to the databases' state and is responsible for the execution of the SQL statements submitted by clients. It runs inside a TEE in isolation from both the local OS and mobile applications. The DBStore ensures that the SQL commands issued by the clients are securely forwarded to and executed on the intended database and the results routed back to the respective client.

The SQL commands are issued to a database residing on a remote DBStore by invoking *SQL Remote Procedure Calls* (SRPC). To this end, an API is provided by the DBStore library. An SRPC comprises one or more SQL statements, such as "CREATE TABLE", "INSERT INTO", and "SELECT" commands, and are performed as a single transaction. An SRPC can be parameterized by inputs provided by the client application, and can optionally return an output to the client application. The DBStore library and DBStore manager implement security protocols which involve the exchange of information in the form of specific Protocol Data Units (PDUs). An SRPC is translated into PDUs that are sent over the network to the mobile application. The mobile application must then issue a request to a DBStore driver in order to forward the PDUs to the DBStore. This driver is required because the context switch to SW requires kernel privileges. The DBStore, then, handles the PDUs, assigns the SRPC commands to the internal SQL engine, and returns the results all the way back to the client.

We consider the same threat model as the one described in Section 3.1.1. More specifically, we want DBStore to provide to applications confidentially and integrity of sensitive data. We use the ARM TrustZone technology, as described in Section 2.4.2, to provide a secure world isolated from the untrusted Rich OS. In our work, the SW is used to implement a TEE where the DBStore resides; OS and mobile applications run in the non-secure world (see Figure 3.1). As a result mobile applications and OS have no direct access to the DBStore state.

Interaction between worlds requires a world switch operation triggered by a special SMC instruction. In our system, the SMC instruction is invoked by the DBStore driver to enter the secure world and by the DBStore to exit and return to the OS. Parameters can be passed in both ways for each SMC call. Peripheral access protection is achieved by configuring interrupts to be handled by a specific world. TrustZone devices equipped with a secure boot mechanism provide integrity and authenticity protection of the SW software. We use this mechanism to check the DBStore binary upon boot. This binary must be part of device firmware and signed by the device manufacturer. When the device is powered, the processor enters SW, initializes the DBStore, and then switches to NW and hands over control to the OS bootloader.

Data Structures	*Meaning*
SQL Databases	Databases containing the applicational data
TK^-	Private part of the attestation keypair embed in the devices firmware, only available to DBStore if it passes secure boot
TK^+	Public part of the attestation keypair
C_{TK}	Certificate issued by the firmware provided, containing the public part TK^+ of the attestation keypair
n_i	Current nounce value, used to guarantee message freshness
SK	First session key generated by DBStore at the end of the initialization protocol
K_i	Current session key, derived by hashing the previous session key concatenated with the current nounce value

Table 3.1: DBStore's data structures.

3.2.1 Data Structures

DBStore requires managing and maintaining a number of data structures for it to function correctly and be able to implement its protocols. The most obvious data structures it needs to maintain are the SQL databases. On them, DBStore is able to store data and operate over it, according to the SRPCs it receives. DBStore does not maintain a list of the databases it manages; instead, that task is left to the remote client, which issues an SRPC over a certain table and DBStore, via the I/O API, is able to access the corresponding table.

Besides the databases, DBStore needs to maintain the cryptographic keys required by its security protocols. The first of these keys is the attestation key TK. The attestation key is a keypair that the device manufacturer must embed into the device firmware. The private part TK^- is accessible within the SW (if the secure boot passes). Considering that the private part is only accessible in the condition that the secure boot passed, it gives the assurance that the DBStore binary running on the SW has not been tampered with and thus it can be trusted. Additionally, DBStore also maintains a certificate C_{TK}, which is issued by the device manufacturer in order to authenticate the attestation key by containing the public part TK^+. Thus, by checking the signature against a legitimately certified attestation key, the client can verify the authenticity of the remote DBStore. After mutual authentication is achieved, communications between the remote client and DBStore are protected by a session key K_i, that is altered upon each exchanged message. K_i is derived by concatenating the first generated session key SK with the current nounce value, n_i. This nounce also has to be stored by DBStore. Table 3.1 sums up these data structures. The use of each of these keys will be detailing in the following sections.

3.2.2 Initialization Protocol

Before deploying databases on a remote DBStore, a client must first verify that the remote peer runs a legitimate DBStore inside a TrustZone-enabled device. Conversely, the DBStore must be able to authenticate the client and implement an authorization policy that prevents other applications from accessing the remote client's databases. To obtain these guarantees, the client and the remote DBStore must first

Initialization Protocol	
(M_1)	C $\rightarrow$ S: $\langle$*AppId*, $n_1\rangle_{AK-}$
(M_2)	S $\rightarrow$ C: $\langle n_1, \{SK\}_{AK+}\rangle_{TK-}$, C_{TK}
SRPC Invocation Protocol	
(M_3)	C $\rightarrow$ S: n_2, $\{$*SrpcReq*$\}_{K_i}$, hmac(K_i)
($M_{4.a}$)	S $\rightarrow$ C: n_2, *Ok*, $\{$*SrpcRes*$\}_{K_i}$, hmac(K_i)
($M_{4.b}$)	S $\rightarrow$ C: n_2, *Fail*
Resync Protocol	
(M_5)	C $\rightarrow$ S: n_3, *Resync*, hmac(SK)
(M_6)	S $\rightarrow$ C: n_3, $\{i,$ *salt*$\}_{SK}$, hmac(SK)

Table 3.2: DBStore Protocols: C: Remote Client, S: DBStore.

perform an initialization protocol by exchanging two PDUs: M_1 and M_2 (see Table 3.2).

To prove that the DBStore service is authentic and resides inside a valid TrustZone-enabled environment, the first message (M_1) contains a challenge to the remote DBStore. This challenge consists of a nonce n_1 that the DBStore must sign with a valid *attestation key* (TK). The attestation key is an asymmetric cryptography keypair that the device manufacturer must embed into the device firmware. The private key TK^- is accessible within the SW only and under the condition that the secure boot passes. Otherwise, if a spurious DBStore binary has been detected upon boot, the system will halt and access to this key will be locked. If the DBStore is able to sign n_1 with such a key, then it means that the DBStore has not been tampered with and is running inside the SW. The DBStore must then sign n_1 with TK^- and send the signature along with a certificate (C_{TK}) that contains the public key TK^+ required to check the signature. This certificate is issued by the device manufacturer in order to certify the authenticity of the attestation key. Thus, by checking the signature against a legitimately certified attestation key, the client can be convinced about the authenticity of the remote DBStore.

To authenticate the remote client and restrict access to the local databases by alien applications, M_1 contains nonce n_1, an application ID and both are signed with the private part AK^- of *application keypair* AK. This keypair must be generated by the application provider and the private part must be kept secret by the client. The public part AK^+ is included in the mobile application package. Upon receiving M_1, DBStore can validate the signature by obtaining AK^+ from the mobile application thus authenticating the client. To ensure that future interactions remain mutually authenticated, a *session key* (SK) is exchanged as a token of successful client authentication and service remote attestation. This is reflected in message M_2 which includes a session key SK created by the DBStore and encrypted with AK^+. This result is also included in the signature by TK^-.

This way, the DBStore can ensure the authenticity of the client endpoint, because only the legitimate client application holds the private key AK^- which will enable the session key to be decrypted. On the

other hand, by checking the signature of M_2, the client can be assured about the authenticity of that session key (i.e., that it was issued within a legitimate DBStore-enabled device) and of the freshness of the session key (because the tuple includes nonce n_1 which has been chosen by the client itself). The session key is then crucial in securing SRPCs.

3.2.3 SQL Remote Procedure Calls

After the initialization, the client can invoke SQL Remote Procedure Calls (SRPCs) on DBStore. Each SRPC generates a message to the DBStore (M_3), which will then return one of two possible messages: $M_{4.a}$ if the request message was well formed, or $M_{4.b}$ otherwise (see Table 3.2). The SRPC request message (M_3) includes field *SrpcReq* which contains the SQL commands and inputs to be executed on the remote DBStore. The *SrpcRes* field is included in message $M_{4.a}$ and contains the execution output to be sent to the client. For confidentiality protection, *SrpcReq* and *SrpcRes* are encrypted with a symmetric key shared between the client and the DBStore. For message integrity protection, the same key is used to generate HMAC codes for the whole message. Nonce n_2 helps the client detect replay attacks of old M_4 messages.

Although the symmetric key shared between both endpoints is the session key SN, this key is not used in messages M_3 and M_4. Instead, the client creates a new symmetric key K_i for each invoked SRPC. K_i is generated by calculating a hash of SK concatenated with a counter value i, which is initialized to zero and incremented for every SRPC request issued by the client. To determine K_i, the DBStore must also keep a local counter for the i value. To calculate the key of an incoming SRPC message, all the DBStore has to do is to append i to SK, and compute the hash of the result, yielding K_i, which is then used to decrypt SRPC $\#i$ and encrypt the corresponding SRPC response. After submitting the response to the client, the DBStore increments its local i counter.

This mechanism serves two purposes. First, it prevents replay attacks of old SRPC requests. Since the DBStore uses a single key K_i for each request, then old messages (i.e., corresponding to request numbers less than i) cannot be decrypted. Furthermore, the i counter of the DBStore is implemented using a TrustZone monotonic counter, thereby preventing an adversary from decrementing i. This mechanism also avoids overuse of the session key SN. By generating a new symmetric key K_i, we reduce the exposure of SN thereby enhancing the overall system security.

If for some reason the client gets out of sync with the DBStore about the current counter value, and the legitimate client sends a request encrypted with K_j different than K_i expected by the DBStore (i.e., $j \neq i$), the DBStore will not be able to properly decrypt the SRPC request field of message M_3 and then returns a failure message $M_{4.b}$. To overcome this issue, the client triggers a resync protocol by which the DBStore will return the current i value encrypted with the session key (see M_5 and M_6 in Table 3.2).

3.2.4 Protection of DBStore Databases

To secure the DBStore databases at rest, we leverage the secure storage allocated to the SW. We use additional TrustZone mechanisms to prevent rollback attacks and protect data confidentiality. In particu-

lar, we use secure monotonic counters to keep track of the most updated version of DBStore databases. Every time a database is updated, its version number is incremented and written into the database, and the secure monotonic counter is also incremented. If the device is rebooted and the database is replaced with an older version, the version number of the recovered database and the current value of the counter will be inconsistent, forcing DBStore to raise an exception. To preserve confidentiality, the DBStore databases (and associated version numbers) are encrypted with a symmetric key which is then sealed, i.e., encrypted with a key which is released only if the secure boot is successful. This prevents an adversary from retrieving the encryption key and decrypt the database.

3.3 DBStore to Protect HCE Mobile Ticketing

The attacks described in Section 3.1 are possible because the security depends on the integrity of the OS. This section describes how to thwart these attacks using DBStore. In our solution, rather than preserving the virtual card data inside an encrypted file (and the respective encryption key secured in Android keystore), this data will be secured inside a DBStore database. The DBStore clients will be two: the ticketing server and the validation reader devices. An application keypair (see Section 3.2.2) must be generated: the public part is included in the mobile app package and the private part is secretly maintained by both ticket server and reader devices.

We now detail how to integrat with DBStore comparing with what we presented in Section 2.2, namely in regards to the three main operations available, Purchase, Recharge and Validation, and the SRPCs we propose. It is important to note that the HCE Mobile Ticketing has to be modified and the Ticketing Library (see Figure 2.5) replaced with the DBStore Lib (see Figure 3.1). As such, the new architecture for HCE Mobile Ticketing will be as shown in Figure 3.2.

3.3.1 Initialization

```
-- SQL code to initialize a mobile ticketing DB
DROP TABLE IF EXISTS Tickets;
CREATE TABLE Tickets(SN INT, Type TEXT, Credits INT);
```

Listing 3.1: SQL instructions for HCE Mobile Ticketing Initialization.

The user starts by downloading HCE Mobile Ticketing from Google Play Store, and installs it on his device. Afterward, it is time to run it for the first time. Creating an account works almost as was described in Section 2.2.1, with the key difference being the execution of the Initialization Protocol at the end of the registration, described in Section 3.2.2. This establishes an authenticated connection between the local HCE Mobile Ticketing application (and thus, the local DBStore) and the remote client, the Mobile Server. After this channel is established, it is then possible for the remote client to issue SRPCs following the protocol shown in Section 3.2.3, starting by initializing a DBStore table which will be able to hold the card data, which we named ”Tickets”. Listing 3.1 shows the SQL commands required to initialize the ticketing

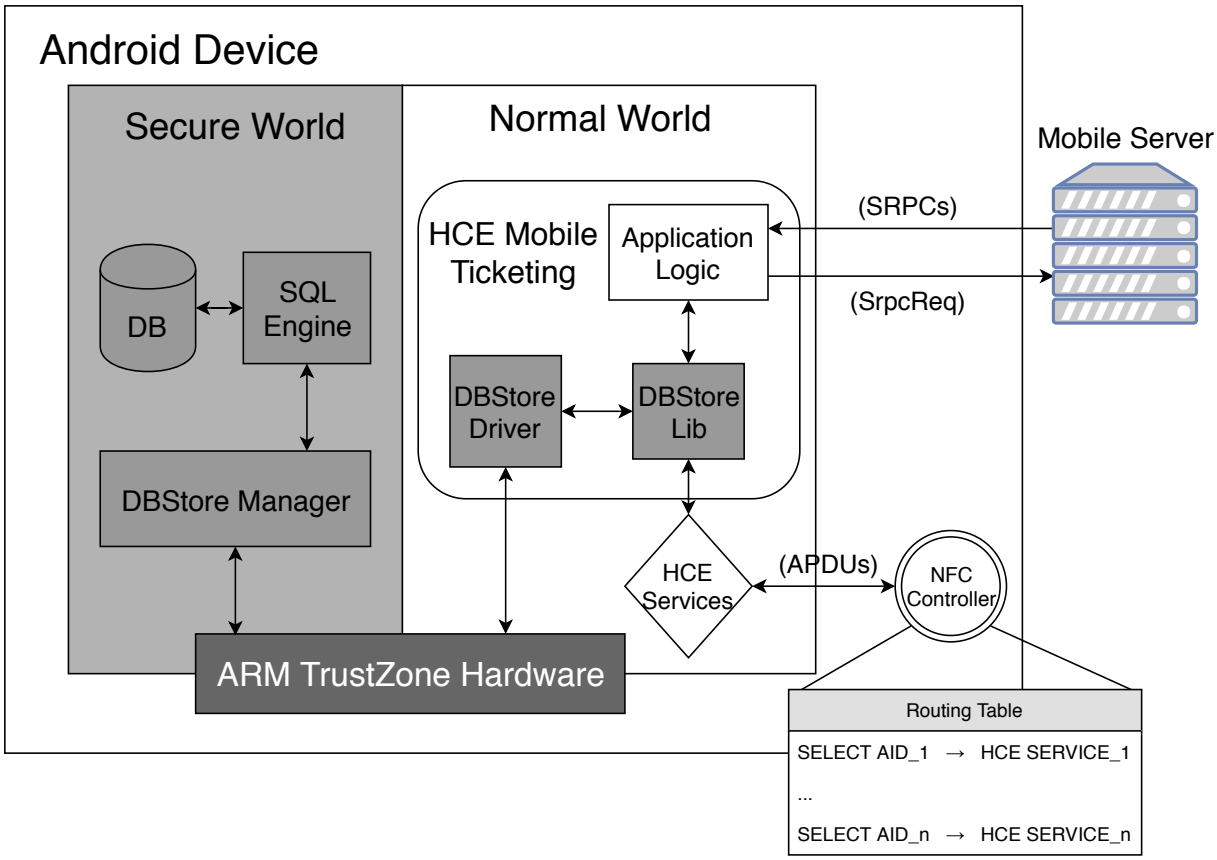


Figure 3.2: HCE Mobile Ticketing's architecture when integrated with DBStore.

table. First, it verifies if a table with the same name already exists, and if so deletes it. Then, it creates a new table able to keep the card information. In this simplified version, the table consists of rows, each representing a card. A card consists of an integer as the serial number, a string defining what type of card it is and an integer that indicates the amount of credits remaining. After this initialization is terminated, it is then possible to start a Purchase operation and add a new card to the table, e.g., a new row.

3.3.2 Purchase

```
-- SQL instructions for card purchasing
INSERT INTO Tickets VALUES(@sn, @cardtype, @credits);
    WHERE SN = @sn;
```

Listing 3.2: SQL instructions for HCE Mobile Ticketing Purchase.

The Purchase operation consists of HCE Mobile Ticketing communicating with the Mobile Server, which talks with the Ticketing Portal, via HTTPS and requesting the purchase of a new card, followed by a payment and response from the Mobile Server. On the original model, the Mobile Server replies by sending back an APDU containing the card data as per the EN 1545 (see Section 2.2.1) standard. With our proposed model, the Mobile Server now replies by performing a secure SQL Remote Procedure

Call. Listing 3.2 shows the SQL command required to add a new card, which essentially consists of adding a new row to the ticketing table. The user chooses the type of card and the number of initial credits it wants via the UI and the Ticketing Server simply builds the SQL command according to that, adding a randomly generated serial number. After this card is added to the ticketing table, it is now possible to do Validation or Recharge operations with it. We will detail both next.

3.3.3 Validation

```
-- SQL instructions for card validation
UPDATE Tickets
    SET Credits = CASE
        WHEN Credits > 0 THEN (Credits - 1) ELSE -1 END
    WHERE Type = @cardtype;
SELECT SN, Credits FROM Tickets WHERE SN = @sn;
    WHERE SN = @sn;
```

Listing 3.3: SQL instructions for HCE Mobile Ticketing Validation.

The Validation operation works differently from Initialization, Purchase and Recharge as it is the only operation where SQL commands are not issued from the Mobile Server; instead, the communication is done directly with a ticketing terminal possessing NFC capabilities. The SRPC protocol works exactly the same, with only the channel changing. The Validation SQL command is shown in Listing 3.3. It starts by verifying if the card to be validated exists by checking if there is a row with the corresponding serial number. Next, confirms if the card has enough credits and, if so, decrements the available credits and sends an acknowledegement (see message $M_{3.a}$ in Table 3.2); otherwise, sends a Fail response (see message $M_{3.b}$ in Table 3.2 and sets the current credit value to -1, indicating that it has been exhausted. The card to be validated is chosen by the user via the UI, which leads to the corresponding serial number being sent to the ticketing terminal.

3.3.4 Recharge

```
--SQL instructions for card recharging
UPDATE Tickets SET Credits = (Credits + @amount)
    WHERE SN = @sn;
```

Listing 3.4: SQL instructions for HCE Mobile Ticketing Recharge.

The Recharge operation is quite simple and works very similarly to Purchase. As with this last one, it starts by establishing a connection with the Mobile Server, which is connected to the Ticketing Portal, via HTTPS, and sending a request for a Recharge. The Mobile Server replies by sending an SRPC containing the SQL command necessary to recharge the card. The SQL command is shown in Listing

3.4. First, it checks if there exists a card (i.e., row) on the table with the same serial number as the one the Recharge was requested. If it exists, the number of current credits is incremented with the credits requested. Both the card to recharge and the number of credits it will be recharged with are chosen by the user via the UI.

3.4 Summary

This chapter began with a vulnerability analysis of a typical HCE Mobile Ticketing application that aims to offer offline validations, meaning that the card data must be stored on the device. Although this application enforces security mechanisms, it relies on the integrity of the OS to protect its sensitive data, i.e., the cards it emulates. We define a threat model by defining three types of attackers, as proposed by [22], and choose the intercepting root attacker model, who has complete control over the Android OS and its filesystem. We show that such an attacker is able to compromise the application because it has free access to the filesystem, more specifically, to the locations where card and key data are stored, while also proposing three attacks that show how such application can be exploited.

We then presented our solution to protect applications that need to manage sensitive data against an intercepting root attacker. We name this solution DBStore, a TrustZone-backed database management system for mobile applications which provides an SQL interface to application-custom databases, allowing SQL commands to be issued to a database residing on a remote DBStore by invoking SQL Remote Procedure Calls (SRPCs). We define security protocols that aim to protect every communication between remote client and local DBStore, including the SRPCs. Given that the sensitive data is now stored and operated on a Trusted Execution Environment, via TrustZone technology, it is now isolated and applications don't need to rely on the integrity of the Rich OS for its protection.

Finally, we showed how HCE Mobile Ticketing can be protected by using DBStore. Considering that card data is no longer stored on the Rich OS's filesystem but on the TEE, an intercepting root attacker is no longer able to access it and none of the proposed attacks remains possible, as all data and operations over it are out of the attacker's reach. We show an example of SRPCs that are able to keep the original functionality of the application, namely storage of card data and the three possible operations over it: Purchase, Validation and Recharge.

In conclusion, we have shown the vulnerabilities that exist on a typical HCE Mobile Ticketing application and how DBStore can be used to protect against them, as with any application that needs to manage sensitive data and relies on the integrity of the OS for its protection. By moving the data and the operations that are able over it to a Trusted Execution Environment, and by providing a well-known SQL interface and security protocols to protect communications, we argue that this is a feasible solution. We also show how the proposed attacks are no longer possible when using DBStore. In the next chapter we present two prototype implementations of our system.

Chapter 4

Implementation

Considering the design of DBStore presented in the previous section, we implemented two different prototypes. Both have the DBStore Trusted Service, which was integrated with an emulated version of the HCE Mobile Ticketing system. The first prototype was developed essentially to serve as proof-of-concept, thus being very simple, while the second aimed to be an accurate implementation of DBStore's architecture that could be integrated with a real application. As such, the first prototype was developed using Genode and is described in Section 4.1; the second was developed using OP-TEE and is described in Section 4.2. It is important to note that neither of these prototypes uses TrustZone's secure boot mechanism, although on the OP-TEE prototype it is assumed that DBStore is able to pass the secure boot and thus has access to the private part of the attestation key (AK^{-}).

4.1 Genode Prototype

This prototype was developed in a very early stage of this work. As such, our goal was mainly to show that a trusted service like DBStore could indeed be developed and that it could be used to protect applications that need to manage sensitive data, such as HCE Mobile Ticketing. This lead to a simple prototype can presented the basic functionality of DBStore, i.e., could run SQL commands inside the TEE. This prototype is then comprised of a set of building blocks, the DBStore trusted service and a NW application.

To develop this prototype, we had to leverage some pre-existing building blocks, both hardware and software. Regarding hardware, we developed this prototype to run on the i.MX53 Quick Start Board. In terms of software, we adopted the Genode environment to generate the system image, more specifically the tz_vmm demo that offers a TrustZone implementation (see Section 2.4.3), with the Linux kernel as the NW rich OS. To boot Genode's system image on the board, we chose Das U-Boot as the bootloader. Finally, we chose SQLite as SQL engine.

As for the prototype itself, we had to develop two different programs: the DBStore trusted service and the NW application. The first one runs inside the TEE and integrates SQLite. The NW application works both as a Remote Client that issues SRPCs and as the mobile device's HCE Mobile Ticketing application.

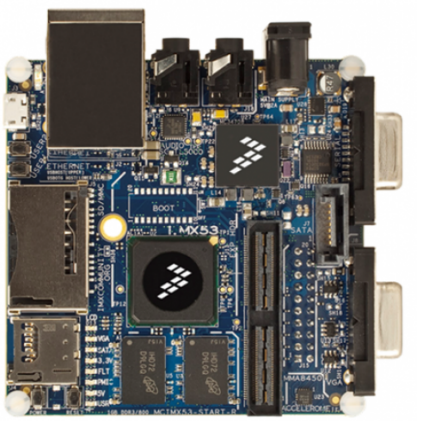


Figure 4.1: The i.MX53 QSB.

As such, it has the behavior of a ticketing server (issues Purchase and Recharge SQL commands) and a ticketing terminal (issues a Validation SQL command). It is also responsible for triggering the SMC to change context. Both follow the security protocols described in Table 3.2.

4.1.1 Building Blocks

Some building blocks were required as a starting point for the development of the prototype. This included choosing a Secure World Operating System in which we would be implementing DBStore, what SQL engine to use and the hardware in which we would be running and testing the prototype.

This subsection aims to give insight into why these particular Building Blocks were chosen by presenting their properties and advantages. It is important to note that these building blocks were used without any kind of alteration, except in terms of configurations.

i.MX53 Quick Start Board

"The i.MX 53 Quick Start Board (shown on Figure 4.1) is an open source development platform integrated with an Arm Cortex-A8 1 GHz processor and the NXP MC34708 PMIC" [82]. It was designed with the intent to be cost-effective and be a feature-rich platform for development. It has a 1 GB DDR3 SDRAM memory.

Development boards are particularly useful when it comes to testing software. As such, to test this prototype, we required a board that supported the ARM TrustZone technology, while also being to run Genode. Given that the i.MX53 QSB is supported by Genode and that it had been used by previous researchers at INESC-ID, we chose this board as the hardware testbed to run our prototype.

Linux Kernel

A kernel is the central piece of an Operating System; it controls everything that happens in the system, from memory management to handling interrupts. It is the first component of the OS to be loaded to memory on booting, and it stays there until the end of the computing session, as its services are continuously needed [83].

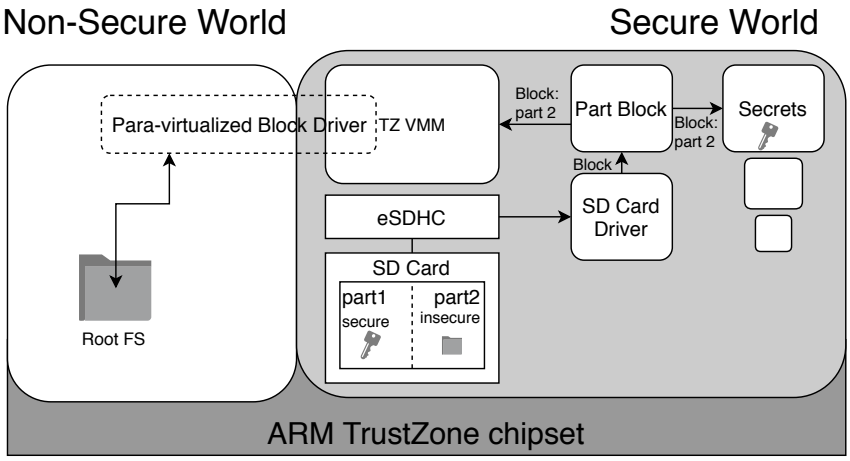


Figure 4.2: Genode's tz_vmm architecture (adapted from [84]).

Linux's kernel is one of the kernels supported by Genode. For one to be able to access the TEE, it is necessary to add a SMC instruction, defining the parameters to be exchanged between worlds. The Virtual Machine Monitor is able to detect this SMC and trigger the necessary procedures for world-switching.

Tz_vmm Demo

Genode offers a TrustZone demo, named tz_vmm, which uses a virtual machine monitor to leverage the TrustZone capabilities available on the underlying platform. Its architecture is illustrated in Figure 4.2. This demo is able to create a system image, containing the guest OS (Linux Kernel, in our case), a full TEE and a filesystem. It is possible to extend this demo by adding code to both NW and SW to develop a TrustZone-based prototype. Adding code to the NW is required for having an application that is able to invoke a function running inside the TEE. To do so, one also needs to add a new SMC to the kernel that is able to pass the arguments needed to the SW. This SMC then needs to be called by the NW application, triggering world-switching, managed by the monitor. Modifying the SW code allows for additional functionality, which will be executed when the NW application triggers the world-switch.

After these alterations are added to the tz_vmm demo, it is only necessary to run a script that will compile all these components into an uImage to boot on a development board. This practicality led us to choose this demo as a starting point for the development of our prototype.

Das U-Boot

U-Boot is an open-source bootloader for embedded systems "based on PowerPC, ARM, MIPS and several other processors, which can be installed in a boot ROM and used to initialize and test the hardware or to download and run application code" [85]. Its development is aligned with Linux's, as some parts of U-Boot's source code actually originate from the Linux source tree, namely some headers and special provisions put in place to support booting Linux images. It provides many configuration options for various architectures and board types.

In order for bootstrapping the system, a bootloader must be present on the device. This allows to

load the kernel from flash memory into operating memory, while also setting up parameters that will be used by the kernel when booting, such as avaialable RAM or the underlying machine type. Due to its popularity and compatibility with the board we would be using, we chose U-Boot as the bootloader.

SQLite

"SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine" [86]. It offers developers a wide range of settings that allows one to personalize how SQLite will be built; for example, in terms of threading (single-threaded or multi-threaded) or the amount of available memory. It is open-source and free to use for any purpose, making it the most widely deployed database in the world.

The popularity of SQLite, combined with the functionality it offers, makes it a very enticing SQL library, as using it allows execute the SRPCs described in Table 3.2. Given that the micro-kernel approach to Genode (see Section 2.4.3) allows for the development of extensions, and that SQLite is one of them, the reasons aforementioned made us choose it as the SQL engine to use in our prototype.

4.1.2 DBStore Trusted Service Prototype

The DBStore Trusted Service was developed in C and C++ and follows the description provided in Section 3.2. It runs inside Genode's TEE and can be invoked on Linux programs by triggering the SMC. It uses SQLite as the SQL engine and forwards SQL statements received from the NW to it. The TEE functions by using a switch-case to decide what functions to run. By specifying a certain value in the SMC the desired code can be invoked. State is maintained between invocations and for the duration of the session.

The basis of this prototype lies in a demo provided by Genode named tz_vmm. We adapted some code from Nuno Duarte [24] and Tiago Brito [52], namely related with the TEE invocation procedure and memory address translation that allowed to read from and write to the NW's memory. Only the *Invocation Protocol* was implemented, and with several differences when compared with the protocol defined in Table 3.2. Considering that we would be implementing a better prototype in the future, and using a better framework, we were mostly concerned with presenting a simple proof-of-concept. Contrary to the architecture shown in Figure 2.5, there isn't exactly a DBStore service running inside the TEE. Instead, there is the tz_vmm code and we simply added DBStore's code to it. While this doesn't make any significant difference, it is still relevant pointing out. Next, we'll present the modified Invocation Protocol.

Invocation Protocol

Table 4.1 shows a modified version of the Invocation Protocol. The first alteration is that the "session key" K is not changed upon each exchanged message; instead, it is a hardcoded key shared between client and DBStore. The second alteration is that instead of using hmac, we simply compute a hash of the message and nonce. We use the mbedTLS library [87] as the source for our cryptographic algorithms.

SRPC Invocation Protocol
(M_1) C $\rightarrow$ S: $\langle n_i, \textit{SrpcReq} \rangle_K$, digest(*SrpcReq,* n_i)
($M_{2.a}$) S $\rightarrow$ C: $\langle n_i + 1, \textit{OK} \rangle_K$, digest(*OK,* n_i)
($M_{2.b}$) S $\rightarrow$ C: $\langle n_i + 1, \textit{FAIL} \rangle_K$, digest(*FAIL,* n_i)

Table 4.1: DBStore Genode prototype protocols: C: Remote Client, S: DBStore.

Upon receiving a request from the NW application (M_1, decrypting it and verifying its freshness and integrity, DBStore forwards it to the SQL Engine. The result of executing the operation (success ($M_2.a$) or fail ($M_2.b$) is sent back. The SRPCs that DBStore receives are exactly the same as described in Listings 3.1-3.4.

4.1.3 Non-Secure World Application Prototype

The NW application is responsible for issuing SRPCs and communicating with DBStore running in the SW via the SMC. The prototype has two main components: $server\_mock$ and $ticket\_client$, both developed in C. Both these programs are very abstract representations of real ticketing software. Three operations were defined, Purchase, Recharge and Validation, which are the fundamentals of every ticketing system. For the scope of this work, these operations were simplified: Purchase requested the creation of a card with type "Demo" and balance 10, Recharge added 1 balance to the card and Validation discounted 1 balance from the card. These were based on pre-constructed SQL queries kept by $server\_mock$, based in Listings 3.1-3.4. Only one card at a time is supported, but modifying the prototype to support more is trivial. The $ticket\_client$ program functions as a proxy; it essentially requests operations to $server\_mock$ and sends them through the SMC to the SW. Communication between these two programs is done through sockets. $server\_mock$, in this prototype, has to emulate both a server and a ticketing validator.

Upon receiving a request from the $ticket\_client$, the server prepares the message by first creating a hash using SHA256 and then encrypting it symmetrically using keys that are only shared by it and the code running on the SW. To leverage the development of the prototype, the keys and the initialization vector are hardcoded. If it is a Recharge or Validation request, $server\_mock$ first adds a counter to the message. This counter is also shared by the modified hypervisor, which stores the most recent received counter, whether the operation is successful or not. This protects the application against replayed messages, as the SW code will only accept messages whose counter is greater than the last one stored on the card. When $server\_mock$ finishes preparing the message, it sends it back to $ticket\_client$, who sends it to the SW via SMC. The hypervisor, upon detecting the SMC, starts by identifying the operation. It decrypts the message and obtains the cleartext, which includes the hash. It then verifies the hash by comparing the received one with a hash computed from the message. If all these operations succeed, then the message is accepted as coming from the server. In the case of a Recharge or Validation, the SW code also verifies if the counter is greater than the one stored in the card. If not, the message is

```
asmlinkage long sys_ticketing(int operation, char * ciphertext, int cipher_size)
{

        /* variables for buffer */
    static dma_addr_t linux_buf_phys;
    static unsigned char * linux_buf;
    static long size_ci;

    /* buffer alloc for ciphertext and hash*/
    size_ci = cipher_size;
    linux_buf  = dma_alloc_coherent(NULL, size_ci, &linux_buf_phys, GFP_KERNEL);

    memcpy(linux_buf, ciphertext, cipher_size);

    asm volatile(".arch_extension sec\n\t");
    asm volatile("mov r0, #3      \n"
                         "mov r1, %0      \n"
                         "mov r2, %1      \n"
                         "mov r3, %2      \n"
                         "dsb             \n"
                         "dmb             \n"
                         "smc   #0        \n"
                         : /*no outputs*/
                         : "r" (operation), "r" (linux_buf_phys), "r" (size_ci) /*inputs
                             */
                         : "r0", "r1", "r2", "r3", "r4"); /*used registers*/
    return 0;
}
```

Listing 4.1: DBStore's SMC

discarded and signaled as a replayed message. If the message manages to pass all these checks, it can be executed. If it was a Recharge or Validation message, the client prepares an acknowledgment to send back to the server, which is essentially an "OK" concatenated with the counter received. It is encrypted and sent back to the NW $ticket\_client$, which sends it to $server\_mock$. Upon receiving this acknowledgment, the server confirms if the operations was done successfully. This is why $server\_mock$ also works as a ticketing terminal; in real case, the acknowledgment from the client in a Validation operation is what would trigger the opening of the gates.

4.1.4 Developing the Prototype

To start the development of this prototype, it was necessary to first study Genode's documentation, especially the one concerning the tz_vmm demo. We used Nuno Duarte's and Tiago Brito's thesis works as a ground base for the development of our prototype, as it already had several extensions configured that would be helpful for us. One of these extensions, part_blk, would allow us to write the card data to a separate partition from where the NW filesystem is.

The first step was being able to run the prototype on the board. As was detailed before, the tz_vmm demo had a run script that would compile both NW and SW into a bootable uImage. To be able to boot this uImage on the board, a bootloader was required, and thus we flashed U-Boot into the SD card we would be using. The card had to be formatted first into the EXT4 filesystem. After this was done, it was time to start developing the prototype itself. As was said, the work developed by Nuno Duarte and Tiago Brito gave us a big headstart, as it could serve not only as a reference for us to develop our work, but to even reuse some of it. Namely, a function responsible for translating memory addresses on the SW into memory addresses on the NW. This was particularly helpful because it was the only way

```
-- Loads the Genode image into memory
ext2load mmc 0:1 0x70200000 uImage

-- Booting
bootm 0x70200000
```

Listing 4.2: Steps for booting Genode on the i.MX53

to exchange strings between worlds. All that was required was the starting memory address and the length of the string and the translation function would be able to retrieve it. We developed both DBStore and Non-Secure World application as described in Sections 4.1.2 and 4.1.3, respectively. A new SMC, shown in Listing 4.1, was also added to the Linux Kernel, which would be used by the NW application to be able to communicate with the SW and send the arguments shown in M_1. This requires a kernel recompilation. When the development of all these components was finished and added to the tz_vmm demo, we simply had to run the script, get the uImage and transfer it to the card.

After having the uImage on the card, we could boot it using U-Boot. Listing 4.2 shows the commands required for booting the image. Upon reaching Linux' userspace, it was possible to execute the binary of the NW application and choose what type of operation to run, according to the description provided in 4.1.3.

4.1.5 Limitations

This prototype was developed knowing it was preliminary until we developed the next one using OP-TEE. As the i.MX53 QSB was available, as well as previous work from other students, we decided to develop this simple prototype first. As such, it presents quite a number of limitations, which can be summed up as follows:

- Invocation Protocol: In this prototype, we implemented neither the Initialization nor the Resync protocols. While the lack of the latter is not particularly serious considering the scope, the first one is of paramount importance in showing the security aspects related to DBStore. As such, this prototype lacks any mechanism able to establish an authenticated connection between the two endpoints. As for this implementation of the Invocation Protocol itself, the encryption key is never changed (concatenation of the previous key hashed with the current counter value) and an hmac is not sent along the message, but instead a simple hash;

- Microkernel Approach: Genode's microkernel approach means that one is able to customize the resulting OS by adding whatever components it sees fit. This is problematic, however, as it may increase the TCB of the TEE;

- Portability: Given that this prototype was developed by following the tz_vmm demo, it is infeasible to be able to port it into another SW OS as there is no standardization defined.

These limitations, namely the first and last one, make this implementation unsuited to be adopted in a real-world scenario. However, and for research purposes, at the time it was developed it was able to

illustrate that the solution we proposed was possible. It is able to show a service running on TrustZone, which is able to receive remote SQL commands from the NW and run them using an SQL Engine. Despite the simplicity of the implementation, this functionality was a motivation to keep developing this work and know what to improve in a future prototype.

4.2 OP-TEE Prototype

The previous prototype provides a simple proof-of-concept, being quite limited and not presenting much functionality. It was then our goal to develop an improved prototype, using now OP-TEE instead of Genode. Since OP-TEE follows Global Platform's Internal Core API, we would be able to develop a trusted service that could be portable to any platform following this specification. That, combined with the great support that OP-TEE has, made this a very interesting TEE choice. It is also important to note that OP-TEE's TCB is much smaller than Genode's. While these is advantageous in regards to security, it also makes development more complicated, as functions that one requires – namely libc or POSIX functions – may not be available.

Just as with the Genode prototype, we had to leverage some pre-existing building blocks, both hardware and software. Regarding hardware, we developed this prototype on a different board: the i.MX6 Nitrogen6x. In terms of software, we adopted the OP-TEE SW OS (see Section 2.4.3), with the Linux kernel as the NW rich OS. To boot both OP-TEE OS and Linux kernel, we again chose Das U-Boot as the bootloader. Instead of SQLite, this time we used the LittleD library as SQL engine.

As for the prototype itself, we had to develop two different programs: the DBStore trusted service and the NW application. The first one runs inside the TEE and integrates LittleD. The NW application works both as a Remote Client that issues SRPCs and as the mobile device's HCE Mobile Ticketing application. As such, it has the behavior of a ticketing server (issues Purchase and Recharge SQL commands) and a ticketing terminal (issues a Validation SQL command). It is also responsible for triggering the SMC to change context. Both follow the security protocols described in Table 3.2.

The rest of this Section will analyze in greater detail each of these three major components and the work that was done to put it all together and functional.

4.2.1 Building Blocks

As we did with the Genode prototype, we studied which building blocks were required as a starting point for the development of the new prototype. Again, this included choosing a Secure World Operating System in which we would be implementing DBStore, what SQL engine to use and the hardware in which we would be running and testing the prototype. This subsection aims to give insight into why these particular Building Blocks were chosen by presenting their properties and advantages.

i.MX6 Nitrogen6X Embedded Single Board Computer

Considering that the main goal of our work was to demonstrate how DBStore could tackle the security

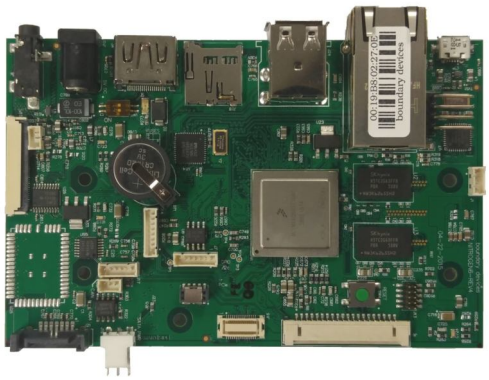


Figure 4.3: The i.MX6 Nitrogen6x.

issues of HCE Mobile Ticketing, our idea was to develop a prototype that ran on top of Android while using the DBStore trusted service running inside OP-TEE's SW. Unfortunately, the i.MX53 QSB does not support Android and, as such, we had to acquire a new development board that did.

"The Nitrogen6X (shown in Figure 4.3) is an embedded single board computer (SBC) based on the NXP i.MX 6 applications processor. It can be built with the i.MX 6 Single, Dual Lite, Dual, and Quad processors, with the Quad core standard on all of our boards" [88]. By following commercial standards, the Nitrogen6X can either be used for development purposes or mass production. Its CPU is based on ARM Cortex-A9 and supports a "wider and faster memory bus (64-bit DDR3 1066)" [88], as well as HDMI and Gigabit Ethernet. The i.MX6 processors aim to deliver scalable platforms combined with good integration and power-efficient computation capabilities. Additionally, there is a line of available accessories for the board, including displays with touch support.

Although the i.MX6 Nitrogen6X board is not one of the default platforms supported by OP-TEE, there are other supported boards that share a very similar architecture to this one. Also, the fact that it is able to run Android (besides Linux) made this a very interesting choice. As such, future researchers on INESC that aim to work on TrustZone would have a very versatile board available. For these reasons, we chose this board.

Linux Kernel

The popularity of the Linux kernel, detailed in Section 4.1.1, made Linaro choose it to complement OP-TEE. As such, they developed a driver for the kernel that allows the Rich OS to access the TEE OS via the GlobalPlatform TEE Client API Specification v1.0, i.e., to know the memory address for the TEE OS entry point, to be able to create shared memory buffers between both worlds and pass arguments between them, etc.

Buildroot

Contrary to Genode, where the run script generates a full system image, including the userspace and corresponding filesystem, OP-TEE only provides the TEE implementation and NW client API to access the SW. This means that it is necessary to generate a filesystem that can be used by the Linux kernel in

Initialization Protocol	
(M_1)	C $\rightarrow$ S: $\langle$*AppId*, $n_1\rangle_{TK+}$, Sign$_{AK-}\langle$*AppId*,$n_1\rangle$, AK^+
(M_2)	S $\rightarrow$ C: $\langle n_1 + 1\rangle_{SK}$, $\langle SK\rangle_{AK+}$
SRPC Invocation Protocol	
(M_3)	C $\rightarrow$ S: $\langle n_2$, *SrpcReq*$\rangle_{K_i}$, hmac(*SrpcReq*, K_i)
($M_{4.a}$)	S $\rightarrow$ C: $\langle n_2 + 1$, *SrpcRes/OK*$\rangle_{K_i}$, hmac(*SrpcRes/OK*, K_i)
($M_{4.b}$)	S $\rightarrow$ C: $\langle n_2 + 1$, *FAIL*$\rangle_{K_i}$, hmac(*FAIL*, K_i)

Table 4.2: DBStore OP-TEE prototype protocols: C: Remote client, S: DBStore.

the NW.

Buildroot is "a tool that simplifies and automates the process of building a complete Linux system for an embedded system, using cross-compilation" [89]. It is able to generate a root filesystem, a cross-compilation toolchain, a Linux Kernel image and a bootloader. However, Buildroot provides all these components independently, i.e., one can use Buildroot simply for the generation of the filesystem and ignore the rest. It is mainly targeted for embedded systems, as they typically use processors that differ from traditional PCs, such as ARM processors. It comes with various default configurations for a number of boards and, besides the expected features of a filesystem, it is highly configurable. Due to its good documentation and easiness to use, Buildroot is a great solution to create a filesystem and userspace to use in conjunction with the Linux Kernel.

Das U-Boot

For the same reasons as detailed in Section 4.1.1, we chose to use U-Boot as the bootloader.

LittleD

LittleD [90] is a relational database aimed for microprocessor devices. At the moment of this writing, LittleD supports "SELECT-FROM-WHERE syntax, including inner joins, projections, and selections over arbitrary expressions" [90], while using around 1kB for most queries. It also supports CREATE TABLE and INSERT statements, but only works with integers and fixed-width strings. It does not support UPDATE statements.

As was described in Section 3.2, DBStore requires an SQL engine that runs on a TEE. Considering that the TCB for OP-TEE's TEE OS is quite small, many functions from libc are not available. Although LittleD proves that sometimes size is not all that matters, it is quite limited. However, given that its code is considerably small and does not use many libc functions, it could be modified to use the Global Platform TEE Internal Core API v1.1 within a feasible amount of time.

LittleD was developed to be used in arduinos. For this reason, as well as the aforementioned ones, we chose LittleD as SQL engine. We had to develop some alternatives due to its limitations, which will be described in greater detail in Section 4.2.4.

4.2.2 DBStore Trusted Service Prototype

The DBStore Trusted Service was developed in C and follows the description provided in Section 3.2. It runs inside OP-TEE's TEE OS and can be invoked on Linux programs via the TEE Client API v1.0. It uses LittleD as the SQL engine and forwards SQL statements received from the NW to it, and is identified by a Universally Unique Identifier (UUID). This UUID allows for the OP-TEE driver to find the Trusted Application being invoked and forward the calls to it. Although a session is established on every call to DBStore, it doesn't keep state between invocations; this means that a new instance of DBStore is created and subsequently destroyed every time it is invoked.

Our DBStore prototype has two main functions: *initialization* and *invocation*. We did not implement *resync*. Both follow the logic of the protocols defined in Section 3.2.2, but with some alterations due to limitations on the available API. The altered version is shown in Table 4.2. Even though the solution described in Section 2.2.1 is the best in terms of security, the alterations we performed to implement this prototype were made with the intent to overcome both technical and time limitations, and mainly to show that it is possible to develop a solution such as DBStore on a TEE. We will detail next how we implemented each of these functions in regards to the alterations that were required.

Initialization Protocol

The Initialization Protocol of our prototype aims to establish a mutually authenticated channel between DBStore and Remote Client. However, due to some missing functionality in the OP-TEE's Internal API and thus the cryptographic library it uses, LibTomCrypt [91], some changes were necessary.

In the first message, (M_1), the Remote Client sends three arguments: the Application's ID and a counter n_1, both encrypted with DBStore's public key TK^+; the second argument is a signature of the AppId and the counter using the application's private key AK^-; lastly, the Remote Client sends its own public key AK^+. In the second message, (M_2), DBStore responds to the Remote Client after verifying the contents of the (M_1) message. This includes verifying the signature and associating the public key to the AppID, as well as storing the initial counter value. Furthermore, DBStore generates a session key SK which will be used henceforth to encrypt the exchanged messages. The response then has two arguments: the counter incremented, to keep ensuring freshness, encrypted with SK; the second one is SK itself, encrypted with the application's public key AK^+. The Initialization Protocol is then completed, and both Remote Client and DBStore share a counter and a session key, which will be used for the messages exchanged in the SRPC Invocation Protocol.

SRPC Invocation Protocol

After establishing a mutually authenticated channel via the Initialization Protocol, the Remote Client can now send SQL statements to be executed on DBStore. These SQL Remote Procedure Calls differ from the ones shown in Listings 3.1-3.4, due LittleD's limitations: given that it does not support either DELETEs or UPDATEs of any particular row, the SQL statements had to be altered to reflect this. The new version is shown in Listing 4.3.

There is now a database for each card. The purchase of a card creates a new table for that card,

```
-- SQL instructions for card purchasing
DROP TABLE IF EXISTS Tickets;
CREATE TABLE Card(SN INT, Type TEXT, Credits INT, Counter N);
INSERT INTO Card VALUES(@sn, @cardtype, @credits, @counter);

--SQL instructions for card recharging
SELECT * FROM Card WHERE Counter = @counterf;
DELETE Card;
CREATE TABLE Card(SN INT, Type TEXT, Credits INT, Counter N);
INSERT INTO Card VALUES(@sn, @cardtype, @newcredits, @counter + 1);

-- SQL instructions for card validation
SELECT * FROM Card WHERE Counter = @counterf;
INSERT INTO Card VALUES(@sn, @cardtype, @credits - 1, @counter + 1);
```

Listing 4.3: SQL instructions for mobile ticketing app.

where each entry corresponds to a validation (a decrement on the credits and an increment on the counter). The name of the table can be a secure random. Recharging a card is now very different from what was proposed in Listing 3.4. Given that it is an operation in which an extra time penalty can be tolerated, we use this as an opportunity to reset the Card database as for it not to grow very large. As such, we get the last row, which corresponds to the latest validation and thus the actual state of the card, identified by the current counter *@counterf*. We then DELETE the database and CREATE it again, and INSERT on it the new card state with new credits and the incremented counter. For the validation, given that it is an operation that needs to be as fast as possible, we get the last row of the table (containing the latest state) and INSERT into it a decrement of the credits and an increment of the counter. These workarounds are not nearly as efficient as what was proposed in Listings 3.1-3.4, but are able to maintain the same degree of functionality.

As for the messages exchanged between Remote Client and DBStore, they are quite similar to the ones shown in Table 3.2. In message M_3, the Remote Client sends an SRPC to DBStore, alongside the current counter value. As was described in Section 3.2.3, the Session Key SK is not used to encrypt messages M_3 and M_4. Instead, SK is combined with the current counter value and hashed, resulting in K_i. Then, on each use, a new K_i is generated and used as a symmetric key to encrypt the messages. In M_3, the Remote Client also sends an *hmac* of the SRPC. DBStore can then answer with two different messages, ($M_{4.a}$) or ($M_{4.b}$), depending on the result of executing the SRPC. Message $M_{4.a}$ corresponds to a successful operation, meaning that DBStore was able to use decrypt the message with the current K_i, verify the freshness and the *hmac*, and that LittleD executed the SRPC successfully. DBStore then answers with the incremented counter, the SRPC value in the case of a SELECT or an OK if it was a CREATE, DELETE or INSERT, both encrypted by K_i. It also sends an *hmac* of the SRPC value or the OK. If any of these conditions is not met, DBStore answers with $M_{4.b}$, which is almost identical to $M_{4.a}$ but with FAIL as a response.

4.2.3 Non-Secure World Application Prototype

As was said at the beginning of this Section, while DBStore is meant to be a standalone service, we developed this prototype with the HCE Mobile Ticketing application in mind. As such, our goal is to show how DBStore can be used to protect the application against the attacks proposed in Section 3.1.

```
-- Initial Cast
db_int size = *POINTERATNBYTES(ptr, -1*sizeof(db_int), db_int*);

-- Where POINTERATNBYTES is the macro
POINTERATNBYTES(s, n, t) ((t)(((unsigned char*)((s)))+(n)))

-- Is converted to
db_int size;
memcpy(&size, ((unsigned char*) ptr) + (-1*sizeof(db_int)), sizeof(db_int));
```

Listing 4.4: C cast to memcpy example

The Non-Secure World Application was then implemented in C as an emulation of a ticketing server, a validator, and the mobile device's HCE Ticketing application. As a ticketing server, it issues SRPCs for both Purchase and Recharge operations; as a validator, it issues an SRPC for a Validation operation; as the HCE Mobile Ticketing application, it essentially provides a simple interface and is responsible for forwarding the SRPCs to DBStore. The functionality and message exchanges were already described in Section 4.2.2. Contrary to DBStore, which uses LibTomCrypt for cryptography, the Non-Secure World Application uses OpenSSL [92]. It uses the TEE Client API v1.0 to establish a session with DBStore and forward the SRPCs to it, as well as receive and process the responses. As DBStore, it has Initialization and Invokation functions, and provides a simple command UI for interaction:

Welcome to DBStore!
Write "init" to start initialization protocol
Write "inv" followed by an SQL statement to start invocation protocol
Write "exit" to quit the program
>--

Writing "init" will send M_1. The AppId is hardcoded, as well as both public keys and private key. Writing "inv" followed by an SQL statement will send M_3. M_2, $M_{4.a}$, and $M_{4.b}$ are printed on the command line after being processed. If the application detects that the message is not fresh (reused counter) or that the *hmac* is not valid, it will signal it. An use case showcasing the functionality of this prototype will be provided in Section A.2.

4.2.4 Developing the Prototype

Now that every component of this prototype has been detailed, we can start analyzing the steps required to implement it and put it running. We started by implementing DBStore and the Non-Secure World Application. To do so, it was necessary to study OP-TEE's documentation as to understand how it works. Then, while we were waiting for the i.MX6 Nitrogen6x board to be delivered, we started implementing the prototype on QEMU [93], which is able to emulate ARM and thus a TEE. Thanks to OP-TEE's portability, having the prototype running on the board later would be trivial. Given that the prototype had to be entirely programmed in C, this posed a challenge from the very beginning.

Another big challenge was the SQL library. Initially, our idea was to use SQLite, but this ended up proving unfeasible due to the sheer size of it. Considering the limitations the TEE has in terms of

available POSIX functions or even multi-threading capabilities, it required changing too much with little time available. For this reason, we had to search for alternative libraries that required fewer alterations, and alterations that would be easier to make. We found LittleD, which used few POSIX functions and, as such, would not require as much work to adapt: we only had to exchange the POSIX I/O functions that LittleD used with the ones provided by the TEE Internal API. We then faced another problem, as LittleD used many casts on its code, which were causing memory alignment compilation errors. To fix these errors, we had to replace the casts with memcpys. An example of such a fix is shown in Listing 4.4; this had to be done to pretty much *every* cast present in the library, which was cumbersome and error-prone. While most of the available functionality remained working correctly, some didn't, such as the case of the *SELECT FROM WHERE* statement. Because this statement would be needed for the HCE Mobile Ticketing SRPCs, we had to add a parser that was able to convert a *SELECT * FROM* to a *SELECT * FROM WHERE* statement.

After having the prototype running on QEMU, using LittleD as the SQL engine, it was time to port it to real hardware. Initially, our idea was to use a custom build of OP-TEE, which worked with Android, and for the i.MX6 Nitrogen6x board [94]. Unfortunately, we ran into several compilation errors that we ended up not being able to fix, and for this reason we decided to build OP-TEE for the Nitrogen6x ourselves. We started by building the bootloader U-Boot for the Nitrogen6x and flash it, following the instructions provided by Boundary Devices [95]. We first had to format the card into two partitions: the first, a FAT filesystem, would contain all the required boot files; the second, an EXT4 filesystem, would contain the fileystem generated by Buildroot, as well as DBStore TA's and NW application's binaries. After the bootloader was flashed and working, we moved on to OP-TEE itself. However, due to the fact that this board is not natively supported by OP-TEE, this proved to be more challenging than we initially anticipated. Luckily for us, other similar boards, also made by Boundary Devices, were supported, which meant that little work had to be done to make OP-TEE compatible. Following a guide posted by Peng Fan [96] for the similar i.MX6Q SDB, we were able to build the OP-TEE components, which includes OP-TEE's OS and user space interface to access the TEE, the Linux Kernel image, and the Device Tree file, which contains general information needed by the kernel and bootloader for booting. OP-TEE OS was compiled with "CFG_RPMB_FS=y", which guarantees that the secure storage files are protected against rollback attacks. OP-TEE's Trusted Storage mechanism stores the files encrypted in the NW, using a key derived from the TA's UUID. As such, only the TA that owns the files can access them. The encryption and decryption of these files are done using a hash tree, whose header stores data, including IV and keys [55]. An object list database indexes the files, and a hash of both these data stores – object list database and hash tree – is stored in the Replay Protected Memory Block (RPMB) partition of the eMMC. This assures that a file cannot be brought back to a previous state as the hashes will not match. To generate the user space itself and filesystem, we used Buildroot. After having all the required components, it was time to put them in an SD card and try to boot. Booting was getting stuck on CPU world switching, probably when the OP-TEE driver was giving back control to the Linux Kernel. After some research, we discovered that disabling the Cryptographic Accelerator and Assurance Module (CAAM) in the Kernel, as well as booting with the bootargs "*console=$console*

```
-- Setting boot arguments
setenv bootargs "console=${console} root=/dev/mmcblk1p2 rootwait
    earlyprintk"

-- Loads the kernel image into address 0x12000000. Boot files are on the
     first partition of the SD card
fatload mmc 0:1 0x12000000 zImage

-- Loads the Device Tree Blob into address 0x18000000
fatload mmc 0:1 0x18000000 imx6q-nitrogen6x.dtb

-- Loads the TEE image into address 0x20000000
fatload mmc 0:1 0x20000000 uTee

-- Booting
bootm 0x20000000 - 0x18000000
```

Listing 4.5: Steps for booting OP-TEE with Linux Kernel

root=/dev/mmcblk1p2 rootwait earlyprintk" would solve the problem. The steps required for booting are shown in Listing 4.5.

After we were able to boot, it was time to run the prototype. First, we tried to run XTest to make sure everything was set up correctly. XTest ran into a *SIGILL* raised by a *udiv* instruction in OpenSSL (*_armv7_tick()*), not supported by Cortex-A9. We had to recompile OpenSSL targetting a lower platform, which fixed the issue. XTest ran successfully and so we tried to run the prototype, but it crashed with an *alignment fault* error. After searching for a solution to this problem, we discovered that rebuilding OP-TEE with the configuration option *CFG_SCTLR_ALIGNMENT_CHECK=n* would fix the problem. We were then able to have the prototype fully running on the i.MX6 Nitrogen6x board, with the only problem being long I/O times to write files. Unfortunately, we haven't been able to fix this issue, which may be due to some kernel configuration that is missing.

4.2.5 Limitations

The limitations of the OP-TEE prototype can be summed up as follows:

- Resync Protocol: We only implemented the Initialization and Invocation protocols. As such, our prototype assumes that neither part will be getting out of sync and thus we chose not to implement it at this stage;
- SQL Engine: LittleD does not have the same degree of functionality as SQLite does, for instance, which required some alterations to the proposed SRPCs. Additionally, due to the alterations required to port LittleD to the TEE, some additional functionality stopped working correctly, which limited an already limited library;
- TEE Internal API: Due to the available API for cryptographic functions not supporting encryption with private key, for example, some small adjusts had to the made to DBStore's security protocols;
- I/O: For reasons unknown, the writing of files on the TEE takes an abnormally long time (5-10 seconds) even when writing 128-bit AES keys, making it more difficult to measure performance

accurately. Additionally, the files are being written to the Non-Secure World filesystem, although they are protected against rollback attacks via the Trusted Storage mechanism. However, our goal was to have them being written to an isolated partition that only the TEE has access. Unfortunately, we did not have time to implement this;

- Android: Our initial goal was to have Android running on the board, and were expecting to do so by using the build made at the Max-Planck Institute for Software Systems. When this was not possible, we took more time than we expected to have OP-TEE running on the Nitrogen6x. For this reason, we decided it was best not to have Android at this time as would create an additional layer of possible issues which we may not have time to fix.

4.3 Summary

This chapter presented two different prototype implementations of DBStore, a detailed analysis of the required components to build them, how they were built and an analysis of their limitations.

We started by analyzing our first prototype, built using Genode. This prototype is very rudimentary, only supporting a single protocol, the Invocation Protocol, which was slightly altered. This prototype was built by leveraging the tz_vmm demo provided by Genode, which has an hypervisor monitoring the Guest OS and taking advantage of the TrustZone capabilities of the underlying hardware to implement a TEE. This prototype serves as a proof-of-concept of DBStore's applicability in the context of mobile ticketing by demonstrating how a typical HCE Mobile Ticketing application could use it for the storage of cards and operations over them.

Next, we presented our second and current prototype, which uses OP-TEE as a SW OS. The fact that OP-TEE follows the Global Plataform's specifications makes this a very interesting development platform, specifically due to the available documentation and portability. This prototype, however, was quite challenging to implement when compared with Genode's. The reduced TCB made us scrap SQLite as the SQL Engine and had us search for an alternative and lighter approach. We chose LittleD, a very crude SQL library that requires few libc/POSIX functions and thus easier to port. The fact that our board was not natively supported by OP-TEE also made running the prototype on the board difficult. Given that OP-TEE supports QEMU, an virtualization software, we were able to develop the prototype on it so that we could later port it to the board. Although we had to perform some alterations to the security protocols and to the HCE Mobile Ticketing's SRPCs, this protocol has much more functionality than the Genode prototype.

To conclude, it is clear that although these are two prototypes of DBStore, they differ quite a lot. Each prototype brings advantages that would be helpful in the other: Genode's using SQLite and having TEE files written to an isolated partition; OP-TEE's by offering a standardized API, following Global Platform's specifications and having much more functionality. A perfect prototype would benefit from the advantages from both, but unfortunately this was not possible at the moment. However, we argue that these two prototypes are enough to show the feasibility of our solution, but also the challenges associated with developing for TrustZone.

Chapter 5

Evaluation

In the previous chapter we presented the functionality of the two prototypes we developed: one using Genode and the other using OP-TEE. This chapter aims to provide a performance evaluation of both implementations. We start by evaluating the Genode prototype in Section 5.1. Given that this prototype was developed very early into this work, and that it would be a preliminary proof of concept, we performed a very basic evaluation of it. Considering the OP-TEE prototype we would be later implementing and which be the main one, we performed a more thorough evaluation, which is presented in Section 5.2.

Experimental testbed. The evaluation testbed for our prototypes is different. For the Genode prototype, our evaluation testbed consisted of an i.MX53 Quick Start Board, featuring a 1 GHz ARM Cortex-A8 Processor, and 1 GB of DDR3 RAM memory. The board executed our system from a mini SD card, which was flashed with the modified Genode and Android versions. We measured the performance in both worlds through libc's time library; more specifically, by using the function 'clock()', which returns the number of clock ticks elapsed since the program was launched, which we then converted to seconds.

For the OP-TEE prototype, our evaluation testbed consisted of the QEMU emulator running on a PC that features an SSD and a 2.8GHz Intel Processor and the i.MX6 Nitrogen6x, which features a 1 GHz ARM Cortex-A8 Processor and 1 GB of DDR3 RAM memory. The board executed our system from a mini SD card, which was flashed with the modified OP-TEE and Linux versions. As with the Genode prototype, we measured time using libc's time library, but only in the NW. In the SW, we used OP-TEE's implementation of the Global Platform TEE Time API.

For each experiment in either prototype, we performed 50 runs and we report the mean and standard deviation, which is on top of each bar.

5.1 Evaluating the Genode Prototype

This section provides an overview of the evaluation we performed of the Genode prototype, which consists of the evaluation of a very simple NW application emulating HCE Mobile Ticketing and an also

simplified version of DBStore. We start by providing a performance evaluation, in Section 5.1.1, of each of the three operations typically found in these types of applications – Purchase, Recharge and Validation – comparing their execution fully in the NW and in the SW. Then, in Section 5.1.2, we discuss how HCE Mobile Ticketing would function integrated with this prototype, namely what vulnerabilities would be tackled and those that would remain.

5.1.1 Operations Performance

To study the performance overhead of our security fix for the HCE-based mobile ticketing application, we measure the execution time of three main card operations tested on our prototype DBStore testbed. More specifically, we measure these times using a simple execution flow where the user first purchases the card and subsequently charges and validates it. Note that although the execution time of SQL queries depend on the size of the tables against which they are executed, this simple scenario mimics a real use case, since users commonly only have a single and intransmissible transportation ticket. To be able to do a more in-depth analysis of these numbers, we performed measurements both in the normal and secure worlds. Since we are interested in assessing the performance overhead of our approach within the mobile environment, the numbers we report exclude network communication costs, i.e., communication with external servers.

Figure 5.1 presents the execution time and standard deviation (atop each bar) of the tested card operations: purchase, recharge, and validation. Results cover the execution cost in the non-secure world (NW) and in the secure world (SW). The figure also details the time spent by these operations on two main phases. The first one, labeled “operation”, refers to the core logic behind all the three cases, which is the same on both NW and SW settings. The second, “context-switch”, refers to the logic associated with changing between the normal and secure world contexts, and only exists in the SW setting.

The results shown that the “operation” logic has negligible cost differences between worlds. Generally, purchase queries are faster since they consist of a simple insertion in the ticket table. In contrast, the card charge query is slower since it involves a write operation on the table to update the card balance. Finally, the card validation query is the slowest since it must first read the table to get the balance and validate the operation, and only then write the new updated card balance to the table. When comparing both worlds, we can see that the SW setting incurs a constant performance penalty of roughly 3ms associated with the context switches. Nevertheless, considering the brought about security benefits, we argue that this overhead to cause little usability impact within the context of mobile ticketing.

5.1.2 Functionality and Security Assessment

Considering that the context of this work revolves around a typical HCE Mobile Ticketing application, and even though DBStore was proposed for any application that needs to manage sensitive data, it is important to show what vulnerabilities DBStore is able to tackle and those that it can’t, specially considering the analysis done in Section 3.1. It is also necessary to show how HCE Mobile Ticketing

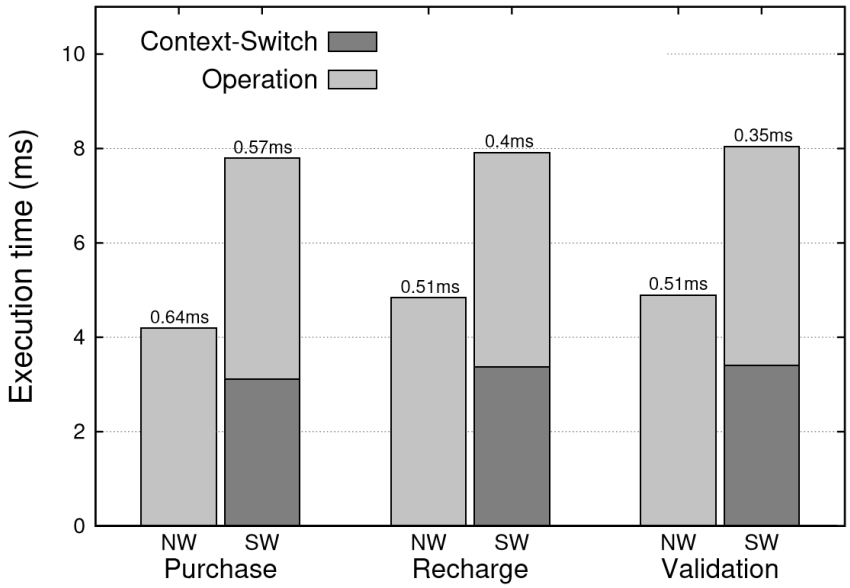


Figure 5.1: Card operation execution times, in the NW and in the SW (Genode).

would be able to assure the correct behaviour of DBStore, which would have to follow the protocols detailed in Table 3.2. Table 5.1 illustrates this. Darkened rows indicate the functionality that the prototype fails to implement.

Starting from the first row, we can see that HCE Mobile Ticketing's data would be now safely stored by the TEE in an isolated partition, thus thwarting Attacks 1 and 2 (see Section 3.1). Attack 3 is also thwarted because operations would now be done by DBStore in the SW. The first problem arises with the lack of the Initialization Protocol. Given that it is not possible to establish a mutual authenticated channel, HCE Mobile Ticketing has no assurances that is indeed communicating with a legitimate DBStore instance. Also, if the application falls out of sync with DBStore it has no means to resynchronize due to the Resync Protocol not being implemented in this specific proof-of-concept. However, it is possible to assure message freshness because a counter is exchanged on each message. It is also possible to guarantee the integrity of the message because both counter and message are hashed, with the hash being sent alongside encrypted. The implented protocol does not authenticate the message because hmac is not used. Finally, the same key is always used to encrypt the messages, which leaves it vulnerable to a key-recovery attack.

Although HCE Mobile Ticketing would be able to maintain the same degree of functionality by using this DBStore prototype, the security guarantees offered are not enough. The main problem is the lack of the Initialization Protocol, which leads to HCE Mobile Ticketing not being able to be assured it is communication with a valid DBStore. This means that an attacker could create a fake DBStore and trick HCE Mobile Ticketing to communicate with it. If this attacker was able to recover the key by analyzing the exchanged messages, it would be able to decrypt the messages and alter the SQL commands to his will. This makes this prototype implementation insecure in the real-world. However, it was not our goal to fully implement all the security mechanisms proposed in Section 2.2.1, but to test if the architecture we had developed was feasible, particularly in the context of HCE Mobile Ticketing. To that end, our goal was reached.

HCE Mobile Ticketing's Security Requirements	***Genode Prototype***
Card and Key Data Protection	Protected by storing them encrypted in an isolated SD card partition
Reverse Engineering Resistance	Given that the remote client issues SRPCs that will be executed in isolation in the SW, this is no longer possible
Mutual Authentication	The prototype does not implement the Initialization Protocol, and as such it cannot guarantee Mutual Authentication
Resync	The Resync protocol was not implemented and thus HCE Mobile Ticketing has no means to resync with the remote DBStore
Message Freshness	A counter is maintained by each party and exchanged on every message. Message freshness is assured
Message Integrity	A hash is sent alongside each SRPC, which guarantees integrity
Message Authentication	Considering that the prototype sends a hash and not an hmac, the authentication of the message cannot be assured
Encryption key renewal	The prototype always uses the same key for encrypting requests and responses

Table 5.1: Analysis of the guarantees provided by the Genode prototype to HCE Mobile Ticketing.

5.2 Evaluating the OP-TEE Prototype

In the previous section we presented an evaluation of our preliminary Genode prototype. This section looks at our current OP-TEE prototype, which is much more robust and functional. Considering the alterations we had to perform to the SRPCs proposed in Table 3.2 and that we are using a library not as popular as SQLite, we start in Section 5.2.1 by analyzing the cost of executing each supported SQL command individually, both in the NW and SW. Afterwards, in Section 5.2.2, we analyze the performance of each of the three typical ticketing operations – Purchase, Recharge and Validation – performed both fully in the NW and in the SW with OP-TEE. Lastly, we present in Section 5.2.3 a study of how HCE Mobile Ticketing could be integrated with this prototype.

5.2.1 SQL Commands Performance

Contrary to the Genode prototype, where the very popular and researched SQLite was used as SQL Engine, in this prototype we used the LittleD library. Given that this library is considerably less known that SQLite, we think it is important to start by analyzing the performance of each of the supported SQL commands we used in the modified SRPCs. We have analyzed the execution times both using an emulated solution (QEMU) and on real hardware (Nitrogen6x). The idea is to show a comparative measure between executions in different platforms, specially due to the long I/O times on the board.

We defined a set of testing SQL commands to execute on both platforms. These commands are shown in Table 5.2 and are examples of the SQL commands supported by the LittleD library, and which are applicable in the HCE Mobile Ticketing scenario. We start by testing LittleD's performance of running these commands in the NW, i.e., without being integrated in DBStore. We evaluate the time it takes to write the testing tables in CREATE commands and to write/read from them in INSERT and SELECT

Command Name	***SQL Command***	***Meaning***
SQL1	CREATE TABLE tabletest (testone int);	Creates a table 'tabletest' containing only one integer parameter, 'testone'
SQL2	INSERT INTO tabletest VALUES (100);	Inserts the value 100 in 'tabletest'
SQL3	SELECT * FROM tabletest WHERE testone=100;	Selects every row in 'tabletest' where 'testone' has the value 100
SQL4	CREATE TABLE tabletest (testone int, testtwo string(10), testthree int);	Creates a table 'tabletest' containing two integer parameters, 'testone' and 'testthree', and on text parameter up to 10 characters, 'testtwo'
SQL5	INSERT INTO tabletest VALUES (100, 'testvalue', 100);	Inserts 100, 'testvalue' and 100 in 'tabletest'
SQL6	SELECT * FROM tabletest WHERE testtwo = "testvalue";	Selects every row in 'tabletest' where 'testtwo' has the value 'testvalue'
SQL7	INSERT INTO tabletest VALUES (500, "another", 500);	Inserts 500, 'another' and 500 in 'tabletest'
SQL8	SELECT * FROM tabletest WHERE testthree = 500;	Selects every row in 'tabletest' where 'testthree' has the value 500

Table 5.2: SQL Commands to be tested on both NW and SW, running on QEMU and the Nitrogen6x.

commands as the I/O, and the time it takes to execute the rest of the command, such as parsing it, as the operation. LittleD creates some metafiles during operations to aid in processing commands. We did not measure the time to handle metafiles as part of the I/O, but as part of the operation itself. Figure 5.2 shows the results for executing the commands on both platforms, including the I/O times. As expected, the first three operations are faster than commands SQL4-SQL8 because the table may only contain one parameter. It is also easy to conclude that the operations performed faster on the emulator than on the board due to the underlying hardware, as the computer running QEMU is much more powerful than the Nitrogen6x board.

After testing the commands in the NW, we tested them in the SW where OP-TEE runs, both on QEMU and the Nitrogen6x. The results are quite different and as such we are going to discuss these platforms separately. Figure 5.3 shows the results of running the SQL commands on OP-TEE with QEMU. As expected, and as shown in Figure 5.2, the first three commands execute faster due to the smaller size of the table. As such, creating the table, inserting rows and selecting values from it requires less indexing than operations SQL4-SQL8, and thus reducing execution time. Overall, comparing with the values shown on Figure 5.2, we can conclude that the operations run slower on the SW than on the NW. Given the changes that were required for compiling and running LittleD on the SW, namely the 'cast to memcpy' (see Section 4.2.4), this incurred in an additional time penalty. Furthermore, and as we mentioned before, LittleD creates some metafiles during the processing of commands. Considering that the Secure Storage API offered by OP-TEE requires constant context-switches between worlds, as files are stored in the NW's filesystem, the creation, writing, reading and deletion of these metafiles incurs in

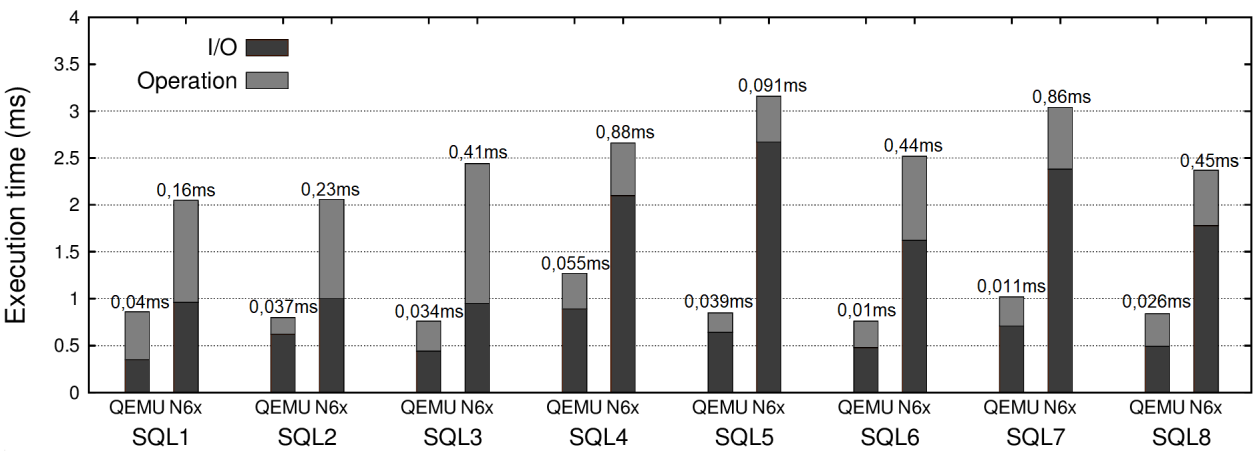


Figure 5.2: SQL Commands defined in Table 5.2 running on the NW of QEMU and Nitrogen6x.

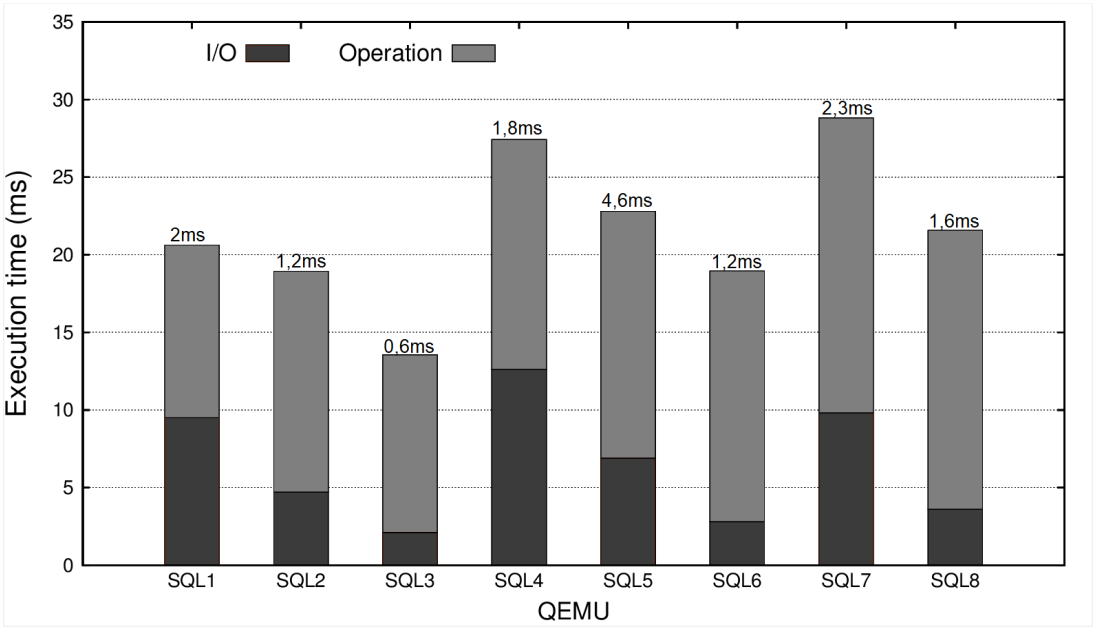


Figure 5.3: SQL Commands defined in Table 5.2 running on the SW of QEMU.

an additional overhead and, as such, operations on the SW take longer than on the NW.

The results we obtained on the Nitrogen6x were quite unexpected and are illustrated on Figure 5.4. The left graphic shows that operations SQL1, SQL2, SQL4, SQL5 and SQL7 performed in the magnitude of seconds, mainly due to a huge chunk of time required for writing files. The reason why these times are so slow is unknown at the time of writing this dissertation, although we suspect it may be due to some configuration option in either kernel, as we ran into some problems when trying to deploy OP-TEE on the board that were fixed by enabling or disabling some configurations in both kernels (see Section 4.2.4). On the bottom figure, the operations that only require reading files (e.g., SELECTs) take a normal ammount of time. Given that CREATE and INSERT commands require the generation of more metafiles, this incurs in an additional time penalty. SELECTs, on the other hand, only require reads and thus executes faster. Comparatively to the NW execution, and as with QEMU, operations take longer on the SW, for the same reasons aforementioned.

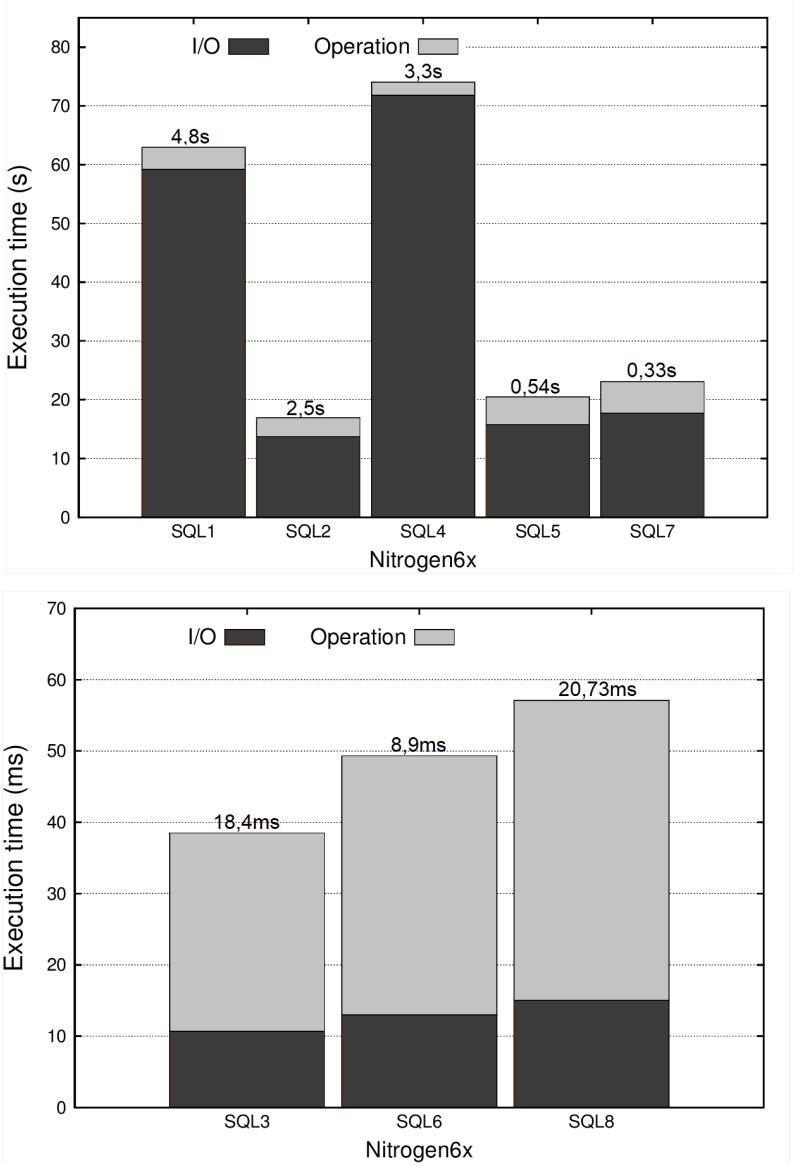


Figure 5.4: SQL Commands defined in Table 5.2 running on the SW of the Nitrogen6x.

5.2.2 Protocol Performance

In our prototype, we implemented only the Initialization and Invocation protocols and with slight alterations when compared with the original model (see Tables 3.2 and 4.2). To evaluate the performance of each protocol, we conducted tests again on both platforms (QEMU and the Nitrogen6x). We consider that the protocol begins the moment the NW application triggers the SMC, and only ends when it is given back control from the SW. In the Invocation Protocol, we are discounting the execution time of the SQL command (we will analyze it further down in Section 5.2.4), and are only evaluating the execution time of cryptography and parameter exchange. We also evaluate the cost of I/O on both protocols, as well as the time penalty of context-switch.

Figure 5.5 shows the execution times of running each protocol on QEMU and on the Nitrogen6x board. We observed similar trends on both platforms; the Initialization Protocol is the slowest – taking 2.44 seconds on QEMU and 39.07 seconds on the Nitrogen6x – since it uses Public Key Encryption

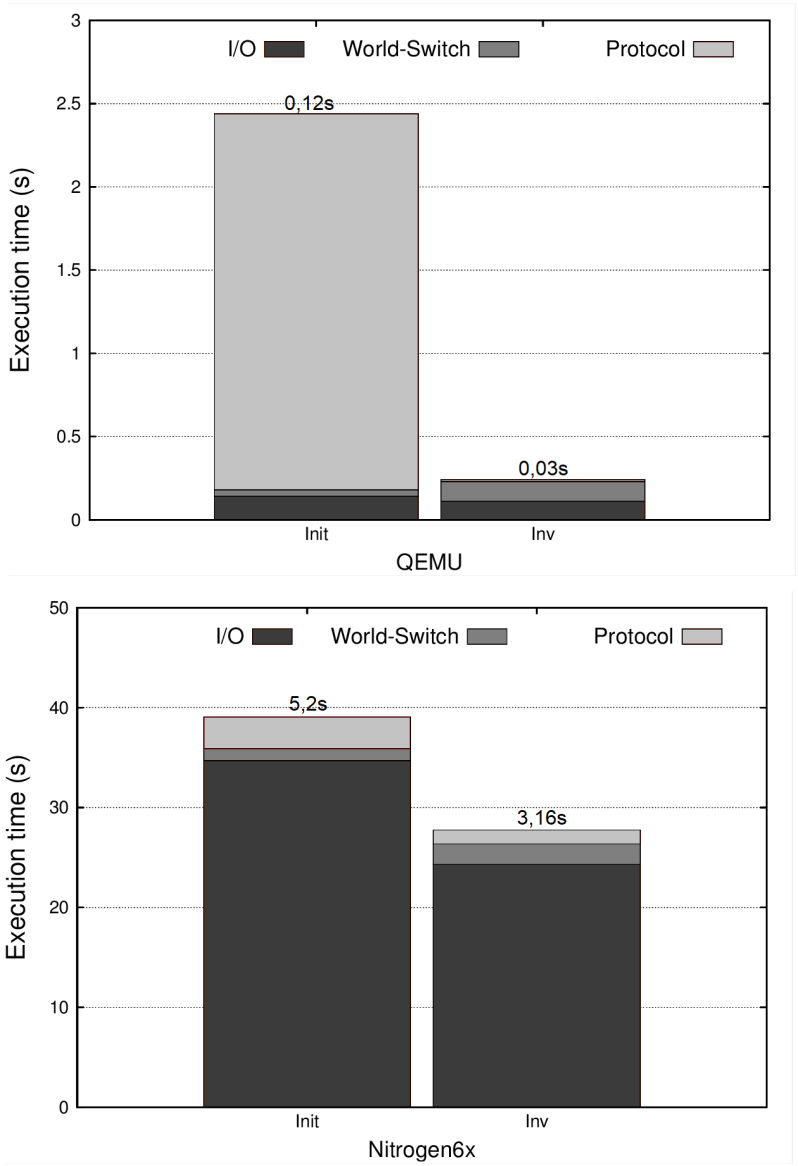


Figure 5.5: Execution times of the protocols defined in Table 4.2, running on QEMU (top) and on the Nitrogen6x (bottom).

(PKE) to encrypt M_1 and generate a signature (see Table 4.2). The Initialization Protocol also needs to create files to store the counter and the session key, thus making the I/O take longer. The cost of world-switching is lower than on the Invocation Protocol, as there are less parameters being exchanged between worlds. Given that we are discounting the cost of running the SQL command, the Invocation Protocol takes less time – 0.2414 seconds on QEMU and 27.75 seconds on the Nitrogen6x – because it only needs to decrypt the exchanged arguments via symmetric cryptography, which is less costly than PKE. The I/O is also faster as it only needs to update the session key and counter files with the newly generated ones. World-Switching takes about twice as long because there are more parameters being exchanged between worlds. Although we discounted the time it takes to run an SQL command, we measured the time it takes to switch context when passing an SQL command as well. As such, it is understandable that the world-switch penalty is greater on the Invocation Protocol when compared with the Initialization Protocol due to the size of the arguments being exchanged, leading to longer memory

reads of the shared buffers. Unfortunately, the I/O problems existing in the Nitrogen6x platform lead to very long execution times, specially when compared with QEMU's performance.

5.2.3 HCE Mobile Ticketing and the OP-TEE Prototype

HCE Mobile Ticketing	***OP-TEE Prototype***
Card and Key Data	Stored in the NW filesystem
Reverse Engineering	Given that the remote client issues SRPCs that will be executed in isolation on the SW, this is no longer possible
Mutual Authentication	The Initialization Protocol assures HCE Mobile Ticketing that it is communicating with a legitimate DBStore instance
Resync	The Resync protocol was not implemented and thus HCE Mobile Ticketing has no means to resync with the remote DBStore
Message Freshness	A counter is maintained by each party and exchanged on every message. Message freshness is assured
Message Integrity	A hmac is sent alongside each SRPC, which guarantees integrity
Message Authentication	The hmac sent alongside each SRPC is also able to guarantee the authenticity of the message
Encryption key renewal	The prototype renews the session key upon each exchanged message, by concatenating the previous session key with the current nonce value and hashing it

Table 5.3: Analysis of the guarantees provided by the OP-TEE prototype to HCE Mobile Ticketing.

Like we did in Section 5.1.2, this section analyzes how HCE Mobile Ticketing can be integrated with DBStore, the alterations that are required and the security guarantees are in place. Table 5.3 illustrates this. Darkened rows indicates the issues that this prototype presents. It is assumed that DBStore passes the secure boot check and thus has access to the private part of the attestation key (AK^{-}).

The first row deals with how the prototype is able to secure the card and key data. Contrary to the Genode prototype, where it is able to store this data securely in an isolated partition, in the OP-TEE prototype it is stored in the NW filesystem, meaning it can be accessed by a possible attacker. This does not mean that Attack 1 is possible; OP-TEE's Secure Storage mechanism, and as explained in Section 4.2.4, stores the files encrypted in the NW. The encryption and decryption of these files is done via an hash tree, whose header stores data like encryption key. Rollback protection is ensured by the RPMB, which stores file hashes. As such, backing up the card and key data, using the card and then restoring the backed up files will not work as the Trusted Storage will detect that files' hashes do not match with the stored in the RPMB. Attack 2 is also not possible; just like Attack 3, as reverse engineering of the application to remove the validation check, for instance, cannot be done as the operations are executed inside the TEE. Also, given that we implemented the Initialization Protocol (and assume that DBStore passes the secure boot), it is possible for both endpoints to establish a mutual authenticated channel and thus have the assurance that are communicating with a legitimate party. As with the Genode prototype, the Resync Protocol was also not implemented, hence HCE Mobile Ticketing has no means of resynchronizing. Message freshness is guaranteed by a counter that is exchanged and incremented upon each traded message. Message integrity and authentication is ensured by creating a hmac of

the message and counter using the current session key. The session key is also updated every time a message is exchanged, thereby making a key-recovery attack much more difficult.

With our second prototype, HCE Mobile Ticketing is able to maintain the same level of functionality and many more security guarantees than the version presented in Section 2.2. The three attacks proposed are all thwarted, and the isolation guaranteed by the TEE assures that even in the presence of a compromised Guest OS, e.g., a user with root permissions, the integrity of the sensitive data maintained by the application is not jeopardized. By not having to rely on the correct behaviour of the OS for the security of the data, the application becomes much more robust. The security protocols also guarantee that message exchanges between both endpoints are authenticated and securely encrypted. The portability of this solution, supported by the adherence to the Global Platform's specifications, makes it much more feasible to be deployed in the future.

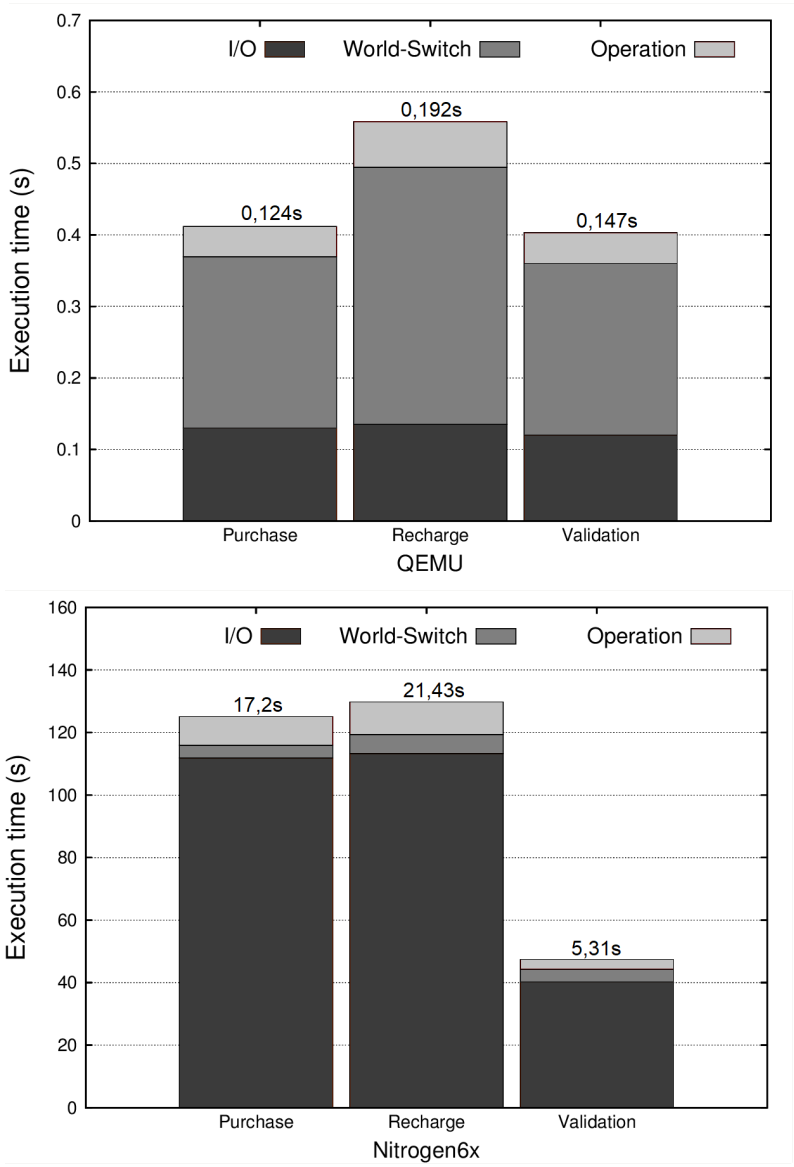


Figure 5.6: Execution times of the three main ticketing operations, running on QEMU (top) and the Nitrogen6x (bottom).

5.2.4 HCE Mobile Ticketing Prototype Performance

Now that we have analyzed the performance of each individual SQL command and protocol, we can start analyzing the performance of each operation, Purchase, Recharge and Validation, as shown on Table 4.2. Operations are essentially an agglomerate of these individual SQL commands, which are then packed in an SRPC and encrypted like detailed in the Invocation Protocol. Again, like in Section 5.2.1, we provide numbers for both an emulated evaluation (QEMU) and evaluation on real hardware (Nitrogen6x). Figure 5.6 details the execution times of the operations on each platform.

Looking at both graphics, we can easily conclude that Recharge is the slowest operation, taking 0.56 seconds on QEMU and 129.79 seconds on the Nitrogen6x. This makes sense, as Recharge is comprised of four steps: first, selecting the most recent row in the 'Tickets' table; then, delete the table, followed by the creating on a new one; it ends by inserting a new row by incrementing both the credits and counter of the selected row. This can be tolerated, as Recharge is an operation the user can perform online at any given moment. The second slowest operation is Purchase, as it consists on the creation of the table 'Tickets' and the insertion of the initial state. It takes 0.41 seconds to execute on QEMU and 125.12 seconds on the Nitrogen6x. Both these operations are I/O intensive, making them a bit slower than Validation, which takes 0.40 seconds in the QEMU evaluation, and a lot slower in the Nitrogen6x evaluation, which takes 47.40 seconds. Validation is then the fastest operation, as it only requires selecting the latest row and decrementing the number of avaialable credits. This is ideal, as Validation is performed at the ticketing terminals and, in a pessimist scenario, in a case where the user has a long line behind of him, meaning that the operation must be fast in order not to anger the other passengers.

We consider that the prototype's performance when using QEMU is acceptable; in comparison, the performance when using the Nitrogen6x is very far from the desirable, which should be in the order of a few seconds. Although the evaluation on the Nitrogen6x board was not positive, QEMU's evaluation offers much more consistent results and can serve as an indication that such implementation can indeed perform well in a properly configured platform. Even though we did not implement all the features, the security guarantees that are offered to HCE Mobile Ticketing are substantial. As with the Genode prototype, HCE Mobile Ticketing would now be protected against rollback attacks and reverse engineering. Although the prototype stores the files in the NW's filesystem, the rollback protection assured by OP-TEE's Trusted Storage protects them, although it would still be preferable if the prototype was able to store the files in a partition isolated from the Rich OS and only accessible from the TEE. Given that it implements the Initialization Protocol, this prototype can establish a mutual authenticated channel through which the SRPCs and responses will be sent. Given these security guarantees, which are summed on Table 5.3, this prototype could be deployed in the real-world.

5.3 Summary

In this chapter we have analyzed both prototypes that were implemented, the first one using Genode and the second one using OP-TEE. We are interested in measuring the performance of each prototype and the security guarantees it provides when used by an application, namely HCE Mobile Ticketing.

We started by analyzing the performance of our first prototype, built to run on Genode. This prototype was transitory and was developed in a very early stage of this work, and thus we aimed it to be mostly a proof-of-concept. Given that we knew that we would be implementing a more robust and functional prototype in the future, this prototype implements only the Initialization Protocol. However, we felt it was enough to show that the solution was viable, and the added cost of running the SQL commands inside the TEE (i.e., world-switching penalty) was negligible considering the additional security guarantees that DBStore provided.

Then, we analyzed our current prototype, which was developed to run on OP-TEE. This prototype offers much more functionality than Genode's, mainly by implementing both Initialization and Invocation Protocols, however with some alterations to the proposed model were required to implement this prototype, specially due to the SQL library we would be using. As such, the first step for evaluating the performance of the OP-TEE prototype consisted on evaluating the performance of the LittleD library itself, and the SQL commands it supports, by defining a set of SQL commands we would be executing on each platform and in each world, concluding that the SQL commands were executed faster on QEMU as it was running on more powerful hardware, both on NW and SW. Comparing between worlds, it took longer to run the commands on the SW. We then measured the performance of the Initialization and Invocation protocols. For both platforms, the Initialization Protocol is the slowest, with the cost for switching context being slower on the Invocation Protocol. The chapter finishes with an analysis of how the OP-TEE prototype could be integrated with the HCE Mobile Ticketing application and its performance. As with the Genode prototype, HCE Mobile Ticketing would now be protected against rollback attacks and reverse engineering. while also being able to establish a mutual authenticated channel through which the SRPCs and responses will be sent. Given these security guarantees, which are summed in Table 5.3, this prototype could be implemented in the real-world. As for the performance itself, we can conclude that Recharge is the slowest operation on both platforms, followed by Purchase and finally Validation, the fastest.

Chapter 6

Conclusions

The main drive of this work was to develop a solution to the vulnerabilities of the HCE Mobile Ticketing application, without requiring any changes to the front-end of the validators. This meant design a system that allows for sensitive data to be securely stored on the device and providing means to operate over it. We designed DBStore, a TrustZone-backed database management system for mobile applications which provides an SQL interface to application-custom databases. This work proved to be an extraordinary learning experience, resulting in two publications [3, 4], and allowing me to confidently say I know a lot more than I did one year ago. It was, however, also very challenging.

The first main challenge when developing for TEEs was related to programming. Given that Trusted Applications have to be written in C, which is a language that although we had some familiarity, induces in additional considerations that are not present in some other popular programming languages, such as Java or Python. This meant taking into account memory management and pointers, which led to a long time debugging SEGFAULTs. Also, the number of available quality-of-life functions and data structures present in the aforementioned programming languages were mostly not available in C, meaning that it took twice as long to implement something.

In terms of developing the Trusted Applications themselves, this also posed another challenge because we were quite unsure of where to begin. Although there is plenty of documentation available online explaining what Trusted Applications are, there isn't exactly a guide that explains to someone new to the area how to start developing them. Fortunately, OP-TEE's repository contains the source code of examples of NW applications interacting with Trusted Applications. This served as the basis to develop DBStore. Another very helpful piece of documentation was Global Platform's Internal Core API specification, which explains each of the available APIs and the functions they provide. It also details the arguments each function takes and the outputs they provide, as well as error codes and their meanings. This proved essential when I wanted to implement the security protocols – and thus, needed to implement cryptography on the TEE – and secure storage. The fact that OP-TEE's source code is available on GitHub was also immensely helpful, as any issue I was having and that I could not understand would quickly be explained by someone from OP-TEE's community, including developers. This helped crossing many roadblocks we faced and that we did not know how to handle.

Lastly, it is also important to emphasize how difficult it is to set up a development board to develop for TEEs, specially if you're not familiar with the specifics of operating systems. Although there are some guides online explaining how to boot for some boards, they lack any explanation about what you're actually doing. Other main problem is that when you run into a problem, it is very difficult to find a solution because you're unsure of what's the root of it. Is it related to the board or some kernel configuration? What do we need to adapt to be able to boot for this board? These questions sometimes remain unanswered, as proven by our prototype taking long to write files from the SW, on the i.MX6. Although we searched online for an explanation, or even someone with the same problem, we were not able to find a solution. We personally consider this as the biggest challenge when developing for TEEs and aiming at running solutions on real hardware.

In retrospective, and even after going through these challenges, TEEs are very interesting area to do research on. The amount of knowledge one can get, which is not only related to the TEEs themselves, but to kernels and hardware as well, makes this an enticing learning experience. Particularly in this work, it was also very encouraging that we were designing a system that could be deployed on a real application. It is recommendable to someone new to this area to first gather as much information as possible regarding TEEs, Trusted Applications, and specifications before actually start developing. It is much easier to know what one want to do if he know what he's developing for. Furthermore, it is important not to be afraid of not knowing something. Having doubts is normal and there are plenty of channels to get support from; either on online forums or directly to someone who has worked on this before.

Future Work

There are many possible angles to work towards in the future. The most urgent one is being able to fix the problems on the Nitrogen6x board and having the prototype running with better performance, as for us to be able to measure it accurately. Additionally, and considering that one of our initial goals was to have Android running on the board, this is another aspect that should be achieved in the future.

Another goal of this work was to propose alternatives to protect HCE Mobile Ticketing. Due to the difficulty of having a TrustZone-based solution applied to a commercial application, it is interesting to conceive other security mechanisms that could be in practice enforced by the application's developers. Although we propose several security systems for mobile devices in Section 2.3, we don't specify how they could be used to thwart the vulnerabilities present on HCE Mobile Ticketing Furthermore, it may even be possible to improve the application's security by analyzing its source code and altering some already implemented protocols.

Finally, and given how challenging it was to develop for a TEE, we aim to create a guide for any future developer or researcher that aims to work in this are. Although there is much documentation available on the internet, for instance in OP-TEE's repositories and on the Global Platform's specifications, there isn't any document or source that is able to summarize this information and explain to someone new to the area. As such, we aim to create this source by providing it through the eyes of someone who faced the same challenges the reader may be facing.

Bibliography

[1] http://www.comscore.com/Insights/Presentations-and-Whitepapers/. Digital Future in Focus. Technical report. Comscore Technical White Paper, 2015. http://genode.org.

[2] Google - Host-based Card Emulation, . https://developer.android.com/guide/topics/connectivity/nfc/hce.html.

[3] P. S. Ribeiro, N. Santos, and N. O. Duarte. DBStore: A TrustZone-backed Database Management System for Mobile Applications. In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications - Volume 1: SECRYPT,*, pages 396–403. INSTICC, SciTePress, 2018. ISBN 978-989-758-319-3. doi: 10.5220/0006883603960403.

[4] P. S. Ribeiro, N. Santos, and N. O. Duarte. DBStore: A TrustZone-backed Database Management System for Mobile Applications. In *Proceedings of INFORUM18*, 2018.

[5] NFC-enabled e-ticketing in public transport: Clearing the route to interoperability. NFC Forum White Paper, .

[6] JR-East - Suica, . http://www.jreast.co.jp/e/pass/suica.html.

[7] Sony - About Felica. https://www.sony.net/Products/felica/.

[8] Aaron Preston - Technology Review: Suica Card, . https://atpreston89.wordpress.com/2012/06/18/technology-review-suica-card/.

[9] Transport for London - What is Oyster? https://tfl.gov.uk/fares-and-payments/oyster/what-is-oyster.

[10] Easy Card Corporation. https://www.easycard.com.tw/english/easycard/index.asp.

[11] R. Want. Near field communication. *IEEE Pervasive Computing*, 10(3):4–7, July 2011. ISSN 1536-1268. doi: 10.1109/MPRV.2011.55.

[12] Host Card Emulation (HCE) 101. Smart Card Alliance White Paper, .

[13] Rambus - Rambus HCE Ticketing. https://www.rambus.com/security/smart-ticketing/hce-ticketing-app/.

[14] Consult Hyperion - HCE and SIM Secure Element: It's not black and white, . http://www.chyp.com/wp-content/uploads/2015/01/HCE-and-SIM-Secure-Element.pdf.

[15] Medius - CloudSE. https://cloudsecureelement.com/.

[16] Xuxian Jiang - Security Alert: New RootSmart Android Malware Utilizes the GingerBreak Root Exploit. https://www.csc2.ncsu.edu/faculty/xjiang4/RootSmart/.

[17] NFC Forum - Suica Case Study, . https://nfc-forum.org/resources/jr-east-passengers-travel-freely-with-mobile-suica-and-nfc/.

[18] Pillow Talks - How Japanese Stations and Railway Companies are Reinventing Smart Mobility — Part 1, . https://18pillowtalks.io/how-japanese-stations-and-railway-companies-are-reinventing-smart-mobility-1-3-a54ae7290008.

[19] Google - Android Keystore System, . https://developer.android.com/training/articles/keystore.html.

[20] BSI - ISO/IEC 1545:4, . https://shop.bsigroup.com/ProductDetail/?pid=000000000030272412.

[21] BSI - ISO/IEC 1545:5, . https://shop.bsigroup.com/ProductDetail/?pid=000000000030094326.

[22] 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, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3155-5. doi: 10.1145/2666620.2666627. URL http://doi.acm.org/10.1145/2666620.2666627.

[23] Legion of the Bouncy Castle Inc - Bouncy Castle. https://www.bouncycastle.org/.

[24] N. O. Duarte. On the Effectiveness of Trust Leases in Securing Mobile Applications, 2015.

[25] Spotify. https://www.spotify.com.

[26] Netflix. https://www.netflix.com.

[27] Android DRM Framework, . https://source.android.com/devices/drm.

[28] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy Oriented Secure Content Handling in Android. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 221–230, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0133-6. doi: 10.1145/1920261.1920295. URL http://doi.acm.org/10.1145/1920261.1920295.

[29] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of NDSS*, 2013. URL https://www.ndss-symposium.org/ndss2013/ndss-2013-programme/security-enhanced-se-android-bringing-flexible-mac-android/.

[30] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association. ISBN 1-880446-10-3. URL http://dl.acm.org/citation.cfm?id=647054.715771.

[31] Z. Zhao and F. C. C. Osono. "TrustDroid^{TM}": Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 135–143, Oct 2012. doi: 10.1109/MALWARE.2012.6461017.

[32] T. H. Toshiharu Harada and K. Tanaka. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Linux Conference*, 2013.

[33] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-936-7. doi: 10.1145/1755688.1755732. URL http://doi.acm.org/10.1145/1755688.1755732.

[34] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce Component-level Access Control in Android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 25–36, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2278-2. doi: 10.1145/2557547.2557560. URL http://doi.acm.org/10.1145/2557547.2557560.

[35] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related Policy Enforcement for Android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 331–345, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-18177-1. URL http://dl.acm.org/citation.cfm?id=1949317.1949355.

[36] G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: Supporting Operation Modes on Smartphones. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 3–12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1295-0. doi: 10.1145/2295136.2295140. URL http://doi.acm.org/10.1145/2295136.2295140.

[37] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3919-5. doi: 10.1109/ACSAC.2009.39. URL http://dx.doi.org/10.1109/ACSAC.2009.39.

[38] T. U. Darmstadt, S. Bugiel, L. Davi, R. Dmitrienko, T. Fischer, A. reza Sadeghi, D. Tr, S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. reza Sadeghi. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks.

[39] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 49–54, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0649-2. doi: 10.1145/2184489.2184500. URL http://doi.acm.org/10.1145/2184489.2184500.

[40] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST'11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21598-8. URL http://dl.acm.org/citation.cfm?id=2022245.2022255.

[41] S. Bugiel, S. Heuser, and A.-R. Sadeghi. myTunes : Semantically Linked and User-Centric Fine-Grained Privacy Control on Android. 2012.

[42] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924971.

[43] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android Security Framework: Enabling Generic and Extensible Access Control on Android. 04 2014.

[44] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 1005–1019, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL http://dl.acm.org/citation.cfm?id=2671225.2671289.

[45] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552, Bellevue, WA, 2012. USENIX. ISBN 978-931971-95-9. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/xu_rubin.

[46] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda. PatchDroid: Scalable Third-party Security Patches for Android Devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 259–268, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2015-3. doi: 10.1145/2523649.2523679. URL http://doi.acm.org/10.1145/2523649.2523679.

[47] ARM - ARM TrustZone, . https://www.arm.com/products/security-on-arm/trustzone.

[48] ARM Security Technology: Building a Secure System using TrustZone Technology. ARM White Paper, .

[49] Samsung Knox. https://www.samsungknox.com/en.

[50] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena. DroidVault: A Trusted Data Vault for Android Devices. In *2014 19th International Conference on Engineering of Complex Computer Systems*, pages 29–38, Aug 2014. doi: 10.1109/ICECCS.2014.13.

[51] GlobalPlatform - Trusted Execution Environment Guide, . https://www.globalplatform.org/mediaguidetee.asp.

[52] T. Brito.

[53] Genode Labs - Operating System Framework. https://genode.org/.

[54] Genode - An Exploration of ARM TrustZone Technology. http://genode.org/documentation/articles/trustzone.

[55] OP-TEE's Secure Storage, . https://github.com/OP-TEE/optee_os/blob/master/documentation/secure_storage.md.

[56] Linaro - Leading software collaboration in the Arm Ecosystem. https://www.linaro.org/.

[57] TEE Internal Core API v1.1, . https://members.globalplatform.org/kws/compliance1/GPD_TEE_Internal_Core_API_Specification_v1.1.2_CC.pdf.

[58] TEE Client API v1.0, . https://members.globalplatform.org/kws/compliance1/TEE_Client_API_Specification-V1.0_c.pdf.

[59] OP-TEE Explained, . https://github.com/OP-TEE/optee_os/blob/master/Notice.md.

[60] OP-TEE OS GitHub repository, . https://github.com/OP-TEE/optee_os.

[61] OP-TEE Client Library GitHub repository, . https://github.com/OP-TEE/optee_client.

[62] Linux Kernel Git repository. https://github.com/linaro-swg/linux/tree/optee.

[63] OP-TEE XTest GitHub repository, . https://github.com/OP-TEE/optee_test.

[64] OP-TEE Build GitHub repository, . https://github.com/OP-TEE/build.

[65] OP-TEE Examples GitHub repository, . https://github.com/OP-TEE/optee_examples.

[66] sierraware - sierraTEE. https://www.sierraware.com/open-source-ARM-TrustZone.html.

[67] armMBED. https://tls.mbed.org/.

[68] A. Merlo, L. Lorrai, and L. Verderame. Efficient trusted host-based card emulation on TEE-enabled Android devices. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 454–459, July 2016. doi: 10.1109/HPCSim.2016.7568370.

[69] H. Sun, K. Sun, Y. Wang, and J. Jing. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 976–988, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813692. URL http://doi.acm.org/10.1145/2810103.2813692.

[70] D. Liu and L. P. Cox. VeriUI: Attested Login for Mobile Devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, HotMobile '14, pages 7:1–7:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2742-8. doi: 10.1145/2565585.2565591. URL http://doi.acm.org/10.1145/2565585.2565591.

[71] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 8:1–8:7, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3024-4. doi: 10.1145/2637166.2637225. URL http://doi.acm.org/10.1145/2637166.2637225.

[72] D. Rosenberg. QSEE Trustzone Kernel Integer Overflow Vulnerability. In *Black Hat conference*, 2014.

[73] D. Shen. Exploiting TrustZone on Android. In *Black Hat conference*, 2015.

[74] S. Zhao, Q. Zhang, G. Hu, Y. Qin, and D. Feng. Providing Root of Trust for ARM TrustZone Using On-Chip SRAM. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, TrustED '14, pages 25–36, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3149-4. doi: 10.1145/2666141.2666145. URL http://doi.acm.org/10.1145/2666141.2666145.

[75] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[76] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, Vancouver, BC, 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[77] Z. Ning, F. Zhang, W. Shi, and W. Shi. Position Paper: Challenges Towards Securing Hardware-assisted Execution Environments. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*, HASP '17, pages 6:1–6:8, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5266-6. doi: 10.1145/3092627.3092633. URL http://doi.acm.org/10.1145/3092627.3092633.

[78] pxb1988 - dex2jar. https://github.com/pxb1988/dex2jar.

[79] iBotPeaches - Apktool. https://ibotpeaches.github.io/Apktool/.

[80] Java Decompiler - JD-GUI. http://jd.benow.ca/.

[81] Guard Square - ProGuard. https://www.guardsquare.com/en/proguard.

[82] NXP - IMX53QSB: i.MX53 Quick Start Board. https://www.nxp.com/products/power-management/pmics/power-management-for-i.mx-application-processors/i.mx53-quick-start-board:IMX53QSB.

[83] Kernel Definition. http://www.linfo.org/kernel.html.

[84] The story behind Genode's TrustZone demo on the USB Armory note = thestorybehindgenode'strustzonedemoontheusbarmory.

[85] U-Boot Boundary Devices Repository. https://github.com/boundarydevices/u-boot-imx6/tree/boundary-v2018.07.

[86] About SQLite. https://www.sqlite.org/about.html.

[87] mbedTLS. https://tls.mbed.org/.

[88] Boundary Devices - Nitrogen6X. https://boundarydevices.com/product/nitrogen6x-board-imx6-arm-cortex-a9-sbc/.

[89] The Buildroot user manual, . https://buildroot.org/downloads/manual/manual.html#_getting_started.

[90] LittleD. https://github.com/graemedouglas/LittleD.

[91] LibTomCrypt. https://github.com/libtom/libtomcrypt.

[92] OpenSSL. https://www.openssl.org/.

[93] QEMU. https://www.qemu.org/.

[94] 'SeCloak: ARM TrustZone-based Mobile Peripheral Control' source, . http://www.cs.umd.edu/projects/secureio/.

[95] U-Boot v2017.07 for i.MX platforms. https://boundarydevices.com/u-boot-v2017-07/.

[96] OP-TEE on I.MX6Q SDB Linux. http://mrvan.github.io/optee-imx6q-sabresd.

[97] ISO/IEC 14443:1, . https://www.iso.org/standard/70170.html.

[98] ISO/IEC 14443:2, . https://www.iso.org/standard/66288.html.

[99] ISO/IEC 14443:3, . https://www.iso.org/standard/70171.html.

[100] ISO/IEC 14443:4, . https://www.iso.org/standard/70172.html.

[101] ISO/IEC 18092, . https://www.iso.org/standard/56692.html.

[102] Overview of EMVCo, . https://www.emvco.com/about/overview/.

[103] EMVCo L1, . https://www.emvco.com/processes-forms/product-approval/mobile/level1/.

[104] NFC Forum Adds Analog Testing to Certification Program, . https://nfc-forum.org/newsroom/nfc-forum-adds-analog-testing-to-certification-program/.

[105] ISO/IEC 7816:4, . https://www.iso.org/standard/54550.html.

[106] ISO/IEC 7816:5, . https://www.iso.org/obp/ui/#iso:std:iso-iec:7816:-5:ed-2:v1:en.

Appendix A

Appendix

A.1 Standards and Specifications

Mobile devices have to be interoperable with the four following standards and specifications:

ISO/IEC 14443: It is the international standard for Proximity Cards and the transmission protocols to communicate with them. It is divided in four parts:

- **Part 1** [97]: Physical Characteristics;
- **Part 2** [98]: Radio frequency power and signal interface;
- **Part 3** [99]: Initialization and anti-collision;
- **Part 4** [100]: Transmission protocol.

ISO/IEC 18092: It is the international standard for the communication modes for the Near Field Communication Interface and Protocol (NFCIP-1), using inductive coupled devices operating at the centre frequency of 13.56 MHz for interconnection of computer peripherals [101]. It is heavily based on ISO/IEC 14443, but with a key difference; it uses a different command protocol to replace Part 4 of ISO/IEC 14443, while also including two communication modes, active and passive. This allows a compliant device to be able to communicate with other compliant devices via peer-to-peer or with compliant NFC tags. Given the two communication modes, ISO/IEC 18092 defines three modes of operation:

- **Read/Write** - The NFC-enabled device may read or write data on any compliant tags in a standard NFC data format.
- **Peer-to-peer** - Two NFC-enabled devices exchange information.
- **Card-Emulation** - Allows the NFC-enabled device to act as a tag (in this case, a contactless card) to be read by terminals.

EMVCo L1 [102]: EMVCo exists to facilitate worldwide interoperability and acceptance of secure payment transactions, by managing and evolving its specifications and related testing processes (card and terminal evaluation, security evaluation, interoperability issues, etc). Bank cards are an example. L1 [103] attests the compliance of a mobile device to the EMV Contactless Communication Protocol Specification. This level is granted to a mobile product that supports NFC-based payment (e.g. proximity payment).

NFC Forum Analog [104]: The NFC Forum Analog is a specification that aims to clear the path to interoperability, by making it easier for device manufacturers to build NFC Forum-compliant devices. It addresses the analog characteristics of the RF interface of an NFC-enabled device, with the intent of it to be used by manufacturers that want to implement such devices.

ISO/IEC 7816: Another very relevant standard is ISO/IEC 7816, especially parts 4 and 5. Part 4 [105] defines the structure of the messages that are to be exchanged between smart card and smart card reader. These are called Application Protocol Data Units. These messages include, but not only, means to read/write data on the card. It also specifies how to identify an application via Application ID. ISO/IEC 7816:5 [106] specifies how to register a new AID. APDU commands are divided into two categories: command APDUs and response APDUs. Command APDUs are sent by the reader to the card, which contains an obligatory 4-byte header and command-specific data. Cards respond by sending a response APDU to the reader, containing an obligatory 2-byte header and response-specific data. Additionally, ISO/IEC 7816:4 is independent of the physical interface technology, i.e., it supports contactless cards, close coupling and radio frequency.

With the exception of EMVCo L1, the standards must support reader mode.

A.2 OP-TEE Prototype Demonstration

In this Section we will present a simple use case of our OP-TEE DBStore prototype. It was conducted on QEMU, and we are able to show two terminal windows corresponding to the Normal and Secure Worlds. This use case consists of conducting the Initialization Protocol to establish a mutual authenticated channel, through which we are then able to issue SRPCs to DBStore running on the SW via the Invocation Protocol. This implementation follows the description provided in Section 4.2. This use case will be explained using the terminology present on that Section, so we advise the reader to refer to it. In this example, the NW application functions both as the local application, as it forwards messages to DBStore and receives the replies, and the remote client, responsible for issuing the SRPCs and conducting the protocols.

After the user is prompted with a simple command line, as shown on Figure A.1, he is then able to execute the Initialization Protocol and, afterwards, the Invocation Protocol, issuing the SQL commands he so desires.

A.2.1 Initialization Protocol

Figure A.2 illustrates the prototype's behaviour when executing the Initialization Protocol. The user first writes 'init', as shown on Box 1, thus beginning the protocol. In this simple example, the AppId being sent is '1' and the initial value for the counter is '0'. After incrementing the value of the counter, DBStore then generates the AES session key and the corresponding IV, as shown on Box 2. With this session key, DBStore is now able to encrypt the counter. The session key is itself encrypted using RSA via the Public Key the client sends to DBStore on M_1. This is illustrated on Box 3. Before replying back to the NW application with message M_2, the DBStore service first needs to persistently store the current session key and the IV. To do so, we use OP-TEE's Secure Storage API to store these files securely and protected against rollback attacks. It also updates and stores the current counter value. Box 4 illustrates the storage of these files. Finally, after DBStore replies back to the NW application, the NW application can decrypt M_2, verify if the message is fresh by checking if the counter corresponds to the expected value (n_1+1) and, if so, get the session key, as shown on Box 5.

A.2.2 Invocation Protocol

Now that the Initialization Protocol was finished, it is possible to start sending SRPCs, which will be encrypted with K_i, which is the hash of the previous session key concatenated with the current counter value. Figure A.3 illustrates this: the user starts by issuing a CREATE TABLE command, on Box 1, which is encrypted with K_i. M_3 also contains an hmac of this SQL command. Upon receiving this SRPC, DBStore starts by retrieving the session key and IV that it persistently stored before. It updates it, generating K_i just like the NW application did, and stores it back. This is shown on Box 2. Then, using K_i, it is able to decrypt M_3, thus obtaining the counter and the SQL command. It then verifies the freshness of the message, its authenticity and integrity (via the hmac), as shown on Box 3. If these security guarantees are assured, DBStore can then forward the SQL command to LittleD. After the command is executed, DBStore updates and stores the current counter value, encrypts the reply with K_i, generates the corresponding hmac and replies back to the NW application (Box 4). Upon receiving the reply from DBStore, the NW application decrypts it with K_i, getting the reply itself and the counter value. It then verifies freshness and authentication/integrity, like shown on Box 5.

Figure A.4 shows the issuing of two other SRPCs, which function exactly like we just described. First, the value '100' is inserted into the created table 't' (Box 1), after which a SELECT * is issued (Box 2). The result of this last command is shown on Box 3.

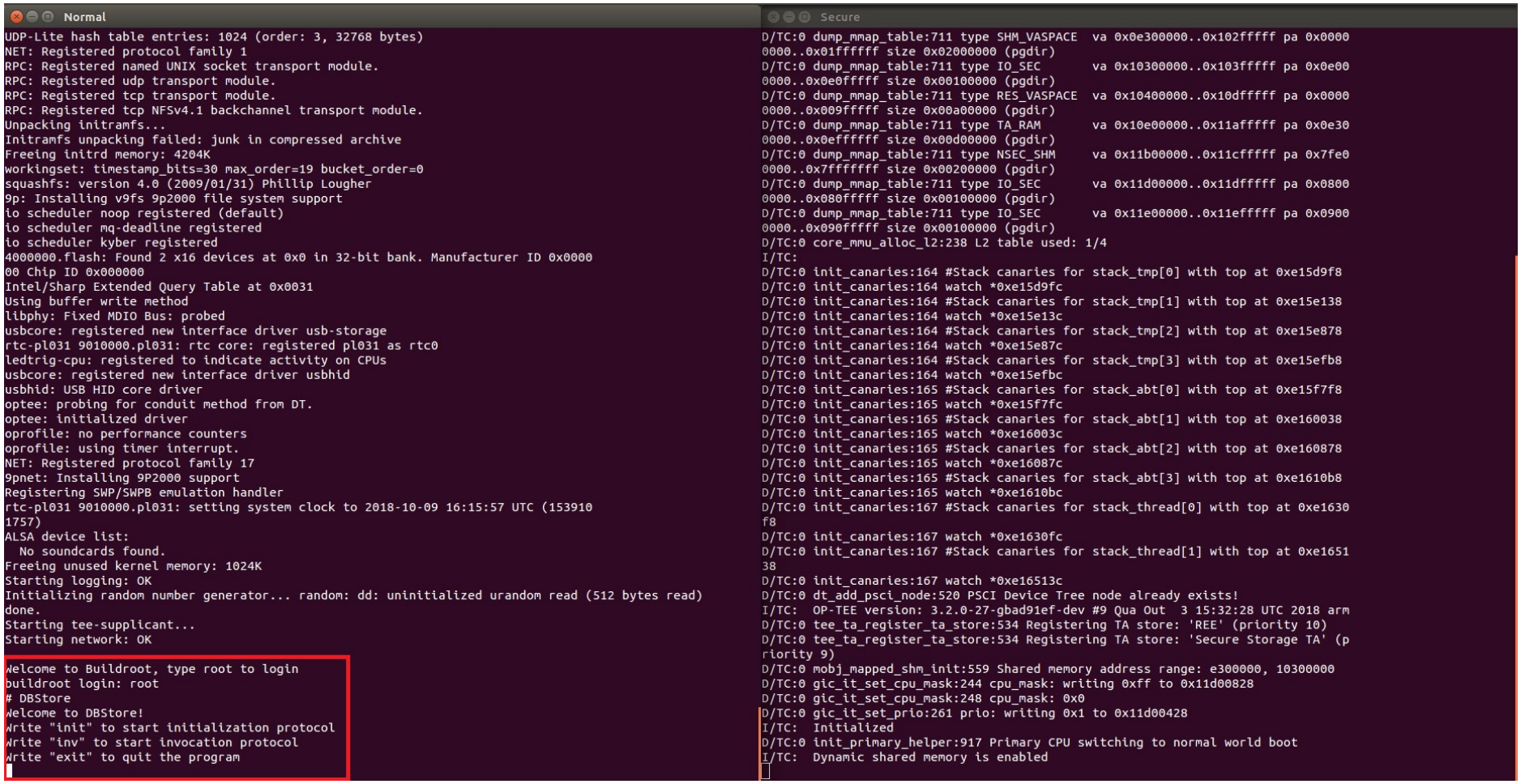


Figure A.1: Initial command line of the OP-TEE Prototype.

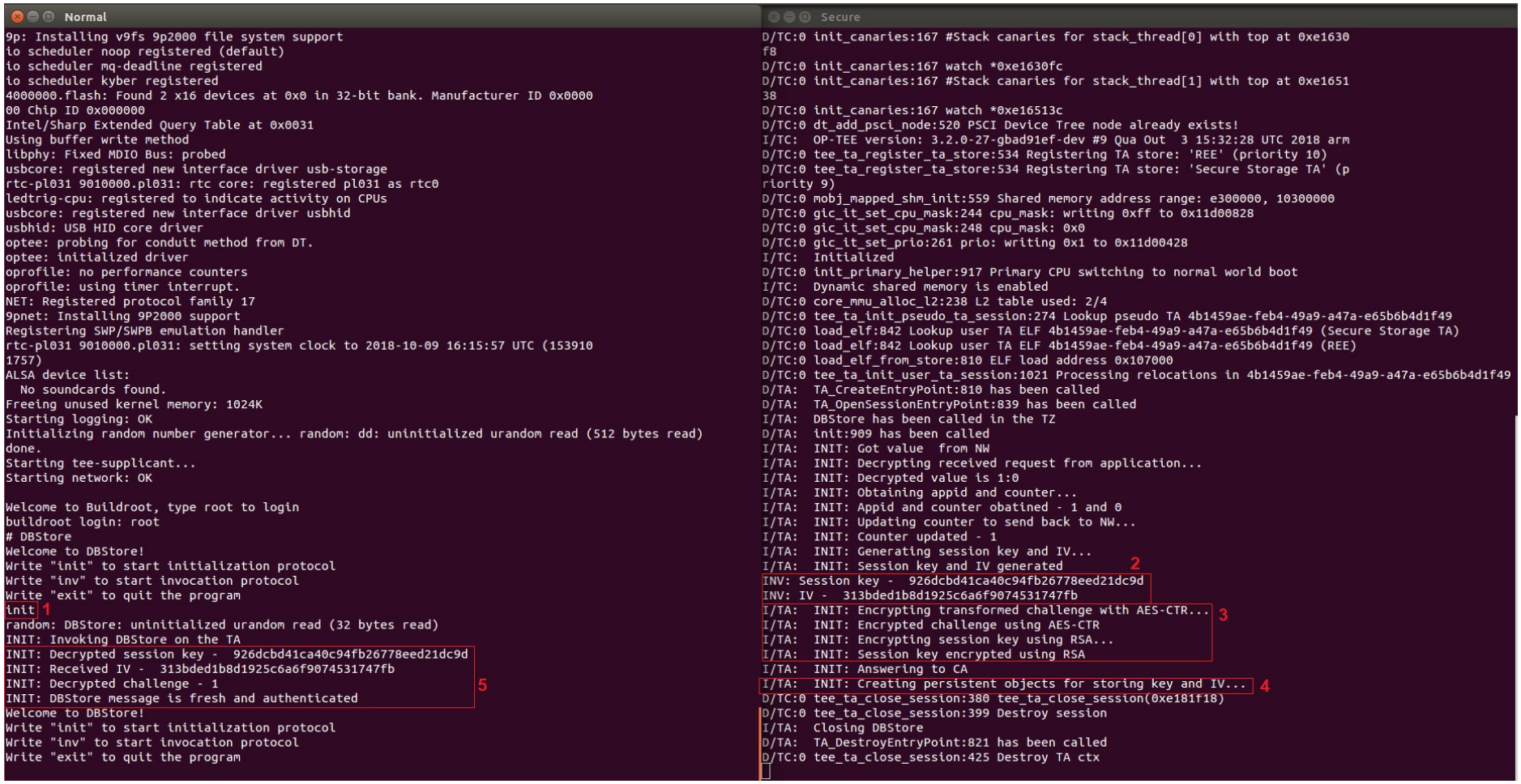


Figure A.2: Initialization Protocol with the OP-TEE Prototype.

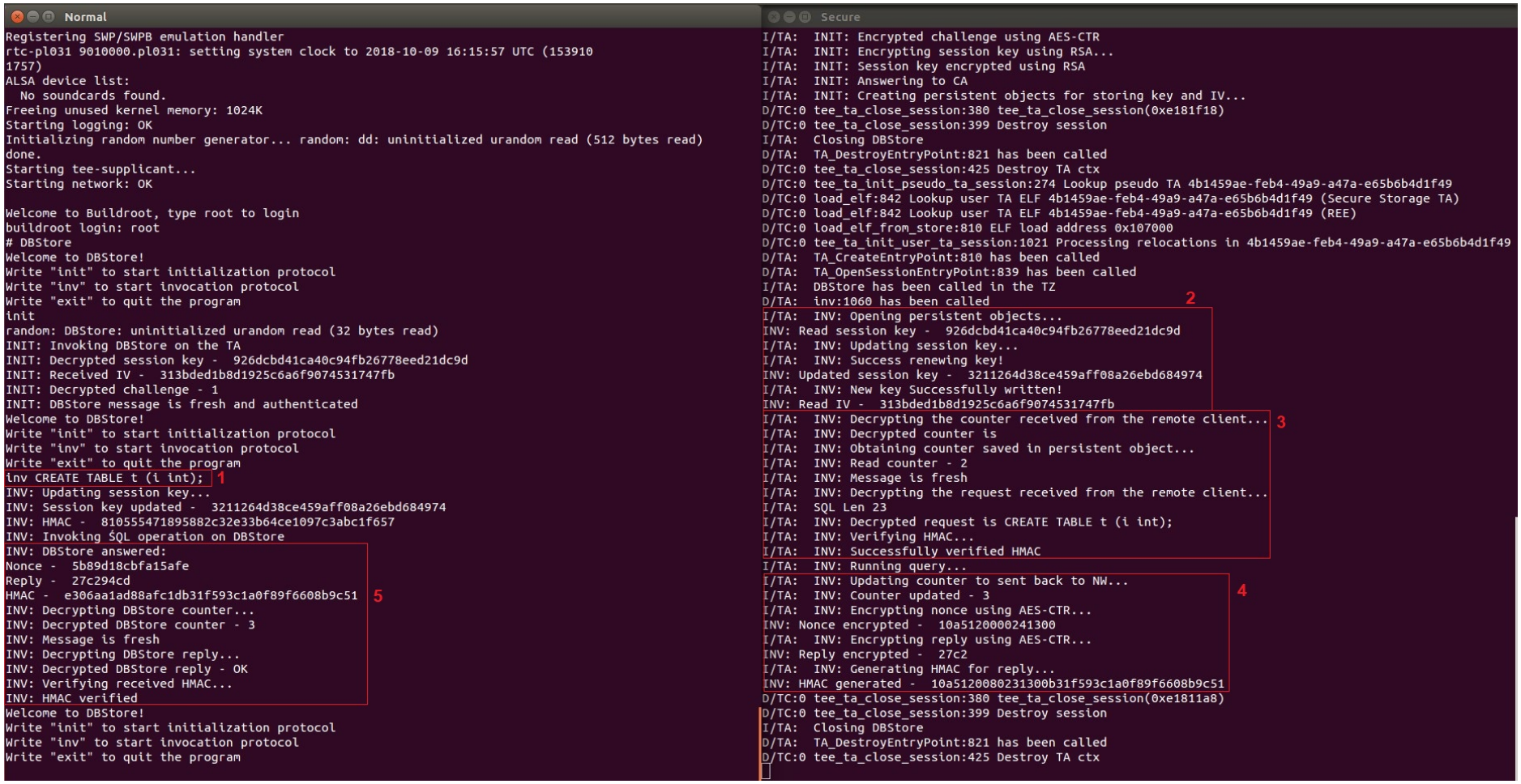


Figure A.3: Invocation Protocol with the OP-TEE Prototype, issuing a CREATE TABLE command.

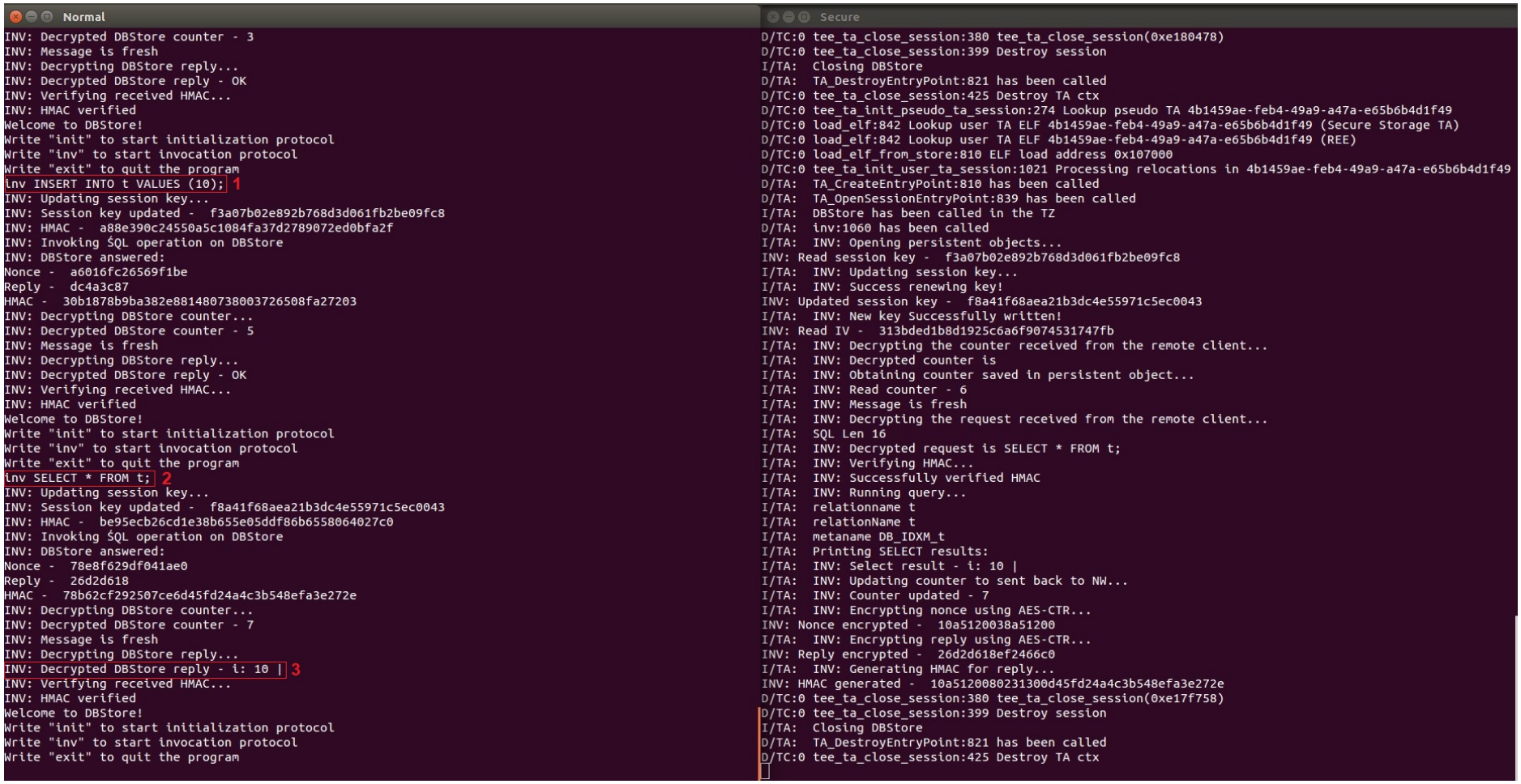


Figure A.4: Invocation Protocol with the OP-TEE Prototype, issuing INSERT and SELECT commands.