Université de Limoges

ÉCOLE DOCTORALE SCIENCES ET INGÉNIERIE POUR L'INFORMATION, MATHÉMATIQUES

On the Security of Microcontrollers: from Smart Cards to Mobile Devices

THÈSE

présentée et soutenue publiquement le 24-11-2016

pour l'obtention du

Doctorat de l'Université de Limoges

(spécialité informatique)

par

Tiana RAZAFINDRALAMBO

Composition du jury

Rapporteurs : André Seznec, Professeur, IRISA/INRIA, Rennes
David Naccache, Professeur, Ecole Nationale Supérieure, Paris

Examinateurs : Benoit Feix, Responsable Sécurité Cryptographique, UL
Jean-Pierre Seifert, Professeur, Université Technique de Berlin
Guillaume Bouffard, Ingénieur de Recherche, ANSSI
Eric Vetillard, Architecte Sécurité, Prove & Run

Directeur : Professeurs Christophe Clavier et Jean-Louis Lanet

Laboratoire de recherche XLIM UMR CNRS 7252

Xlim institut de recherche

eshard

UL

Mis en page avec la classe thesul.

À ma famille, mes amis et à tous les passionnés.

Remerciements

Je remercie tout d'abord Christophe Clavier et Jean-Louis Lanet qui m'ont beaucoup inspiré depuis la License et qui m'ont permis de sortir de la routine scolaire de la FAC, grâce à des projets et des sujets de discussion passionnantes et stimulantes. La curiosité, l'envie de comprendre, et la persévérance ont été les meilleurs enseignements qu'ils m'ont inculqué, et qui ne se trouvent dans aucun manuel scolaire. Un grand merci à eux deux d'avoir accepté d'être mes directeurs de thèse.

Je suis très reconnaissant envers les professeurs André Seznec et David Naccache pour avoir accepté de rapporter mes travaux. Je les remercie pour le temps qu'ils ont consacré à la relecture de ce mémoire, ainsi que pour leur présence dans mon jury. Un grand merci en particulier à David Naccache pour avoir accepté de présider ce jury de thèse.

Un très grand et chaleureux merci à Jean-Pierre Seifert, Guillaume Bouffard, Benoît Feix et Eric Vétillard d'avoir accepté de faire parti du jury.

Merci sincèrement Benoît de m'avoir si bien accueilli à UL et de m'avoir communiqué, inspiré et transmis, sans que tu ne le saches, la passion pour notre métier si particulier, même si nous n'avions pas forcément les mêmes domaines d'expertise.

Merci Guillaume d'avoir joué le rôle du grand frère, et de m'avoir guidé de la FAC à Michard, des cartes à puce au Docteur. Un gros bisous à Anne qui m'a aussi beaucoup supporté et encouragé.

Je remercie aussi très chaleureusement, Hugues Thiebeauld, Georges Gagnerot, Lionel Rivière, Antoine Wurcker et Pierre Carru, patrons et collègues passionnés, qui m'ont supporté, écouté, lu et conseillé jusqu'au bout.

Sur une note un peu plus personnelle, j'aimerais beaucoup remercier affectueusement mes parents et mon frère qui m'ont toujours poussé à persévérer, et qui m'ont toujours soutenu quels qu'eussent été les épreuves de la vie, et surtout pendant ces 3/4 dernières années. Bà mamy be!

Finalement, je te remercie infiniment Jenny pour m'avoir accompagné pendant tout ce temps et de m'avoir fais confiance jusque là et d'avoir affronté sur la ligne de front tous les hauts et les bas de la dernière année. Promis, j'aurais un peu plus de temps cette fois-ci <3...

Contents

Partie I Software Security of Java Card-based Secure Elements

Partie III Practical Time-Driven Cache Attack on a Mobile Device

Chapter 1

Introduction

1.1 What makes Smarpthones so "smart"?

1.1.1 From an Actual Phone to a Multi-Purpose Device

Moore's law[1] states that the number of transistors on an affordable CPU (Central Processing Unit) would double every two years. This implies the fact that the more transistors we have, the faster a processor can be. But somehow, more transistors also means more space. For 50 years, following Moore's law, continual shrinking of transistors has helped make computers and any electronic device more powerful, compact, and energy-efficient. Although Moore's empirical observation is now considered as obsolete [Man00], it has helped in bringing powerful Internet services, breakthroughs in fields such as artificial intelligence and genetics and smartphones.

Initially, so-called GSM (Global System for Mobile) phones were actual phones that provided the necessary for performing voice calls and SIM (Subscriber Identity Module) phone book editing features. Hereafter, Personal Digital Assitants (PDA) appeared and introduced touch screen user interfaces and a set of application programs (calendar, scientific calculators, etc.). Basically, PDAs were mobile electronic devices that function as a personal information manager. Afterwards, PDA's functionalities were slowly added to phones. Those phones were called "feature phones".

Nowadays mobile devices are called "smartphones". A very common definition that enables to differentiate between both is given by Welte [Wel10]. On one hand, a feature phone is basically a phone that contains a single application processor (AP) that is also called the baseband processor (BP). The latter is the only one responsible of running the software that handle the GSM protocol, managing the user interface and running the applications. On the other hand, a smartphone is a more evolved phone that may embed two main microcontrollers[2] that are physically separated from each other: the baseband processor that remains the one, amongst other radio-related functions, that handles the GSM protocol and the AP (generally multi-core) that is a general purpose one, and manages the user interface and applications.

However modern smartphones do not always implement such kind of design where the AP and BP are physically separated. As illustrated by the Figure 1.3, on the left we have the Samsung Galaxy S7 (SGS7) and on the right we have the iPhone 6. The latter has the AP (Apple A8) and the BP (Qualcomm MDM9625M) that are physically separated from each other, where the

[1] http://www.mooreslaw.org/

[2] A microcontroller contains the actual CPU and memories (RAM, ROM, flash) with different peripherals, while a microprocessor only contains the CPU.

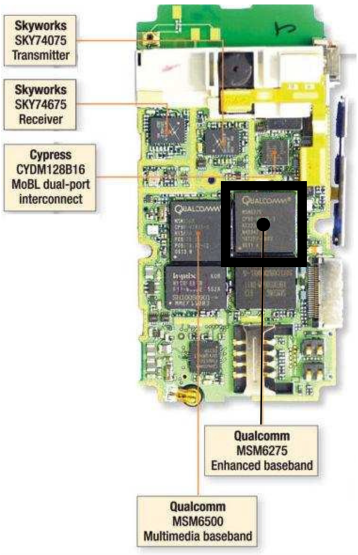

Skyworks SKY74075 Transmitter
Skyworks SKY74675 Receiver
Cypress CYDM128B16 MoBL dual-port interconnect
Qualcomm MSM6275 Enhanced baseband
Qualcomm MSM6500 Multimedia baseband

Figure 1.1: Inside a feature phone that embeds the MSM6275 Qualcomm's baseband processor (src: http://eetimes.com) that packages the application processor and the modem.

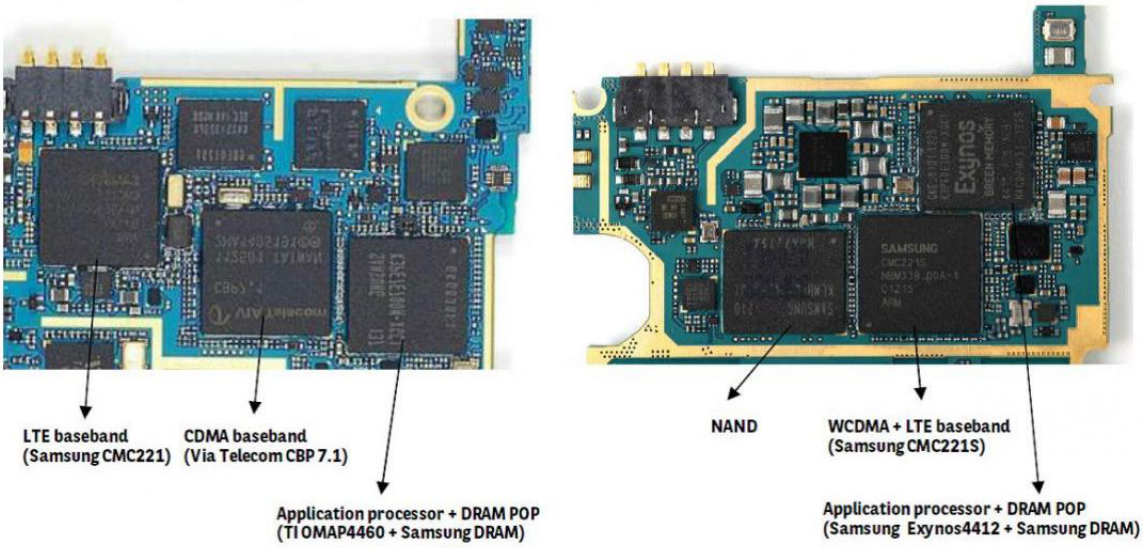

LTE baseband (Samsung CMC221)
CDMA baseband (Via Telecom CBP 7.1)
Application processor + DRAM POP (TI OMAP4460 + Samsung DRAM)
NAND
WCDMA + LTE baseband (Samsung CMC221S)
Application processor + DRAM POP (Samsung Exynos4412 + Samsung DRAM)

Figure 1.2: Example of two devices where the application and baseband processors are physically separated (src: http://www.semiwiki.com).

SGS7 has both integrated within a single chipset, the Qualcomm MSM8996 Snapdragon which is also referred to as a System-on-Chip (SoC).

1.1.2 The Operating Systems that Make the Magic Happen

Smartphones are now so much more than phones or personal organizers. They are mobile web browsers, video and music players, cameras, electronic wallets and more. They are small computers and as we have seen, they feature more than a single microcontroller to make all the magic happen. What exactly differentiate the main CPU with other sub-processors (*e.g.* video

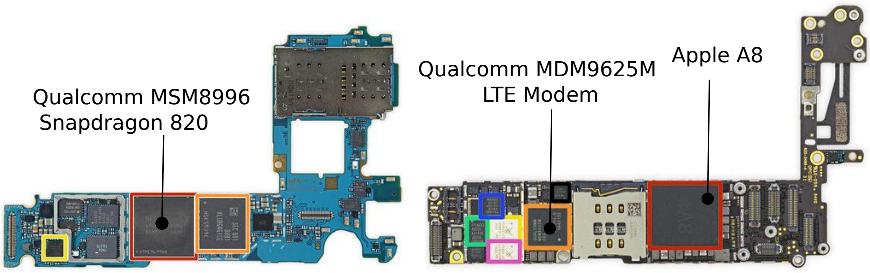


Figure 1.3: Design comparison between the Smasung Galaxy S7 and the iPhone 6 (src: http://www.ifixit.com)

encoders/decoders, audio playback, etc.) is that it has an Operating System (OS), the software responsible of providing the base management of the whole device (*e.g.* setting the device while booting, managing the underlying hardware and so on). There are many different mobile operating systems but the most popular ones are *Android* (Google Inc.), `iOS` (Apple) and `Windows Phone` (Microsoft).

Nevertheless, the AP is not the only hardware unit that has an OS. Nowadays, we can at least count two other microcontrollers that are controlled by other OSs that can run independently from the main one. We have already seen the first one, the baseband processor, and it is controlled by a proprietary real-time baseband OS such as for example `QuRT` (Qualcomm) or `ThreadX` (Apple). The second one is the secure element (SE) which is a tamper resistant integrated circuit (IC) capable of running smart card applications and is also controlled by a proprietary OS. One of the most popular OS for SEs is Java Card that provides platform interoperability and post-issuance application management.

SEs can have different form factors where the most known one is the `SIM` (Subscriber Identity Module) card that can mainly run GSM-related applications. With the advent of more services, a major evolution of the SIM card has been the introduction of the `UICC` (Universal Integrated Circuit Card) which is actually a secure smart card – also known as chip card – that provides a more secure storage, advanced 4G/LTE mobile network capabilities, more storage space and particularly more secure cryptographic capabilities. The other form factors are the `eSE` (embedded Secure Element) a tamper-proof chip, and the `microSD` (Secure Digital) card that provides a flash memory combined with a dedicated `eSE`.

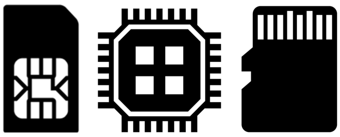


Figure 1.4: Illustration of the different form factors of a Secure Element within a mobile device (src: http://www.applus.com)

1.2 Motivation

1.2.1 Study of the Attack Surface and the Attacks Implementations

The attack surface on mobile devices is amazingly large due to the numerous attack vectors that exist. Those attack vectors can also be referred to the entry points from where an attack can be carried out. Those entry points mainly depends on the different high level features (Messaging, Voice calls, etc.) provided by the software, and the low-level features provided by the hardware (cache mechanism, hardware implementation of a cryptographic operation, hardware protection, etc.) that the device provides. Considering those attack vectors, we can define two categories of attacks. *Software* and *Physical Attacks.*

1.2.1.1 Software Attacks

They are also known as Logical Attacks and can be divided into two sub-groups of attacks: Remote Attacks and Local Attacks.

On the one hand, remote attacks refer to attacks that do not require to touch the device. Different wireless technologies can be considered to carry out such kind of attack (e.g. Wi-Fi, GSM, GPS, NFC, Bluetooth, etc.). They are the most attractive as in some cases, the adversary does not need to be physically near the victim, or at least in other cases, within a certain distance range.

On the other hand, local attacks require the adversary to have the device at his disposal in order to perform them. During our experimental research we mainly performed local attacks. They are useful for learning the target platform and improving the attack techniques, however, in some cases, considering a more sophisticated scenario, it is possible to turn a local attack into a remote one.

1.2.1.2 Physical Attacks

They consist of any attacks that involve any invasive or semi-invasive modification of a given hardware component or any passive observation of the behavior of the latter.

Invasive attacks – Firstly, an invasive or semi-invasive attack both involve physical alteration of a part of the microcontroller. The difference between both resides in whether after the physical alteration the package of the SoC remains intact or not. For example, so-called microprobing attacks exist and require to directly access the chip surface in order to observe, manipulate and interfere with the integrated circuit. Be it a SE, an AP or a BP, according to the type of packaging of the microcontroller, if the chip surface is not directly accessible due to some layers of some types (plastic, resins, metal) some techniques could be used to get access to the latter such as using wet chemical etching, drilling and so on. Any invasive attacks are quite complicated and require highly qualified specialists and a proper budget. The microcontrollers may remain functional, however such kind of attack definitely destroy the package.

Semi-Invasive attacks – Secondly, any attack that consists of modifying hardware to perform a function, either not originally conceived for, or initially hidden in one way or another.

Different entry points can be considered in such kind of attacks such as for example, the CPU itself, the serial ports, any debug ports (if available, *e.g.* JTAG[3], Serial Wire Debug[4]), the flash memories, and more. Such kind of attack may be destructive because of the limited physical access to some hardware components which also may requires to remove a or to add hardware.

Non-Invasive attacks – Finally, physical attacks can also be non-invasive such as the adversary still exploits the specificities of a target hardware implementation, but without any physical access. A very well-known attack technique in the area of cryptography that correctly defines such kind of attacks is called *Side-Channel Analysis* (SCA). Basically, it consists of measurement-based inference techniques to highlight the behavior of a given hardware component such as the main CPU, and to recover information and data manipulated by a targeted algorithm. Examples of techniques can be the measurement of the power consumption, the analysis of the electromagnetic emanation, or the measurement of the execution time of a part of an algorithm or a whole program.

1.2.2 Hardware and Software Security of Microcontrollers in the Context of Mobile Devices

This thesis is concerned with the security of the three main microcontrollers that are managed by independent OSs within mobile devices; the SE, the baseband processor and the application processor, *i.e* the main CPU. Their security depends on the hardware and the software implementation of the numerous features that they provide.

First of all, SEs implements various techniques to implement tamper resistance to prevent from extracting information from it. It is considered as a digital vault as it provides a very secure storage. Furthermore, it generally provides a co-processor with cryptographic computation capabilities. The specification [Ora15] defines its implementation. However, this does not ensure that the actual implementation strictly respects it. Consequently, it is likely that some platforms may embed vulnerabilities due to the specificities of their implementation.

Secondly, one of the key components that has directly access to the SEs is the baseband processor. If we take as an example an open source mobile device's OS like Android, the baseband OS is one of the few close-source software that run on the device. No public documentation is available (unless leaked). Having control over the baseband side allows an attacker to perform different attacks related to the "phone" part of a device. Therefore it is worth to be aware of the security impacts of this companion processor over the whole system.

Finally, the security of software that is run by the application processor and the sensitive assets that are manipulated may also depend on the hardware implementation of the latter. In cryptography, a very well-known technique for passively extracting information from a system without any knowledge of the internal workings is called Side-Channel Analysis. The most popular techniques involve inferring information by measuring the power consumption, or the electromagnetic emanation during the execution of a sensitive operation. However, other techniques that do not require physical measurements also exist and can be used to gather and analyse information.

To summarize, the Figure 1.5 illustrates the major topics that are studied throughout the dissertation. The dotted arrows refer to topics that have not been considered at all.

[3] Joint Test Action Group

[4] For ARM processors, http://www.arm.com/products/system-ip/debug-trace/coresight-soc-components/serial-wire-debug.php

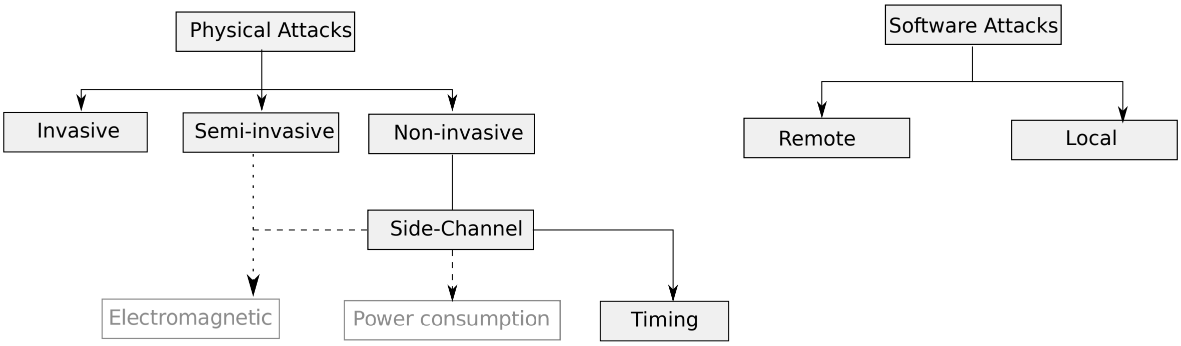


Figure 1.5: Approximate taxonomy of the studied major topics

1.3 Contributions and Outline

Major contributions of the PhD studies research work summarized in this dissertation are as follows:

Software attack on Java Card Platform: The chapter 4 is a proposal of a new software attack technique on the Java Card platform that exploits the Java frame creation mechanism to perform a stack underflow attack. More particularly, our technique is not dependent on the Java stack implementation which may be totally different from one platform to another one.

Memory forensics of a raw Java Card memory dump: The chapter 5 is a proposal of a new heuristic to identify the code of a Java Card application within a raw memory dump that has been obtained after a successful attack. The clear objective in this experimental analysis is to improve memory forensics and reverse engineering capabilities while studying a secure microcontroller.

Security of the main CPU against Cache Timing Attacks: The chapter 10 studies the applicability of Bernstein's time-driven cache attack [Ber05] on real mobile devices. We particularly focus on highlighting the behavior of the main CPU while performing the attack through practical experiments and various adaptations of the original attack. Furthermore, while nobody in the literature explicitly shares the specific configuration within which their experiments have been performed, we particularly describe the different parameters that we have used to carry out our experiments.

The remainder of this dissertation is structured as follows: The next chapter (chapter 2) provides the reader the knowledge about the Java Card platform and an overview of the security of the latter. The chapter 3 introduces, explains and classifies the software attacks that are known in the literature. The Part II of the dissertation summarizes our investigations to answer the following question: what is the attack surface on a baseband processor from a local to a remote attack perspective? What are the means to perform security assessment of a baseband processor? How is the (U)SIM card managed by the baseband OS?

The Part III concerns our investigations regarding the applicability of so-called time-driven cache attacks on actual mobile devices. We provide an introduction to ARM processors and their multi-level cache memories in chapter 7. The chapter 8 summarizes our empirical analysis of the cache memories effects on a mobile device. The chapter 9 summarizes our findings regarding all the known so-called micro architectural attacks that exploit the behavior of the CPU due to the implementation of specific hardware mechanisms. In that chapter, we also particularly try to identify the different techniques that are used up to now in the area of cache attacks to extract information used to carry out further analysis.

Finally, the chapter 10 summarizes our experiments related to the application of Bernstein's time-driven cache attack[Ber05] on an actual mobile device.

Part I

Software Security of Java Card-based Secure Elements

Chapter 2

Security Concepts behind Java Card-based Secure Elements

Contents

In this chapter, we define the main aspects of the Java Card technology and basic knowledge regarding the security of the platform, and the main risks it faces. In the section 2.2, we provide the reader with a general understanding of the platform under study. The section 2.3 aims at introducing fundamental security concepts that gives a general picture of the security mechanisms that are involved in the Java Card technology. The final section 3.1 introduces the different categories of attacks that a Java Card paltform must withstand.

2.1 Introduction

2.1.1 What is a Smart Card?

The most known form factor of smart cards is the traditional credit-card which is a piece of plastic with fixed dimensions housing a small microcontroller at a fixed exact location defined by international standards[5]. Additionally, the contacts and positioning of the input and output (I/O) interfaces are closely specified and fully disclosed in the ISO7816 standard[6]. Smart cards offer a secure storage and a secure computing environment. It is managed by an operating system whose sophistication varies between the most rudimentary data access or update operation, to controlling the loading and the execution of applications with full security management.

The second major use of smart cards is within mobile devices. Inside every phone resides a very small smart card which is commonly known as the SIM card. The SIM card is used to store a secret SIM key that is used in a challenge/response protocol (see Part II, section 6.2.3.2) to authenticate the SIM to the mobile network. However, SIM cards are not the only form factor of secure elements within a mobile device.

2.1.2 What is a Secure Element?

2.1.2.1 A Tamper Resistant Microcontroller

A secure element is a tamper resistant smart card chip that provides the ability to run smart card applications, to securely store data and to execute cryptographic functions by means of co-processors that implement common algorithms such as RSA, DES, AES. Smart Cards use various hardware and software techniques to implement tamper resistance, making hard the analysis of the chip to extract data by means of either physical attacks (see section 1.2.1.2) or software attacks (see section 1.2.1.1). They are generally pre-programmed with a multi-application OS, namely an OS that enables to load and execute multiple applications. The memory protection features provided by the platform (hardware + software) ensures the segregation of each application and their data.

2.1.2.2 A Smarter Contact and Contactless Smart Card

Typical smart card has contacts set aside to enable the communication between the chip and a card reader. However, smart cards can also be contactless and they do not need to be physically connected to a reader. The delivery of power and the transmission of data to and from the card are achieved at a distance by means of different technologies: close-coupling (ISO/IEC 10536), remote-coupling (ISO/IEC 14443 and ISO/IEC 15633) or NFC (Near Field Communication) that uses contactless identification according to the ISO/IEC 14443 standard and operates over the air interface according to the ISO/IEC 18000-3 standard.

Nowadays, modern smartphones also support NFC technology. In this case, the secure element is connected to a NFC chip making the wireless communication with the SE possible. An illustration of the possible layouts of the NFC Chip, the SIM Card and the secure element within a mobile device is illustrated by the Figure 2.1.

The secure element has at least equivalent capabilities to that of a smart card. It also can have superior capabilities since it can host multiple applications, it enables to manage within a

[5] http://www.iso.org/iso/catalogue_detail?csnumber=31432

[6] http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx

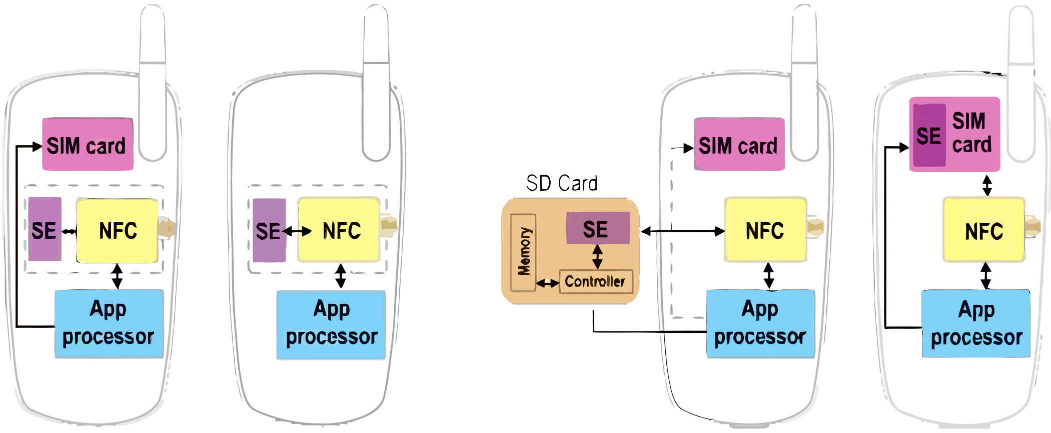


Figure 2.1: Secure Element Solutions (src: *http://smartcardalliance.org/*)

single device many different types of cards or security tokens (access cards, loyalty cards, One Time Password generators, Public Key Infrastructure credential storage, and so on).

2.2 Java Card Technology

2.2.1 Java Card Architecture

Java Card is a multi-application platform that enables programs written in Java to run Java bytecode on smart cards. It supports only a carefully chosen and customized subset of the features of the Java platform. Therefore smart card manufacturers can design their own Java Card technology-based implementations. It defines a runtime environment, the Java Card Runtime Environment (JCRE). The latter is on top of the hardware, the native operating system and the Java Card Virtual Machine (JCVM). The JCRE is paired with a high level standard interface to Java Card applications, the Java Card API (JCAPI). As a result, it enables for rapid application development. A card issuer may also provide its own API on top of the JCAPI.

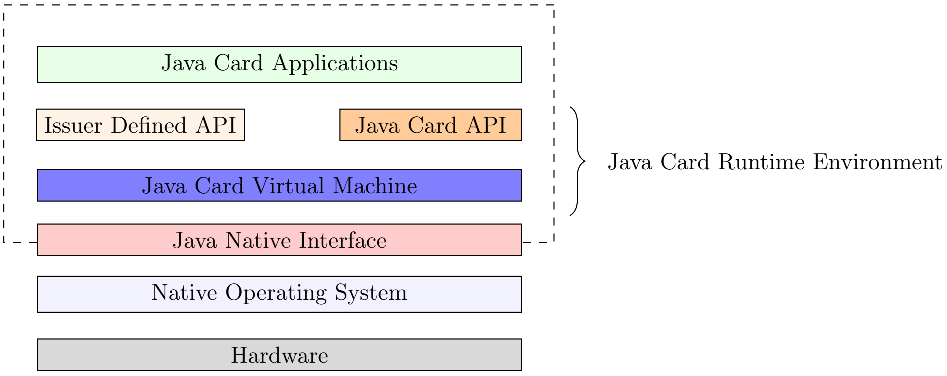


Figure 2.2: Java Card Architecture overview

2.2.2 Java Card Virtual Machine

The Java Card Virtual Machine mainly consists of two separate components: the *converter* and the *interpreter*.

The *converter* is the off-card component of the JCVM and runs on a host computer. As any other Java program, Java Card applets are first compiled and translated into a binary form that respects the `CLASS` file format. The *converter* is then in charge of converting the latter into a Converted APplet (`CAP`) file that suits better the Java Card platform and the underlying hardware. The *converter* takes as an input two files, `CLASS` files and an `EXPORT` file. For the sake of memory constraints, all the public information needed for linking every classes and methods within the `CAP` file, are stored within the `EXPORT` file and are never stored on the device.

The *interpreter* runs on the Java Card device, and is responsible for executing the code of the application, and enforcing the runtime security.

2.2.3 Java Card Runtime Environment

The JCRE is a masterpiece in the Java Card architecture and can be considered as the device's operating system, as it manages the resource management, I/O communication, the general security model enforcement and the applet life cycle. The JCRE is initialized at card initialization time and takes care of instantiating the applets. It is also responsible of handling the commands sent from the external world that are also called Application Protocol Data Unit (APDU) commands as defined by the standard `ISO7816-4`[7]. As many applets can be installed on the device, one of the main step to perform is to request the JCRE for selecting a specific applet thanks to its Application ID (AID). The JCRE forwards the APDU packet to the `select` method of the applet and tells the JCRE if the applet is ready or not for processing other commands. In case the applet is ready, it takes the control and passes the received command to its `process` method that will handle the command.

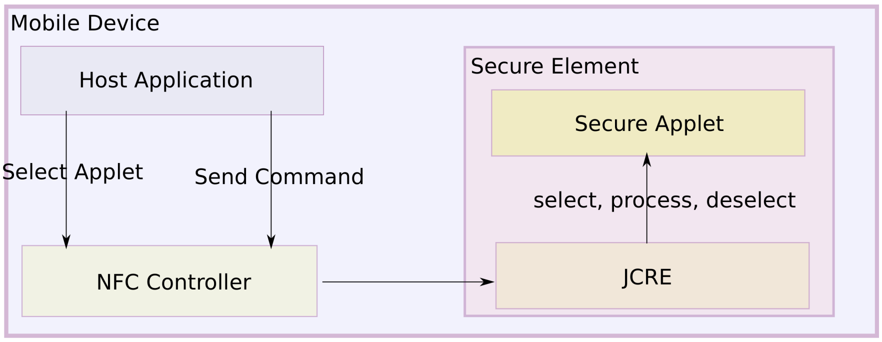


Figure 2.3: Communication between a host application and a secure applet installed on a secure element embedded within a mobile device

[7] http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4_5_basic_organizations.aspx

2.3 Security Concepts in a Multi-Application Environment

2.3.1 Application Verification

To preserve the Java Card platform from being attacked with tampered applications, a very important component of the JCVM, called the Byte Code Verifier (BCV), is run for statically verifying the correctness of the internal structure of the application. While in the regular Java Virtual Machine (JVM), this piece of software is completely integrated within the VM itself, on Java Card platform, due to memory and performance constraints, it is common to retrieve the BCV as an off-card piece. However, there is no doubt that in a near future, all the devices will be able to run such resource-consuming process.

2.3.2 Application Loading

Another corner stone of the Java Card platform security is the application loading process. In case it is not secured, it would give the freedom to arbitrarily install applications. Therefore, for security purpose, a code loading mechanism is generally implemented for enforcing the integrity and authenticity of the application through public key or symmetric key cryptography. Consequently, one has the notion of "loading keys" that may be owned either by the card issuer, or by a third party. In the second case, the loading keys are derived from a master key that is also either only owned by the card issuer or shared among different third parties. They are required for installing an application on the device. For instance, practically all SIMs used today implement GlobalPlatform card specification[8] so applet can be installed by authenticating first to a pre-loaded applet called the `Card Manager`, also called the `Issuer Security Domain` (see section 2.3.3.2). The latter has to be used for uploading and installing new applets by issuing `LOAD` and `INSTALL` commands after selecting it and performing an optional authentication.

An important difference SIM cards have compared to regular smart cards is that it supports Over-The-Air (OTA) updates via binary SMS (Short Message Service) or over TCP/IP (Transmission Control Protocol/Internet Protocol) through the mobile network (GPRS/3G etc.). For Java Card-based SIM cards, they are pre-personalized by the network operator with an "OTA profile" that enables them to retrieve and execute OTA commands. Hence, file management and more particularly applet management can be performed remotely by network operators on the SIM card.

2.3.3 Application Isolation

2.3.3.1 The Firewall

The Firewall is a mechanism that provides a means to guarantee the segregation between applets' data on the device. It mainly prevents an arbitrary applet from unwanted interference from another applet. To each application is assigned a security context, or also called, execution context. Furthermore, to several applications can be assigned the same security context. Subsequently, all objects belong to some security context. The latter defines the access rules that protect a given object (field access, method call, etc.), and thus, it can only be accessed by the current running security context. Any attempt to illegally perform an access may result in a `SecurityException`. Only the JCRE is able to bypass those rules, as it is the one that has the highest privileges.

[8] http://www.globalplatform.org/specificationscard.asp

To enable data exchange, the shareable interface mechanism is provided for giving the ability of two applets to communicate through a shareable interface object (SIO). The SIO is the only interface that a client applet must request to a server applet for getting access to the functions exposed by the SIO.

2.3.3.2 The Security Domains

Security domains are privileged applications that manage their own cryptographic keys. They enable the coexistence of multiple applications on the same device without violating the privacy and integrity of each application provider. Security domains are grouped in 3 types:

1. the issuer security domain: represents the on-card representative of the card administrator

2. the application providers security domain: consist of representatives of application providers, or card issuer

3. the controlling authorities security domains, which are optional and offer further confidential personalization services to authenticated application providers

2.3.4 Operation Atomicity

For the sake of data consistency and integrity, it is critical to ensure that the data that are manipulated at runtime are protected against external events that would abruptly stop an ongoing operation due to a power loss. Thus, while such events occur, the platform must implement a mechanism that should be able to recover the previous state of the data. The platform provides the ability to protect critical data by means of a Transaction Mechanism, that mainly keeps track of the data state before the transaction begins until it ends.

2.4 Summary

We have provided the reader an overview of various security concepts around Java Card platform. The latter can benefit from a layered security to provide a very secure storage and computing environment. From the lowest level to the highest one we have:

- a multi-application environment that can run and segregate multiple secure applets thanks to hardware and software protections

- a semi-open platform that enables to perform post-issuance applet loading only on condition one has the right keys to authenticate to a given security domain that ensure that one has the right to load and install a given apple

- a split virtual machine architecture that provides an off-card applet verifier that could be run by a third-party validation authority to ensure that the applet is structurally and semantically correct

From this layered architecture, one can ask the corresponding questions:

- is the current Java Card platform implementation fully bullet-proof, and what a malicious applet can do from the inside?

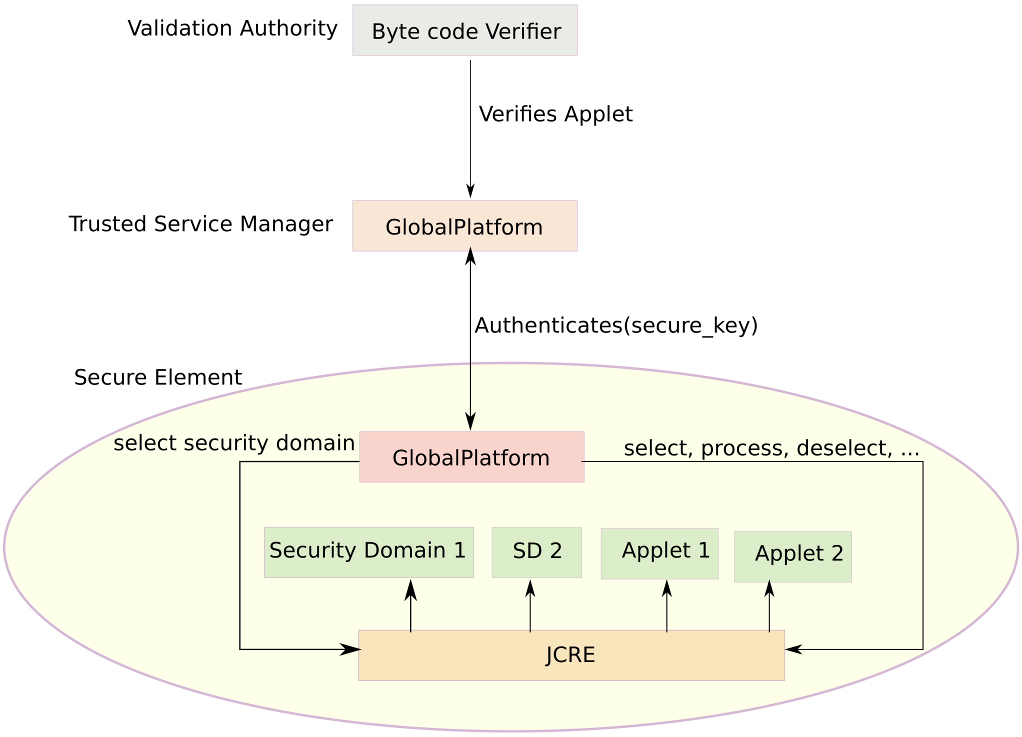


Figure 2.4: Java Card: a layered security with different actors

- how much can we trust our "business partners" that may also be present in the same secure element, can we fully trust the security domains implementations that provide the powerful ability to perform post-issuance applet loading and installation?
- how much can we trust to the so-called validation authority that has to verify applets that have to be installed in the same secure element as ours?

To answer these questions, the next chapter provides the reader the knowledge about software attacks applied to Java Card platforms following a basic assumption that defines our adversary: regardless how a malicious applet has been uploaded onto the secure element, it may potentially embed malicious code that may threaten other applications or the whole platform.

Chapter 3

Introduction to Software Attacks on Java Card Platforms

Contents

This chapter presents the related work regarding pure software attacks on the Java Card platform. This topic is crucial for the reader, in order to give thorough insights regarding the possible attack techniques that exist on the platform. Therefore the section 3.2 will mainly focus on the study of the different attack techniques that are presented in the current literature applied to the Java Card platform. The section 3.3 briefly presents our generic framework that enables to test both the platform implementation and the applet that has to be executed on the latter. But first, we provide a brief overview of the security risks of Java Cards in the section 3.1.

3.1 Security Risks of Java Cards

Java Card devices are prone to various attacks, however, we will only focus on a given type of attacks. Furthermore, in order to give the reader an overview of the large attack surface in this field, a brief description of the different possible attacks are given in the next sections.

3.1.1 Invasive attacks

Such kind of attacks are destructive, and employ advanced methods for de-packaging the chip, in order to directly access the memory, or for data analysis by means of micro-probing techniques. Such kind of attacks are often quite complicated and require high-end laboratories and qualified specialists. In [AK96], the authors describAirlineTablineRefreshAirlineTablineRefreshe various destructive attacks that they carried out in order to retrieve sensitive assets that are meant to be protected.

Other attacks less destructive exist, though they still require in most of the case, de-packaging the chip. Those attacks refer to semi-invasive attacks.

3.1.2 Semi-Invasive attacks

Semi-invasive attacks are less destructive, as the protective layer of the chip remains untouched, and is still functioning. Some examples of this type of attack include the observation of the electromagnetic emanations, or the injection of faults by means of a light source, such as a laser light for instance [Sko05, BBKN12, Sko10, SHP09, Bar12]. While this kind of attack does not require as expensive equipment as the invasive attacks, the overall effort that is needed to carry on an attack still remains relatively high.

3.1.3 Non-Invasive attacks

3.1.3.1 Side-channel attacks

Side-channel attacks consist in passively analysing an operation that is being executed by the processor by means of measurement techniques that do not tamper with the device. Such attacks include Timing Analysis [Koc96], Simple Power Analysis [Man02] or Differential Power Analysis [KJJ99, CCD00].

As an example, the algorithm 1 illustrates a PIN comparison that clearly introduces time variation in the execution time of the comparison process. The execution time will be shorter in case the comparison stops (line 3). On the opposite, it will be obviously longer if all the characters of the PIN is checked. Such timing variation can be exploited to guess a secret PIN code.

Algorithm 1 Dummy PIN comparison

```
1: procedure CheckPIN(InputPIN, RealPIN)
Require: $InputPIN = \{cA_1, cA_2, \cdots, cA_N\}$, $RealPIN = \{cB_1, cB_2, \cdots, cB_N\}$
2:     for each $i^{th}$ character $\in InputPIN$ do
3:         if $cA_i! = cB_i$ then
4:             return $PIN\_VERIFICATION\_FAILED$
5:     return $PIN\_VERIFICATION\_SUCCEEDED$
```

In addition to the time execution, nowadays, it is common to also verify during evaluations, if other sources of measurements can leak information, such as the power consumptions, or the electromagnetic emanation. Though those attacks have already proven their efficiency in retrieving secret data, for the sake of measurement precision, expensive equipment are still required.

I will rather focus on another non-invasive type of attack, that only costs the price of a regular computer. This topic is further studied Chapter (3).

3.1.3.2 Software Attacks

They are the cheapest ones as they only require at least the targeted device and a computer. They can be divided in two different attack vectors: locally or remotely. In the context of secure elements, a local attack consists of any techniques that implies that the adversary has the target device at his disposal. In some case local attacks can also be specifically designed to be turned into remote attacks.

Two attack settings can be defined according to the privileges that our adversary has. In the first attack setting he has no privilege, thus he does own any loading keys (see section 2.3.2) that would enable him to authenticate to the `Card Manager` which could grant him the rights to load and install his applet(s). However, he can at least communicate with the secure element by issuing commands (APDUs, binary SMS, etc.). In the second attack setting, he has the loading keys which is a powerful situation.

Non-Privileged Attacks. Lee *et al.* in [LJL05] proposed an approach for detecting trapdoors, namely, hidden features that can be triggered through an APDU command. While the authors applied timing and power analysis for identifying possible trapdoors, Putt *et al.* work in [PPL14], particularly searched for hidden commands using the fuzzing technique, *i.e*, identifying the entry point of a target and characterizing the input format, generating pseudo-random or chosen inputs, and injecting them for testing the target and analysing the outcomes.

Nohl [Noh13] demonstrated that SIM cards that use weak cryptographic algorithms such as the `DES` (Data Encryption Standard) could be locally and remotely attacked to recover the secret `DES` key that is used to sign binary SMSs (see ETSI TS 102 225 and ETSI TS 131 115 standards) that are issued during an OTA software updates by the network operator (as specified by the standard ETSI/3GPP for GSM 03.48). This could be done by explicitly requesting a proof of request (PoR) to the SIM card so a `DES`-signed SMS response is returned. The author carried out a so-called *code book* attack [Bir11, Noh] to recover the secret key. During one year, the author pre-computed very large tables (several petabytes) that enable him through only one hard disk access to make a correspondence between a ciphertext and a plaintext. The consequence is that it is then possible to simulate an OTA update and issue a binary SMS that embeds a (malicious) applet that can be downloaded onto the SIM card. Moreover, as demonstrated by Alecu [Ale] it is possible to craft the SMS such as it is directly sent to the SIM card without any notification.

Roland demonstrated [RLS12] a relay attack against the Google Wallet application, namely, an attack technique that involves the attacker to initiate the communication with both the secure element and the card reader. The application uses a secret PIN code generated and stored in the secure element that is normally used to grant access to the application. However, on the version of the application he has carried out the attack, the PIN code verification was performed outside the secure element. Instead, within the latter, the Google Wallet Java Card applet only expects a single "unlock" command through an APDU packet, so the author could initiate an actual payment.

Privileged Attacks through Malevolent Applications. In this case, the attacker performs the attack from the inside through a malicious application. We particularly put the focus on these attacks in the next sections.

3.2 Attack Techniques through Malevolent Applications

3.2.1 Concepts of Well-formed and Ill-formed Applications

So far, one can divide pure software attacks on smart cards in two main attack vectors: either using a well-formed applet, or an ill-formed application.

A malicious well-formed application consists of an application that has malevolent behaviors, but remain structurally and semantically correct, namely, well-formed. It has the ability to explicitly perform malicious operations such as for instance, exploiting platform bugs or, simply by returning sensitive data to the external world.

However, an ill-formed application is a `CAP` file that has been modified in order to embed a particular sequence of bytecode that is not usually produced by a compiler. The specific characteristic of an ill-formed applet is the fact that after modification, it may not be structurally or semantically correct any more. Therefore, it is not likely that such kind of attack passes the checks performed by an off-card BCV nor an on-card one.

There are two other attack vectors: attacks that combine the use of external means (e.g: laser beam) to modify the behavior of a well-formed code, also known as *combined attacks* and attacks that combine those external means and an ill-formed code, known as *ill-combined attacks*. However, they are not pure software attacks as they involve the use of a laser beam to alter the application and thus its behaviour in order to turn it into an ill-formed application. Those attacks may be powerful where the targeted secure element embeds an on-card BCV. These attacks are out-of-scope of this study.

3.2.2 Malicious Well-formed Application based Attacks

Malicious applets are well-formed applications that are structurally and semantically correct. These applets can either attempt to gather information on the platform and other loaded applets, or to attack the platform with the aim of circumventing the security mechanisms by misusing some features. Such applets can try to discover weaknesses in the platform and its environment.

For instance, an applet can try to use all the cryptographic services defined in the Java Card specifications [Ora15] in order to figure out which one is present on the card. As an example, Svenda developed an automated tool, JCAlgTest [Svea, Sveb], that includes an applet that perform various tests: mainly, identifying the cryptographic algorithms that is supported by the card, and their performance. The main reason behind such kind of tests, is that the set of cryptographic algorithms supported by a given device is sometimes hard to obtain from vendor's specifications. Though this testing tool does not have any malicious intention, for reverse engineering purpose, such kind of applet enables to gather valuable information regarding the targeted device.

However, in the literature, some possible attacks through well-formed applications have already been studied. An introduction to those attacks is given in the next two sections.

3.2.2.1 Abuse of the Java Card Firewall

The Java Card firewall mechanism enables to guarantee separation of applet data on the card. It provides a security context that is assigned to each applet. It prevents an applet from undesirable intrusion from another applet. For instance, it must prevent references to internal data to be leaked to the outside world. Any attempt to access data outside of the current security context would result in a `SecurityException`. Montgomery et al. [MK99] and Perovich [Per02] first pointed out the deficiencies of the Java Card firewall mechanism. Then, the Coq theorem prover [ACL03a] has been proposed in order to be able to perform verification of applet isolation properties. Shortly after, Werner et al. [DMPH05] proposed a type system to statically verify the absence of firewall violations. In the meantime, Mostowski *et al.* [MP07] decided to test the firewall mechanisms implemented in different smart cards based on the Java Card technology and from four different manufacturers. Although their results did not reveal any major security issues, they pointed out the fact that despite a standardized specification, the latter can allow different interpretations while implementing a given platform. Consequently, minor issues where implementations did not comply with the specification could still be encountered.

From the literature, the main attack vector for abusing the Firewall through a well-formed applet is through a Shareable Interface Object.

Abusing Shareable Interface Objects – The Shareable Interface mechanism is a feature that has been introduced in the Java Card 2.1 API Specification. It aims at enabling applets to explicitly share objects through the firewall by defining a set of shared interface methods. Such shareable objects are called Shareable Interface Objects (SIO).

Applets that provide SIO are considered as "server applets" since they provide access to their services via the SIO. The other ones that use the SIO of another applet are called, "client applets".

Witteman first suggested [Mar03] SIO abuse. He gives general principles of software attacks and gives some examples of pre-installation bytecode manipulation to exploit Java Cards. He also put forward the idea of using the firewall mechanism to achieve similar results to bytecode manipulation, namely introducing type confusion in the JCVM.

The idea has been experienced further by Mostowski et al. [MP07] and [MP08a] and consists of using shareable interface mechanism to create type confusion between applets without any direct editing of CAP files. The authors relate an exploit of a bug in the implementation of the sharing mechanism that they found on several platforms.

The general idea behind SIOs abuse, as depicted by the example in Figure 3.1, is to let two applets communicate through a shareable interface, but by using different definitions of the shareable interfaces after the CAP files generation. A type confusion can then be induced, namely, confusing the virtual machine about the types of data objects it is manipulating. The strength of this attack is the fact that there is no need to modify the CAP file for performing the attack.

3.2.2.2 Transaction Mechanism Abuse

In a nutshell, transaction mechanism enables bytecode instructions to be turned into atomic operations. It offers a "roll-back" mechanism in case operation is explicitely aborted. This mechanism should be responsible of deallocating any objects allocated during an aborted transaction and references should be set to *null*. An operation can be aborted by a card tear or via the invocation of the API method `JCSystem.abortTransaction`. Therefore in both cases such kind of mechanism is very important in order to preserve the data that were being manipulated during the transaction.

Listing 3.1 Example of programmatically aborted transaction and the possible execution flow

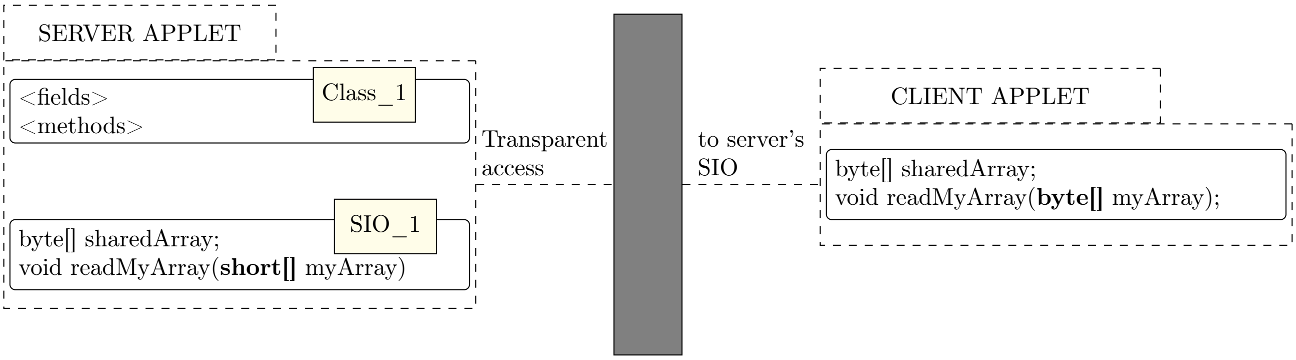

SERVER APPLET
<fields>
<methods>
Class_1
SIO_1
byte[] sharedArray;
void readMyArray(**short[]** myArray)
Transparent access to server's SIO
CLIENT APPLET
byte[] sharedArray;
void readMyArray(**byte[]** myArray);

Figure 3.1: Type confusion attack by abusing the Shareable Interface mechanism

```
public class MyApplet extends Applet {
    private short[] array1, array2;
    private byte[] bArray1;
    ...
    public void foo()
    {
        short[] localArray = null;
        JCSystem.beginTransaction();
        array1 = new short[N];
        array2 = localArray = array1;
        JCSystem.abortTransaction();

        bArray1 = new byte[N];

        if(array1 == null && array2 == null)
        {
            //array1 and array2 have been reset
        }
        else { //Oops, this should not happen...}

        if(localArray == null)
        {
            //localArray has been reset
        }
        else { //Oops, this should not happen...}

        if((Object)bArray1 == (Object)localArray)
        {
            //This sould definitely not happen
        }
        else { //Everything went fine, let's keep going }
    }
}
```

The Listing 3.1 depicts an example of a programmatically aborted transaction, and illustrates the issue that may happen if the transaction mechanism is not properly implemented. The issue resides in some buggy implementations of the transaction mechanism on some cards. Indeed, according to the specification, after the transaction has been aborted (line 11, Listing 3.1), the three arrays, `array1`, `array2`, and `localArray`, should be reset to *null*. However, if the transaction mechanism has not been properly implemented it is likely that the involved data during the transaction may remain set instead of being unset.

In their paper [MP08a], Mostowski *et al.* encoutered a buggy implementation where `array1` and `array2` have been reset, but not localArray. Therefore, in this particular case, the *else* statement at line 25 (Listing 3.1) can be reached. Furthermore they also noticed an additionnal strange behavior: the reference of `array1` was reused during `bArray1`'s instantiation at line 13. This unexpected behavior could lead to a type confusion that could be verified as depicted at line 27. It should be noticed that `bArray1` is an array of type *Byte* and localArray is an array of type *Short*. In case this equality test succeeds, it means that despite they are two distincts objects, they can be used interchangeably.

Hogenboom et al. [HM] exploited this bug and performed a type confusion in order to identify the internal representation of an array on a specific card.

3.2.3 Ill-formed application based attacks

This second category of attacks is the most represented in the literature. This is mainly due to the fact that Java Card specification leaves a lot of freedom for defensive features in the implementation of the JCRE. Therefore the implementation of the platform's security is fully dependent on the developer's choice. As a consequence, this situation leaves a large playground regarding the manipulation of malicious bytecode sequence.

As a reminder, an ill-formed application is an application that has been modified such as the code does not comply anymore with the rules defined in the specification. Therefore, it is likely that the BCV detects the malevolence behind such kind of application, as it has been designed to ensure that the program is type-safe.

However, if a malicious ill-formed application could be installed without being detected by the BCV or by bypassing it, then it opens the door to various techniques that we describe in the next sections. The latter can be used for, either, reverse engineering purposes or for attacking other applications and their sensitive data.

In order to perform a successful attack against the JCVM, a given flaw must exist in its implementation. The goal of an attack is to circumvent Java Card language security or to invoke potentially harmful operation (for applets).

In the next section, various attacks are studied and described.

3.2.3.1 Type Confusion attacks (TC)

Type confusion vulnerabilities are, as the name implies, vulnerabilities that occur when one data type is mistaken for another. Most of the time, they result in an attacker being able to either read sensitive data from a target application, namely, an information leak, or achieve unintended execution. Type confusion is also already a well-known issue in the Java world [oDRG02] as many known attacks are based on type confusion (e.g. CVE-2011-3521, CVE-2012-1723, CVE-2013-2423, CVE-2014-4262).

The type confusion condition occurs in a result of a flaw in the JCVM, which creates the possibility to perform cast operations from one type to any unrelated type in a way that violates the Java type casting rules

This category of attacks are now well-known logical attacks on Java Cards. An overview on the general principles of such attacks has been described by Witteman in [Wit02] with some examples of bytecode manipulation that enable the exploitation of Java Cards.

There are several kinds of logical attacks that can be forged and most of them rely on type confusion as it has already ben seen [MP08b, BTG10, BICL11, HP04, ICL10a].

According to the literature, there are two main ways to perform type confusion within a Java Card applet.

1. One modifies the CAP files after conversion in order to turn it into an ill-formed applet. Only this technique is fully covered in this thesis.

2. The other uses fault injection to alter data at runtime to render a benign applet into a malicious one.

Furthermore, one can identify three kinds of type confusion attacks: Reference-to-Primitive Type Confusion, Primitive-To-Reference Type Confusion, Reference-To-Reference Type Confusion.

In the next paragraphs, we illustrate the concept of the different type confusion attacks with an associated basic example. Obviously, it is very likely that bytecode sequence variants may exist due to the diversity of combinations that can be performed for getting to the same result.

Reference-to-Primitive Type Confusion (RPTC) This kind of type confusion attack, simply consists of converting the reference of a given object (arrays included) into a primitive data type (byte, short, int). The Table 3.1 depicts an example. This technique can lead to C-like pointer arithmetic on the reference that is being used as a primitive data.

Java Code	Java bytecode	Ill-formed bytecode Sequence

byte[] myByteArray = **new** **byte**[2];
short myShort = (**short**)0xBABE;
return myShort;
	sconst_2
	newarray 0x0B
	astore_2
	sspush 0xBABE
	sstore 0x03
	sload_3
	sreturn
		sconst_2
		newarray 0x0B
		astore_2
		sspush 0xBABE
		sstore 0x03
		sload_2
		sreturn

RESULT	Local 3 ($0xBABE$) is loaded and pushed on top of the stack then returned	Local 2 (array's address) is loaded and pused on top of the stack then returned

Table 3.1: Example of Reference-to-Primitive type confusion

In this example, an array is created then stored at the index 2 of the local variables area (local 2). Then, the short value $0xBABE$ is pushed on top of the stack then stored at the local 3. Before the value can be returned, it is reloaded on top of he stack again. If one take a look at the ill-formed byte code sequence, just a single slight change has been performed and would enable one to load a reference value as a short value.

As one will see in the next various examples, gaining access to reference values may open the door to various exploitation combining other techniques (described throughout this document). A good example would be the one Iguchi-Cartigny *et al.* described in [ICL10b] the EMAN1 attack which was inspired by Hypponën's work [Hyp03] and takes benefit of a weakness in the implementation of the firewall. It involves the *getstatic* and *putstatic* instructions. Both their operands consist of a reference to a static field that is fetched or updated. The weakness lies in the absence of context check when the fields are accessed.

Therefore by taking advantage of this weakness, the authors were able to implement a Trojan which basically consists of few steps:

- The very first step of this attack is to perform a Reference-to-Primitive type confusion in order to get the address of an array,
- A malicious code is then embedded within a byte array. This code essentially performs Hypponën's trick [Hyp03] that would enable one to read any arbitrary address with the instruction *getstatic_ a*,
- The hidden malicious code could be then executed with the *invokestatic* instruction This attack enables the authors to dump the entire EEPROM area of a given card.

Primitive-to-Reference Type Confusion (PRTC) This category of attacks (Table 3.2) aims at "forging" the address of an object. It is not allowed to directly assign a primitive data type to an object, thus, such kind of bytecode manipulation is easily detectable by the BCV. An example of ill-formed bytecode sequence is illustrated by the table 3.2. In this example, a value of type short is initially stored in the APDU buffer and is fetched so it can be stored in a local variable. The bytecode modification that is depicted entails that the value that has been previously stored is pushed on top of the stack as a reference, instead of a value of type short. In case the VM does not properly check that incoherence, a type confusion is then performed.

Java Code	Java bytecode	Ill-formed bytecode Sequence
short ref = Util.getShort(apduBuffer, ISO7816.OFFSET_CDATA); Object obj = **null**;	invokestatic 0013 sstore_3 aconst_null astore 04	invokestatic 0013 sstore_3 aload_3 astore 04
RESULT	The local variable *obj* will be set to *null*	The short value stored at local index 3 will be loaded as a reference and stored with *obj*

Table 3.2: Example of Primitive-to-reference type confusion

Reference-to-Reference Type Confusion (RRTC) This subcategory of Type Confusion attacks, can be divided in 4 basic attacks according to their purposes:

1. accessing a Byte Array as a Short Array (RRTC-BASA)
2. accessing a Byte Array as an Integer Array (RRTC-BAIA)
3. accessing an Object as an Array (RRTC-OA)
4. accessing a Class as another Class (RRTC-CC)

(1) and (2) are mainly useful for performing an overflow over an array. If possible, such kind of overflow attack enables to get access to extra data. The location of the data, and subsequently the impact of the overflow attack, depends on where the targeted array has been allocated. It can be either in the non volatile memory or in the volatile one.

(3) can be also very powerful, in case the platform does not implement properly the right countermeasures. Indeed, as explained [MP08a, Mar03], it would enable an attacker to:

- fabricate arrays
- directly access (read/write) to object through an array
- modify an AID object
- spoof references

Java Code	Java bytecode	Ill-formed bytecode Sequence

`Object myObj = new Object();`
`byte[] myBarr = new byte[N];`
`byte tmp = mBarr[0];`
`short dummyShort = (short)0xBABE;`
`return dummyShort;`
	`new 0006`
	`dup`
	`invokespecial 0007`
	`astore_1`
	`sconst_2`
	`newarray 0B`
	`astore_2`
	`aload_2`
	`sconst_0`
	`baload`
	`sstore_3`
	`sspush BABE`
	`sstore 04`
	`sload 04`
	`sreturn`
		`new 0006`
		`dup`
		`invokespecial 0007`
		`nop`
		`nop`
		`nop`
		`nop`
		`astore_2`
		`aload_2`
		`sconst_0`
		`baload`
		`sstore_3`
		`sspush BABE`
		`sstore 04`
		`sload 03`
		`sreturn`

RESULT	The value at index 0 `myBarr` is stored in the local variable "tmp" and the value 0xBABE is pushed on top of the stack and returned	The reference of `myObj` is stored into the local 2 (instead of `myBarr`'s ref) and the insruction *baload* attempts to read the object as a byte array at index 0. The result is stored at local 3

Table 3.3: Example of Object-to-Array type confusion

The example in Table 3.3 depicts a dummy example of Object-to-Array Type confusion in order to briefly illustrate the capability of such attack.

Finally, (4), namely, type confusion between two different classes, have been first described by Govindavajhala *et al.* in [GA03]. They described an attack against the JVM that would enable one to obtain two pointers of incompatible types that point to the same location. An attacker would then have to create in his application several instances of a given class, and should count on a memory error to modify the reference of a given instance of another class.

By going further in the exploitation of the previous attack, Barbu *et al.* [BDH11] described a combined attack (software + fault injection) that targets the operand stack on which references are pushed. This enabled them to perform an instance confusion between two classes.

It turns out that the previous described exploitation can be performed through a pure software attack using an ill-formed application, as illustrated in the Table 3.4.

Java Code	Java bytecode	Ill-formed bytecode Sequence

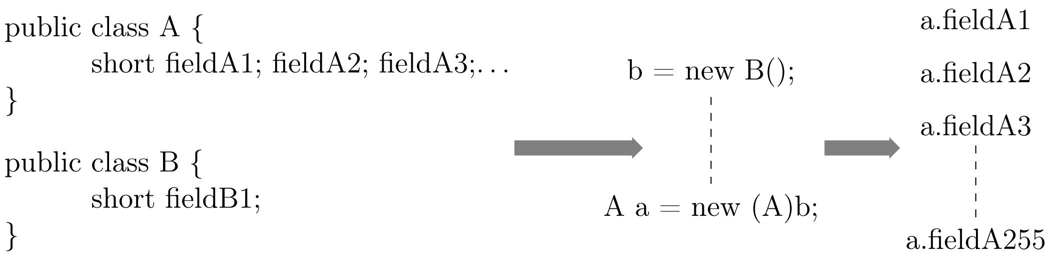
`ClassA objA = (`
`    ClassA)objB;`
	`getfield_a_this 0E`
	`checkcast 00 0096`
	`astore 04`
		`getfield_a_this 0E`
		`checkcast 00 0092`
		`astore 04`

RESULT	The `checkcast` should fail and an exception should be raised as B cannot be casted to A	The `checkcast` succeeds as we now verify the type B against itself. B can now be accessed as if it was A.

Table 3.4: Example of Class-to-Class Type Confusion

As illustrated by the Figure 3.2, on the third column, if the `checkcast` instruction, is not properly implemented, an attacker would be able to get access to several additional fields, through the `class A`.

Indeed, the `class B` is accessed as if it was the `class A`. As the `class B` only has one field, thus, accessing it through `class A` with an instance field of type `short`, would give an overflow of 2 bytes. Furthermore, as it is defined in the specification, a method can only have at most, 255 local variables. Thus, if such attack succeeds, and the memory that is being accessed through the overflow is not read/write-protected, it would give an area of 254 × ($size\_of(type)$) bytes.



```
public class A {
        short fieldA1; fieldA2; fieldA3;...
}

public class B {
        short fieldB1;
}
```

Figure 3.2: Illustration of the possible exploitation of Class-to-Class Type Confusion

3.2.3.2 Stack Overflow/Underflow Attacks

The Java Stack can be represented such as an array in memory. According to the stack implementation, accessing (read/write) to data beyond the boundaries of the stack may endanger another application, or the whole system. Indeed, according to the stack layout, it is possible to overflow or underflow it.

The Figure 3.3 describes a common implementation of a Java Card Stack. One should note that the implementation of the Java stack is highly platform dependent. Therefore, the way each component of a frame is arranged may differ from one platform to another one. During the frame creation, each component has a predefined size. For instance, the number of local variables is known in advance, thus, it is possible to allocate the local variables area accordingly. In case the Java Stack implementation is not represented as the one depicted by the Figure 3.3, if an attacker is able to read beyond the boundaries of a given frame component, he will have to perform some reverse engineering in order to identify where exactly he landed, and what kind of data he is accessing.

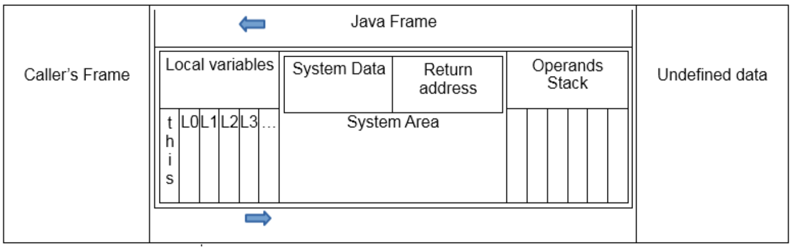


Figure 3.3: Java Card stack common implementation <*to be replaced with tikz*>

According to the direction of the stack access, the attack can be defined as either an underflow, or an overflow attack.

Bouffard *et al.* described [BIL] the EMAN2 attack that is actually an exploitation of a stack overflow attack. This attack is further explained in the section 3.2.3.4. Nevertheless, an illustration of the stack overflow is given in the listing 3.2.

Listing 3.2 Out-of-bound access to the local variables array

```
1 public void AttemptStackOverflow() {
2     // flags:0  max_stack:2
3     // nargs:1  max_locals:0
4     sspush 0xCAFE
5     sstore 0x2
6     return
7 }
```

As one can notice at line 3, the maximum number of local variables `max_locals` that the method has, is 0, yet the `short` value `0xCAFE` is stored at a location that is totally out-of-bound of the local variables array. If the JCVM does not properly verify this overflow attempt, it is then possible get read/write access up to 256 (0xFF) memory slots.

[BL15], Bouffard *et al.* briefly introduced the use of the *sinc*, *sinc_w* and *swap_x* instructions for illustrating other possible stack overflow attacks.

Another trick uses the *dup_x* instruction, and was proposed by Faugeron [Fau13], for underflowing the stack in order to tamper with the current frame header.

The main exploitation that has been the most studied in the literature consists in tampering the system area (see Figure 3.3) that contains the return address that has been saved there by the system for being able to restore the previous context. While all these attacks are restricted to the main assumption that the system area is located between the operand stack and the local variable area, it is important to precise that the platform implementation may differ from one card to another, so as the Java stack. Furthermore, current countermeasures for preventing from stack overflow attacks, mainly focus on bounds checking, such as the one presented by Lackner *et al.* [LBL$^+$13]. Additionally it is worth to notice that the way the Java Frame is structured may be different from one platform to another one.

As part of my contributions, we show in the Chapter 4 that it is still possible to overcome these issues through a new attack that enables to go further in the exploitations.

3.2.3.3 Static Links based attacks (SLA)

On Java Card platform, a linking step is performed while the applet is loading. The JCVM takes care of linking the CAP file with the installed Java Card API. It needs three components for resolving the different methods that are used in the `Method Component`, namely the code of the applet. The first one is the `Class component` that contains various methods tables that store the offsets where to jump for interpreting the corresponding method. The second one is the `Constant Pool` component that is a kind of lookup table that stores the tokens that identify a given method. The last one is the `Reference Location component`, that keeps a list of offsets in order to easily retrieve each token referred in the `Method Component`. Using those two components, the JCVM is able to identify and resolve any methods that are being used in the code.

[Hyp03] Hypponën describes an attack that takes benefit of static class field linkage. The main purpose of the attack is then to remove an item from the `Reference Location` component in order to avoid the address resolution by the linker. As the author describes it, one can exploit this technique on the bytecode instruction *getstatic_ a* in order to try to read any arbitrary address instead of reading the correct resolved address.

[ICL10b] Iguchi-Cartigny et al. targeted the *invokestatic* instruction showing that regarding the internal structure of an on-card applet, it is possible to get the address of a byte array and by passing the latter to the *invokestatic* instruction it would be possible to execute the code at that address. The linker must not resolve the address at the offset corresponding to the address that has been provided to *invokestatic.* To do so, the reference to this offset in the reference location component must be deleted. This would then give the possibility to write an hidden malicious code within an array and to execute it following the previous technique.

[HBL$^+$12] Samiya et al. describe another way to take benefit of the linking mechanism in order to characterize the Java Card API. During the linking step, some platforms do not check if the instruction just before the address to resolve is the appropriate one.

If one take as an example the following operation: *invokestatic XXXX*. The argument *XXXX* would be normally resolved by the linker in order to provide the correct address to the *invokestatic* instruction. Now, if one replaces *invokestatic* by the instruction *sspush* and one does not update the reference location component, *XXXX* will still be resolved. Consequently, the *sspush* instruction would push the resolved address as a short value on top of the stack. Therefore, by using this technique, in the case it is not detected, it would be possible to identify the token that corresponds to any function.

Recently, Lancia *et al.* [LB15] presented a new attack that exploits the token resolution of a public method, for triggering an overflow while the VM attempts to scan the `Class component`, for retrieving the method identified by the token. The trick consisted of removing N entries from the structure `public_virtual_method_table[]` within the `Class component`. Consequently, while the VM will attempt to access any of those entries, it will step outside of the `Class component`, *i.e*, an overflow. As the authors were aware of the memory layout of the targeted device, they knew that right after the `Class component`, the `Method component` was set up. As the `Method component` can be easily modified before the `CAP` installation, according to N, they were able to control the overflow such as, it points out to a location in the `Method component` that contains a sequence of bytes that will be used by the VM, as an offset that points to the new method to call. One of the main thing of interest in their paper is that at that time, the

BCV did not detect the structural incoherence that was created within the `Class component`. Fortunately, the Oracle BCV has been patched since then.

3.2.3.4 Control Flow Modification based Attacks (CFTA)

Return Address Modification - One possible exploitation of the attacks presented in 3.2.3.2, namely, stack overflow/underflow attacks, consists in modifying the return address, that is theoretically stored within the frame header. The latter is a structure that enables the JCVM to restore the previous context. The return address is used for coming back from the callee function to the caller.

As an example, the EMAN2 attack [BIL] aims at changing the return address in order to redirect the current execution flow. It basically consists of:

1. performing the first step of EMAN1 [ICL10b] attack in order to retrieve the address of a given byte array
2. locate the return address of the current function
3. update the return address with the reference of our malicious array

In (1) a Reference-to-Primitive Type Confusion attack (cf. section 3.2.3.1.0) enables to convert a reference into a type short value. In (2), the attacker has to overflow the stack in order to reach the return address which is located in the system area. For this purpose, the authors used the instruction *sload* to go beyond the local variables area boundaries, and read any value as a type short value.

Indeed, the *sload* instruction takes as an argument the value of the index to access within the local variables area. Overflowing this area consists of reading an element at a given index that is out-of-range.

Furthermore, one should also note that the return address's position is not known in advance. Therefore, the attacker has to dump somehow, the area that he has overflowed, in order to verify what are the values that have changed, or remained constant, between two different methods invocations.

Finally, in (3), once located, the return address can be modified using the *sstore* instruction using the same principle as the *sload* instruction, namely, overflowing the local variables area in order to write at a given index that is out-of-range.

Jump Subroutine (JSR) Exploitation - Lately, the authors of EMAN2 proposed another approach [BL15] that is more generic and relies on the exploitation of the couple of `jsr` and `ret` instructions.

The `jsr`/`ret` mechanism was originally used to implement finally blocks. It enables to save some extra code size. However, the extra complexity it introduced for verifying the control flow was not worth, therefore, it got gradually phased out from the Java compiler. Though it is not anymore generated by a regular compiler, it seems that it is still handled by the JCVM.

The `jsr` bytecode is analogous to the `goto` bytecode, except that it pushes a return address on the stack. The return address is normally stored in a register β after a `jsr` jump. When the subroutine is complete, the `ret` bytecode is used to return. Technically, the `ret` instruction set the Program Counter register to the address that has been previously saved by the `jsr` instruction. If one is able to modify the return address that is stored within β, then, the `ret` instruction will enable to jump at that new address. Consequently, the execution flow has been explicitly modified. As a demonstration, the authors used the instruction `sinc` just before the `ret` instruction, for incrementing the value of the address. This involves a type confusion as an arithmetic operation is performed on a reference. In case the authors were facing a typed stack as a countermeasure, they proposed to "hide" the address within an instance field instead of within a memory slot into the current stack. The main reason behind this choice is that main countermeasures against type confusion [DBBL13, LBL+13] are focused on protecting the values manipulated on the stack. Therefore, it is likely that the heap memory is not well-guarded. Consequently, they use the instruction `putfield_a_this` and `getfield_s_this` for performing a type confusion between a reference and a short type for saving the return address as a reference a fetching it back as a short value. A thorough analysis of this bytecode manipulation is given in Bouffard *et al.* [BLLL14] paper.

While such kind of ill-formed applet is detected by the verifier, the authors demonstrated by means of the `if_scmpeq_w` instruction, that due to a flaw within the verifier implementation, it is possible to hide the malevolent return address modification within an unreachable code, that is not verified by the off-card BCV. The instruction `if_scmpeq_w` instruction is a conditional branch one, and compares two values from the top of the stack. According to the result, the execution flow carries on toward a given execution path. At runtime, for jumping to the unreachable code, the authors rely on a fault attack for modifying the value of the operand of the `if_scmpeq_w` instruction, that defines the location where the current program counter must be set to.

3.3 Our Generic Framework for Evaluating Java Card Platforms and Applets

3.3.1 Testing Tools Categories

One can identify three categories of testing tools for verifying the compliance of a given platform against attacks that can be performed through ill-formed applets: (1) Having a tool that enables to statically and manually modify pre-designed Java Card applets, into ill-formed ones, (2) having a tool that is able to automatically generate all the possible malicious byte code sequences, (3) a semi-automatic tool that is a compromise between both (1) and (2).

In the next sections, we are going to describe thouroughly, the three approaches, with their benefits and inconvenients, but, above all, few clarifications should be made regarding the terms, "attack" ,"exploit" and "attack/exploit implementation".

Attack versus Exploit - An attack is a single event that can be triggered in order to reach a target. A target can be refered to a vulnerability. For instance, a platform that does not properly verifies the stack boundaries while executing the instruction *sload* (the event), is vulnerable to a stack overflow attack (the vulnerability). A single event is only useful in highlighting this vulnerability. However, an exploit is a series of choosen attacks that can lead to more malevolent actions.

Attack/Exploit implementation - Implementing an attack or an exploit consists of 2 steps that may seem basic and simple at the first sight: (a) Developing an applet, (b) modify it for integrating a malicious byte code sequence. (b) is constrained by (a) according to the following properties: let A an applet of size N that would embed a set of tests $T = \{T_0, T_1, \cdots, T_{i-1}\}$. In anticipation of implementing all the T_i, A needs to provide a set of methods $M = \{m_0, m_1, \cdots, m_{k-1}\}$, where Sm_k is the size of the k^{th} method. Let $V = \{V_0, V_1, \cdots, V_{j-1}\}$ be the set of vulnerabilities to be tested, hence the following is considered: $\forall V_j \exists T_i \mid V_i = f(T_i)$.

Consequently, for $f : T \to V$, any V_j is "hit" at least once. Furthermore, considering the following, $T_i = \{b_0, b_1, \cdots, b_{n-1}\}$, where b_n is the n^{th} Java Card byte code and S_{T_i} is the size of the byte code sequence that compose T_i. There is a strong constraint between T and M, such as,

$$\forall m_k \in M \exists x \mid Sm_x >= S_{T_i}, 0 \geq x \geq k-1 \tag{3.1}$$

otherwise, it will not be possible to implement the corresponding T_i. The only way for overcoming this issue would be to implement a tool that will automatically manage the interdependencies between the `Method Component` and the other CAP components. Indeed, for instance, each reference that are used in the `Method Component` of a CAP file, for referencing fields or other methods, are referenced within the `Constant Pool Component`. As the `Reference Location Component` is responsible of storing the necessary information to make the link between the `Constant Pool Component` and the `Method Component`, hence, any modification that involves modifying pre-existing references, implies to update those CAP file components, otherwise, the application will not behave properly, or it will just crash.

3.3.1.1 Model 1: Static One-to-One approach

Let A be the domain that contains any testing applets, and B its codomain. The latter contains the vulnerabilities within a given platform that can be attacked. An element of A will be noted x and an element of B is noted y. Therefore, given the function $f : A \to B$, we have, $\forall x, y \in A, f(x) = f(y) \implies x = y$. This solution, involves a one-to-one correspondence, as depicted by the Figure 3.4, and may be very constraining as one should have an applet for each specific attack and exploit. If the naming and the classification of each applet is not properly done, it would make waste a lot of time in case, one has a large set of testing applets.

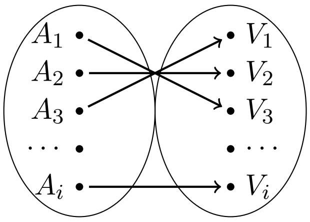


Figure 3.4: One-to-One correspondence between the set of testing applets and the set of vulnerabilities to test

One should note, that due to the fact that each exploit is highly dependent on the implementation of the platform, therefore, each new attack variant, may lead to the development of a new testing applet. It might be possible to turn this model into a One-to-Many one, however, in term of maintainability, it would be difficult to foresee every possible cases in a single applet.

In term of tooling, a Java editor and the compiler distributed in the Java Card Development Kit is required for generating an applet. Additionnally, an hexadecimal editor, or more specifically, a Java Card byte code decompiler with advanced features (binary modification) is required for modifying the applet to particularly fit a given exploit scenario.

The main inconvenient regarding this kind of approach, is that, the corresponding tool must automatically ensure that the generated applet is compliant with the CAP file format as described in the specification [Ora15].

Moreover, as the applet will not be structurally correct, it is likely that the Byte Code Verifier detects the issue and will prevent from installing or running the applet.

3.3.1.2 Model 2: Automated One-to-Many approach

While formal methods, have been mainly used for performing so-called "formal verifications" of software [Mos07, DG02, ACL03b, Bec01, ACL03b], we have shown in [AJLM$^+$12] that formal method can also be used for automatically generating byte code sequences that can be used for testing a given platform. As a summary, we have shown that the Java Card set of instructions could be modeled using the Event-B formal method [fm]. From the different models, it is then possible to generate a set of byte code sequence (mutants) that respect mathematically proven rules. The models must ensure that the generated byte code sequence remains semantically correct.

This case has a one-to-many correspondence between the two domains, and might be considered as the best approach regarding its reliability, and tests coverage. However, despite the robustness of such kind of approach, the difficulty is to accurately describe (model) every instructions and their computation, as defined in the Java Card Virtual Machine Specification [Ora15].

Since then, though the tool has developed very well in term of performance and robustness, it appears in the recent results [SFLL15] that, only 61 instructions over 184 are taken into account for now.

Once the byte code sequences generation is done, one still has to implement them within a real Java Card applet. Hence, one can either use the first approach, or the third one. The latter will be described in the next section. If one chooses the first approach, according to the number of byte code sequences to implement, the time for implementing them must be taken into account.

3.3.1.3 Model 3: Semi-Automatic One-to-Many approach

This approach, is a compromise between the two previous approaches. Indeed, the user still has to manually modify the binary file, however, he just has to focus on the design of the malicious byte code sequence. The integration of the byte code sequence to test and the whole applet generation would be implicitenormally performed.

Initially, we have already developed CapMap [ST], a CAP file manipulator that enables to parse and to modify a given CAP file. We have presented the tool in [RBL12] and we have explained that its purposes is mainly threefold: (1) reverse engineering of Java Card applets and the CAP file format, (2) designing software attacks, and (3) designing fault enabled malicious applets, namely, benignn applets that can be turned into malicious ones by means of an induced fault using a laser beam.

The programming language for using the tool remains Java, which makes it more convenient for any Java Card applet developer.

Additionally, Lancia described in [Lan] a python framework (JaCarTA) that mainly enables to design ill-formed applets from an abstract class (the parent class). The author describes a complete toolchain that enables to design testing applets, compile and install them on a given card. It also eases all the installation process on the card. Unfortunately the tool is not publicly available.

3.3.2 Constraints

Four general constraints are considered for choosing the most suitable approach, in our particular case.

1. the time, which is a very strong constraint during an evaluation. The most suitable tool, must enable the evaluator to design an exploit in a short time, or at least, during the amount of time that has been defined lately,

2. the tool must ensure a sufficient test coverage, *i.e*, it must cover at least, all the possible attacks that are defined in the literature,

3. compatibility of the solution with other tools,

4. expertise level of the user must also be taken into account.

3.3.3 Approaches comparison

According to the four constraints that we have previously defined, we are going to compare the three models. For the sake of readability of the chart, the time constraint is going to be redefined as the term "productivity", which defines the level of productivity while using a given model. The expertise level constraint is also going to be redefined as being the term, "usability". The usability aspect of the model strictly depends on the expertise level of the user.

Accordingly, the Figure 3.5 depicts the differences of the three models.

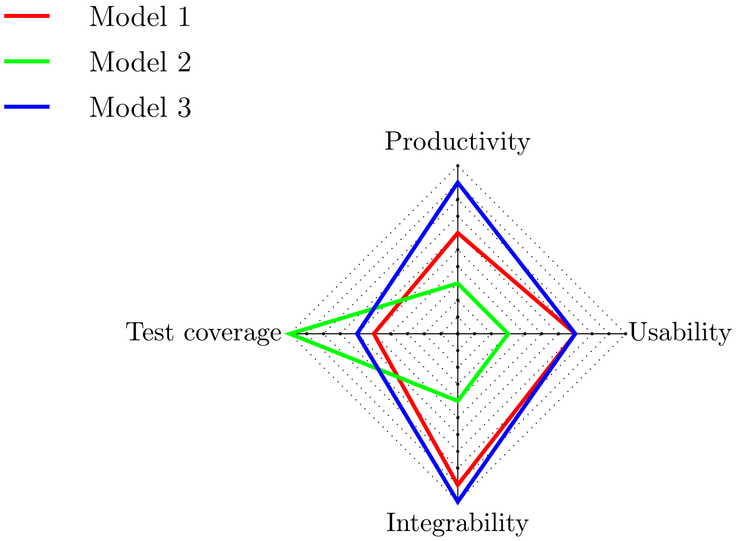


Figure 3.5: Constraints versus Models

According to the chart 3.5, the third model seems to be in average, the best choice. Despite a test coverage that is lower than the second model, the model can be easily integrated with the existing Lab tools, and remains easier to use.

3.3.4 A Framework for Designing Attack Scenarios

We have developed a generic framework in `Python` that allows to easily and quickly design attack scenarios. The framework has a core component which is the `parser` that translates the binary format of a CAP file into a set of objects that can be programmatically accessed (read or modified). An API (Application Programming Interface), namely a set of functions, enables to load a CAP file and modify its code embedded within its Method Component. It can also modify any other CAP components with respect to the interdependencies between the latter and the other ones. Instead of generating a single applet per attack scenario, the resulting applets are kept in memory. Thanks to our API and our core parser implementation, we can load a single template applet that is modified according to the attack scenario. The modified applet is kept in memory and provided as is to another tool responsible of installing the applet on the device. This approach has the great advantage of only storing very lightweight and comprehensive "scripts" and a single applet, instead of a myriad of testing applets.

3.3.5 A Framework for Performing Static Code Analysis

Our framework can also be used as an alternative to perform automated static code analysis or manual code review. Through the API it is possible to write scripts that allow to automatically and statically analyse applets.

Our first version of the framework has been extended by adding an additional significant module: a disassembler. The latter is composed of the core disassembler, an Export file parser, a linker and a view. The linker aims at resolving information within the CAP file related to classes or methods owned by external packages as defined in the corresponding Export file. The core disassembler is used to query the linker and the CAP file parser to provide a human readable content of the whole CAP file or just a part of it (*e.g.* an arbitrary method), namely, the disassembly of the latter. The view aims at giving the ability to display this content in different format according to the needs (*e.g.* plain text, JSON, XML, etc.). The default syntax that is used to describe the code is JASMIN[9]. The resulting disassembled applet can be then displayed in plain text into a terminal or within a dedicated and more convenient user interface that eases the review and the manual analysis of the code.

3.3.6 Example of Usage

A simple example is illustrated by the Listings 3.3, 3.4 and 3.5. In the Listing 3.3, we have a dummy Java method that has two local variables: the APDU buffer and a short value. The latter is returned at the end of the function.

Listing 3.3 Java method that get the APDU buffer and returns a short value

```java

...

public short attackMethod(APDU apdu)
{
    byte[] buffer = apdu.getBuffer();
    short tmp = (short)0xCAFE;
    return tmp;
}

...
```

In the Listing 3.4, from line 2 to line 9 we:

[9] http://jasmin.sourceforge.net/guide.html

- load the CAP file
- initialize the indexes of the class and the targeted method
- get the target method
- disassemble the method
- display the disassembly

From line 13 to 23 the disassembly of the method is displayed in order to have an overview of the targeted method.

- the virtual method `getBuffer` is invoked, and the reference of the buffer APDU is pushed on top of the stack
- the reference is stored within the local variable area at index 2
- the short value `0xCAFE` is pushed on top of the stack
- it is then stored within the local variable area at index 3
- the latter is fetched and pushed on top of the stack
- the instruction `sreturn` fetch a short value on top of the stack and returns it

Listing 3.4 Example of script that disassembles the targeted method and prints the result in the terminal

```
1
2       capfile = CAP(PATH_TO_CAPFILE)
3
4       classIndex = 1
5       methodIndex = 6
6
7       attackMethod = capfile.get_method(classIndex, methodIndex)
8       disassembly = disassembler.translate_method(capfile, attackMethod)
9       view.display(disassembly)
10
11      ...
12
13      .method public sub_0040(Ljavacard/framework/APDU;)S
14          .limit stack 1
15          .limit locals 2
16          .limit args 2
17
18          invokevirtual 0056 Ljavacard/framework/APDU;->getBuffer()[B
19          astore_2
20          sspush 0xCAFE
21          sstore_3
22          sload_3
23          sreturn
24
25      ...
```

The attack scenario that we chose here as an example consists in disclosing the value of the reference of the `buffer APDU`. It basically consists of a Reference-to-Primitive Type Confusion (see 3.2.3.1.0). For doing so, as we know that its reference is stored at index 2 within the local variable, we can just reload it in order to push it on top of the stack right before the `sreturn` instruction instead of returning the short value `0xCAFE`. The listing 3.5 illustrates the script that can be used to perform the bytecode transformation. The steps are as follows:

- the cap file is loaded and parsed
- we initialize the class and method indexes
- we fetch the targeted method object
- we search for the offset of the instruction `sreturn`
- we replace the `sload_3` instruction by `sload_2`
- the CAP file object is updated according to the new method object that is provided as an argument. In this case our modification does not implicitely affect any other CAP components therefore the update process will execute without updating any other CAP Components except the `Method component`.
- the changes are committed so the binary version of the CAP file that is stored in memory is also updated. This way, we can directly push that buffered CAP file to another tool responsible of loading and installing it on the tested device so we do not have to actually save it on the disk.

Listing 3.5 Example of script that perform a bytecode modification within a targeted method

```

    capfile = CAP(PATH_TO_CAPFILE)

    classIndex = 1
    methodIndex = 6

    attackMethod = capfile.get_method(classIndex, methodIndex)
    returnOffset = attackMethod.search(JC.instructions.sreturn)
    attackMethod.bytecodes[returnOffset - 1] = JC.instructions.sload_2

    capfile.update_method(attackMethod)
    capfile.commit()
```

The listing 3.7 illustrates a simple script that allows to automatically search for such kind of bytecode sequence in an arbitrary CAP file, namely the sequence `astore_x sload_x sreturn` where `x` refers to an index in the range `[0,...,3]`. The main assumption here is: if the last two instructions of a given method are `sload_x, sreturn`, then consider the current method as a suspicious method if a `astore_y` instruction is executed before, such as we have $x = y$.

Listing 3.6 Example of script that is used to automatically perform a specific static analysis over the functions of a CAP file

```
1
2  capfile        = CAP(PATH_TO_CAPFILE)
3  TYPE_SLOAD_X   = JC.instructions.types.SLOAD_X
4  TYPE_ASTORE_X = JC.instructions.types.ASTORE_X
5
6  for class in capfile.classes:
7      if class.is_external:
8          continue
9
10     for method in class.methods:
11         lastInstruction = method.bytecodes[-1]
12         ins1            = method.bytecodes[-2]
13
14         ins_type = JC.instructions.get_type(ins1)
15
16         if lastInstruction == JC.instructions.sreturn \
17             and ins_type == TYPE_SLOAD_X:
18                 index1 = JC.instructions.get_load_index(ins1)
19                 instructions = method.bytecodes[:-2]
20                 results = JC.instructions.search_by_type(instructions, \
21                                                          TYPE_ASTORE_X)
22
23                 for instruction in results:
24                     index2 = JC.instructions.get_load_index(instruction)
25
26                     if index1 == index2:
27                         print("[+] Suspicious locals manipulation found
28                               in method %s ", \
29                               disassembler.translate_method_ref(capfile,
                                   method))
30
31                         print("[+] Method at offset %d returns a short
32                               value stored at local index = %d",\
33                               method.offset,\
34                               index1)
35
36                         print("[+] Yet the same index is used to store a
37                               reference at offset %d ",\
38                               results[instruction])
```

We loop through all the methods of each class (line 6 and 10). Classes and interfaces that are defined in an imported packages, and thus external to the current CAP file, are not considered in the search (line 7, 8). In line 11 and 12, we retrieve the two last instructions of the current method. If the last instruction is the `sreturn` and the one just before it is of type `SLOAD_X` (line 14, 16, 17), then proceed to the next step. Get the value of the local index of the `SLOAD_X` instruction, namely the actual value of X (line 18). Get all the instructions in the current method except the two last instructions and search an instruction of type `ASTORE_X` (line 19, 20). At line 20, `results` is a dictionary structure that contains key-value elements, where keys refer to the decimal value of an instruction, and values are the offsets where the instructions have been found. In our example, `results` should contain for instance the following instructions: `astore_0`, `astore_1`, `astore_2`, and so on.

In the line 24, the index value x in `astore_x` is retrieved and we check (line 26) if the latter is equal to the index y in `sload_y`. If it is then print a log in the terminal (lines 26 to 36). The `capfile` object that we have modified in the listing 3.5 can then be used in our previous script, and as a result, we have the following output.

Listing 3.7 Output of the script in Listing 3.7 while applied to the CAP file generated in Listing 3.5

```
[+] Suspicious locals manipulation found in method class_0001.sub_0040(
    Ljavacard/framework/APDU;)S
[+] Method at offset 0x0040 returns a short value stored at local index =
    2
[+] Yet the same index is used to store a reference at offset 0x0001
```

3.4 Summary

In this Chapter, we have seen the different categories of pure software attacks that exist in the current state of the art of Java Card platform security. Two main families of pure software attacks have been identified according to the way they are implemented within the applications: attacks through well-formed malicious applications, and attacks through ill-formed malicious applications. For the first family, they may take benefit from platform bugs to produce unexpected behaviors, and they remain structurally and semantically correct. Therefore it is likely that they pass the static verifications of the BCV. Regarding the current state of the art they are quite rare, and interests are put more on so-called "fault attacks" to modify the code at runtime, and turn a benign application into a malicious one.

On the other hand, for the second family of attacks, they thrive on the fact that the JCVM and JCRE implementations are platform dependent. Therefore, though the specifications give the general rules for being compliant, the variety of instructions and their possible combinations give the choice for the attacker/evaluator to implement variants for a given malicious bytecode sequence.

Consequently, according to different constraints related to the evaluation of Java Card platforms and applications, we have developed a generic framework that allows to automatically perform static analysis, to disassemble and review the code of an application and it also helped us for our research to design attack scenarios. This framework is in constant evolution so as our scripts library that implement attack scenarios and test automation.

Amongst those different attack scenarios, we have found out a new one that Java Card platform developers should take care of. This new software attack is further described in the next chapter.

Chapter 4

Misuse of Frame Creation to Exploit Stack Underflow Attacks on Java Card Platform

Contents

We have seen in the section 3.2.3.2 that stack underflow attacks against Java Card platform attempt to access undefined local variables or operands to corrupt data that are not supposed to be accessible. For instance, exploiting a stack underflow vulnerability may enables to change system data (return address, execution of context, etc.).

The current attacks that are presented in the literature are restricted to the main assumption that the frame system data is located between the operand stack and the local variable area. However, the platform implementation may differ from one card to another, so as the Java stack. Therefore the way they are structured may be different from one platform to another one.

After investigation, we present in the following Chapter a novel approach to perform a stack underflow attack and which does not rely on the Java stack implementation model.

4.1 Problem analysis

The Java virtual machine is a stack machine that uses Last-In-First-Out data structures to store data and partial results. As illustrated in the Figure 4.2, it is common that its implementation has a stack of frames related to method execution and a stack of call-return data structures to recover the previous runtime execution variables on method completion. The memory design of the Java stack is dependent on the VM implementation. Therefore, it is likely that two devices from different manufacturer have different stack layout.

We analyse first the constraints imposed by common stacks implementation. Second of all, we show the limitations of theses implementations and the technique that can be used for bypassing them. Finally, we present various attacks scenarios that can be set up according to the result of the attack. The latter also highly depends on the platform's implementation.

4.1.1 The Java Virtual Machine stack

4.1.1.1 Basic Stacks Implementations

One can identify two different ways for implementing a stack. Either it can be seen as an array, or as a linked-list. However, in either case, what defines a stack is not its implementation but the way it is accessed. As illustrated by the Figure 4.1, data can be pushed on top of the stack for storing it. It also can be popped for removing it from the stack. These are the two basic operations that almost all the instructions in the Java programming language can perform, either, explicitely or implicitely. Dedicated instructions exist for explicitely pushing or popping values from the stack such as *<T>push* and *<T>pop* instructions. *<T>* here refers to the type of the data that is manipulated by the instruction on the stack, such as, $\{a, b, s, i\} \subset T$. $b, s, i \in PrimitiveTypes$ and refer to byte, short, int, and $a \in Reference$, consists of the reference of an array or an object.

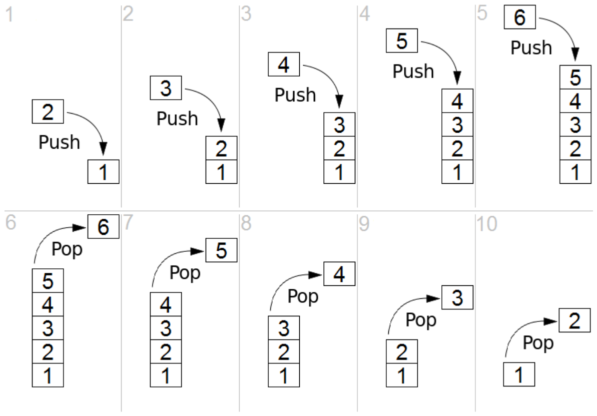


Figure 4.1: Last-in-First-Out stack manipulation

4.1.1.2 Stack Structure

From a global perspective, the stack is a bounded area that may be splitted in different substructures.

Frames usually store references, data and partial results related to a specific method execution. That is, each frame corresponds to a reserved area that is used by the method that owns it. Therefore, a new frame is created when a method is invoked and the frame is destroyed when the method completes. In Java card, a frame is mostly limited to a set of local variables and an operand stack. The Figure 4.2 illustrates stacked frames in the Java stack, and their current state during the execution of a given method. The allocated memory used by the latter is depicted by the active frame.

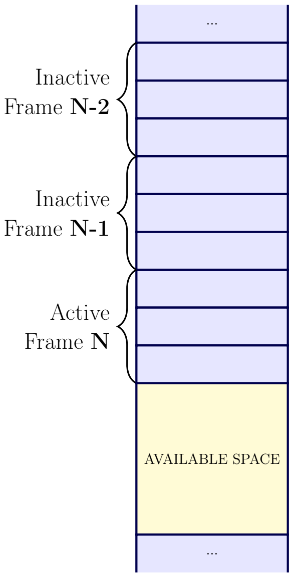


Figure 4.2: Stacked Frames

Local variables (LV-Array) are the variables defined within the current method block and that are used by the current subroutine. The latter frequently needs memory space for storing the values of its local variables, namely, variables that are known only by the current subroutine. This area, is an array-like structure, in the method frame. These variables are addressed by index and the first local variable index is 0. Furthermore, the VM uses the local variable area to pass parameters. On method invocation, parameters are passed in the form of consecutive local variables.

Operand stacks (OP-Stack) are used to store constants, values from local variables, object or static fields related to the current method execution. The VM provides instructions to load temporary data to the operand stack and applies arithmetic operations to values on the top of the operand stack.

Call-return data structure (SYS-DATA) saves previous method execution data. Its content is not specified, however, it usually holds data for restoring the previous frames and VM state variables. The minimum information stored in this structure is the previous execution context, the previous Java Program Counter (JPC) and the previous Java Frame Pointer (JFP). These data belongs to the system and shall not be directly accessible to Java applets running on the platform.

A summary of the stack structure is depicted by the Figure 4.3.

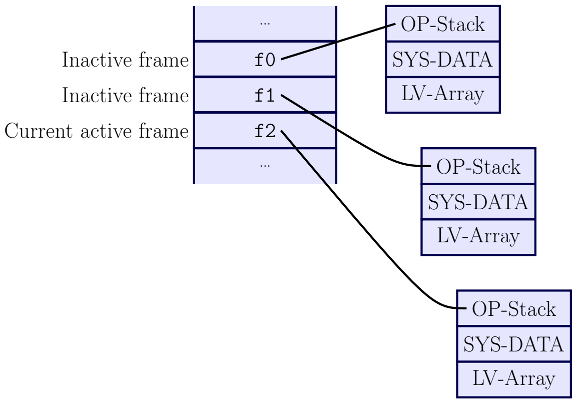


Figure 4.3: Stack Structure

4.1.1.3 Stack Manipulation and Bounds Checking

We have seen in the previous section that the frame is composed of different array-like structures. The way they are arranged in memory is platform dependent. To manipulate the stack in memory, the Java Card programming language exposes various instructions. Four basic operations exist for manipulating the operand stack and the local variables array: push, pop, store and load. Pushing and popping values affect the operand stack, and load and store operations are performed on the Local variables array. They also can be noted as *<T>{push, store, load}* and *pop*, such as $T \in \{b, s, i, a\}$ with $b = byte$, $s = short$, $i = int$ and $a = reference$.

What defines the most an array-like structure is its bounds. Consequently, the main vulnerability of such kind of structure is that they are prone to overflow/underflow attack. Therefore, any instruction that explicitely or implicitely manipulates the stack must implement bounds checking. Otherwise, it lets the freedom to go beyond the bounds of the structure. In general, virtual machines use specific dynamic registers for controlling the bounds of the Java Stack. We address this in the section 4.1.2.2.

A recent countermeasure in the literature was proposed by Lackner *et al.*. They described in [LBL+13] an hardware accelerated defensive virtual machine that enables to peform type checking on the value that is used by a given instruction, and also, OP-Stack/LV-Array overflow/underflow detection. The security checks that are performed are based on a security policy that ensures both type and bound protection.

4.1.2 Compile-time and runtime assignment

4.1.2.1 Compile-time attribution

For allocating the OP-Stack and the LV-Array, the JCVM requires their size. The latters are defined at compile-time. The size of the OP-Stack can be noted max_stack, and the size of the LV-Array $nargs + max\_locals$ where $nargs$ refers to the number of arguments that the current function takes, and max_locals is the number of local variables it uses. As the sizes are computed at compile-time, this means the size of method frames are computed by the compiler and hardcoded into the resulting Java byte code, for each method. Consequently, according to the CAP file format, the Method Component structure contains a stream of $method\_header\_info$ structures that are depicted by the listing 4.4.

```
method_header_info {
    u1 bitfield {
        bit[4] flags
        bit[4] max_stack
    }
    u1 bitfield {
        bit[4] nargs
        bit[4] max_locals
    }
}
```

```
extented_method_header_info {
    u1 bitfield {
         bit[4] flags
         bit[4] padding
    }
    u1 max_stack
    u1 nargs
    u1 max_locals
}
```

Figure 4.4: *method_header_info* structure

4.1.2.2 Runtime behaviour

During the runtime the Java Card platform interprets the bytecode and dynamically allocates frames on the stack. To do so, in general, Virtual Machines use the following registers to keep track of its execution states:

- The JPC register: contains the address of the current executed bytecode instruction. The bytecode interpreter loop increments the JPC to point to the next instruction.
- The JSP register: points to the top of the operand stack in the current frame. This is usually the index of the last value on the operand stack.
- The JFP register: refers to a fixed location in the current frame structure. It is common that it points to the bottom of the local variables area.
- The current execution context: keeps track of the context of the currently running application. This variable is mainly used by the Java Card firewall to verify the object access rules.
- The address of the call-return structure: is either contained in the frame or contiguous with the Java Frame stack.

4.1.2.3 Method invocation

During a method invocation, the JPC points to the method header which has three compile-time data: The maximum size of the operand stack (*max_stack*), the number of local variables (*max_local*) and the number of arguments (*nargs*). Such information is generally used to dynamically allocate a newly created frame with the right size. The Figure 4.5 depicts the runtime behaviour before and after frame allocation during a method invocation.

4.2 Corruption of method frame during a method invocation

In the previous, sections we have shown that the Java Stack is a memory area in which Java Frames are stacked. The latters are allocated during a method invocation and contain array-like structures that the invokee needs during the execution of its code. We also have seen that those array-like structures are vulnerable to overflow/underflow attacks, if the right bounds checking is not properly performed.

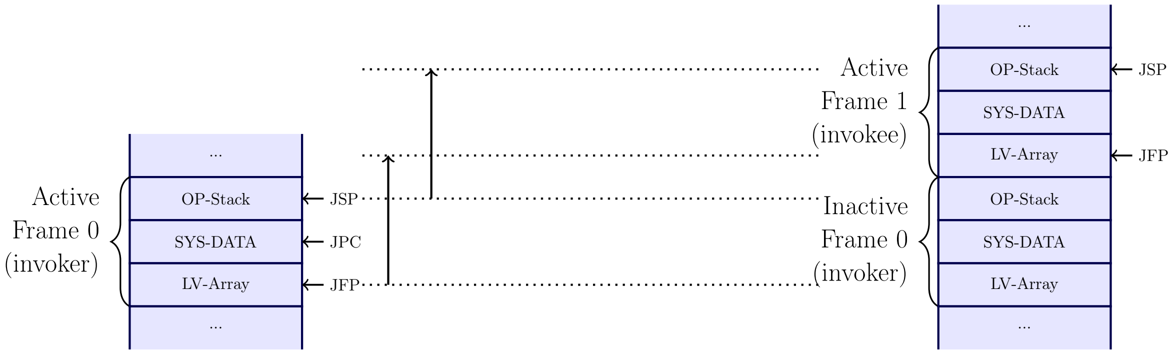


Figure 4.5: Runtime behaviour before and after frame allocation

In the next section, we are going to demonstrate, that despite a bound protection on these structures in memory, one can adopt a new approach by exploiting the frame allocation mechanism during a method invocation to fool the JCVM while performing a stack underflow.

First of all, let us define the method *ExploitFrameAlloc* as being the malicious method that is used for the attack. Its normal behaviour consists in creating a local variable, and storing it within a static byte array. The Figure 4.6 depicts different views of the body of the method *ExploitFrameAlloc*. From the left column to the right one, the first one shows the Java code, the second one the decompiled code at instruction level, and the last one would be what one should be able to see, if one opens the binary within an hexadecimal editor.

```
1 private satic void
      ExploitFrameAlloc(APDU
       apdu)
2 {
3     byte local_0 = 0;
4     bArray[0] = local_0;
5     //[...]
6 }
```

```
1  ExploitFrameAlloc(Ljavacard/
       framework/APDU;)
2  {
3      sconst_0
4      sstore_1
5      getstatic_a @ref.bArray
6      sconst_0
7      sload_1
8      sastore
9      ...
10 }
```

```
1  80 03 01 10
2  {
3       03
4       30
5       7B @ref
6       03
7       1D
8       39
9       ...
10 }
```

Figure 4.6: Different views of the body of the method *ExploitFrameAlloc*

If one takes the following bytes sequence `80 03 01 10` from Figure 4.6, line 1, it actually depicts the method header. According to the structure defined in the Figure 4.4, `03 01 10` respectively refers to *max_stack*, *nargs*, *max_locals*. They are all defined such as their value V respects the following property: $V \in \mathbb{Z}$ with $0 \leq V \leq 255$. They are useful for the JCVM to properly allocate the frame of the invokee, such as, $Frame\_size = max\_stack + nargs + max\_locals + \alpha$, where α refers to the potential additional structure that contains the system data, useful for mainly supporting normal method return.

In general, the parameters passed by the invoker are popped out from their operand stack after the completion of the invokee. Therefore, it is common that the parameters are pushed into the new frame consecutively, in order to initialize the local variables of the invokee. The normal behaviour of the frame stacking during a method invocation is illustrated by the Figure 4.5. At this point, the JFP of the newly created frame is equal to the previous JSP minus the number of argument. Therefore, as the JCVM may use these registers for checking overflow/underflow attempts, it is possible to exploit the relation $new\_JFP = prev\_JFP - nargs$ for affecting the frame's bounds during its allocation.

The attack is illustrated by the Figure 4.7. If one compares to the Figure 4.5, the allocated local variables area will overlap with previous data that are outside the newly created frame. If no control is done during the frame creation, the attack would not be detected. That is, when the overlapped area is bigger than the currently allocated Java stack, it results in a stack underflow that gives access to an undetermined memory area.

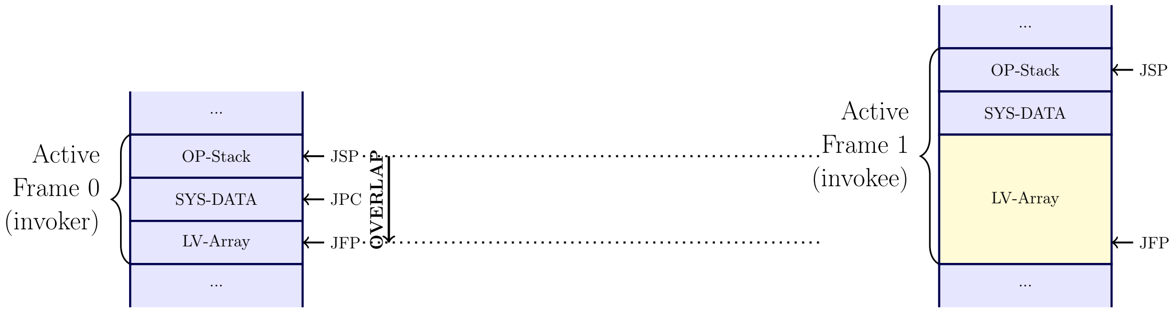


Figure 4.7: Frame bound expansion during method invocation

Due to the memory overlap, the extra data is mapped to the LV-Array structure. Therefore, accessing those data can be performed by using the instruction *<T>load <index>* that is used for fetching a value of type *<T>*, from the LV-Array at a given *index*.

At this stage the *<T>load* instructions are not interpreted as an underflow access because the local indexes remains within the newly created frame.

4.2.1 Java Card bytecode mutation

Depending on the ability of an attacker to access and modify the applet code or its environment, I am going to highlight four different techniques to perform this attack on a vulnerable Java Card platform.

4.2.1.1 "*nargs*" modification

First, it is possible to perform the attack by directly modifying the *nargs* value in the $method\_header\_info$ as described in the previous section. Then the objective is to copy each local variable word into the operand stack using a sequence of *sload* instructions. The consequence of such modification results in an ill-formed applet that may be able to access a corrupted range ($max\_locals + corrupted\_nargs$) of local variables with an underflow of ($corrupted\_nargs - original\_nargs$) words.

The easiest implementation is to create a static function with 0 arguments and 255 local variables. Then by swaping the *nargs* and *max_locals* values in the $extended\_method\_header\_info$, it is possible to perform the attack. This attack implementation requires only two steps:

1. swap the *max_locals* and *nargs* in the $extended\_method\_header\_info$,

2. discard the local variable initialisation to avoid data corruption.

Listing 4.1 Java source code of the malicious method

```
private static void maliciousMtd()
{
    short s000 = 0, s001 = 0, // ...
          s254 = 0;
    MyStaticShortArray[0] = s000;
    MyStaticShortArray[1] = s001;
    // ...
    MyStaticShortArray[254] = s254;
}
```

Listing 4.2 Byte code of the malicious method

```
//private static void maliciousMtd
flags = 0; max_stack = 3 ; nargs = 0 ;
max_locals = 255
{
   sconst_0                  // push zero value on the stack
   sstore_0                  // pop the stack value and store it into s000
   sconst_0                  // push zero value on the stack
   sstore_1                  // pop the stack value and store it into s001
   // ...
   getstatic_a      XXXX     // push MyStaticShortArray reference
   sconst_0                  // push array's index (0)
   sload_0                   // load s000 on the stack
   sastore                   // pop and store s000 in MyStaticShortArray[0]
   getstatic_a      XXXX     // push MyStaticShortArray reference
   sconst_1                  // push array's index (1)
   sload_1                   // load s001 on the stack
   sastore                   // pop and store s001 in MyStaticShortArray[1]
   // ...
   return
}
```

The Listing 4.1 shows the original Java source code and the Listing 4.2 highlights the corresponding bytecodes to be changed.

The Listing 4.3 shows the ill-formed bytecode on *extended_method_header_info*.

This ill-formed bytecode copies the data into the static array *MyStaticShortArray*. Designing this attack on an *extended_method_header_info* enables to read around 500 bytes below the stack. Theoretically, the attack would allow to dump up to 256 words, but some are discarded as they belong to the previous frame.

Listing 4.3 Exploitable ill-formed byte code

```
//private static void maliciousMtd
flags = 0; max_stack = 3; nargs = 255 ; max_locals = 0
{
    nop                          // do nothing
    nop
    nop
    nop

    // ...
    getstatic_a     XXXX    // push MyStaticShortArray reference
    sconst_0                // push array's index (0)
    sload_0                 // load s000 on the stack
    sastore                 // pop and store s000 in MyStaticShortArray[0]
    getstatic_a     XXXX    // push MyStaticShortArray reference
    sconst_1                // push array's index (1)
    sload_1                 // load s000 on the stack
    sastore                 // pop and store s000 in MyStaticShortArray[0]
    // ...
    return
}
```

4.2.1.2 Method's reference modification

Another way to carry out the same attack is to forge the method signature to be invoked. This can be achieved by several different ways:

- modify the method offset in the constant pool table to redirect the call to a different private method
- modify the method token in the constant pool table to redirect the call to a different public method
- change the *invoke<>* operand to point to another method index in the constant pool.

For example if the original method signature is

`static void myMethod1()`

the forged method signature might be

`static void myMethod2(short s_0, short s_1,[...] short s_254)`

This attack technique is an improvement of the previous technique. It is preferable as it requires less bytecode modification. Indeed, $myMethod2$ has just to carry out a sequence of *getstatic_a /.../ sastore* instructions to pull out the full underflow in a persistent array. The related Java source code is depicted by the Listing 4.4.

If *myMethod2* is successfully invoked in the place of *myMethod1*, it would have the same effect as the $nargs$ modification. The $nargs$ would be directly corrupted and no modification is required regarding the local variable initialisation. Then the underflow data are directly saved within the persistent array.

Listing 4.4 Java source code of a method that would carry out the full underflow attack

```
private static void myMethod2(short s_0, short s_1,[...] short s_254)
{
    MyStaticShortArray[0] = s_0;
    MyStaticShortArray[1] = s_1;
    [...]
    MyStaticShortArray[254] = s_254;
}
```

4.2.1.3 Token modification on *invokeinterface*

This technique is an extension of the previous techniques. It involves modifying the *method_token* operand on the *invokeinterface*. Indeed, this instruction takes as operands the number of passed arguments (XX), the interface class reference (YYYY) and the public method token (ZZ) as shown below:

```
invokeinterface XX YYYY ZZ
XX:     nargs,              1 byte
YYYY:   constantpool_index, 1 short
ZZ:     method_token,       1 byte
```

Those three operands are used to resolve the public method reference at runtime. That is, the device has to resolve the class reference that implements the interface and has to identify the method to be invoked with the *method_token*. When modifying the *method_token*, it is possible to redirect the *invokeinterface* to a malicious method that has a specific crafted signature. If the malicious method has more arguments than the original method, it induces the same attack consequences and it would be possible to illegally expand the frame size for underflow accesses.

The advantage of such implementation is that it can be taken into consideration for combined attack. A fault attack can be attempted to precisely target the *method_token* or its resolution. Fault attack might set the *method_token* value to zero or corrupt the method resolution in the class component (either on $implemented\_interface\_info$ or *public_virtual_method_table*).

Additionally, this combined attack can also be used to perform multiple type confusions at once. For example, if the interface defines the following method signatures:

```
public static void myMethod1(Object o_0, short s_0,
                             Object o_1, short s_1, ...)

public static void myMethod2(Object o_0, short s_0,
                             short s_1, Object o_1, ...)
```

the fault attack can perturb the resolution of *myMethod2* to *myMethod1*. As a result the arguments types would be interchanged between the two different signatures. Thus, it is possible to achieve as much type confusions as the number of permutation that can be defined between the two method signatures.

4.2.1.4 The export file modification

An export file contains all the public API linking information of classes in a given package. It is generated by the Java Card Converter tool during the CAP file generation. Class, method and field names are assigned with unique numeric tokens. Therefore, method signature forgery can also be achieved with rogue export files. In that case, the ill-formed application does not require post-compilation modification and it behaves the same as the attack described in "***Token modification***". This technique is well-known in the literature and it has been described by different papers, such as [Fau13], [BKKS13], [Bou14].

4.3 Attacks scenarios for Java Card

The potential attack scenarios mainly rely on the memory mapping. Depending on the nature of the dump, an attacker can get a better comprehension of the data and go further in his exploitation. To do so, a reverse engineering step is needed. As the underflow is performed on the Java stack, it is likely that the leakage data remains is the RAM area. The RAM area holds different buffers, stacks and RAM registers that can be exploited.

4.3.1 Underflow on sensitive buffers

This attack directly exploits the exposed sensitive data that can be exploited. Depending on the card memory mapping, the attack might be successful at this stage. On some JCVM implementation, we observed that the transient memory segments were below the stack. In such implementation, an attacker is able to read and write access to the transient data directly and can potentially expose transient key or tamper application RAM data. In other cases, an attacker may access other sensitive data such as the crypto input/output buffers or system key structure.

4.3.2 Underflow on Runtime Data

If no asset are directly exposed, an attacker can investigate further to exploit the underflow data. On some tested cards, it turns out that RAM registers can be found below the stack. To have write access to these runtime data, an applet needs to carry out the same attack technique by using the *sstore* instructions.

Before to exploit those runtime data, one should first identify them. Therefore, it is required to perform some reverse engineering, while analysing the data that have been dumped from the device.

4.3.2.1 Frame Exploitation on JSP

Reverse – The JSP is an easy VM variable to reverse. One can use push instructions to increase the JSP in the current frame and observe which element of the dumped data is increased or decreased.

Consequences – Depending on the platform countermeasures, the JSP can be difficult to exploit. The attack does not grant more privileges to the attacker than direct stack overflow or underflow with *pop/push/dup_x* instructions because the interpreter still applies its security policy on each instruction. It only gives more control and flexibility on the JSP

4.3.2.2 Frame Exploitation on JFP

Reverse – The JFP can be identified by observing the JSP before the *invoke*. An easy implementation is to invoke the same fake header from different methods that have different local variable array sizes. Then, on each dump only the JFP and the JSP change.

Consequences – If an attacker manages to change the JFP value, he would shift the pointer of the fetched local variables as, usually, they are relative to the JFP. This might allow him dumping even more memory from the corrupted JFP and the attacker has control over the address of the dump. This could enable to read the whole memory by varying the JFP from 0x00000000 to 0xFFFFFFFF.

However some hardware can limit the logical memory access range. Either the card detects the illegal reading on some range or not at all. In the latter, a full memory read can be performed. [MP08b] describes a full memory read attack which enable the authors to identify the different memory contents (which are platform-dependent). This attack succeeded in reading out and modifying all the code and data belonging to all other applets.

4.3.2.3 Frame Exploitation on JPC

Reverse – To characterize the JPC, an applet needs to invoke several ill-formed method headers. Then on each dump, the JPC could be identified because it is incremented according to the differences of method header offsets.

Consequences – Taking control on the JPC register would enable one to change the execution flow. As a result, a shell code might be executed as demonstrated in [BICL11].

4.3.2.4 Frame Exploitation on Execution Context

As explained in [Fau13], an attacker can deactivate the firewall by switching the execution context to the JCRE context. The author used the `dup_x` instruction for carrying out an underflow attack. That instruction is normally used for duplicating the top m operand stack words. The copied words are then inserted n words down in the operand stack. m and n form the operand of `dup_x` instruction. In case n is equal to 0, hence, the copied words are placed on top of the stack. Consequently, let us say that the operand stack is empty, calling the `dup_x` instruction will result in a stack underflow. Therefore, exploiting the `dup_x` instruction, the author was able to either, fetch data from the bottom of the stack, or overwrite data. In the particular implementation of the JCVM of the targeted device, the structure that was right below the stack was the frame header. It contains the necessary information for returning to the previous context (the caller's context). However, that structure is implementation dependent, and as a consequence, it is first required to perform some reverse engineering in order to identify the values that are stored within that structure.

Reverse – One of the value of most interest in the frame header is the current execution context. Indeed, as it has been explained in the chapter 2, section 2.3.3, it is used by the JCVM for controlling the access rights to objects assigned to an application. In order to identify it, the author carried out the underflow attack through two different applications. Both of them have to extract the data below the stack using the `dup_x` instruction. Since to each application is assigned a value that represents the execution context, an analysis of the extracted values may highlight the differences.

Consequences – Being able to change the execution context of the current executed method may enable one to spoof the identity of another application. This may grant further access rights. The worst case would be, to be able to spoof the JCRE identity. As it is likely that it has the highest privileges, it would give a no-limit access to the method that is being invoked.

4.3.2.5 Frame Exploitation on call-return structure

Reverse – The call-return structure is difficult to identify. To reverse its structure, a combination of different techniques described above should be applied when calling the ill-formed methods.

Consequences – The consequences are the same as above because the structure is supposed to store the previous VM state related to the execution of the caller method. In this case, the exploitation is only efficient when the ill-formed method completes and the VM returns into the caller method without throwing exception.

4.4 Attack assumptions

We have presented different reverse engineering methods and potential exploitations. However, these attacks rely under certain assumptions:

The Verifier checks – Applet to be installed must pass the different static checks performed by the Oracle Bytecode Verifier. However, as we previously pointed out, combined attacks are well-known in the Java Card security, because they can render benign application to a malign one. Unfortunately, this type of attack has not been attempted during this thesis. Furthermore, it is important to notice that the BCV is still worth to test for finding potential bugs. As any other software, the BCV may be prone to some lack of checks that an attacker can exploit to silently embed malicious ill-formed code within the application. The recent attack of Lancia *et al.* in [LB15] that has been described in chapter 3, section 3.2.3.3, clearly demonstrates that.

Loading keys – The attacker needs to get the loading keys that is required for installing his malicious application onto a given Java Card. In the case of Karsten Nohl's attack in [Noh13], the weakness resides in the SIM when using a Toolkit Application key for message integrity. However, if the SE is properly managed and configured, the exposure of such secret does not grant full installation privilege. Therefore, the danger does not necessarily comes from the outside world, but instead, from the inside. Indeed, nowadays, as the underlying hardware is getting more efficient and reliable, consequently, more and more application vendors could share spaces on the tiny device. The risk that a given application attempts to endanger other applications, or the whole platform, cannot be discarded.

4.4.1 Reliance on secret of other entities

GlobalPlatform provides security policies and authentication protocols for Card Content management. However, due to the different business and issuance models in the field, it might be difficult to assert in some cases that the secure channel keys confidentiality is fully enforced. It is true to say that GP brings the security towards by limiting the attack environment but if no efficient countermeasures have been implemented, devices may remain in the field with their vulnerabilities for several years. Moreover, if the “Over The Air” (OTA) channel keys are exposed, attacks can be performed remotely without any physical access to the devices.

4.4.2 The bytecode verification

The Oracle BCV verifications are based on static analysis and they do not cover attacks through bug exploitations or memory allocation mechanisms. Furthermore, most of the ill-formed bytecode sequences might be hidden behind a rogue export file. For all of these reasons, validation authorities must take the utmost care that the export files used for verification, match the on-card CAP files. Even so, it is difficult to assert that all the deployed applications have undergone the Oracle BCV, yet it is highly recommended.

4.4.3 New edition, new vulnerabilities

Java Card products must pass through functional testing and security evaluation to gain high assurance on the overall product security. However, with the new emerging technologies, Java Card environment is evolving quickly. Since 2009, Sun released the Java Card 3.0 specification that defines a new JCVM and a new JCRE for the deployment of high end SE and USB tokens. To satisfy the needs of the SIM card industry, some the SIM Toolkit (STK) APIs were introduced and standardized by ETSI for the GSM world. It has already proved their efficiency in helping the telecom operators adding some features to the mobile telephony through the SIM Java Cards. However, nowadays, with the introduction of sophisticated user interfaces in telephony, the simplicity of the STK menus turned out to be too poor. The Smart Card Web Server concept was then introduced to provide SIM card software developers the means to design richer user interfaces for SIM card applications. Consequently, more functional capabilities may also introduce new vulnerabilities.

A well detailed analysis of the vulnerabilities introduced with Java Card 3 Connected Edition can be seen in [Cal13]. It shows that various new features have actually increased the attack surfaces.

For instances:

- Dynamic Class Loading may significantly complicate the type safety enforcement. Then reference forgery by type confusion will be more threatening than the actual type confusion attacks.
- Dynamically loaded classes may embed malicious code.
- Multithreading makes it more difficult to analyse security.
- Web-applications may introduce various code injections attacks well known in the desktop world.

However, at the time this is being written, there is no significant deployed products with Java Card 3.0 Connected Edition yet, although it supports Java 6 and thus offers a lot of richer features (*e.g.* threads, full types, generics [Bra04]).

4.5 Conclusion

We have presented in this chapter, a software based attack that leverages the execution of ill-formed applet on Java Card device to induce an underflow that might lead to a full exploitation of the targeted platform. We have demonstrated through different attack techniques that an exploitable flaw might exist on card Operating System (OS). This flaw can lead to data leakage and can be exploited to gain knowledge of a particular device. It potentially enables one to access or change the Java Card assets (such as code/secret data of the OS or resident applications). We have presented a concrete and powerful attack that exploits potential vulnerabilities caused by the lack of runtime checks of the JCVM. This can lead to various attack scenarios with read and write access to a part of the memory. We have also shown that because an embedded virtual machine is implementation dependent, it implies some reverse engineering steps to go further in the exploitation. We briefly explained, how it is possible to identify the system data on a partial memory dump. We have also described potential exploitations that can be achieved depending on the leaked data. This might enable one to threaten additional Java Card assets.

One should note that this technique is only harmful to open platforms. That is, only devices with post-issuance applet management capability are prone to this attack. Besides, the ill-formed applets used in this attack do not pass the Oracle's off-card BCV and this attack has been performed only on cards that do not have on-card BCV.

Chapter 5

Visual Forensic Analysis of a Java Card Raw Memory Dump and Automatic Code Extraction

Contents

Up to now, one of the most common attack that targets Java Card-based platform consists in dumping its memory. We have shown in the Chapter 4, a new approach for performing an underflow attack. A possible exploitation of the latter can be, dumping the memory. However, while the raw binary content is extracted, the attacker still has to retrieve the sensitive asset that he is looking for.

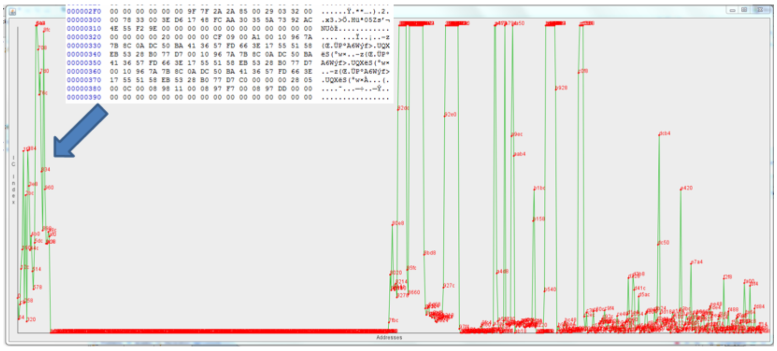
The application's code is of particular interest for an attacker. However, distinguishing the code and data from a memory dump is not trivial and requires some reverse engineering skills. A technique has already been proposed in the literature [LBL$^+$14] in order to delineate the interesting areas. It mainly uses the Index of Coincidence (IC) computation technique to determine to which extent the frequency distribution of the raw memory dump is similar to the Java Card set of byte codes. Computing a proper IC value requires to gather a lot of Java Card applets in order to perform the computation on them. It would enable to get an IC value that should best reflect the real "aspect" of a Java Card code. The more samples are used during this step, the more accurate is the IC value that is then used as a reference. The main challenge consists in gathering the largest possible set of applets on which the IC value can be computed.

To overcome this very important step, we present a new approach for identifying Java Card code within a raw memory dump and for extracting it without the need of gathering a wide range of different applets.

5.1 Problem Analysis

5.1.1 Memory Forensics of a Java Card Dump

Lanet *et al.* described in [LBL+14] a technique that uses the Index of Coincidence as a heuristic in order to determine the area in a memory dump that is likely to be similar to a Java Card application code. The Index of Coincidence basically shows how the frequency distribution of a set A is similar to a set B. It is usually used in cryptography for breaking substitution ciphers. For instance, it measures how alike cryptogram frequencies are to a given language (e.g: English). In their paper, Bouffard *et al.* state that an acceptable IC for Java Card language is located between 0.020 and 0.060. In order to locate Java Card code, they use a sliding window with a certain size that compute an IC value while parsing the memory dump. However, they do not precise among which kind of byte code they performed their study phase. Furthermore, they also do not give any information regarding the size/number of samples they used to determine the proper IC. The more samples are used, the more accurate should be the IC value.



```
000002F0  00 00 00 00 00 00 9F 7F 2A 2A 85 00 29 03 32 00   ......Ÿ.**…).2.
00000300  00 78 33 00 3E D6 17 48 FC AA 30 35 5A 73 92 AC   .x3.>Ö.Hü*05Zs'¬
00000310  4E 55 F2 9E 00 00 00 00 00 00 00 00 00 00 00 00   NUòž............
00000320  00 00 00 00 20 00 00 00 CF 09 00 A1 00 10 96 7A   .... ...Ï..¡..–z
00000330  7B 8C 0A DC 50 BA 41 36 57 FD 66 3E 17 55 51 58   {Œ.ÜP°A6Wýf>.UQX
00000340  EB 53 28 B0 77 D7 00 10 96 7A 7B 8C 0A DC 50 BA   ëS(°w×..–z{Œ.ÜP°
00000350  41 36 57 FD 66 3E 17 55 51 58 EB 53 28 B0 77 D7   A6Wýf>.UQXëS(°w×
00000360  00 10 96 7A 7B 8C 0A DC 50 BA 41 36 57 FD 66 3E   ..–z{Œ.ÜP°A6Wýf>
00000370  17 55 51 58 EB 53 28 B0 77 D7 C0 00 00 00 28 05   .UQXëS(°w×À...(.
00000380  00 0C 00 08 98 11 00 08 97 F7 00 08 97 DD 00 00   ....˜...—÷..—Ý..
00000390  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Figure 5.1: Index of Coincidence computation trace (x-axis: offsets in the memory dump, y-axis: IC values)

The Fig. 5.1 depicts the resulting trace after computing the Index of Coincidence values over a memory dump using a sliding window. Different areas of interest can be noticed. The first one on the leftmost is the one that contains the actual code. The area of peaks in middle of the graph are just noises. And the third area does not reflect Java Card code because the width of the multiple blocks that compose this area are too small.

Over 64Kbytes of memory dump, it appears then that only 380 bytes are necessary and can be considered as potential Java Card code, so it can be extracted/analysed/disassembled later on. Once the block extracted, additional analysis must be performed in order to retrieve the real start of the code area, and also its real end (dynamic analysis of the control flow graph, execution/simulation/analysis of the stack, etc.)

Despite the fact that this heuristic seems to be enough for distinguishing code from data within a given raw memory dump, some questions remain, in case one tries to reproduce the experiment for recomputing a new IC value. As any other statistic experiment, the latter may certainly vary according to the conditions of the experiment, therefore, in the next section, we are going to explain a bit further the IC computation, in order to get a better understanding of its application and the parameters to take into account for computing it.

5.1.2 Understanding the Index of Coincidence Computation

The index of Coincidence, denoted IC, was first introduced by Friedman in 1920 [Fri87]. The author describes the IC as a measure of how likely it is to draw two matching letters by randomly selecting two letter from a given text. The IC computation technique was mainly used in the cryptography area. For instance, it was used for attacking the Vigenere cipher which is a very well known cipher that is based on polyalphabetic substitution.

In cryptography randomness is a very important aspect to take in consideration for both the design of the cipher, and its output. By definition, randomness is the lack of pattern or predictability in a given sequence of events, symbols or steps. The principle of the IC computation comes from the probability of the same letter occuring in two different texts at the same position, hence, the author introduced two terms for defining a text. Either it is random, then it is described as "flat", or it fails to be random in some way, then it is called "rough".

To illustrate this, the Figure 5.2 illustrates the difference of shape between two histograms: the first one (left) depicts the frequency of letters in a sample noted A. Letters are taken from the English alphabet, which counts 26 letters. The sample A is generated randomly, using the Linux command shown in the Listing 5.1. The second histogram (right), depicts the letters frequency from the sample B, which is a text taken from the first three Chapters of the book: "*Frankenstein, or the Modern Prometheus*" [10]. As one can notice, the shape of the first histogram is flatter than the second one.

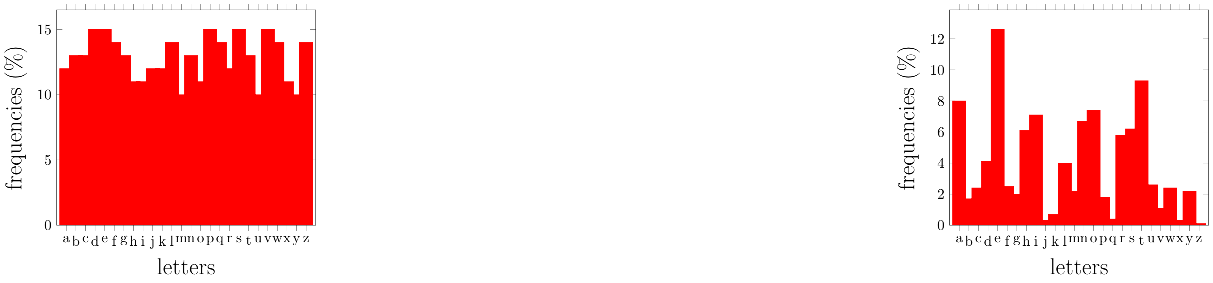


Figure 5.2: Letters frequency comparison: Random text versus English text

Listing 5.1 Linux command for randomly generating a sequence of 30635 letters of the alphabet

```
< /dev/urandom tr -dc a-z | head -c30635
```

Therefore, one can identify two extremes: a rough text, and a flat one. Consequently, Friedman proposed to measure how likely two texts are "rough" in the same way. The test he proposed consists in writing the texts one above the other and counting the number of occurences for which the same letters come together. Two letters appearing at the same position in two different texts is then called a "coincidence". Therefore, the IC computation is defined such as being the ratio of the number of coincidences to the number of coincidences that are expected in a random text. It can then be noted:

$$I.C = \frac{ActualCoincidences}{ExpectedCoincidences} \tag{5.1}$$

[10] http://gutenberg.org/files/84/84-h/84-h.htm

5.1.2.1 Index of Coincidence of a Language

From a theoritcal point of view, let E be the event such as, all the letters (5.2) in a language with C characters, occured with the same frequency in a given text T. The probability that E occurs can be noted

$$P(E) = \frac{1}{C} \tag{5.2}$$

The $I.C$ value has two extremes. Considering a text T, then two cases exist:

1. either every letter only occupies one position at a time, then $I.C = 1$

2. or a given letter occupies all the position in T, then $I.C = C$

One can then conclude that the expected $I.C$ value of a given language L, is such as

$$1 \leq I.C(L) \leq C \tag{5.3}$$

5.1.2.2 Index of Coincidence for measuring the roughness of a text

We have seen in the previous paragraph that if all of the letters of a given language L occured with the same frequency, hence, the probability for each letter would have to be $\frac{1}{C}$ (see Equation 5.2).

Let p_i the probability of finding the i^{th} letter a certain amount of times in a given text. The probability that p_i differs from $\frac{1}{C}$ is

$$P_{diff} = p_i - \frac{1}{C} \tag{5.4}$$

Consequently, the amount of all the letter probabilities that differ from $\frac{1}{C}$ would be

$$I.C = \sum_{i=1}^{C}(p_i - \frac{1}{C})^2 \tag{5.5}$$

We square for ensuring that the result is always positive, because some differences $p_i - \frac{1}{C}$ may be negative. The total difference would result in a sum of 0, because $\sum_{i=1}^{C}(p_i - \frac{1}{C}) = \sum_{i=1}^{C} p_i - \sum_{i=1}^{C} \frac{1}{C} = 1 - C \times \frac{1}{C} = 0$.

If one exapands 5.5, and considering the condition 5.2, one would have

$$I.C(T) = \sum_{i=1}^{C} {p_i}^2 - \frac{1}{C} \tag{5.6}$$

Therefore, what one can deduce is that if one has access to the list of p_i for every letter in a given language L, then, it is possible to precisely compute the Index of Coincidence of the text T under examination, in order to determine how alike it is to L. Consequently, if $T \to T_{rough}$, we have $I.C(T) \longrightarrow I.C(L)$. Otherwise, if $T \longrightarrow T_{flat}$, then we have $I.C(T) \underset{p_i \to \frac{1}{C}}{\longrightarrow} 0$.

5.1.2.3 Approximating the I.C measure

The accuracy of the Equation 5.6 mainly depends on the accuracy of each individual p_i. Thus, the main caveat is that generally speaking, getting the list of p_i cannot be easily performed, because, one should have a certain amount of samples, for enhancing the accuracy. Fortunately, Friedman proposes an approximation of the I.C measure that does not involve computing all the p_i. Furthermore, his proposition only involves a single sample.

Indeed, let S be a sample with length equal to N. Let f_i the occurences of the i_{th} element in S. There are $f_i(f_i - 1)$ different ways to choose that i_{th} element within S. As a pair of i element can be counted in 2 orders in S, a refined count would be $\frac{f_i(f_i-1)}{2}$. Furthermore, the number of ways to choose 2 of the same element in S is

$$\sum_{i=1}^{C} \frac{f_i(f_i - 1)}{2} \tag{5.7}$$

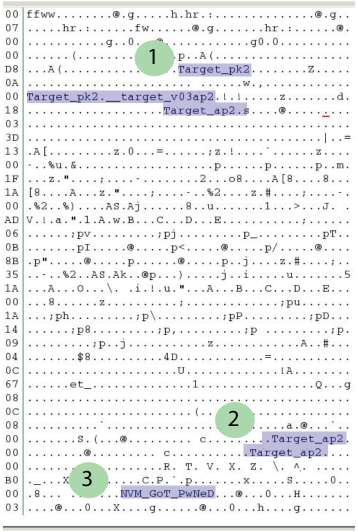
Finally, if we consider N the length of S, there are $\frac{N(N-1)}{2}$ different ways of arbitrarly choosing 2 elements in S. The probability that the two elements are the same is

$$\frac{\sum_{i=1}^{C} f_i(f_i - 1)}{N(N - 1)} \tag{5.8}$$

As a consequence, theoritically, (5.6) is a more robust approach, however, practically (5.8) is more feasible.

5.1.3 Analysis of unknown Binary file

A common habit for analysing any “unknown” binary file is to identify the existing strings within it. Those strings may be valuable hints for identifying what is in the memory dump (e.g: a package name, com.mycompany.mypackage). For instance the Fig. 5.3 depicts some strings that help in identifying: (1), (2) some meta-data related to a given installed applet, (3) the start of an array that is located in the Non-Volatile Memory area. In case of files recovery, there are

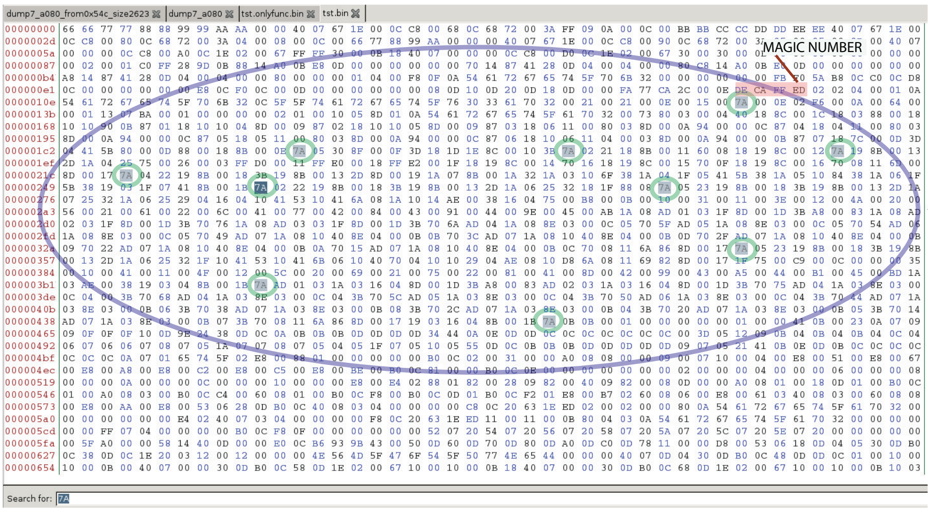


```
00 ffww........@.g.....h.hr.:..............@.g..
07 .....hr.:......fw......@.g.......hr.:......@.
00 ...........g..0..@...............g0.0..........
00 ......(...........p..A(....................
D8 ...A(.............Target_pk2........Z.....
0A ........................ ........w.,............
00 Target_pk2.__target_v03ap2.!.!......z.......d.
18 ...................Target_ap2.s....@..........
03 ..............................................
3D ...........................................|..=
13 .A[.........z.0...=......;z.!....`.......z....
00 -..%u.&...................p......p......p..m.
1F ...z."....;....-........2...o8....A[8....8...
1A [8....A...z."....;.....-..%2....z.#....;....-.
00 .%2..%)....AS.Aj......8..u.......1...>...J. .
AD V.!.a.".l.A.w.B...C...D...E............;......
06 .......;pv.........;pj.........p_.........pT..
0B ........pI.....@.....p<.....@.....p/.....@....
8B .p".....@.....p......@.....p..j....z.#....;..
35 ..-..%2..AS.Ak..@p...).....j..i.....u.......5
1A ...A...O...\. .i.!.u."...A...B...C...D...E...
00 ...8......z.............;.............;pu.......
1A ...;ph.........;p\.........;pP.........;pD...
14 .......;p8.........;p,.........;p .........;p.
09 ..........;p..j...........z............A..#...
04 .......$8..........4D............=...........
0C ....................U.............!A........
67 ......et_..............1................Q...g
08 ..............................................
0C ...........................(...................
08 ..........`....................a.@....`..
00 .......S.(...@......... c............Target_ap2.
00 ........@......... c...............Target_ap2....
00 ..............R. T. V. X. Z. \. ^. .....
B0 ._...X.........C.P.`.p.......x.....S.....0..
00 .8...  ........NVM_GoT_PwNeD...@...0...H.......
03 ....@...0...X....g......@...0...h....g.......
```

Figure 5.3: Identification of meta-data and a sample of the Non-Volatile Memory area

plenty of file carving tools [11]. The most known are probably *volatility*, *foremost*, *photorec*. Most file carvers operate by simply “carving out” blocks after looking for file headers and/or footers. The majority of file carving programs only recover files that are not fragmented.

In practice, while analysing Java Card raw memory dump, one should bear in mind that the output is platform dependent. This means that objects, or code structures, or simply, memory layouts, highly depends on how the Java Card platform handle them. Therefore, there is no specific structure, or meta-data, one may always rely on, for automatically identifying the start or the end of the code. In the example depicted by the Fig. 5.4 one can identify a magic number ($0xDECAFFED$) that identifies the Java Card CAP file format. While in this case it is obvious to determine the start of the applet, this case cannot be generalized on other platforms. However, a trained eye is relatively capable of quickly identifying some specific details. For instance, a method always ends with a *return* instruction according to the value to be returned: *return* (0x7A, return void), *sreturn* (0x79, return a short), *areturn* (0x77, return a reference). If we still take the same example depicted by the Fig. 5.4, if one highlights the byte value $0x7A$ (*return* instruction), it is possible to distinguish the area where the latter is the most concentrated.



```
00000000 66 66 77 77 88 88 99 99 AA AA 00 00 40 07 67 1E 00 0C C8 00 68 0C 68 72 00 3A FF 09 0A 00 0C 00 BB BB CC CC DD DD EE EE 40 07 67 1E 00
0000002d 0C C8 00 80 0C 68 72 00 3A 04 00 08 00 0C 00 66 77 88 99 AA 00 00 00 40 07 67 1E 00 0C C8 00 90 0C 68 72 00 3A [illegible] 00 40 07
0000005a 00 00 00 0C C8 00 A0 0C 1E 02 00 67 FF FF 30 00 0B 18 40 07 00 00 00 0C C8 00 D0 0C 1E 02 00 67 30 00 30 00 0B [illegible] 00 00 00
00000087 00 02 00 01 C0 FF 28 9D 0B 88 14 A0 0B E8 0D 00 00 00 00 00 00 70 14 87 41 28 0D 04 00 04 00 00 80 C8 14 A0 0B E8 0D 00 00 00 00 00 00
000000b4 A8 14 87 41 28 0D 04 00 04 00 00 80 00 00 00 01 04 00 F8 0F 0A 54 61 72 67 65 74 5F 70 6B 32 00 00 00 00 00 00 FB F0 5A B8 0C C0 0C D8
000000e1 0C 00 00 00 00 00 00 E8 0C F0 0C 00 0D 00 00 00 00 00 00 08 0D 10 0D 20 0D 18 0D 00 00 FA 77 CA 2C 00 0E DE CA FF ED 02 02 04 00 01 0A
0000010e 54 61 72 67 65 74 5F 70 6B 32 0C 5F 5F 74 61 72 67 65 74 5F 76 30 33 61 70 32 00 21 00 21 00 0E 00 15 00 7A 00 0E 02 F6 00 0A 00 64 00
0000013b 00 01 13 07 BA 00 01 00 00 00 00 02 01 00 10 05 8D 01 0A 54 61 72 67 65 74 5F 61 70 32 00 73 80 03 00 04 40 18 8C 00 1C 18 03 88 00 18
00000168 10 10 90 0B 87 01 18 10 10 04 8D 00 09 87 02 18 10 10 05 8D 00 09 87 03 18 06 11 00 80 03 8D 00 0A 94 00 00 0C 87 04 18 04 11 00 80 03
00000195 8D 00 0A 94 00 00 0C 87 05 18 05 11 00 80 03 8D 00 0A 94 00 00 0C 87 06 18 10 06 11 04 00 03 8D 00 0A 94 00 00 0B 87 07 18 7C 00 0D 3D
000001c2 04 41 5B 80 00 0D 88 00 18 8B 00 0E 7A 05 30 8F 00 0F 3D 18 1D 1E 8C 00 10 3B 7A 02 21 18 8B 00 11 60 08 18 19 8C 00 12 7A 19 8B 00 13
000001ef 2D 1A 04 25 75 00 26 00 03 FF D0 00 11 FF E0 00 18 FF E2 00 1F 18 19 8C 00 14 70 16 18 19 8C 00 15 70 0F 18 19 8C 00 16 70 08 11 6D 00
0000021c 8D 00 17 7A 04 22 19 8B 00 18 3B 19 8B 00 13 2D 8D 00 19 1A 07 8B 00 1A 32 1A 03 10 6F 38 1A 04 1F 05 41 5B 38 1A 05 10 84 38 1A 06 1F
00000249 5B 38 19 03 1F 07 41 8B 00 1B 7A 02 22 19 8B 00 18 3B 19 8B 00 13 2D 1A 06 25 32 18 1F 88 08 7A 05 23 19 8B 00 18 3B 19 8B 00 13 2D 1A
00000276 07 25 32 1A 06 25 29 04 16 04 10 41 53 10 41 6A 08 1A 10 14 AE 00 38 16 04 75 00 B8 00 0B 00 10 00 31 00 11 00 3E 00 12 00 4A 00 20 00
000002a3 56 00 21 00 61 00 22 00 6C 00 41 00 77 00 42 00 84 00 43 00 91 00 44 00 9E 00 45 00 AB 1A 08 AD 01 03 1F 8D 00 1D 3B A8 00 83 1A 08 AD
000002d0 02 03 1F 8D 00 1D 3B 70 76 1A 08 AD 03 03 1F 8D 00 1D 3B 70 6A AD 04 1A 08 8E 03 00 0C 05 70 5F AD 05 1A 08 8E 03 00 0C 05 70 54 AD 06
000002fd 1A 08 8E 03 00 0C 05 70 49 AD 07 1A 08 10 40 8E 04 00 0B 0B 70 3C AD 07 1A 08 10 40 8E 04 00 0B 0D 70 2F AD 07 1A 08 10 40 8E 04 00 0B
0000032a 09 70 22 AD 07 1A 08 10 40 8E 04 00 0B 0A 70 15 AD 07 1A 08 10 40 8E 04 00 0B 0C 70 08 11 6A 86 8D 00 17 7A 05 23 19 8B 00 18 3B 19 8B
00000357 00 13 2D 1A 06 25 32 1F 10 41 53 10 41 6B 06 10 40 70 04 10 10 29 04 AE 08 10 D8 6A 08 11 69 82 8D 00 17 1F 75 00 C9 00 0C 00 00 00 35
00000384 00 10 00 41 00 11 00 4F 00 12 00 5C 00 20 00 69 00 21 00 75 00 22 00 81 00 41 00 8D 00 42 00 99 00 43 00 A5 00 44 00 B1 00 45 00 BD 1A
000003b1 03 AE 00 38 19 03 04 8B 00 1B 7A AD 01 03 1A 03 16 04 8D 00 1D 3B A8 00 83 AD 02 03 1A 03 16 04 8D 00 1D 3B 70 75 AD 04 1A 03 8E 03 00
000003de 0C 04 00 3B 70 68 AD 04 1A 03 8E 03 00 0C 04 3B 70 5C AD 05 1A 03 8E 03 00 0C 04 3B 70 50 AD 06 1A 03 8E 03 00 0C 04 3B 70 44 AD 07 1A
0000040b 03 8E 03 00 0B 06 3B 70 38 AD 07 1A 03 8E 03 00 0B 08 3B 70 2C AD 07 1A 03 8E 03 00 0B 04 3B 70 20 AD 07 1A 03 8E 03 00 0B 05 3B 70 14
00000438 AD 07 1A 03 8E 03 00 0B 07 3B 70 08 11 6A 86 8D 00 17 19 03 16 04 8B 00 1B 7A 0B 0B 00 01 00 00 00 00 00 01 00 00 41 0B 00 23 0A 07 09
00000465 09 0F 0F 0F 10 0D 9E 24 38 0D 0C 0A 0B 0B 0B 0D 0D 0D 0D 34 44 0A 0E 0D 0D 0C 0C 0C 0C 0C 0C 0C 00 3D 05 12 09 0B 04 0B 04 0B 04 0C 04
00000492 06 07 06 06 07 08 07 05 1A 07 07 08 07 05 04 05 1F 07 05 10 05 55 0D 0C 0B 0B 0B 0D 0D 0D 0D 0D 09 07 05 21 41 0B 0E 0D 0B 0C 0C 0C 0C
000004bf 0C 0C 0C 0A 07 01 65 74 5F 02 E8 00 88 01 00 00 00 00 00 B0 0C 02 00 31 00 00 A0 08 08 00 00 09 00 07 10 00 04 00 E8 00 51 00 E8 00 67
000004ec 00 E8 00 A8 00 E8 00 C2 00 E8 00 C5 00 E8 00 BE 00 B0 0C 81 00 00 B0 0C 0E 00 00 00 00 00 00 00 02 00 00 00 04 00 00 00 06 00 00 00 08
00000519 00 00 00 0A 00 00 00 0C 00 00 00 10 00 00 00 E8 00 E4 02 88 01 82 00 28 09 82 00 40 09 82 00 08 0D 00 00 A0 08 01 00 18 0D 01 00 B0 0C
00000546 01 00 A0 08 03 00 B0 0C C4 00 60 08 01 00 B0 0C F8 00 B0 0C 0D 01 B0 0C F2 01 E8 00 B7 02 60 08 06 00 E8 00 61 03 40 08 03 00 60 08 08
00000573 00 E8 00 AA 00 E8 00 53 06 28 0D B0 0C 40 08 03 04 00 00 00 00 C8 0C 20 63 1E ED 02 00 02 00 00 80 0A 54 61 72 67 65 74 5F 61 70 32 00
000005a0 00 00 00 00 00 00 E4 02 40 07 03 04 00 00 00 00 F8 0C 20 63 1E ED 11 00 11 00 0B 80 04 03 0A 54 61 72 67 65 74 5F 61 70 32 00 00 00 00
000005cd 00 00 FF 07 04 00 00 00 00 B0 0C F8 0F 00 00 00 00 00 00 52 07 20 54 07 20 56 07 20 58 07 20 5A 07 20 5C 07 20 5E 07 20 00 00 00 00 00
000005fa 00 5F A0 00 00 58 14 40 0D 00 00 E0 0C B6 93 9B 43 00 50 0D 60 0D 70 0D 80 0D A0 0D C0 0D 78 11 00 00 D8 00 53 06 18 0D 04 05 30 0D B0
00000627 0C 38 0D 0C 1E 20 03 12 00 12 00 00 00 4E 56 4D 5F 47 6F 54 5F 50 77 4E 65 44 00 00 00 40 07 0D 04 30 0D B0 0C 48 0D 0D 0C 01 00 10 00
00000654 10 00 0B 00 40 07 00 00 30 0D B0 0C 58 0D 1E 02 00 67 10 00 10 00 0B 18 40 07 00 00 30 0D B0 0C 68 0D 1E 02 00 67 10 00 10 00 0B 10 03
```

Search for: 7A

Figure 5.4: Identification of return instructions within a Java Card raw memory dump

In this particular example, the methods do not contain that much instructions. Consequently it is relatively easy to highlight the area where the code is located. The larger is the size of the raw memory dump and the different methods, harder is such kind of reverse engineering exercise. Therefore, one must find an automatic way for handling a large raw memory dump, and for distinguishing the code from data.

5.1.4 Visual Reverse Engineering of Binary and Data Files

In their paper Conti *et al.* [CDSS08] present some design principles for files analysis. By using different graphing techniques, they show how to identify the contents of binary files. Indeed, they demonstrate how quick you can detect certain patterns within the binary file. This way, it is easier to highlight what is contained within the file. One can notice then that every type of document (jpg, word, etc.) has its own ”visual signature“ (see figure Fig.5.5). Their results

[11] list of file carving tools: http://forensicswiki.org/wiki/Tools:Data_Recovery

indicate that visual approaches help analysts for identifying files or analyzing unfamiliar file structures. These principles are the main essence that led me to follow their approaches.

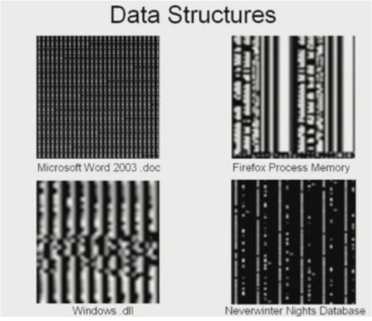


Figure 5.5: Visual signature of (from left to right): a Word document, a Firefox process Memory, a Windows dll, a Neverwinter Nights Database

5.1.5 Experiments

Inspired from Conti *et al.* work [CDSS08] one can check how CAP files look like by using one of their graphing technique, in order to check if it is possible to distinguish the code from the data. During the different tests we have performed, we worked with Java Card applets that are different in size, and also in purpose. The first one, named TargetApp, is an application that we have installed on a given smart card that we name TargetCard. The loading keys of the latter were known, that is why we could authenticate and load TargetApp on the device. The others are just different Java Card applications found on the internet [12]: JavaCardApp1 (PKIApplet) and JavaCardApp2 (OpenPGPApplet). The memory dump used for experiments in the rest of the paper comes from the Non-Volaile Memory (NVM) area. we successfully dumped the NVM using a malicious ill-formed applet as presented in [TB15]. After some static/manual reverse engineering, we retrieved the code of TargetApp. The aim of the different experiments is then to try to find a way to perform the previous job (almost) automatically in order to speed up the reverse engineering step during an evaluation.

5.1.5.1 Visual Signature of a Java Card Application

As an example we took two different applications and we applied the "*Byte Plot*" technique to visualize their representation, depicted in Fig. 5.7 and Fig. 5.8.

Fig. 5.6 shows how Byte Plots are drawn. The "Height" value can be arbitrarily set and enables define the output size of the image. According to size of image, patterns can be better spotted.

5.1.5.2 CAP components extraction

A CAP file behaves like a JAR/ZIP archive file format. For instance one can use the software "*Unzip*" to extract its content. Once extracted, one can retrieve the different CAP file components.

[12] https://github.com/martinpaljak/AppletPlayground

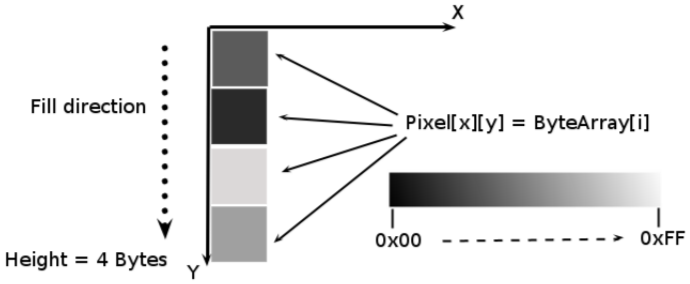


Figure 5.6: Byte Plot of a byte array.

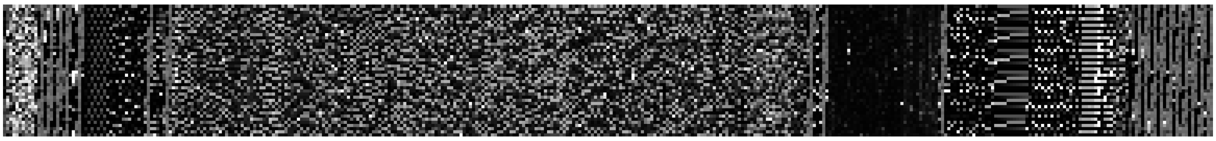


Figure 5.7: Visual signature of Java CardApp1 , size = 16KB

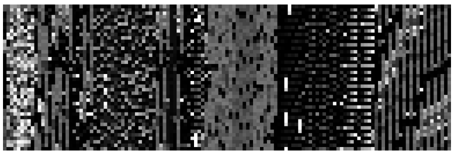


Figure 5.8: Visual signature of a TargetApp, size = 8KB

Once Java CardApp1 (Fig. 5.7) has been decompressed one can independently draw and check the "Byte Plot" for each CAP file components. For example, below are the plots of the Method component, the Reference Location and the Descriptor component:

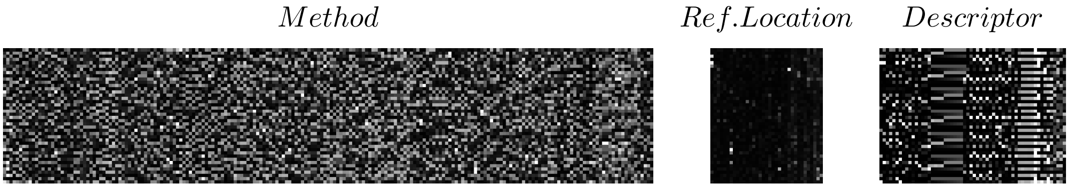


Figure 5.9: Byte plots of three specific CAP file components

5.1.5.3 Byte distribution analysis and characterization

Byte distribution – What we can deduce from the figures Fig. 5.7, Fig. 5.8, Fig. 5.9 is the fact that the code of an application (stored in the Method Component) has a byte distribution different from the other data that compose a Java Card application. Fig. 5.10 shows a comparison between the byte distribution of different CAP components (Method component included). Furthermore, one can see in figure Fig. 5.11 the byte distribution of 8 Method Components from 8 different Java Card applications.

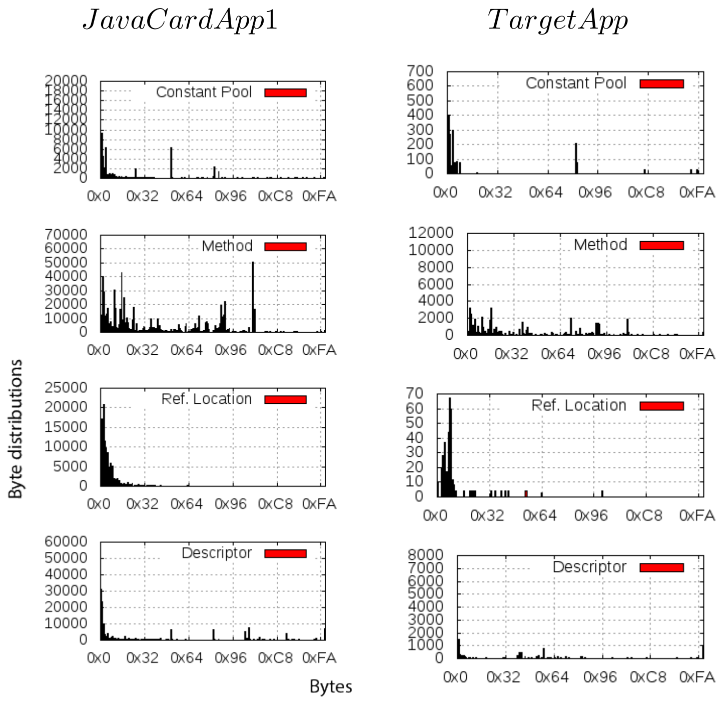


Figure 5.10: comparison of byte distribution within 4 components of 2 different applications

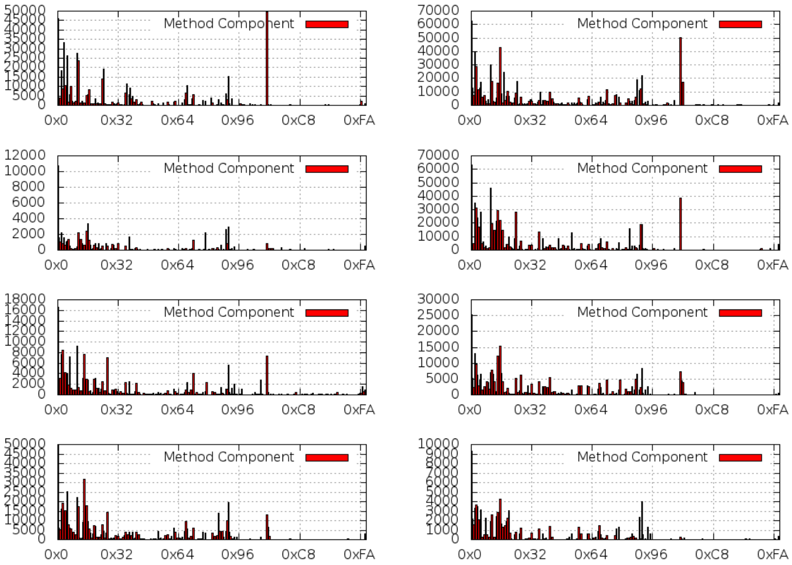


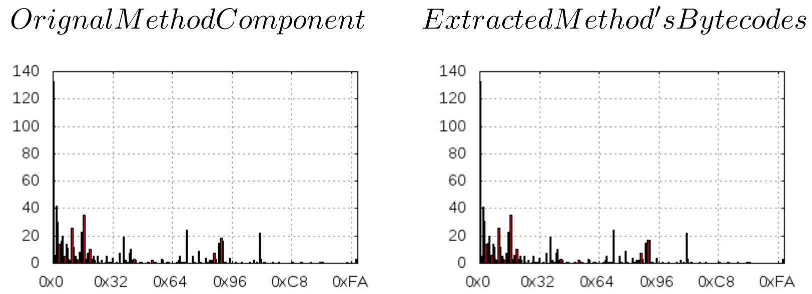
Figure 5.11: Byte distribution of different Method Components

Statistical tests for comparing two sets of byte distribution – As all is a matter of byte distribution, we need to find a way to test the byte distribution of a given sample of raw memory dump against a kind of "template". This template can be for instance the byte distribution of the Method Component of another CAP file.

Let A be the byte distribution of a given Method Component, and B a sample of the byte distribution from the raw memory dump. Furthermore, let C be the byte distribution of another CAP component. The statistical test is noted ST and X, Y are two arbitrary samples of byte distribution. The latter should be able to verify the following properties: $ST(A, B) \approx 1$, $ST(C, B) \approx 0$, $0 \leq ST(X, Y) \leq 1$. A is a reference that can be compared to B. The statistical test should be able to check if A looks like B. It should also verify that C does not look like B. Choosing the right test to compare two measurements is a bit tricky, as one must choose between two families of tests: parametric and non-parametric. Tests are referred to as parametric tests while the data are sampled from a "*Gaussian*" or "*normal*" distribution. Otherwise if one does not want to make any assumption regarding the data distribution one should choose non-parametric tests.

Analysis of the byte distribution of a Java Card Memory Dump In order to confirm that we properly extracted the right portion of memory that normally contains the target applet's code, we visually compared the byte distribution of TargetApp against the byte distribution of A.

As one can see from the figure Fig. 5.12 the byte distribution are definitely the same. This gives us the information that the code installed on the card has not been altered in any way.



$OrignalMethodComponent$ $ExtractedMethod'sBytecodes$

Figure 5.12: comparison of byte distribution between the original target applet's method component (distribution of reference) and the extracted byte codes

Correlation tests – The Pearson Correlation enables to compare two data sets in order to determine if similarities exist. In our case a data set is represented by byte distribution.

As we can see in the Table 5.1 the value of the correlation computation while we compare the extracted code and the method component of the TargetApp is nearly equal to 1 (0.999802). If we compare A against the byte distribution of other CAP components (Constant Pool, reference Location, Descriptor components), the correlation computation result is lower. However, if we perform the same tests against other (totally different) applications (Java CardApp1 and Java CardApp2, 2nd and 3rd lines) the results do not really help. Indeed, according to the 2nd line of the table, the Constant Pool(0.755622), the Method Component (0.7452), and the Descriptor Component (0.714169), are all three similar to TargetApp's Method Component, which is obviously not true.

	Pearson correlation values			
	Cst. Pool	Method	Ref. Loc.	Desc.
TargetApp	a.669712	0.999802	0.224208	0.823772
Java CardApp1	0.755622	0.7452	0.270308	0.714169
Java CardApp2	0.652619	0.699482	0.180559	0.822216

Table 5.1: Pearson Correlation values comparison between dumped code and different CAP components

	Spearman (Rho) Ranked Correlation values			
	Cst. Pool	Method	Ref. Loc.	Desc.
TargetApp	0.3935	0.99762	0.38933	0.328617
Java CardApp1	0.431883	0.658014	0.4720988	0.382773
Java CardApp2	0.33328	0.65953	0.492259	0.174243

Table 5.2: Spearman Ranked Correlation values comparison between dumped code and different CAP components

Another interesting statistical tool is the Spearman's rank test. The main difference between Pearson and Spearman is regarding the evaluation of the relationship between the two data sets to compare. Pearson actually evaluates the linear relationship between two data sets, that is, it expects that one change in A is associated with a proportional change in B. However, for Spearman, it evaluates the monotonic relationship, that is, both A and B can change together but not necessarily at a constant rate.

From the Table 5.2 one can notice that Spearman Ranked correlation values seem to be more reliable. Any CAP Components that is not code is under the value 0.5, otherwise it is above.

In the same way as Spearman, Kendall's Tau (see Table 5.3) correlation coefficient uses statistical associations based on the ranks of the data. While Spearman can be thought of as the regular Pearson product moment correlation coefficient (except it is computed from ranks), on the other hand Kendall's Tau rather quantifies the difference between the percentage of concordant (ordered in the same way) and discordant (ordered differently) pairs among all possible pairwise events.

Now that it is possible to compare two samples of byte distribution, one can use these statistical tools for analysing the whole raw memory dump.

	Kendall's Tau values			
	Cst. Pool	Method	Ref. Loc.	Desc.
TargetApp	0.280127	0.9942593	0.34823	0.2814312
Java CardApp1	0.3777	0.52729	0.41179	0.30677
Java CardApp2	0.295751	0.547625	0.428279	0.14866

Table 5.3: Kendall's Tau values comparison between dumped code and different CAP components

5.2 Automated Java Card Application Code Detection

While the memory dump to reverse engineer is not too large, a trained eye can manually/statically locate the code. However, in the case where one has a bigger memory dump, an automated tool would be necessary to highlight the potential area where it appears likely to be code and not data

In the previous section we tried to characterize a Java Card applet's code in order to determine a kind of quantifier that would enable one to determine if a given set of extracted bytes looks more similar to a code than data. As we have now some statistical tools to rely on, we can use them for identifying code within a given memory dump, and extract the code.

5.2.1 The Running Correlation Coefficient Computation Trace (RCCT)

The idea is to compare two sets of data, namely, two sets of byte distribution. The first one comes from a reference Java Card applet. Let us note this reference set as S. The second set of byte distribution comes from a raw memory dump. While the memory dump is parsed, a window of size w is defined for analysing a single sample from the raw memory dump. Let S_i be a given sample where i identifies the sample among other ones. i varies from 0 to $W - \frac{W}{w}$, where W is the size of the raw memory dump. While this window is moved from the start of the raw memory dump to its end, every S_i is compared to S using a statistical tool noted (ST). The outcome from this computation is a correlation coefficient value c_i, that defines how S_i is similar to S. A set of c_i values will result in a correlation coefficient trace. For instance, the Fig. 5.13 depicts a trace that has been computed while computing $ST(S_i, S)$ all along a raw memory dump, where w has been chosen empirically $(= 128)$ and $W = 4KB$.

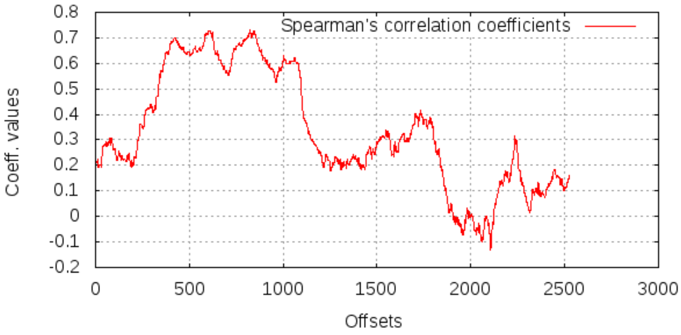


Figure 5.13: Running Correlation Coefficient Traces

As we can notice, the correlation coefficient does not even reach 1. This is normal because we do not compare exactly the same thing. However, there is an area between the offsets 470 and $\approx$ 1100, where the correlation coefficient is higher than 0.5. This gives the information that between those offsets, there is a set of bytes that have a distribution that approximate the distribution of the bytes that one can find within the Method Component of the reference Applet that has been chosen for the experiment. Further analysis can reveal which range of window' size can be chosen so the correlation coefficient is the nearest possible of 1. The Fig. 5.14 depicts a trace where each point represents the number of correlation coefficient that is over 0.9 according to a given window size (absciss). This enables to highlight the range of window size values that can be used for getting the highest correlation coefficient values possible.

Indeed, as one can notice in Fig. 5.14, the correlation coefficient value is quite high between

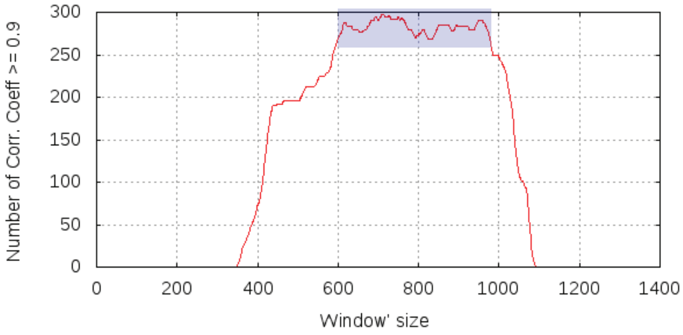


Figure 5.14: Number of Correlation Coefficient greater or equal to 0.9 according to the window' size over 2653 bytes

the offsets 600 and 800 (window' size). Hence, if one recomputes the correlation trace, one can notice that the area of interest can be refined (Fig. 5.15). However, if one refines it too much, the actual start and the end of the code may be missing. On the contrary, if the threshold is too low (e.g: $<$ 400), additional data can be comprised before and after the actual start and end of the code.

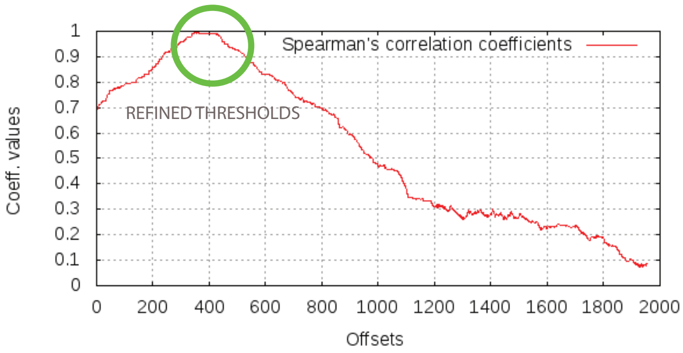


Figure 5.15: Correlation Coefficient computation with a window' size equals to 700

5.2.2 Data Extraction

A RCCT trace can be used for defining the area of interest to extract. A min/max threshold can be then used to accomplish that job. However, we decided to check visually how looks like a RCCT trace while it is plotted as an image instead of a graph. This would give another way to visually appreciate the variations.

To do so, we define the color of a pixel as being the absolute value of the correlation coefficient value that has been rounded up. we kept the gray-scale mode so it is more convenient to process the image. Therefore as a pixel is defined by its RGB values (Red, Green, Blue).

The resulting image is depicted by the figure Fig. 5.16.

What we can notice is that the lighter the colour is, the higher the correlation is. It clearly defines some more visible area . But we need a more accurate delimitation. The gray section on the rightmost is an ignored part due to the window size.

From the previous image (cf. figure Fig. 5.16) we need to create a “mask” that would enable me to extract from the original raw image the area that interests me.

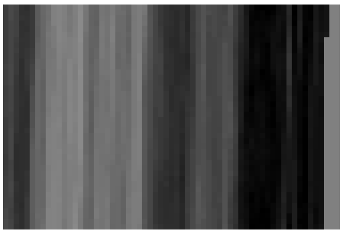


Figure 5.16: Running Correlation Coefficient plots

Let us say an image is represented as a multidimensional array of RGB values (pixels) such as:

image = [[R_1, G_1, B_1], ... , [R_n, G_n, B_n]]

To create my mask we define it such as, every pixel's value greater than the mean value of the flattened multidimensional array is set as "white" , otherwise it is set to "black".

The result is the figure Fig. 5.17.

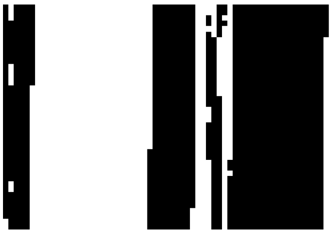


Figure 5.17: Mask processed from Fig. 5.16

Last but not least, we overlap the new mask over the memory dump and we only extract pixels that actually overlap pixels that correspond to the white color. All the transformations are summarized by the figure Fig. 5.18.

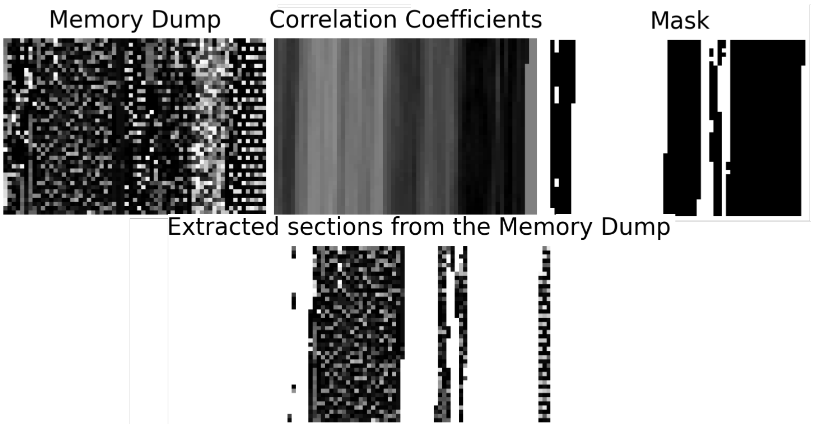


Figure 5.18: Image processing steps for extracting relevant blocks

In this example it is easy to decide/choose the right block to select. The small blocks on the left and the ones just after the big one cannot be code because an actual method component cannot be that small or too fragmented. The rightmost block corresponds to the last part that has been left due to the size of the window during the running correlation coefficient computation. It remains the biggest block.

Once extracted from the raw memory dump, it turns out that due to the sliding window's size, the beginning and the end of the method component is actually not properly delimited. It may remain some bytes before and after the extracted data that are not related to the code. They can be easily identified while we try do decompile the extracted data due to incoherences in the code's semantic.

5.3 Countermeasures

This technique has a strong reliability just because of one fact: the Method Component (meta-data + methods' body) and/or a set of grouped methods' body has a very specific byte distribution compared to other CAP components (or other non-related data or meta-data that surround the code). However, countermeasures already exists as addressed in [Gui12] [TBTL12].

5.3.1 Instructions Scrambling

The first countermeasure just consists in scrambling the Java Card instruction set. To do so, each instructions are XORed with a randomly generated byte.

$$\text{scrambled\_instruction} = \text{original\_instruction} \oplus \text{KEY}$$

Consequently, two devices can then have two different instruction sets. As the instruction set is unknown to the attacker, this is enough to circumvent the different analysis that have been proposed in this paper.

However, as it as been noted in [TBTL12] the XOR key can be revealed. First of all, if an attacker is able to change the control flow of the program to a *return* instruction or if he is able to identify its scrambled version, the XOR key can be easily broken by brute-forcing it. Indeed, the *return* instruction's value is $0x7A$. Its scrambled version can only take 256 possible values which is nothing in term of brute-forcing. Secondly, if a scrambled instruction is repeated too many times, this can give some hints to the attacker regarding the real instruction's value. If his assumptions turned to be true, then the same brute-force attack can be applied.

Hence another countermeasure has been proposed in [TBTL12] and is explained in the next section.

5.3.2 Improved Instructions Scrambling

To improve the above countermeasure, as explained in [TBTL12],in addition to the XOR key, a dynamic element has been added in order to circumvent the weakness that has been noted previously. The Java Program Counter is then included in the scrambling operation:

$$\text{scrambled\_instruction} = \text{original\_instruction} \oplus \text{KEY} \oplus \text{JPC}$$

In this case every scrambled instructions have different values.

5.4 Conclusion

This chapter described a new approach for identifying and extracting Java Card code from a raw memory dump. The aim of this work was to find another technique that would overcome the gathering of a wide range of different applications that is required for computing a proper Index of Coincidence (IC). As it has been described in [LBL+14], that IC value should characterize at best the Java Card set of instructions. Some countermeasures have already been proposed in the literature [Gui12], [TBTL12] and are efficient enough for circumventing such kind of analysis. However, in the case the right countermeasure is not properly set up, the new approach we propose enables to characterize the code embedded within a Java Card applet. The main characteristic of the code is its byte distribution. It has been shown that using some statistical tools enables to use this characteristic for distinguishing the code from data. It has also been shown that instead of characterizing the whole Java Card programming language, a single reference sample (a Java Card applet) suffices for the characterization and the identification steps. This technique thrives on the fact that the Method Component within a Java Card applet has a particular byte distribution that can be recognized amongst data that do not have any relation with the latter. This technique enables to speed up the reverse engineering process while one faces a large raw memory dump.

Part II

Security of Baseband Processors

Chapter 6

Attack Surface Analysis of Baseband Processors and Related Work

Contents

For several years, people had only interest in phone "unlocking", namely, finding a way to untie a given handset to mobile network operators that "lock" their devices, so their customer cannot use multiple operators on the same device. However, since 2009, researchers gradually pointed out security issues that reside within the baseband. On Android platform, the baseband is one of the few close-source software that run on the device. No public documentation is available (unless leaked). Having control over the baseband side allows an attacker to perform different attacks related to the "phone" part of a device. *e.g:* monitoring incoming/outgoing calls and the microphone, performing unnotified calls and message sending, etc.

One can take as an assumption that, the telephony stacks are historically old pieces of code, thus, it is likely that vulnerabilities can be found inside basebands.

In this part, we are going to provide the reader, basic knowledge about what is exactly a baseband (section 6.1.1). Furthermore, in order to have an overview of the attack surface, and the techniques that can be used for analysing the baseband, we present in 6.2, the related work around the security of basebands in mobile devices. One of Qualcomm's proprietary Real-Time Operating system has been particularly studied (section 6.4) in order to understand how was implemented the (U)SIM interface. Finally, from the analysis of the current literature on this subject, we end this chapter with a conclusion and a short discussion (section 6.6) regarding the current issue in this area or research.

6.1 Introduction to the Baseband Processor

6.1.1 Baseband Architecture Overview

A baseband processor is an integrated circuit that is mainly used in mobile devices for processing all the radio functions (all functions that require an antenna). This may (or not) include WiFi, and/or Bluetooth. Basically, it is composed of two sections: the analog and digital baseband processors. They are illustrated by the block diagram in 6.1.

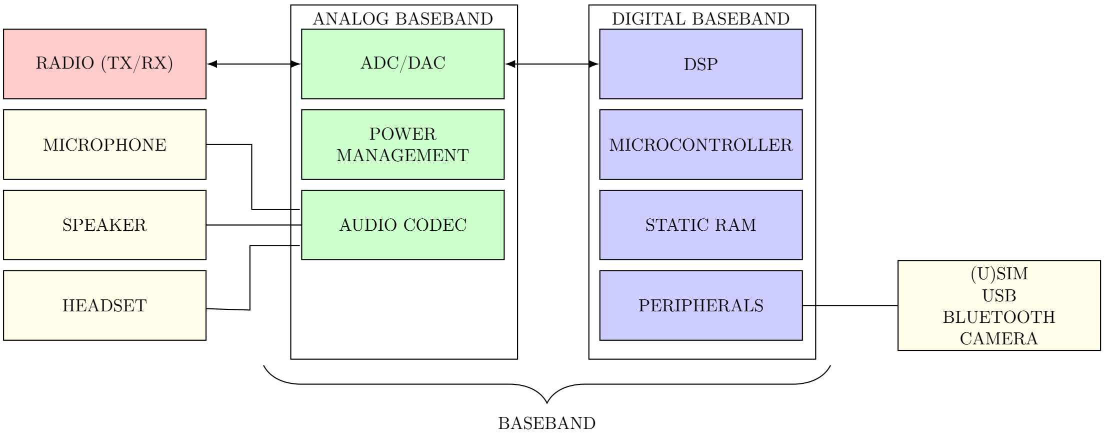


Figure 6.1: Block diagram of the baseband main components. ADC = Analog to Digital Converter, DAC = Digital to Analog Converter

Digital Baseband Processor – This section is the main "brain" of the baseband processor, and it is responsible of running applications, handling data input and output signal, managing peripherals (SIM, camera, bluetooth, etc.) and its dedicated memories (RAM, ROM, Flash)

Analog Baseband Processor – This part is responsible of converting and processing analog to digital signals and digital to analog signals.

Universal Subscriber Identity Module (USIM) – Although it is not part of the baseband, it is important to note that the USIM is connected to the baseband processor, which is a smart card that holds in its read-only memory (ROM), an operating system (e.g: Java Card OS, see chapters 2 and 3), and the security algorithms for authentication and key generation. In its EEPROM is stored the subscriber's identity information: the International Mobile Subscriber Identity (IMSI) and the Temporary Mobile Subscriber Identity (TMSI).

6.1.2 Communication between the Baseband and the Application Processors

Smartphones' SoC has a multi-processors architecture, but as we have seen, the main ones are the baseband and the application processors. The baseband processor and the application processor can communicate and exchange information. Indeed, despite the fact that both processors operate independently, performing specialized tasks, somehow they have to exchange information (e.g: voice, any multimedia data). Both can communicate with each other through a serial line (or multiple ones) such as for instance, UART (Universal Asynchronous Receiver Transmitter), USB (Universal Serial Bus), SPI (Serial Peripheral Interface), HSI (High-speed Synchronous Serial Interface), HSIC. (High-Speed Inter-Chip). The serial interface that is used highly depends on the bandwidth requirement of the cellular technology(ies) that is/are used. Dual-Port RAM (DPRAM), a type of random-access memory, can also be used as a shared memory, or a combination of interfaces such as c2c (Chip-to-Chip Link).

The interface that is made between both processors is generally referred as IPC (Inter-Processor Communication). However, one should note that not all devices can be designed this way, as we have already seen that some SoCs may have a single chip package that hosts both the application processor and the baseband.

6.1.3 The Baseband Operating System

The baseband Operating System (OS) is also known as the firmware of the baseband processor. Such kind of OS is different than regular OSes one can have on regular computers. Indeed, in the embedded world, we have Real-Time Operating Systems (RTOS), and common OSes that we can find on any laptop, desktop computers, are called General Purpose Operating System (GPOS). The main basic difference that characterizes them, and that is the most important, is the way they schedule the tasks they must execute. On a GPOS-based system, a user can run multiple tasks, that are not time critical, while, RTOS-based systems must ensure that the processor executes a single task for a period of time. The baseband OS is mainly responsible for cellular capabilities, and sometimes the baseband may include a Digital Signal Processor (DSP) which is a piece of optimized microprocessor for handling the operational needs of digital signal processing, *i.e*, analog signals filtering/compression, etc.

Significant baseband manufacturers include MediaTek, Broadcom, Icera, Intel (former Infineon wireless division), ST-Ericsson. However, nowadays, the one that mostly leads the market is Qualcomm, as they are one of the largest sellers of cellular chipsets in the world, and moreover, they hold many of the patents on CDMA. Some examples of popular RTOS are Nucleuse RTOS (iPhone 3G/3GS/iPad), REX/Blast/QuRT for devices with Qualcomm's baseband chips. ThreadX (iPhone 4).

6.1.4 Communicating with a baseband

The baseband processor and the application processor use different mechanisms for communicating and sharing data.

6.1.4.1 The Radio Layer Interface

All the applications, related to the telephony, like SMS/MMS, Dialer, GPRS and so on, will require an interface for communicating with the baseband processor. On Android, that interface is called, the Radio Interface Layer (RIL), and on iPhone/iPad/iPod Touch devices, it is called CommCenter. Basically, such interface is a daemon that is run in the background, and the communication with the baseband is made through a serial device in the `/dev` filesystem. The Figure 6.2 illustrates the interaction between Android applications, and the baseband.

6.1.4.2 Hayes commands for communicating with a baseband

Hayes commands are also known as AT commands. They are used for controlling the features provided by the baseband (dial up, hang up, set parameters etc.). It is the most common command language for modems designed in 1981, and it is still used on modern baseband processors

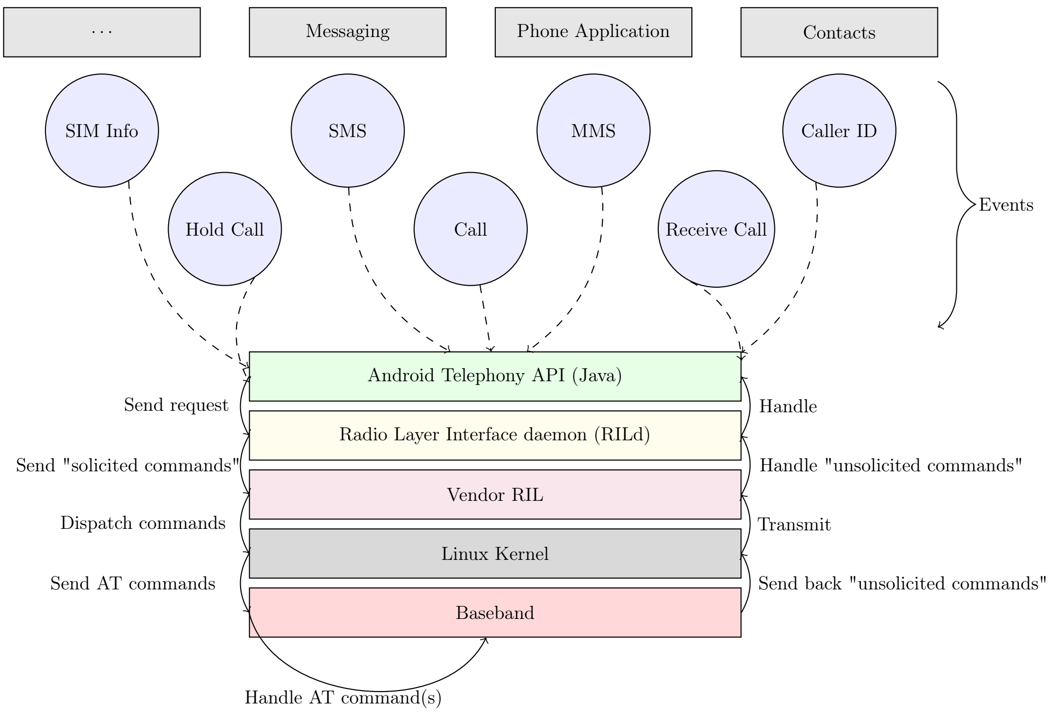


Figure 6.2: Android telephony architecture overview

found in smartphones today. Standards like GSM and 3GPP use this type of commands as a standard way to communicate with the baseband. Having the ability to issue these commands allows, for instance, to issue calls, send SMSs, alter the PIN, and more. The listing 6.1 depicts an example of AT commands, where the PIN code "0000" is set using the command `AT+CPIN=...`, and the PIN verification is requested through the command `AT+CPIN?`, which is answered with the response `+CPIN:READY`.

Since the very first versions of the AT command set, additional public or proprietary command sets have been added, therefore, AT command implementation and support is fragmented across the industry.

Listing 6.1 Example of AT commands: sending PIN for validation

```
AT+CPIN="0000"
...
AT+CPIN?
+CPIN:READY
```

Certain values, or parameters of the baseband, are stored in memory locations called, `S-registers`. For instance, the very first register `S0`, defines the number of rings the baseband has to wait before auto-answering. If it is set to 0, then the auto-answer feature will not be activated.

GSM/3G modems typically support the ETSI GSM 07.07/3GPP TS 27.007 AT command set extensions, however, vendors can implement them differently, and furthermore, additionnal proprietary extended commands can also be added.

6.2 Related Work

6.2.1 Local Attacks

Up to now, the main interest that drove people to reverse engineer mobile devices firmwares has been the "unlock scene". It mainly consists of removing the restriction of a given handset on using another SIM card of another network operator. Indeed, in addition to the user's phone number and other user identification data, a SIM card may also contain various unique data. For instance, it has a unique serial number called the integrated circuit card identifier (ICCID), and it also stores an international mobile subscriber identity (IMSI) that is used for identifying the SIM card over the network of a given mobile network operator (MNO).

As many mobile devices are subsidized by the network providers, in order to sell cheaper devices, they may be designed to only accept certain network and consequently, certain SIM cards. A network/SIM check is then implemented in the baseband for preventing the user to access another network operator.Consequently, by definition, a software unlock is the process of modifying a given mobile device such that the baseband will accept the SIM card of any other MNO.

6.2.1.1 Software unlock

Most of unlocks are made through direct modification of specific values within the baseband firmware, or by modifying a specific system directory that is kept "hidden", like for instance, on Android devices: the EFS (Encrypting File System) folder. It is stored in the internal file system, and mainly contains sensitive information like the device's unique IMEI (nternational Mobile Station Equipment Identity) number, the Wireless MAC address for the mobile's radios, as well as the baseband version, and more. On iPhone devices, the most known unlock technique that has been used consisted in exploiting the AT command parser. Most of the attacks rely on an unsigned code injection vulnerability that would enable one to execute arbitrary code in the baseband. Basically, memory corruptions (stack overflow or heap overflow) vulnerabilities could be exploited within the latter, and could enable one to inject a payload code.

In 2011, Miras presented in [Mir11], `yellowsn0w`, the first unlock that was initially discovered by the iPhone Dev Team [13]. The exploit basically consisted in injecting the AT command `AT+STKPROF` that embeds a payload. The vulnerability resides in the AT command parser that did not properly check the size of that particular command. This resulted in a stack-based buffer overflow that enabled to inject and execute code on the iPhone 3G baseband.

6.2.1.2 Hardware unlock

While software "hacks" do not work, generally, hardware "hacks" are performed as an alternative way. As an example, in 2007, one of the most famous carrier unlock has been performed by George Hotz (geohot). He succeeded to uncover the JTAG interface of the iPhone 2G's baseband, namely the S-Gold 2. In this case, he particularly targeted the baseband processor, as its bootloader has an interactive mode which can be used to download a firmware. The JTAG interface enabled him to extract the content of the ROM memory that contains the bootrom, namely, the very first code that is run by the processor while it is powered up. The code that is loaded and executed after the bootrom, is normally signed and verified before executing it. However, by studying the code of the bootrom, he found out that, unsigned code could be loaded and executed by

[13] http://wikee.iphwn.org/

the bootrom of the baseband, by tricking the latter with and hardware trick, as depicted by the Figure 6.3.

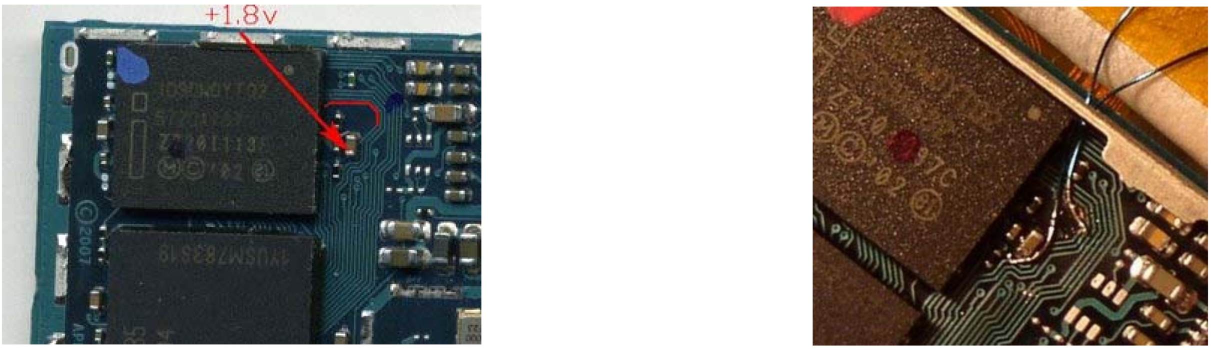


Figure 6.3: On the left, test points (red path and arrow) near the baseband processor where thin wires had to be soldered to pull the capacitor up (as depicted by the picture on the right), so the chip is tricked and will consider that the flash is erased. As a result, it will enter into a "recovery mode", where unsigned code can be downloaded without any verification. (images source: http://geohot.blogspot.fr/)

As he could bypass the signature verification of the code, he succeeded in downloading onto the NOR flash memory, a modified baseband firmware with the unlock patched in.

6.2.1.3 Bootloader unlock

While the purpose of this technique does not have the baseband as the main target, it is worth to note that the latter has been exploited to reach the actual target: the application processor's bootloader, *i.e*, a program that is used during the boot process to load - and in some cases, verify - other programs (e.g: another bootloader, one or several OSes).

NAND unlocking This technique have been most used for some HTC devices which implemented a protection to prevent people from installing custom software such as custom ROMs, custom kernels. The NAND flash memory is a type of non-volatile storage technology that does not require power to retain data. On some HTC devices, NAND lock prevents from writing to system, kernel or recovery partitions. A flag in the baseband's NVRAM is checked by the bootloader `HBOOT`, and it defines if the protection is set or not. Generally, rooting the device permanently will require to install on the file system, some binaries, that will grant the user or any processes, root privileges, However, with the NAND protection, any writes on the file system will be lost after a reboot. To overcome this restriction, either the NAND lock must be removed by modifying the bootloader and/or the baseband NVRAM.

6.2.2 Attack Through USB Connection

We have seen that AT commands was the most common language any modem could talk. Some devices may allow USB connection to issue such commands. Consequently, it gives the ability to directly communicate with the baseband through a USB channel, thus, provides another vector attack.

In 2014, Pereira *et al.* discribed in [PCB14] vulnerabilities they discovered through this attack vector. Their work aimed at demonstrating that it is possible to use proprietary AT commands to perform malicious operations, such as silently flashing a new boot image and installing persistent malicious applications (e.g: AndroRAT). Kies is a Samsung software that is used to interact with a Samsung device using (usually) USB connection. Kies uses standard AT command set, plus an extended proprietary AT command set to perform various operations, such as for instance, to update the firmware on the device. The authors eavesdropped the USB communication that is performed between Kies and the device in order to gain knowledge on how the interaction is performed between both.

They discovered that proprietary commands were handled by processes at the OS level, and they were not forwaded to the baseband, but instead, interpreted and executed within the application processor.

While their paper does not describe an attack against the baseband, it is still interesting to notice that USB connection can be used for issuing AT commands.

6.2.3 Remote Attacks

6.2.3.1 Basics on Cellular Network

In a cellular network, so-called "cells" contain each a Base Station (BS) and are grouped in "cluster". The BS houses the transmitter/receiver antennae and the switching equipment. All BS are connected to a central point, called the Mobile Switching Office (MSO), either by fixed lines or microwave. The MSO is generally connected to the Public Switched Telephone Network) PSTN which is operated by national, regional or local telephony operators and provides infrastructure and services for public communication.

Cellular technology allows the "hand-off" of subscribers from one cell to another one as they travel around. This is the key feature which enables the mobility of users. A computer constantly tracks the subscribers within a cell, and the computer automatically hands-off any call to assign it to a new channel in a different cell whether the user has reached the border of a cell.

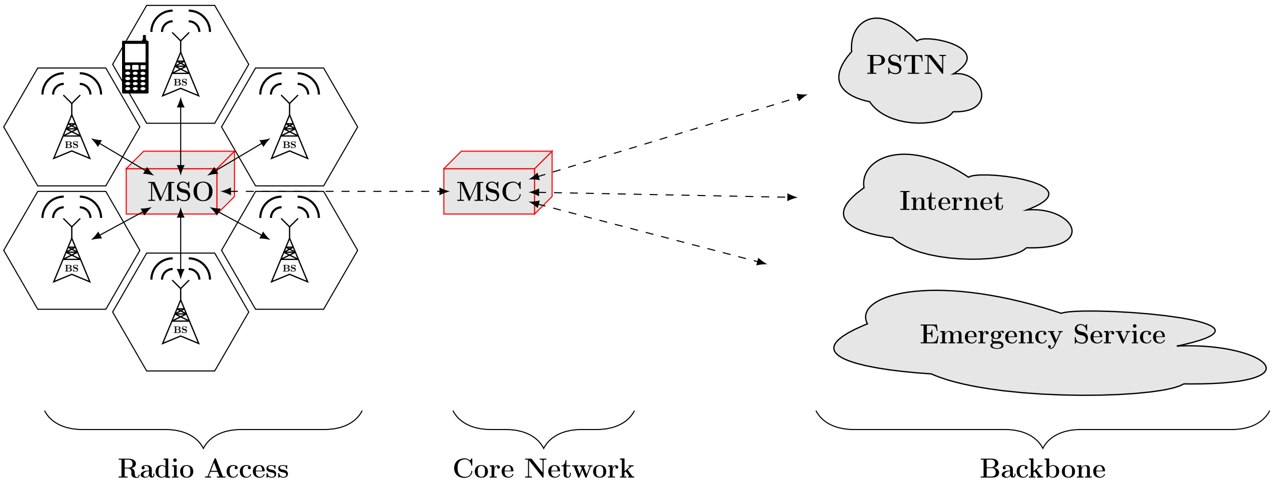


Figure 6.4: Rough overview of a cellular system architecture

6.2.3.2 Overview of GSM Security Features

Cryptographic Algorithms. Three algorithms have been specified to provide security services in the GSM standard:

- A3 is used for authentication in a GSM network before a subscriber can use any service provided by the network. It takes as an input a 128-bit `RAND` random challenge, and a K_i 128-bit private key. As a result, it produces a 32-bit `SRES` Signed RESponse.

- A5 for encryption

- A8 for the generation of a cipher key. It takes the same input as A3 and produces a 64-bit cipher key K_c which is used by A5.

Authentication. As illustrated by the Figure 6.5, it allows MNO to verify the identity of the subscriber. If a MS (Mobile Station) wants to access a network, a challenge-response protocol is used to authenticate the subscriber. After the MS sent its security capabilities, it is induced to transmit its IMSI to the network. The Authentication Center (AuC) is in charge of sending a random value `RAND` to the Subscriber Identification Module (SIM) card of the user's MS. The MS sends back a signed response `SRES` which is generated by the SIM. The Mobile Switching Center (MSC) compares both values, and accepts (authenticates) or not the subscriber.

As a security measure, the network can assign a Temporary Mobile Subscriber Identity (TMSI) to the MS to reduce the frequent transmission of the IMSI. This normaly helps to avoid being identified or tracked.

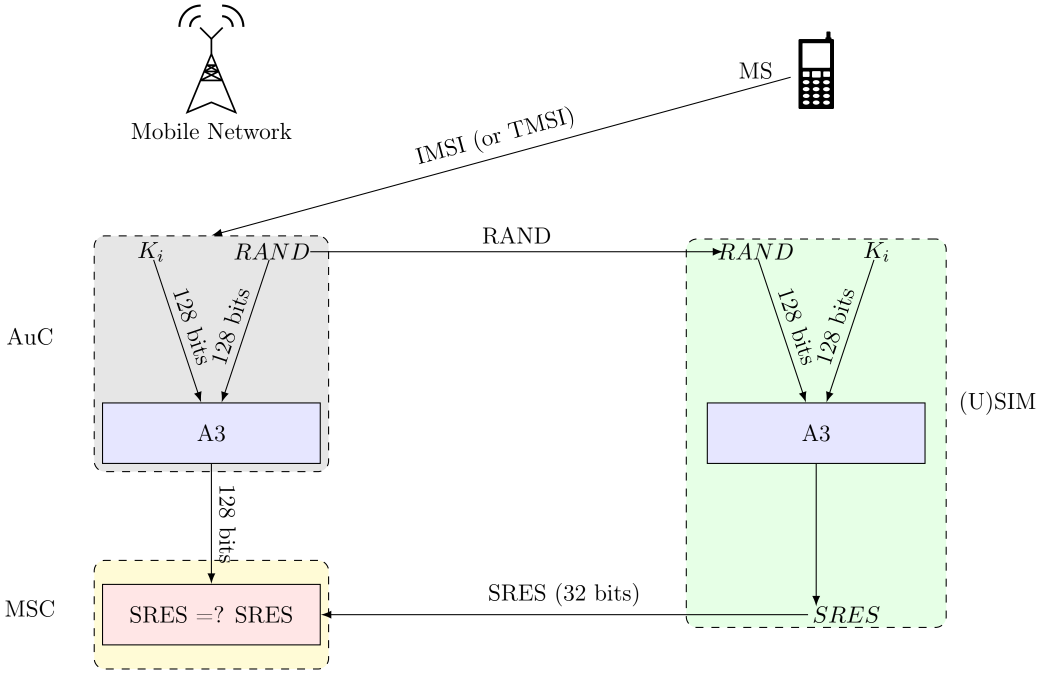


Figure 6.5: Authentication in GSM with K_i an individual subscriber authentication key and $SRES$ a signed response

Confidentiality/Encryption. Protects voice, data and sensitive signalling information against eavesdropping on the radio interface. After authentication, the Base Transceiver Station (BTS) and MS apply encryption to all the user-related data using a cipher key K_c, which is generated using an individual key K_i and a random value by applying the `A8` cryptographic algorithm. K_c

is never transmitted Over The Air (OTA) interface and is calculated by both the SIM and the network which is based on the random value `RAND`. The confidentiality only exists between the MS and the BTS, and they can now encrypt/decrypt data using the algorithm A5 and K_c.

This is illustrated by the Figure 6.6.

We have previously seen that the MS has to specify to the network, which encryption algorithms it supports. In return, the BTS chooses one of these and informs the MS which one is going to be used during the session.

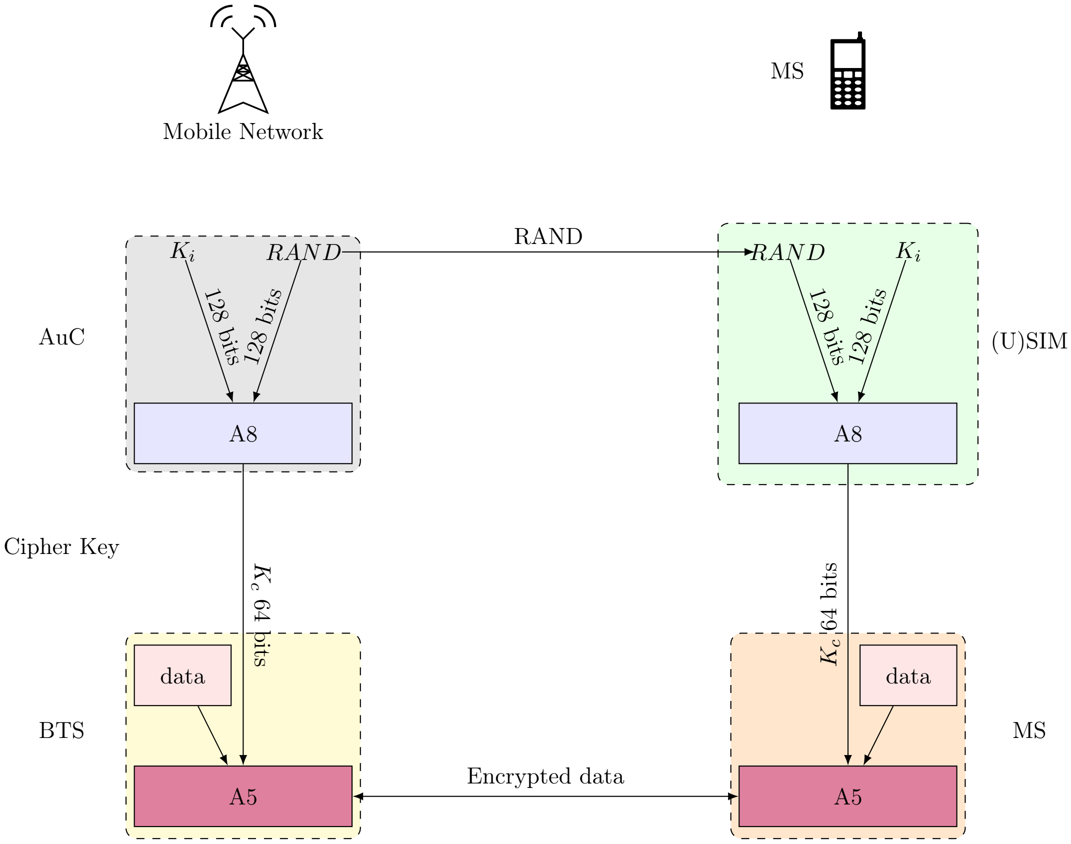


Figure 6.6: Key Generation and Encryption

Anonymity This prevents from being tracked or identifying calls made to or from a user. We have seen that all data is encrypted, which consequently ensures anonymity. Any user identifier is never used OTA, however, we have seen that a Temporary Mobile Subscriber Identifier (TMSI) is rather transmitted and is assigned by the Visitor Location Register (VLR) after each location update. The VLR is the entity which contains temporary data that exists for only as long as the subscriber is active in the particular area covered by it.

6.2.3.3 Remote attack through the GSM/UMTS network

Nowadays, a particular "buzzword" in the area of remote GSM network eavesdropping is "IMSI Catching". It was one of the first practical attacks on GSM which led to the development of devices called IMSI Catchers. The main objective of such device is to gather IMSIs within a given geographical area. It generally acts as a fake base station that provides a stronger signal which will induce any mobile devices in the area to connect to it. The IMSI catcher will then command each device to identify itself, thus, the latter sends out its IMSI. Two different types of eavesdropping can be identified:

Passive: either it is a passive interception, where, the IMSI catcher just silently sits between the mobile station and the base station and passively collects IMSIs. Advanced passive IMSI catchers can also demodulate signals and decipher GSM cryptographic algorithms (e.g: generally A5.0, A5.1, A5.2), intercept SMS and/or MMS. They operate by tuning into the base station and receiving the uplink signal from the mobile station, and the downlink signal from the base station. An uplink signal is the information being sent from the mobile station to the base station, namely, the content of call and SMS/MMS messages. A downlink signal is the information being sent from the base station to the mobile station: the response to the mobile station.

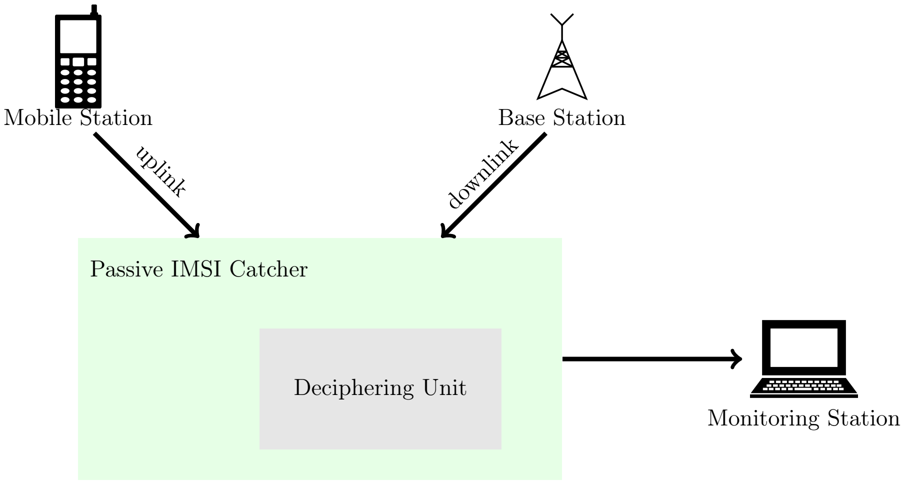


Figure 6.7: Illustration of a passive MiTM performed by a IMSI Catcher

Active: an active interception rather focuses on forcing the mobile station to use the IMSI Catcher as a legit GSM base station. It then determines the device's IMSI and can then attempt to disable or degrade the GSM encryption mode that the mobile station is supposed to use, thus, monitoring the communication is made possible. It can also intercept SMS messages or send fake ones to and from the mobile station.

An example of attack that uses such kind of fake base station is presented by Weinmann in [Wei12a]. This attack is further explained in the section 6.3.2. As a summary, the attack mainly takes benefit of a rogue base station for remotely sending particular crafted SMS messages, with the aim of remotely executing code on the baseband processor. However, the attacker must be within a certain proximity to the target device. Indeed, regular base stations are generally far away from the target device, hence, the latter has to detect the one with the strongest signal to initiate the connection and the authentication. Consequently, an attacker will only need to be close enough to the victim to appear as the one who has the strongest signal. While the victim associates with the rogue base station, the attacker will be able to perform GSM MiTM (Man in The Middle) attack on the victim's traffic, or sending arbitrary malicious traffic.

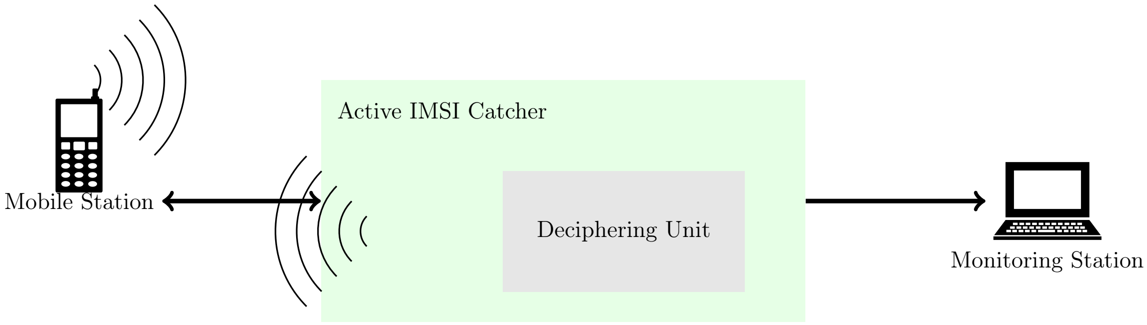


Figure 6.8: Illustration of an active IMSI Catcher

6.2.3.4 Remote attack through the CDMA/3G and 4G networks

As far as we know, the only actual CDMA and 4G attacks that have been performed in the wild, is the one that coderman witnessed at DefCon conference in 2011, and reported in [cod11]. According to the author, as he attempted to monitor the attacks, they consisted in simultaneous MiTM attacks over CDMA and 4G connections with the aim of remotely exploiting Android mobile devices, with *ring0* local exploits, *i.e*, exploits with the highest privileges. In 2014, coderman wrote a write-up in [cod], where he gives further details about the attacks. As a summary , the primary steps of the attacks consisted in the followings :

1. **MiTM between the target's device and its carrier Base Transceiver Station (BTS):** As I previously said in 6.2.3.3, a MiTM attack can be performed between the target device and the carrier BTS as long as the signal of the rogue base station is stronger than the BTS. Consequently, the target device will try to connect to the latter.

2. **Set up a PRL (Preferred Roaming List) Zero[14] attack:** the aim of this step was to silently push PRL updates to the devices in order to reduce the interferences produced by legit carrier bands. Basically, PRL is a database within the device, provided by the carrier, and is used in devices with CDMA capabilities. It is used to filter out the correct tower(s) the device has to connect to, while it is in a given area. What is important to notice regarding PRL is that, it enables the device to connect to the (supposedly) "correct" tower, and not necessarily the one with the strongest signal.

As a consequence the subscriber voice and data traffic were under threat, and led to multiple attacks and infections. As an example, the attackers succeeded in continually invoking the Google Voice Search non-privileged application, for getting an open access to the device's microphone.

6.2.3.5 Remote Attack against Localisation-Based Services

Another possible attack path was presented by Weinmann in [Wei12b] and takes advantage of security issues within the Secure User Plane Location (SUPL) protocol that is used to gather A-GPS (Assisted GPS) information from a server, and potentially, in the IS-801 messages parsing. This last potential vulnerability was not clearly defined in the author's presentation.

[14] While the victim leaves the broadcast area under attack, the next legitimate carrier PRL update will always be greater than zero, thus replace the malicious PRL list

A-GPS A dedicated chip is used for ensuring the GPS feature. The GPS chip needs to acquire the signal of at least three satellites, and this can take time, whether or not the chip is aware of the identities of the satellites, and their orbits. Indeed, traditional GPS receivers work best in open areas that offer an unobstructed line of sight to the GPS satellites. In cases where GPS signals are weak, such as for instance inside an office building, the chip may encounter difficulties to find signals. In order to speed things up, a A-GPS server can be used over internet to gather the necessary information regarding available satellites.

SUPL The SUPL protocol is a standardized method that was developed by the Open Mobile Alliance (OMA) to gather A-GPS information. It is an IP technology that was developed to support Location-Based Services (LBS) for wireless communications. Basic SUPL architecture is composed of a server equipment stack, also called the Home Location Server (H-SLP) and a SUPL-enabled wireless device, denoted as the SET. The secure communication is performed via the Tranport Layer Security (TLS) protocol. SUPL 3.0 , is the latest version from OMA [OMA], and it defines a set of protocols for transporting existing messages as defined by the wireless standards: GSM (RRLP, Radio Resource Location Services Protocol, [ETSc]), WCDMA (RRC, Radio Resource Control, [ETSa]), CDMA (TIA-801, [tia]) and LTE (LPP, LTE Positionning Protocol, [ETSb]). The SLP and the SET communicate through ULP (UserPlane Location Protocol), which is a binary protocol that supports 8 basic SUPL messages. These messages are used to initiate a SUPL session, exchange positionning and authentication data, and end a SUPL session.

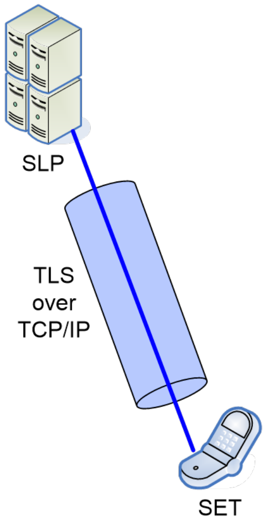


Figure 6.9: Trust relationship between a terminal (SET) and its Home Location Server (SLP)

gpsOne[15] is a dedicated chip manufactured by Qualcomm to ensure GPS functionalities, and is integrated within its Mobile Station Modem (MSM) chipsets since 1999 [Qua11].

[15] https://www.qualcomm.com/news/releases/1999/10/11/qualcomm-cdma-technologies-announces-development-gpsone-global-position

To perform the attack, Weinmann set up a rogue base transceiver station which he used to send malformed SUPL messages over the air to the target devices. Weinmann highlighted some bugs he found within the baseband, regarding the implementation of the SUPL messages parser. The `SUPL INIT` message is used by the SLP to initiate a SUPL session with the SET. That message can be sent either via WAP Push using the Push Access Protocol (PAP) through a Push Proxy Gateway (PPG). While the baseband is processing a given WAP Push SUPL message, it appears that a buffer overflow could be triggered. However, according to the author, the bug was difficult to exploit.

In spite of this, his work highlights the largeness of the attack surface which is reflected by the baseband ecosystem.

6.2.3.6 Global Remote Code Execution Through OTA Device Management Protocols

It is important for device manufacturers and software vendors to have the ability to remotely perform Over-The-Air (OTA) updates. The main advantage of this is the cost reduction of updating the software, as there is no need to recall all the production devices on the field. It includes the ability to remotely configure a single device, an entire fleet of mobile devices or an arbitrary set of mobile devices. The term that describes best this is Mobile Device Management (MDM).

The OMA specified a platform-independent Device Management (DM) protocol called OMA-DM, which was originally developed by The SyncML Initiative Ltd. OMA-DM protocol has been implemented in various devices connected to internet: smartphones, M2M (Machine to Machine) devices, or more generally, IoT (Internet-of-things) devices, laptops, cars, and so on.

Depending on the features provided by the remote control software, having hands on them may provide a powerful (limited) remote control over a given device. Being able to exploit the vulnerabilities inside them may give a full remote control over the device.

In 2014, Solnik and Blanchou presented at BlackHat US their work related to the evaluation of OMA-DM client implementations from the company RedBend, installed on a very wide range of connected devices. They discovered that OMA-DM clients could be either run directly on the baseband processor, or at the OS level, thus on the application processor. While it is run at the OS level, it still has a privileged interface to the baseband. Aside from finding multiple vulnerabilities (buffer overflows, heap corruption, integer overflows, etc.) in the code of the OMA-DM clients, as a Proof-of-Concept (PoC) and for demonstrating the security risks of these vulnerabilities inside such powerful remote control software, they succeeded in exploiting them to perform a remote code execution. For instance, through exploitation chains, they performed a remote iOS Jailbreak, and a remote Android lock screen bypass.

These exploitations were run on the application processor, however, if OMA-DM clients were running on the baseband side, a vulnerable OMA-DM protocol implementation may result to the same consequences, *i.e*, a full access control over the baseband.

6.3 Finding Bugs in a Baseband Processor

6.3.1 Fuzzing the baseband

Fuzzing is a technique for testing software. The main idea is to target an entry point, identify its inputs, automatically generate inputs, and inject them. The bahaviour of the software is then monitored. According to the result(s), one can interpret them and infers some possible security vulnerabilities.

For testing the baseband, the attack vector that has been the most covered in the literature has been SMS injections [MM09a, MM09b, vdBHT14, MGS11, GF11]. The main reason for this is that, first of all, there is no need for explicit user interaction. Secondly, only a phone number is requierd for launching an attack. And most of all, as far as I know, SMS is not firewalled by the current baseband stacks.

As an example, Mulliner *et al.* [MM09b] implemented a "Man In The Middle" (MiTM) technique, for locally injecting SMS. On Android device, the MiTM consists of the followings (illustrated by the Figure 6.10):

1. renaming the pseudo-terminal master (*/dev/smd0*) to */dev/smd0real*. The pseudo-terminal master is responsible of transmitting its input to the slave pseudo-terminal (*/dev/pty/0*),

2. creating a fake pseudo-terminal master (*/dev/smd0*) that will play the role of proxy. As the RILd process will always talk to */dev/smd0*, this tricks enable to intercept data from RILd,

3. running a process with two threads that perform the actual MiTM. The first one noted `Thread1` is responsible of exchanging with the hardware driver through */dev/smd0Real*, and the second `Thread2` manages the communication with RILd, and logs all communication within a log file.

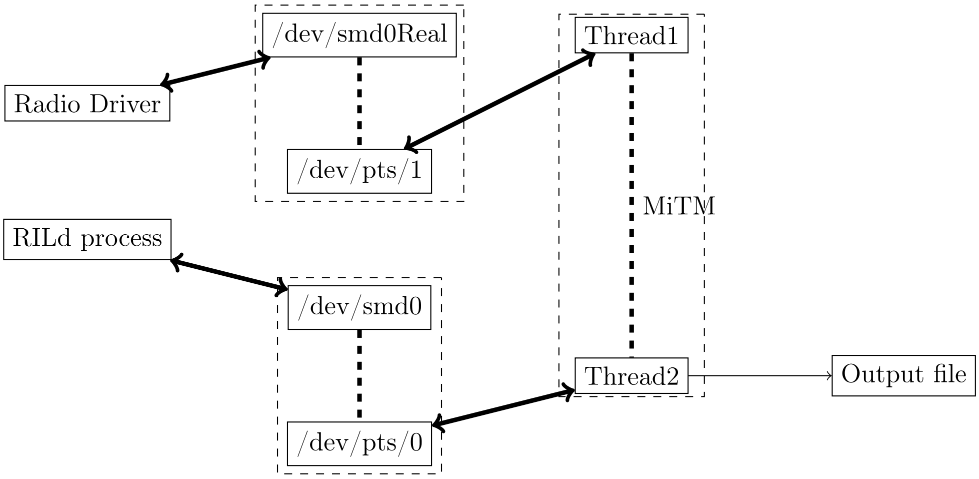


Figure 6.10: MiTM set up for SMS interception + attack vector for reaching the baseband through crafted SMS messages

Fuzzing implies crafting specific SMS messages. According to the specifications [3GPb] and [3GPc], section `9.3.2.10`, SMS messages can be encoded differently. It basically enables to send short 7-bit, 8-bit or 16-bit encoded messages according to the language of the content. Its format enables to wrap much more than a simple text as it also enables to embed multimedia files such as audio, pictures, and so on.

However, one can note six basic types of SMS data format according to the delivery direction, *i.e*, either from the Service Center (SC) to the Mobile Station (MS) or the opposite (see Table 6.1).

`SMS-DELIVER`	$SC \rightarrow MS$
`SMS-DELIVER-REPORT`	$SC \leftarrow MS$
`SMS-SUBMIT`	$MS \rightarrow SC$
`SMS-SUBMIT-REPORT`	$MS \leftarrow SC$
`SMS-COMMAND`	$MS \rightarrow SC$
`SMS-STATUS-REQUEST`	$MS \rightarrow SC$

Table 6.1: Basic types of SMS data format according to the delivery direction

In [MM09b] Mulliner *et al.* mainly focused on the `SMS_DELIVER` format, that is used for conveying a SMS from the Short Message Service Center (SMSC) to the mobile device.

At that time, the main flaw they found within the tested Android device was, `array index out of bounds`. Exploiting this flaw, they succeeded through specifically crafted SMSs, to trigger a Denial of Service (DoS) that led the device to disconnect from the network. Furthermore, the authors have shown that an aggressive DoS, *i.e*, sending a huge number of ill-formed SMSs, could block the user from getting connected to his network, for several hours.

However, there is still a question that remains: does such kind of flaw enable to perform a remote code execution? It is not clearly defined in their paper if it is.

Weinmann investigated further in [Wei12a], and instead of using SMS messages as attack vector, he took a more straightforward approach by directly injecting specifically crafted AT commands. His approach is described in the next section.

6.3.2 Source Code Review and/or Static Analysis of the Baseband Firmware's Binary

This approach for analysing a baseband firmware is the cheapest one, as it only requires to analyse the source code of the firmware (if available: leaked or NDA-protected), or by performing a static analysis of the firmware's binary. The binary of a firmware can either be directly extracted from the device, or from the firmware update that is provided by, for example, phone operators, or directly from the website of the device's manufacturer. Generally, those firmware updates are archives that contain the binary form of the baseband's firmware. Sometimes, that binary can also directly be found within the file system of the device.

As an example, we are going too specifically explain Weinmann's work in [Wei12a] that used both approaches for performing a vulnerability analysis of some baseband firmwares. He had access to the leaked source code of an old GSM baseband, for getting an overview on a real baseband firmware, and then he decided to analyse a specific part of the firmwares' binary.

In his paper [Wei12a], Weinmann demonstrates the risk of remotely exploitable memory corruptions in cellular baseband stacks. He shows that vulnerabilities can be exploited and triggered over the air interface using a rogue GSM base station (see diagram 6.11).

The author puts in evidence that, while all the public software attacks on smartphones are performed on the application processor, only a few tried to actively focus on the baseband processor. He uses memory corruptions for injecting and executing arbitrary code on the baseband processor. The author attempted to fuzz the baseband by sending different formats of AT commands. According to his results, he pointed out that, fuzzing remains inefficient if the outcomes cannot be properly interpreted. Indeed, a lot of crashes occurred, and despite the various logs that could be collected, he highlighted the difficulty to interpret and exploit them.

Consequently, he rather started to perform a code review of the source code of the Vitelcom

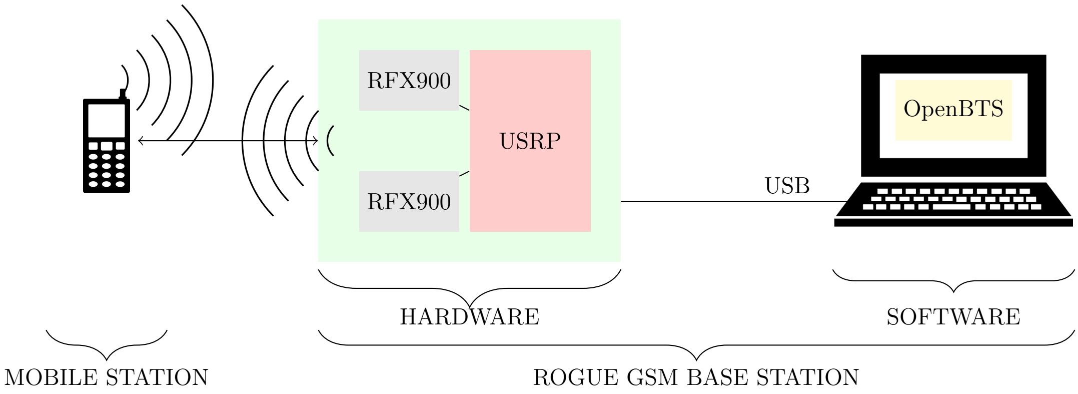


Figure 6.11: Test setup for a rogue GSM base station composed of two parts: (1) the hardware with a USRP (Universal Software Radio Peripheral) for baseband processing of signals, and two transceiver RFX900 daughter boards, and (2) the software with OpenBTS (Open Base Transceiver Station) which is a software-based GSM access point.

TSM30, which was leaked on the website, Sourceforge. It has been removed since then, however, it allowed the author to obtain a general understanding of the structure of the baseband firmware.

He decided to target an Apple iPhone 4 with a Comneon-base baseband on an Intel chip, and an HTC dream that rather uses a Qualcomm-based baseband and chip. For having hands on the firmware, they took the firmware updates, that are provided either by the device's vendor, or from the website of the carrier. These updates may also be called Firmware Over-The-Air (FOTA), and they generally allow manufacturers to remotely install new software updates, bug fixes, features and services on a given device. They also can contain a full baseband binary blob, that can be extracted.

Despite the fact that, those three baseband firmwares that we have mentioned have been developed by different companies, the author highlighted two very important assumptions:

1. the knowledge that can be acquired by studying a baseband firmware, even an older one, is still worth, as no documentation is publicly available,

2. fuzzing is a quite crude way for assessing the security of a given system, furthermore, it does not give a full understanding of the triggered bug, therefore, manual static analysis paired with a dynamic analysis (debugging), if possible, remains the best approach

However, **(2)** may be a very long process, according to the size of the binary under analysis.

During the reverse engineering process, the author had a straightforward approach as he mainly looked for functions that perform memory block transfers, such as for instance, `memcpy()`, `memmove()` or `bcopy()`, and he only focused on a specific part of the firmware. Finding functions that make use of the latter and that do not properly check the length of the data to be copied, would open the door to buffer overflow attacks.

As a result of his reverse engineering, the author mainly found out three different types of vulnerabilities:

Heap/Stack overflows mainly due to insufficient length checks while allocating memory, either in the heap area or in the stack. The Figure 6.12 illustrates an example of possible exploitation of such kind of flaw. The case **(1)**, represents the state of the stack before to call the program `B`. In **(2)**, `B` is executed by `C`, and the space it needs is allocated on the stack, and the **`return address`** is set for being able to return to the program `C` (the caller) once `B`'s execution is finished. Finally, in **(3)**, we illustrate the case where `B` is owned by an attacker, and perform a stack overflow by means of a buffer that he controls, for overwritting the **`return address`** value. This would enable him to modify the execution flow.

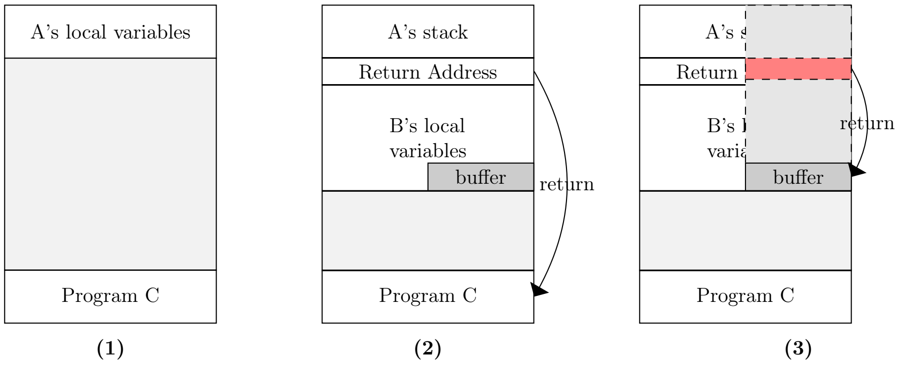


Figure 6.12: Illustration of a stack overflow for controlling the executing flow by means of a buffer controlled by the attacker

Use-after-frees is the result of dereferencing a pointer that points to an object that had already been freed. Such pointers are generally called "dangling pointers". Weinmann points out the fact that the GSM contains a lot of structures, also called “state machines”, that may be vulnerable to such kind of bugs. Indeed, in some cases, in general, a program may use a pointer to a structure whose memory was already de-allocated. If that happens, it is likely that afterwards, the programm will enter into an unintended execution flow, which could lead to a crash, or more dangerous behavior. For instance, if one could control the data that is pointed by the dangling pointer, thus, one would have the ability to execute arbitrary code in the context of the vulnerable program. Afek *et al.* was the first who highlighted such kind of vulnerabilities in [AS07b]. He thoroughly gives more details about the techniques for exploiting dangling pointers.

Interger overflows/underflows is basically the result of attempting to store a value in a variable which is too small to hold it. They do not allow direct overwriting of memory or direct execution flow control, however, their possible exploitations are much more subtle than the previous vulnerabilities. A very basic example is given in the Listing 6.2. In this example, the value `65536` of the local variable `i` is too great to fit into `s`, as the maximum value of an `unsigned short` is `65535`. Consequently, instead of executing the `exit(EXIT_FAILURE)`, the program will execute the function `doSomethingSecret()`.

Listing 6.2 Illustration of an integer overflow for modifying the execution flow

```c

int i = 65536;
unsigned short s = i;

if(s >= 20)
{
        //EXECUTION FLOW A
        exit(EXIT_FAILURE);
}

//EXECUTION FLOW B

doSomethingSecret();
```

For demonstrating the possibility to exploit those security issues, Weinmann attempted and succeeded in corrupting the memory for enabling the auto-answer functionality of the device. He mainly focused on the iPhone 4 and HTC phones:

- on the iPhone 4 with a Infineon/Intel baseband, a N-bit long Temporary Mobile Subscriber Identity (TMSI) was crafted to embed a payload and sent to trigger an heap overflow, where $N \geq 32$ bit, *i.e*, the normal size of a common TMSI. The TMSI is commonly sent between the mobile and the network and is used for ensuring the privacy of the mobile subscriber. It is assigned for the duration that the subscriber is in the service area of the associated Mobile Switching Center (MSC). This attack enabled the author to overwrite an arbitrary location in the heap memory of the RTOS. By exploiting this heap memory corruption, the author succeeded in executing an arbitrary code to enable the auto-answering feature.
- on the HTC phone, with a Qualcomm baseband, he targeted the `S0` register of the baseband such as it must be set to a value greater than 0. For this purpose he chose to exploit a stack overflow vulnerability, by exploiting a bad length check while the baseband is handling a challenge request from the base station during the mutual authentication process (see diagram 6.13). After having found the address of the function that is responsible of setting the `S0` register, the stack overflow helped him in modifying the execution flow to that function (the callee), by taking care of providing to the latter, the new value of `S0`, and also by properly returning to the caller, so the execution flow can carry on without crashing. The diagram 6.14 illustrates the execution flow modification (red path) after the exploitation of the stack overflow vulnerability, and the back to the normal execution flow (orange path).

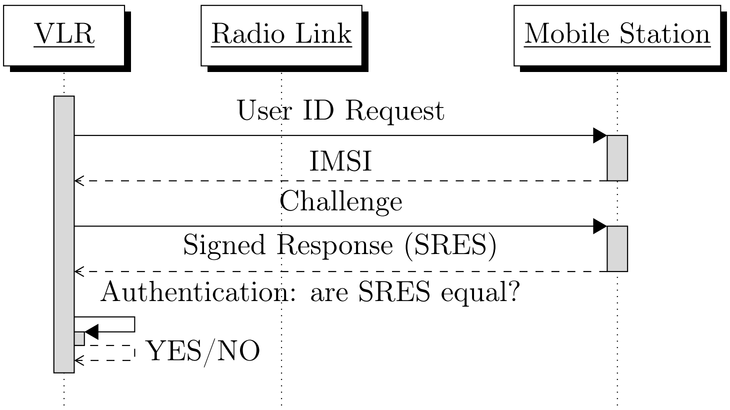


Figure 6.13: Basic diagram on the mutual authentication between the Mobile Station and the VLR. Challenge equals to RAND for GSM or (RAND || AUTN) for UMTS-based network. RAND is a random number, AUTN an authentication token.

Both attacks resulted in the activation of the auto-answer functionality which enables to configure the baseband such as it decides to answer to a given phone call after a certain amount of ring calls.

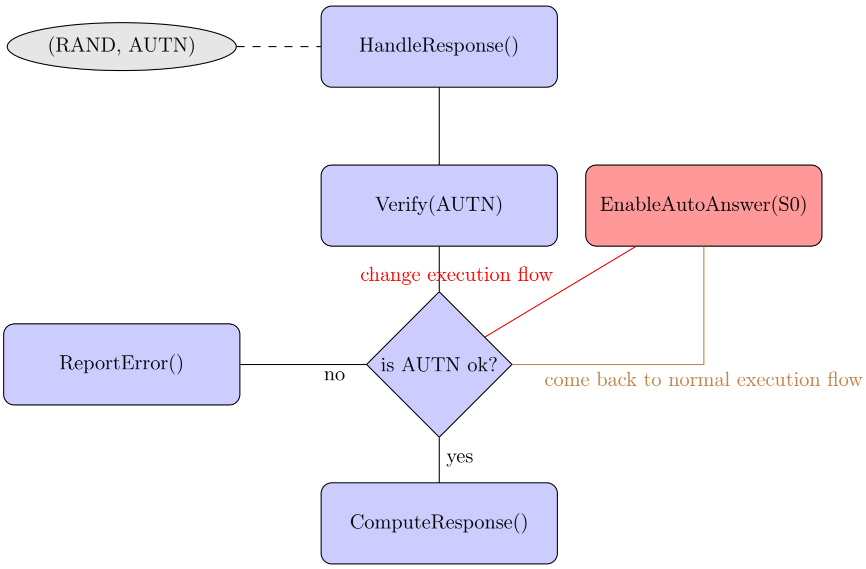


Figure 6.14: Execution flow modification after a stack overflow exploitation for modifying the baseband's `S0` register in order to enable the auto-answer functionality

6.3.3 Debugging the Baseband Firmware

We have seen in the previous sections, that finding vulnerabilities is possible, however, exploitation remains something difficult to do if one does not have a better understanding of the system that is being run. For this purpose, debugging – if possible –, is a powerful technique for carefully analyzing a program as it is running.

There are only three ways for debugging a software running on an embedded microprocessor.

6.3.3.1 Debugging through simulation

It is one of the way to test a software, by running it on a simulator for the embedded chip. A well-written simulator can also behaves like a debugger, namely, getting the ability to stop a process by pausing the simulator. It can easily provide information on the variables, the registers, or anything one wishes to find out by looking up at the simulated program memory. The tool that is used by most embedded developers, or security enthusiasts is probably `Qemu` [qem], which is the most known open source processor emulator. A lot of resources related to analysis and reverse engineering of firmware through `Qemu` exist on Internet and are great sources of information while one starts to have interests in the analysis of embedded software.

6.3.3.2 Software-only solution

It consists of a low-level software that plays the role of interface, that provides the necessary features for pausing/breaking/resuming the execution of the software under study. Such kind of interface is also known as a stub program (e.g: `gdbserver`), that either runs along side the tested program in the case of a GPOS-based system. In case of RTOS-based one, the stub consists of a task that has a high priority, so the OS can schedule it before other tasks. A good example is the Linux Kernel Debugger (KGDB), and it allows kernel developers to use client debuggers such as the GNU Debugger (GDB). However, as far as we know, basebands do not embed such kind of stub program, consequently, it is not possible to connect a client debugger for analysing the firmware that is being run, unless one gets the ability to inject the stub within the device. This has already been attempted once in the literature [Del11]. An illustration of the set up of this kind of approach is given in the Figure 6.15.

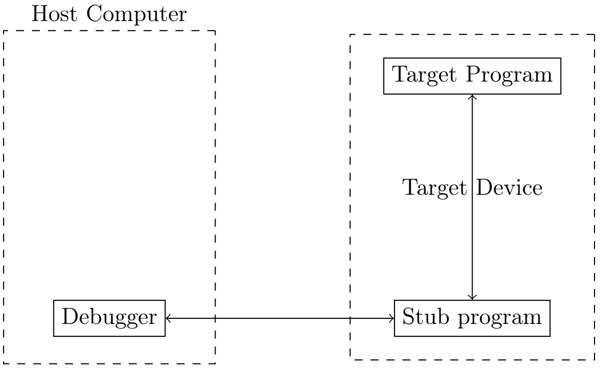


Figure 6.15: Overview of a remote debugging session

6.3.3.3 Hardware-assisted solution

Despite how convenient software-only debuggers can be, they can be complex and unreliable in some cases. Another debugging technique exists, and is more reliable. In the embedded world, it is likely that embedded systems developer provide an hardware component called the JTAG (Joint Test Action Group), or also known as an In-Circuit Emulator (ICE), and provides features that are not available in software-only debuggers. It is used to access on-chip debug modules inside the target processor, and among other things, it enables to halt the processor, inspect its registers and memory, single step through the code, and define breakpoints. It requires to be paired with a software debugger though (e.g: GDB), so the user can control it.

While the JTAG interface are included in many embedded devices to ease the debugging procedure, nowadays, on recent smartphones, it is getting harder to get access to it. Indeed, additional challenges must be overcome: **(1)** finding the JTAG port, and **(2)** activating it. Indeed, the contacts (pins) on which one must connect may be different from a device to another one, so as their location. Furthermore, some vendors may deactivate the JTAG support for security reasons, by means of hardware or software fuses.

In the section 6.2.1.2, we gave as an example, geohot's hardware trick through the baseband processor to perform a carrier unlock on the iPhone 2G. In this example, one of the difficulty geohot met was to find out the exact pinout of the JTAG. While in his case, tracing the JTAG pins required to desolder the baseband chip, some mobile devices such as for instance the Samsung Galaxy S2 (i9100) leave the JTAG pinout visible. The comparison between both is depicted by the Figures 6.17 and 6.16. On the Figure 6.17, the JTAG pinout is not directly visible, and somehow, it has to be traced in order to distinguish the right pins/balls among the various ones, in the Ball Grid Array (BGA). On the other figure, the JTAG pinout is clearly visible, however the head socket connector has been removed, so one cannot easily plug the JTAG. This is not a big issue, and despite the pitch [16] it is fairly easy to solder a socket connector.

However, it is worth to note that most of current modern smartphones have the JTAG disabled, by means of fuse(s). Fuse(s) can be blown in order to disable the JTAG debug interfaces. They can be hardware fuses, or software fuses. While hardware fuses physically disconnect the JTAG lines internal to the processor, software fuses generally change a value in a One-Time Programmable non-volatile memory (OTP NVM).

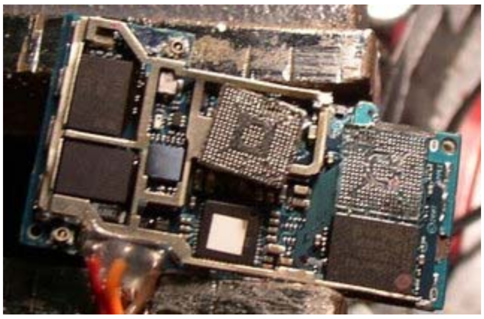


Figure 6.16: BGA removal for tracing the JTAG pads on the baseband processor of the iPhone 2G (source: http://geohot.blogspot.fr/)

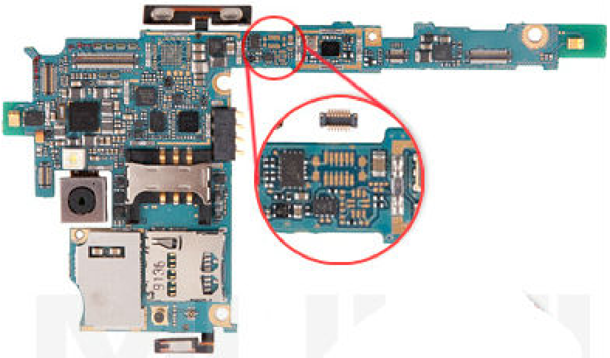


Figure 6.17: JTAG pinouts of the Samsung Galaxy S2

6.3.4 Static Analysis of a Raw Memory Dump of the Baseband at Runtime

Some devices may give the ability to read the memory by means of an interface provided either by the baseband or by the Operating System, or directly through the hardware.

In [Del11], Delugré succeeded in dumping the whole memory of a 3G USB stick. From there, he started to analyse the RTOS named REX. To give himself the means to perform a better vulnerability analysis, he decided to develop a debugger named `qcombbdbg` [Del12].

The target of evaluation was a 3G USB stick that embeds Qualcomm's MSM6280 baseband. After discovering that plugging the USB stick on the computer triggers the creation of three serial ports, he focused on the third one which was actually a diagnostic port used by the baseband. This port is also called the `DIAG` port and speaks a protocol that uses standard HDLC (High-Level Data Link Control) framing with a CRC-16 and a frame terminator of `0x7E`. It also uses `0x7d` as an escape for occurences of `0x7e` and `0x7d`. The escaping is done after computing the CRC and is applied to both the packet and CRC. The CRC is generated using the standard CRC-CCITT-16 generator polynomial of: $f(x) = x^{16} + x^{12} + x^5 + 1$

The HDLC frame format is illustrated by the Figure 6.18.

From, the open source project ModemManager [Fre] he figured out the commands that can be used for instructing the baseband to perform some specific tasks. Further research on internet enables to find out that the proprietary protocol that uses that HDLC format is actually the `Qualcomm Diagnostic Monitor` (QCDM) that is an old protocol found in most (old) Qualcomm devices. Through leaked source codes, people already succeeded in re-implementing the protocol, so it can now be managed through open source projects such as ModemManager (e.g: see `libqcdm`, *dm-commands.h*).

Delugré noticed two particular commands that enable to read and write into the memory at an arbitrary address. Compared to other approaches, such as source code review, or reverse

[16] the center-to-center spacing of pins

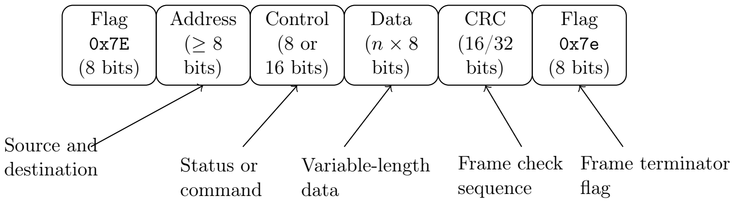


Figure 6.18: HDLC frame format

engineering of a single static binary, his approach provides the ability to analyse at runtime, the heap memory, and the stack states of the running tasks.

As a summary, by means of reverse engineering, he succeeded in giving an high-level overview of the RTOS's architecture:

1. **AMSS**, Qualcomm's proprietary operating system,
2. **REX**, the real-time kernel and its main components (the real-time scheduler, the interprocess communication and synchronisation mechanisms, the timers, etc.)
3. **the boot sequence**, from the primary bootloader (PBL) to the launch of the operating system (AMSS),
4. **the 69 tasks**, namely, the features provided by the baseband (Digital Signal Processing, USIM management, network stacks management, diagnostic tasks, etc.)

Finally, another very instructive work must be cited in this thesis. In [Mic14], Michau *et al.* made a great analysis (in french), regarding the security of basebands in mobile devices. They particularly gave various analysis examples regarding basebands from different manufacturers:

1. Intel XMM6260 from Intel-Infineon, on an Android device with 2G/3G capabilities,
2. BRCM21552 from Broadcomm, on a Samsung Galaxy Y S5360 with 2G/3G capabilities,
3. Novathor from ST-Ericsson on a Samsung Galaxy S3,
4. Qualcomm Snapdragon (Hexagon architecture) on a Google Nexus 5, from LG, with 2G/3G/LTE capabilities.

Furthermore, they set up a GSM network test bed and also a 3G/UMTS one, and they analysed 40 devices (3G USB sticks, and smartphones) with 2G/3G or 2G/3G/LTE capabilities. Their very first finding was about the non-compliance of all the tested devices, to the 3GPP TS 22.101 specification [ETSd], regarding the "ciphering indicator" feature. While the specification states that, devices must indicate to the user that the current radio communication is encrypted or not, it appears that none of the devices implemented it.

Additionally, they also analysed the supported cryptographic algorithms that each modem must announce while they are initiating a connection with a given mobile network. It turns out that one of the tested device returned some fake algorithms. Moreover, while the A5/1 and A5/2 algorithms that are used for encrypting the communication between the device and the base station, have already been deemed breakable [Gol97, BBK, BBK], the authors also found out that, some modern mobile devices just do not provide the A5/3 algorithm which is stronger than A5/1.

Among the other findings, an interesting one is the stack overflow vulnerability that they found out on a given device, which was actually similar to the one Weinmann exploited in 2009, as we previously explained at the end of the section 6.3.2. What is interesting here is that, 5 years after, the same bug can still be replayed and likely exploited.

6.3.5 Dynamic Analysis of an Embedded Firmware

We have already seen that, maliciously crafted SMS could be used as an attack vector for getting hands on the baseband. However, blind injection is not the best approach as the outcomes (mostly only crashes) can not be properly interpreted. While manual static analysis of the code brings a more precise understanding of he system under study, it can take a lot of time.

Dynamic analysis is another approach that relies on testing a software by running it either on the real device, or on a simulator. Zaddach *et al.* introduced in [ZBBF14] a framework that uses both approaches for dynamically instrumenting and analysing embedded firmwares. The whole system architecture is split in two: on the first side, there is an emulator that will run the code, and on the second side, there is the target device that will also run the same code. However, on the emulator side, the data that are required at runtime are fetched from the memory of the target device.

On one hand, the code is totally under control within the emulator, and on the other hand, the state of the target device is faithfully preserved, *i.e*, the state of the registers, and its memory. Both, the emulation, and the target device, while running the code will produce events: instruction translation or execution, memory accesses, exceptions, interruptions, ...The synchronization between both sides is then made through the detection of these events. For the debugging capability, their framework can either use GDB or a JTAG interface. In case, no JTAG interface is available, a GDB stub must be injected on the target device in order to make the bridge between the latter and the GDB that is running on the host computer.

Running the code in a controlled environment has a lot of benefit compared to static analysis as it gives the ability to perform more automated and complex analysis such as:

Dynamic Taint Analysis or also known as Dynamic Information Flow Analysis, consists in running the program and observing which computations are affected by predefined taint sources such as for instance, user input. The information flow between the sources and sinks are tracked, and any value whose computation depends on data derived from a taint source is then considered as `tainted`, otherwise it is `untainted`. Dynamic Taint Analysis have already been successfully used to prevent a wide range of attacks, including format string attacks [NS05, QWL$^+$06], SQL and command injections [HOM06, HOM06], buffer overruns [KZZ06, NS05]. Furthermore, this technique has also been used for reverse engineering purpose, and software testing [LBB$^+$07, MPL04]. Schwartz *et al.* formalized in [SAB10] and give a thorough analysis of this concept.

Dynamic Symbolic Execution Analysis aims at building a logical abstraction that describes a program execution path, which reduces the problem of reasoning about the execution, to the domain of logic. It allows to execute a program through all possible execution paths, thus achieving all possible path conditions. The main drawback from this method would be the exponential number of possible execution paths. Generally, in order to address this limitation, most tools apply a strategy for reducing the number of paths to explore. For instance, one can support setting up a fixed depth until which the execution tree is explored and the path exploration process terminates after that depth.

Concolic Execution Analysis or also known as CONCrete execution and symbOLIC execution. It aims at performing a symbolic execution analysis, with concrete values as inputs. As a result, it would be possible to "guide" it through a specific execution path without traversing all of them, which is more feasible.

The main issue regarding those techniques, in the embedded world, is that target devices generally have proprietary hardware components, and most likely, different undocumented software designs. Consequently, according to the technique that is used, the approach is likely different from a device to another one.

Among the different devices that they tested, they attempted in analysing the baseband firmware of a GSM feature phone, the Motorola C118, with a digital baseband from Texas Instruments "Calypso", and a single ARM7TDMI processor. For the debugging capability, they could get access to the JTAG interface of the target device, so they could pause and resume the processor at will, for performing the synchronization between the emulator and the target device.

6.4 Analysis of the REX micro-kernel

We have introduced in the chapter 6, the necessary concepts regarding basebands in mobile devices, and we also tried to review some of the related work around basebands security in the current literature. We have particularly seen that, blindly testing the baseband, by means of techniques such as fuzzing (section 6.3.1) may give the ability to trigger some unintended behaviors. They can give hints on possible exploitations. Furthermore, as we have seen in the section 6.3.2, manually reversing the baseband in a blackbox way can take time, unless, one exactly knows, what and where to focus on. However, having a precise view on the implementation of a given baseband feature is the best approach for an effective security assessment.

We have presented in the section 6.3.4 the work of Delugré on the reverse engineering of the baseband firmware of a 3G USB stick. His primary intention was to get knowledge on how is implemented an actual baseband firmware. Based on the assumptions that, basebands code within a mobile device may be quite huge, thus, it may take time to reverse engineer it. It appears that for historically reasons, old pieces of code may still be present, it is likely that developers do not start from scratch while implementing a baseband. These statements are mainly drawn from the analysis of Weinmann and Michau's work we have reviewed in the sections 6.3.2 and 6.3.4.

Consequently, rather than directly reversing the baseband of an actual mobile device, we resumed the work of Delugré and we reverse engineered the baseband firmware of the same 3G USB stick in order to get knowledge about how the underlying RTOS works, and more particularly to identify how is made the communication with the (U)SIM card from the software point of view. A summary of our findings is provided in this section.

Beyond the knowledge acquisition about the implementation of a real baseband system, the main benefits of this work is to get the means to ease the reverse engineering of more complex modern baseband systems.

The device that has been studied was an Icon 225 3G USB stick. It embeds a Quaclomm MSM6280 baseband processor with WCDMA, HSDPA, GSM, GPRS, EDGE, and GPS protocol stacks and multimedia system software. It integrates the ARM926EJ-S processor which is paired with a QDSP4000 DSP processor based on ARMv5TEJ architecture.

Figure 6.19: Icon 225 3G USB stick

6.4.1 Methodology

Before blindly digging into the reverse engineering of the target, in addition to the high level description of REX, that was provided by Guillaume Delugré in his paper [Del11], some prior knowledge were needed as listed below:

- getting basic knowledge on RTOS technologies (see 6.1.3)
- analysing ARM's Technical Reference Manual of the ARM926EJ-S to understand the boot sequence
- analysing examples of very basic Multi-Tasking Operating Systems in ARM (a lot of examples exist on internet)
- analysing existing patents related to Multi-Tasking Real Time Operating Systems in general, and more specifically any Qualcomm's patents related to their old RTOS's microkernel, *i.e*, Real-Time Executive (REX). Patents from other companies, but that makes usage of Qualcomm's technologies related to wireless protocol communications, or ICC usage in mobile devices, or Qualcomm's chipsets, provide valuable information and hints.
- getting basic knowledge on firmware analysis (a lot of blog posts exist on this subject on internet)

6.4.2 From the Boot Sequence to the SIM Interface Implementation

The very first thing that is described in Delugré's paper is the boot sequence of the device. Through the static analysis of the firmware, this step is summarized as follows:

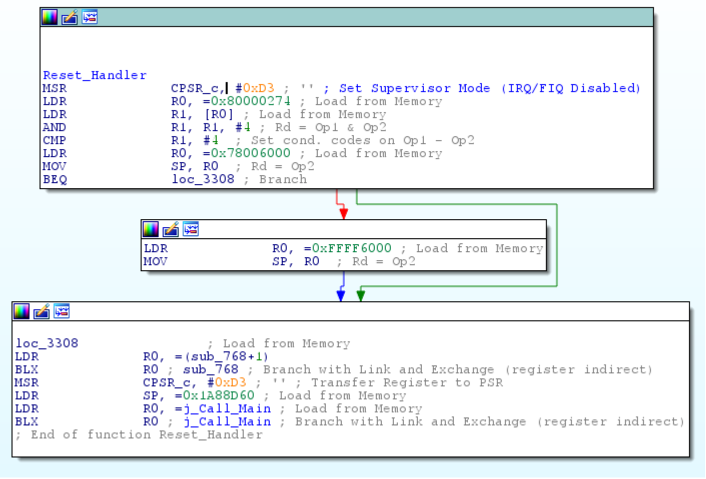
6.4.2.1 The Boot Sequence

The Hardware Bootloader Also called the boot rom, it is a code located into the processor's internal ROM. While the device is powered up, the processor is forced to jump to a `reset vector`, which is, by definition [17], the default location a processor will go to find the very first instruction it has to execute. Reading the technical reference manual[18] of the ARM926EJ-S chip enables to identify the possible location. The signal named `CFGCPUVINITHI` determines the location of the reset vector. According to its state, *i.e*, low or high, the reset vector is either located at address `0x00000000` or `0xFFFF0000`.

On this device, the reset vector was located at `0x00000000`

[17] https://en.wikipedia.org/wiki/Reset_vector

[18] Default vector location, page 60, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0287b/DDI0287B_arm926ejs_dev_chip_



```
Reset_Handler
MSR                 CPSR_c, #0xD3 ; '' ; Set Supervisor Mode (IRQ/FIQ Disabled)
LDR                 R0, =0x80000274 ; Load from Memory
LDR                 R1, [R0] ; Load from Memory
AND                 R1, R1, #4 ; Rd = Op1 & Op2
CMP                 R1, #4  ; Set cond. codes on Op1 - Op2
LDR                 R0, =0x78006000 ; Load from Memory
MOV                 SP, R0  ; Rd = Op2
BEQ                 loc_3308 ; Branch
```

```
LDR                 R0, =0xFFFF6000 ; Load from Memory
MOV                 SP, R0  ; Rd = Op2
```

```
loc_3308                      ; Load from Memory
LDR                 R0, =(sub_768+1)
BLX                 R0 ; sub_768 ; Branch with Link and Exchange (register indirect)
MSR                 CPSR_c, #0xD3 ; '' ; Transfer Register to PSR
LDR                 SP, =0x1A88D60 ; Load from Memory
LDR                 R0, =j_Call_Main ; Load from Memory
BLX                 R0 ; j_Call_Main ; Branch with Link and Exchange (register indirect)
; End of function Reset_Handler
```

Figure 6.20: Reset code from the interrupt vector table of the system at address 0x00000000

The Primary Bootloader (PBL) is the very first component that is loaded from the address 0xFFFF0000, from the ROM memory. Examining the PBL enables to identify the following operations:

- A configuration data is loaded from and is processed in order to configure the NVRAM,
- The start offset of the secondary bootloader (SBL) is retrieved, and the PBL starts to load it into the NVRAM,
- The control is transferred to SBL

The Secondary Bootloader (SBL) also called QC-SBL (for Qualcomm SBL), it is mainly responsible of doing the following operations:

- A vector table is created in the NVRAM
- A configuration data is loaded and is processed in order to configure the NVRAM
- AMSS's (Advanced Mobile Subscriber Software) image is loaded
- REX is initialized
- The control is transferred to AMSS

6.4.2.2 REX's Tasks

As a reminder, a task, related to a RTOS, is a function that performs a given task and runs independently from other tasks.

In the context of REX, every task has its own stack, priority and signal flags which defines the **context** of the task. Signals are used to trigger execution states between tasks. As summarized in the next paragraphs, identifying how are implemented REX's tasks greatly help in the next step of the reverse engineering process.

Task Control Block (TCB). To each task is associated a data structured that is called, the Task Control Block which is a data structure, that is used by REX for keeping track of the context of the current running task. REX maintains a linked list of tasks in which, every TCBs are also linked in decreasing order of task's priority.

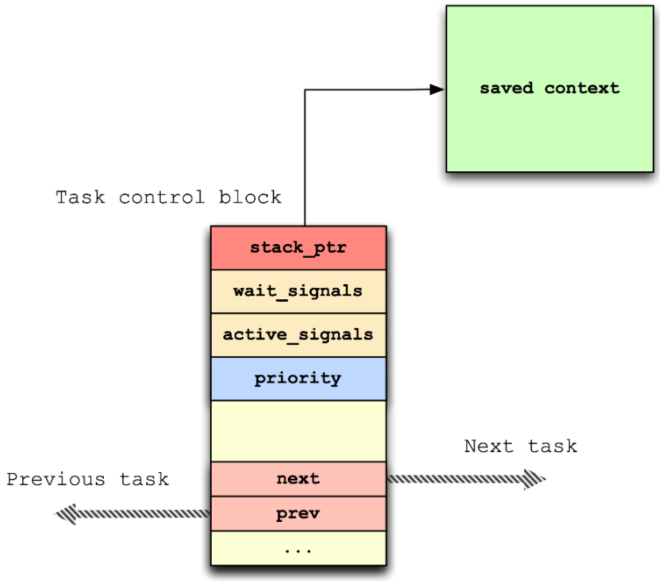
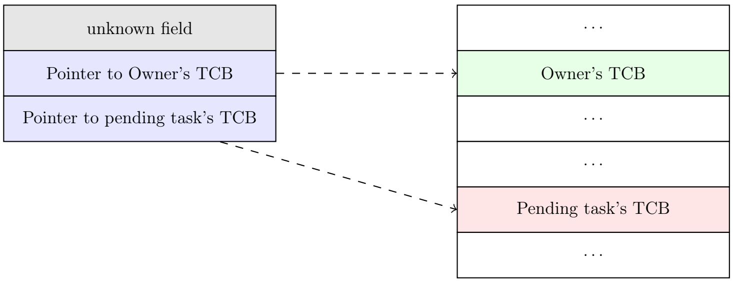


Figure 6.21: Data structure used by REX for keeping track of tasks and their execution context. (source: [Del11])

Task's Stack. To each task is associated its own stack that is retrieved through its TCB (see 6.21). During a task switching, the task's registers are pushed onto its stack, and the address of the top of the stack is saved into its TCB so it can be restored next time the task is resumed.

Task's Signals. Another element that is stored within a TCB are signal flags that are used for notifying the task that some kind of event has occurred. Commonly, a signal refers to a software interrupt that is generated when a particular event has occurred. Signals notify tasks of events that occurred during the execution of other tasks. In the context of REX, signals are defined with a 32-bit flags. As an example, signals can be used to put a task in either a "waiting mode", or to "wake up" it. Each task can be set with up to 32 signals. Different functions are used by REX to set or clear the signals of a given task, and also to wait for a signal to be set. Tasks synchronization are mainly performed through signals and timer management.

Critical sections. Most embedded systems and REX as well, natively supports critical sections, that are used to provide a mutual exclusion mechanism between tasks. It is a data structure that is written by one task and read by another one. REX provides two functions for entering and leaving the critical section. Basically, while a task attempts to get access to a critical section that is already being used by another one, the first one is blocked until the critical section becomes available again.



Timers. REX supports timers for executing tasks at regular intervals. Basically, timers track a countdown in time, and set signals on expiration. Three types of actions can be triggered while a timer is run:

- sending a signal to an arbitrary task
- executing an Asynchronous Procedure Call (APC) in the context of an arbitrary task. (APC is further explained in the next paragraphs)
- executing a task in the current context

Timers may also allow a callback function to be called on expiration.

REX provides various functions to manage the timers, *i.e*, to increment the time, to define a timer with or without a callback function, to start, pause, resume or stop the timer.

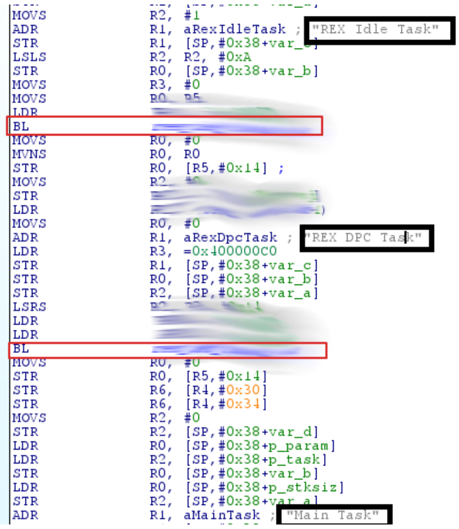
6.4.2.3 REX's Dynamic Memory Management

REX uses various function to allocate and deallocate memory dynamically: `mem_malloc`, `mem_calloc`, `mem_realloc`, `mem_free`. Analysis of strings enables to easily retrieve them. Allocations are performed on REX's heap which is initialized during REX's initialization. Some wrapper functions exist in order to provide further checks.

6.4.2.4 REX's primary tasks

While AMSS is run, three tasks are dispatched as depicted by the Figure 6.22:

- **IDLE task** is the one that has the lowest priority. It has a priority of zero and is run while no other task is ready to run.
- **DPC (Deferred Procedure Call) task**. REX provides the ability to perform asynchronous procedure calls. The DPC task is mainly a background loop that polls for work, that is called, an asynchronous procedure call (APC). An APC is a mechanism that is associated to a task that has to be run in the future. The mechanism consists in pushing a new execution context on the stack of a task that is in a waiting state, and then, once the current execution context corresponds to the associated task, the corresponding APC is executed. Thus, this mechanism mainly enables to defer the execution of a function associated to the execution context of a given task.
- **MAIN task** is a particular task that takes care of starting all the other tasks and services of the system. With the help of strings, one can easily retrieve the structure that stores the name and the function handle of each task, as depicted in the Figure 6.23.



```
MOVS            R2, #1
ADR             R1, aRexIdleTask ; "REX Idle Task"
STR             R1, [SP,#0x38+var_b]
LSLS            R2, R2, #0xA
STR             R0, [SP,#0x38+var_b]
MOVS            R3, #0
MOVS            [redacted]
LDR             [redacted]
BL              [redacted]
MOVS            R0, #0
MVNS            R0, R0
STR             R0, [R5,#0x14] ;
MOVS            [redacted]
STR             [redacted]
LDR             [redacted]
MOVS            R0, #0
ADR             R1, aRexDpcTask ; "REX DPC Task"
LDR             R3, =0x400000C0
STR             R1, [SP,#0x38+var_c]
STR             R0, [SP,#0x38+var_b]
STR             R2, [SP,#0x38+var_a]
LSRS            [redacted]
LDR             [redacted]
LDR             [redacted]
BL              [redacted]
MOVS            R0, #0
STR             R0, [R5,#0x14]
STR             R6, [R4,#0x30]
STR             R6, [R4,#0x34]
MOVS            R2, #0
STR             R2, [SP,#0x38+var_d]
LDR             R0, [SP,#0x38+p_param]
LDR             R2, [SP,#0x38+p_task]
STR             R0, [SP,#0x38+var_b]
LDR             R0, [SP,#0x38+p_stksiz]
STR             R2, [SP,#0x38+var_a]
ADR             R1, aMainTask ; "Main Task"
```

Figure 6.22: Identification of the three main tasks that are run while AMSS is run

6.4.2.5 Analysis of the SIM Interface Implementation

In addition to the three first tasks that we have described previously, a lot more tasks are run. However, we analysed three of them more particularly, as they seemed to be related to the communication and management of the underlying ICC that is connected to the baseband.

1. **UIM:** stands for “User Identity Module” and consists of the task that centralizes commands that have to be processed and sent to the underlying ICC (Integrated Circuit Card). The UIM task manages different ICC types: (U)SIM for GSM or UMTS network, a CDMA SIM (CSIM) or a Removable UIM (R-UIM) for CDMA network. Further reading can be found in the patent [uim].

2. **GSDI:** Generic SIM Driver Interface task. It consists of a set of API that enables to manage SIM card [gsd]. Analysing all the strings related to "GSDI" enables to notice some interesting features, such as for instance:

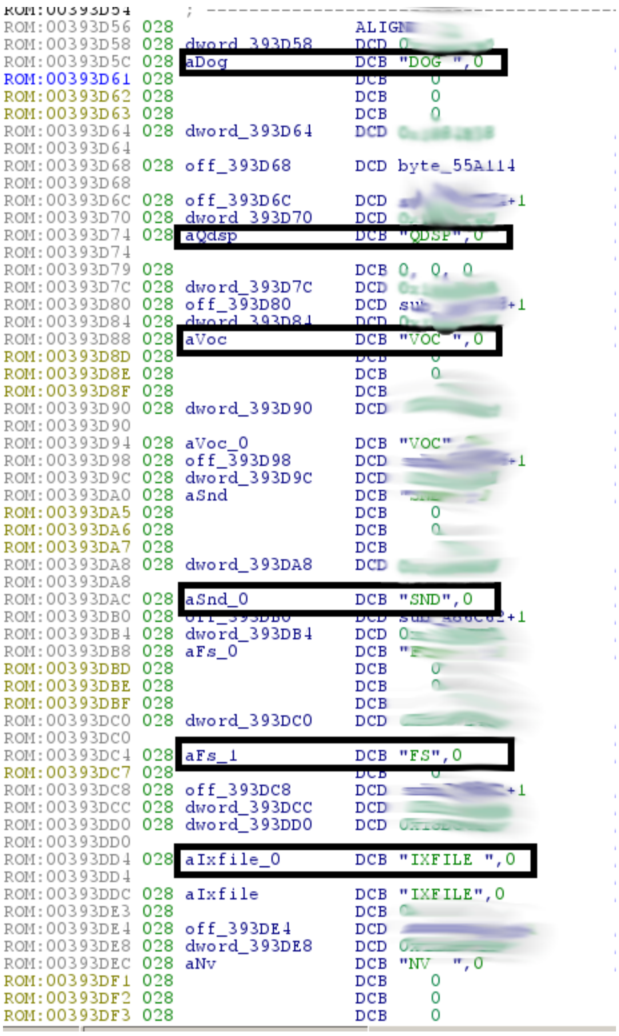
   - `VERIFY CHV` and `CHANGE CHV`, namely, for PIN verification and PIN update (see: 3GPP TS 11.11 specification [3gpa])
   - a `SEND_APDU` (Application Protocol Data Unit) function to directly send commands to the ICC

3. **GSTK:** Generic SIM Toolkit task [gst]. GSTK seems to be paired with GSDI. Analysing strings enables to determine that GSTK is able to handle "proactive commands", *i.e*, commands that can be used to instruct the SIM to perform various tasks, as defined in TS GSM 11.14 [ts1].

Other interesting tasks have been identified, but no further investigations have been made:

1. **FS:** Seems to be related to the management of Qualcomm’s proprietary Encrypted File System. Searching for patents related to this enables to get a basic understanding of the implementation of such File System, however, no further investigation has been performed.

2. **NV:** Non-Volatile memory manager



```
ROM:00393D54          ; ---------------------------------------------
ROM:00393D56 028                       ALIGN
ROM:00393D58 028 dword_393D58          DCD 0
ROM:00393D5C 028 aDog                  DCB "DOG ",0
ROM:00393D61 028                       DCB    0
ROM:00393D62 028                       DCB    0
ROM:00393D63 028                       DCB    0
ROM:00393D64 028 dword_393D64          DCD
ROM:00393D64
ROM:00393D68 028 off_393D68            DCD byte_55A114
ROM:00393D68
ROM:00393D6C 028 off_393D6C            DCD            +1
ROM:00393D70 028 dword_393D70          DCD
ROM:00393D74 028 aQdsp                 DCB "QDSP",0
ROM:00393D74
ROM:00393D79 028                       DCB 0, 0, 0
ROM:00393D7C 028 dword_393D7C          DCD
ROM:00393D80 028 off_393D80            DCD su         +1
ROM:00393D84 028 dword_393D84          DCD 0
ROM:00393D88 028 aVoc                  DCB "VOC ",0
ROM:00393D8D 028                       DCB    0
ROM:00393D8E 028                       DCB    0
ROM:00393D8F 028                       DCB
ROM:00393D90 028 dword_393D90          DCD
ROM:00393D90
ROM:00393D94 028 aVoc_0                DCB "VOC"
ROM:00393D98 028 off_393D98            DCD            +1
ROM:00393D9C 028 dword_393D9C          DCD
ROM:00393DA0 028 aSnd                  DCB "
ROM:00393DA5 028                       DCB    0
ROM:00393DA6 028                       DCB    0
ROM:00393DA7 028                       DCB
ROM:00393DA8 028 dword_393DA8          DCD
ROM:00393DA8
ROM:00393DAC 028 aSnd_0                DCB "SND",0
ROM:00393DB0 028 off_393DB0            DCD sub_           +1
ROM:00393DB4 028 dword_393DB4          DCD 0
ROM:00393DB8 028 aFs_0                 DCB "
ROM:00393DBD 028                       DCB    0
ROM:00393DBE 028                       DCB    0
ROM:00393DBF 028                       DCB
ROM:00393DC0 028 dword_393DC0          DCD
ROM:00393DC0
ROM:00393DC4 028 aFs_1                 DCB "FS",0
ROM:00393DC7 028                       DCB    0
ROM:00393DC8 028 off_393DC8            DCD            +1
ROM:00393DCC 028 dword_393DCC          DCD
ROM:00393DD0 028 dword_393DD0          DCD 0
ROM:00393DD0
ROM:00393DD4 028 aIxfile_0             DCB "IXFILE ",0
ROM:00393DD4
ROM:00393DDC 028 aIxfile               DCB "IXFILE",0
ROM:00393DE3 028                       DCB
ROM:00393DE4 028 off_393DE4            DCD
ROM:00393DE8 028 dword_393DE8          DCD 0
ROM:00393DEC 028 aNv                   DCB "NV  ",0
ROM:00393DF1 028                       DCB    0
ROM:00393DF2 028                       DCB    0
ROM:00393DF3 028                       DCB    0
```

Figure 6.23: Tasks and services initialized by the MAIN task

3. **SECRND:** A service that enables to get variable length random data

4. **SECCRYPTARM:** Cryptographic services

5. **SECSSL:** Support of Secure Sockets Layers (SSL) protocol

6. **IPSEC:** IPSec manager for secure communications over IP

An architecture overview of these tasks is depicted by the Figure 6.24. Two buffers are used and managed by the UIM driver. A buffer is used to issue APDU commands, and the other one is filled with the response issued by the SIM card. GSDI and GSTK queue commands in a linked-list of pending commands that are transmited and processed by the UIM task.

Further investigations have been performed on the GSDI task in order to evaluate potential heap overflow vulnerabilities within the generic APDU command handler that is responsible of sending arbitrary commands. The Figure 6.25 illustrates and summarizes the internal operations that are performed by the GSDI task.

The GSDI task waits for messages coming from other tasks. According to the recipient, the message is routed to another task. We are more interested in the execution path that leads to

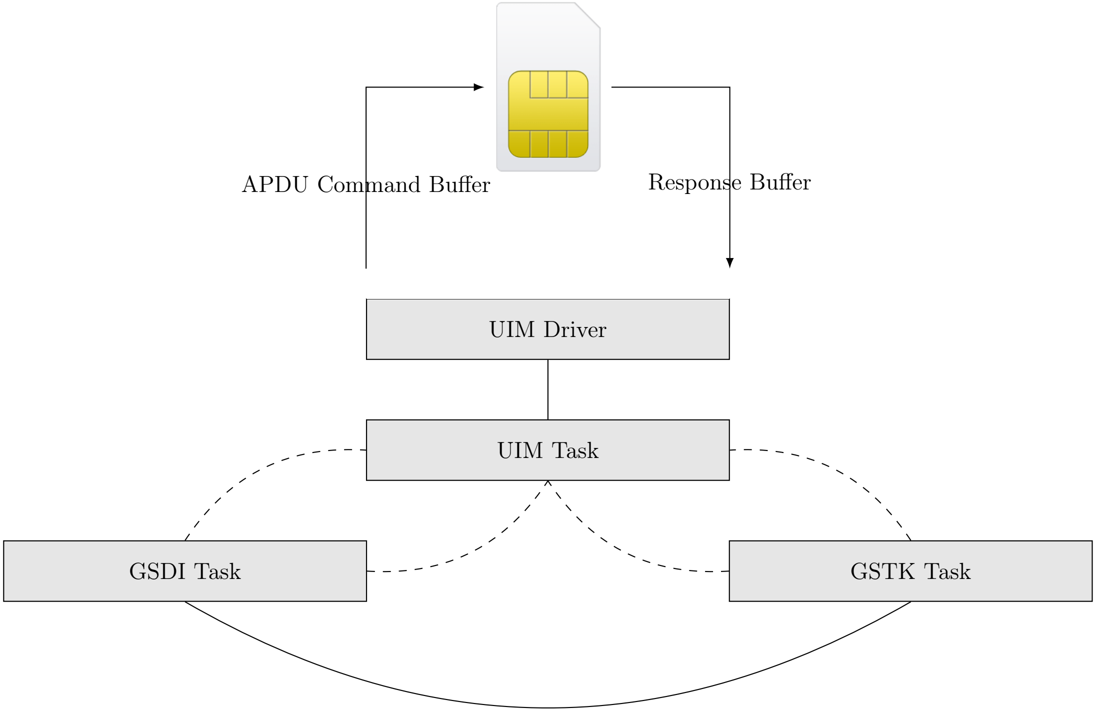


Figure 6.24: Architecture overview of the tasks that talk with an underlying SIM card

the UIM server. The latter can handle 40 different commands. Most of them are pre-defined commands, such as for example for PIN code verification or update, application selection and so on. However there is a particular command handler that seemed to be more interesting, which we called `SendAPDU`, as it enables to send an arbitrary APDU (Application Protocol Data Unit) commands (see figure 6.26).

After reversing this specific part, it has been noticed that the size of the APDU command has been properly checked before to queue it within a buffer. That particular buffer is then stored within another huge and complex structure managed by the UIM server. It turned out that the buffer is large enough to hold 256 bytes of data has it is of size 512 bytes. Consequently, it is unlikely that any overflow could be induced by manipulating the `data` field of the APDU command.

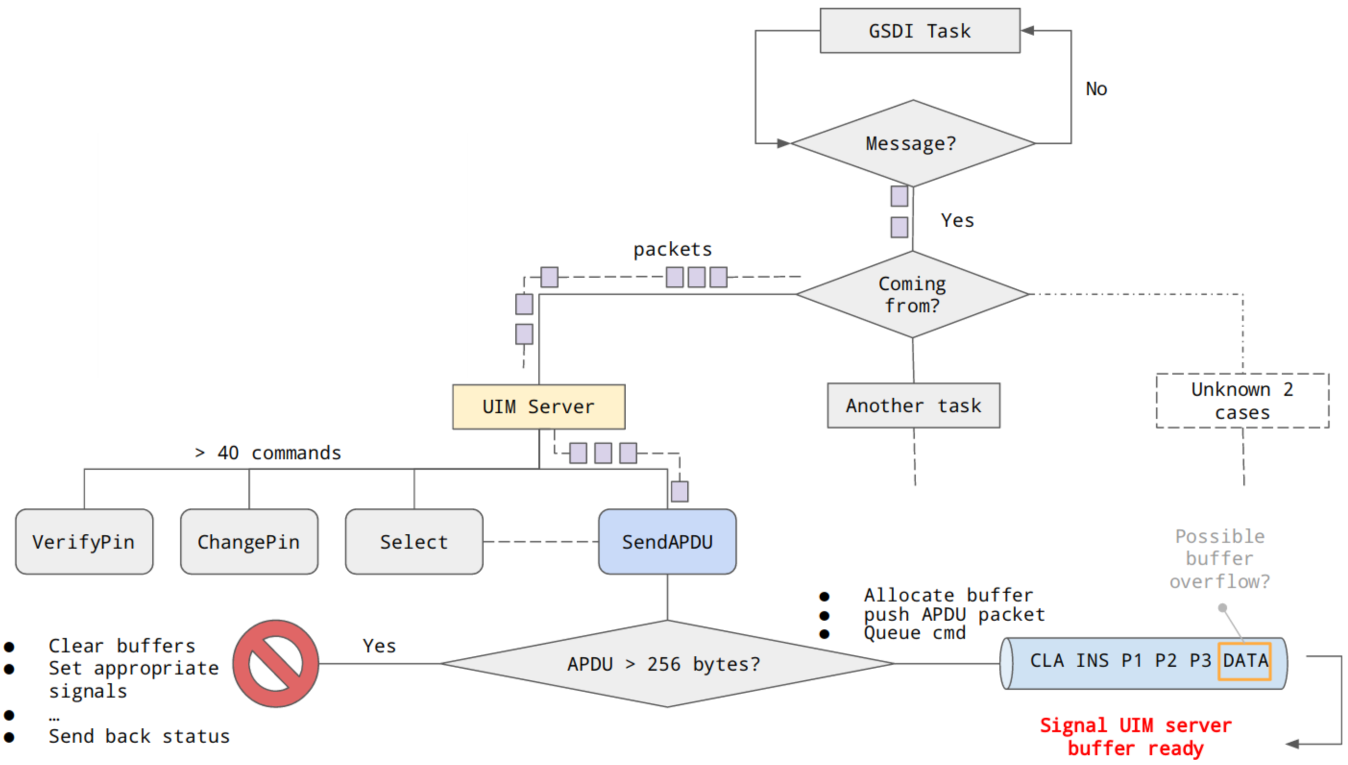


Figure 6.25: Overview of the internal features provided by the GSDI task

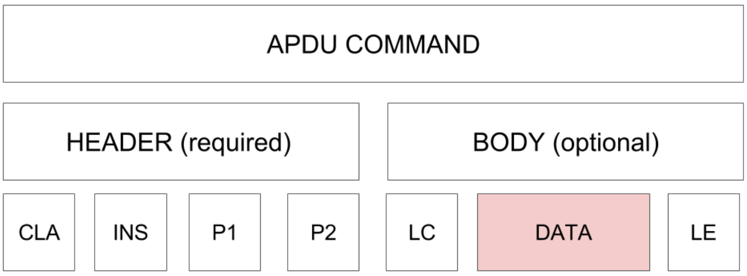


Figure 6.26: APDU command structure

6.5 Analysis of the SIM interface implementation on an actual mobile device

After gathering enough knowledge regarding the SIM interface implementation on the previous 3G USB key, we moved on the analysis of the baseband on an actual mobile device, the Nexus 5. In this case, the baseband's binary has been extracted from an archive containing the firmware update as illustrated by the Figure 6.27.

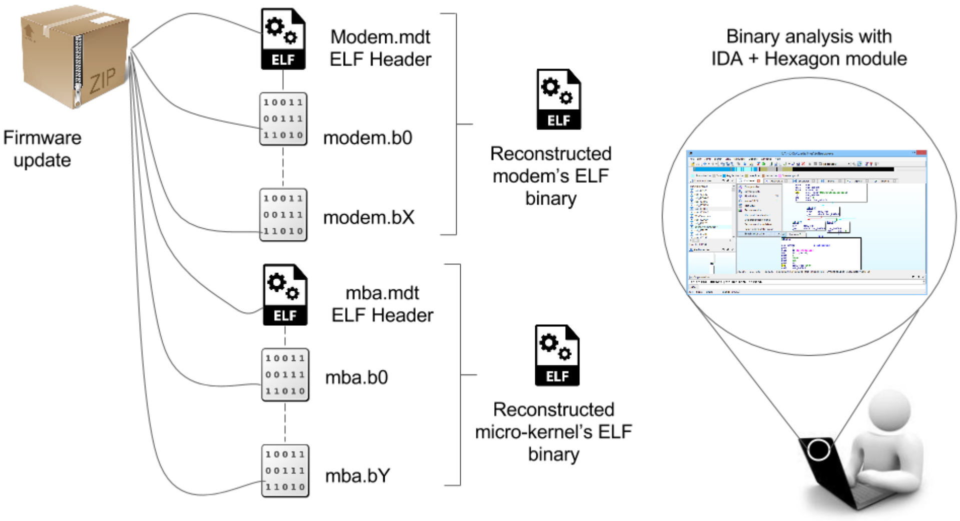


Figure 6.27: Modem's ELF binary reconstruction

The size of the binary in this case is twice larger (64MB) compared to the one extracted from the 3G USB key. Although Qualcomm implemented the baseband OS almost from scratch and is now based on QuRT/Blast, some remnants of REX are still present. It is relatively easy and fast to retrieve the GSDI and UIM tasks by analysing the strings, and by recovering REX's API that perform basic tasks managements. The following illustration illustrates the internal operations that are performed in the GSDI task.

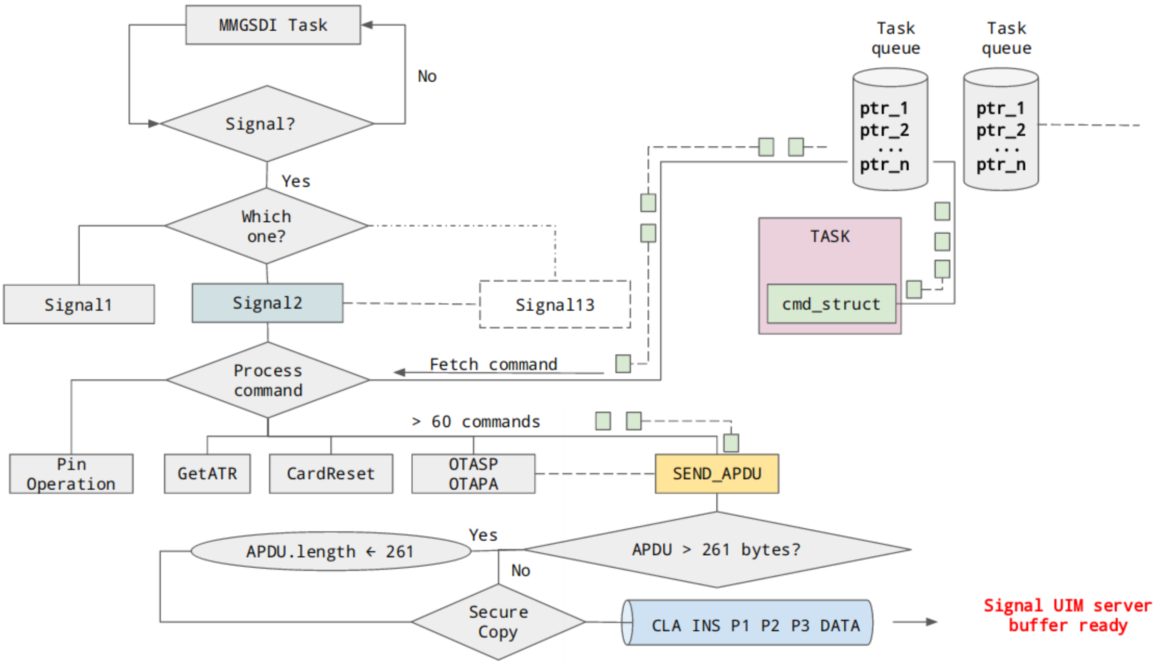


Figure 6.28: Overview of the internal features provided by the GSDI task

One of the very first interesting thing to notice here is that, commands are not anymore directly sent to the GSDI task. Signals (13 differents) are rather sent, and the only way the GSDI task is able to retrieve the actual command to handle, is by retrieving from a queue of tasks the one that is responsible of sending the latter. From the latter, it is then possible to retrieve within a specific structure maintained by that task the command that has to be sent to the UIM server. Secondly, another interesting difference with the 3G USB key, is that the GSDI task can now manage more than 20 additional commands. Among all of these commands, we could retrieve the generic one that can be used to send arbitrary APDU commands. The latter performs the appropriate bounds checking before to queue the command within a buffer. Futhermore, the it turns out that, now the kernel makes usage of secure copy functions, that wrap regular memory copy functions from the C library. The same way as we have noticed on the 3G USB key, the buffer in which the APDU command is stored, is itself saved within another larger structure that cannot be overflowed with only 261 bytes.

6.6 Conclusion and Future Investigations

Regarding the analysis of the current state of the art on the security of basebands in mobile devices, we can relatively draw the following assumption: basebands code is (or likely) a stack of reused code. In Michau's analysis of the Novathor from ST-Ericsson, he highlighted the fact that, ST-Ericsson's developers reused the code of the baseband, initially developed by Nokia, reused by Renesas and provided to ST-Ericsson. In the same way, for the case of the Broadcom's baseband analysis, it also turns out that, a part of the code is also very likely used by manufacturers such as NXP, NEC, NTT-Docomo, and probably more. In [Wei13], Weinmann made an analysis of the Hexagon architecture used by Qualcomm's basebands. An important thing he noticed is that, Hexagon-based baseband firmwares are abandoning the micro-kernel OKL4, that was used as a virtual environment for running the OS on top of it. Qualcomm redesigned a new RTOS from scratch, called BLAST/QuRT. However, apparently, some remnants of the old RTOS REX exist for compatibility reasons. This can be quickly verified: as we compared the modem binaries from a Nexus 5 that implements QuRT, and the 3G USB Icon 225 stick that is still exclusively under REX/AMSS, a lot of strings and especially, functions and state machines related to REX are still present.

While, this practice of code reuse certainly increases the complexity of the OS's architecture, more importantly, bugs or security vulnerabilities may still exist, and are kept unknown until someone finds them.

Our main future investigations aim at keeping analysing the 59 additionnal commands that can be handled by the GSDI task as depicted in the figure 6.28 in order to evaluate how likely it is possible to induce memory corruptions by means of a single APDU command.

Part III

Practical Time-Driven Cache Attack on a Mobile Device

Chapter 7

Introduction to ARM Processors and Their Multi-Level Cache Memories

Contents

The ARM processor core is a key component of many embedded systems, and it is widely used in connected and portable devices such as smartphones. This chapter is an introductory one, and aims at providing the reader with notions about the concepts and mechanisms used by the different types of cache memories that exist within an ARM processors. Section 7.1 first introduces some generalities about ARM Processor. It shows that the processor can implement different cache memories hierarchies. Indeed, modern microarchitectures[19] employ such kind of multi-level cache memories in order to hide the large latency gap between the processor and the main memory. Therefore, we describe in Sections 7.3 and 7.4 the different types of cache memories and the specific multi-level hierarchy that defines the memory organization of the processor.

[19] A set of microprocessor design techniques and resources used to achieve architecture specification in order to reach the target cost and performance goals.

7.1 Generalities on Advanced RISC Machine (ARM) Processor

Basically, an Advanced RISC Machine (ARM) processor is a microprocessor that represents a complex electronic component that contains millions of very smaller components such as transistors. It contains memories to temporarily store some values, and the smallest memory units that it uses are called *registers*. As they can be manipulated at the CPU's cycle, their access time is very fast. However, the number of registers is very limited. Furthermore, as the CPU is capable of processing a very great number of instructions, and as the most used operations are the `load` and `store` operations, the lack of available registers clearly highlights one of its limitations. Nevertheless, in terms of spatial locality, registers are just the closest memory units to the CPU. Other types of memory units exist and the more it is far from the CPU, the more the access time increases. Therefore, the solution to improve the trade-off between processing and memory speed has been the implementation of another small but fast memory close to the CPU in order to hide the main memory's latency. This memory unit is called the cache memory.

However, as we will see throughout the next sections, there are different types of cache memories that have different purposes, such as for instance storing data, instructions, addresses. They are organized along a specific memory hierarchy as illustrated by the Figure 7.3.

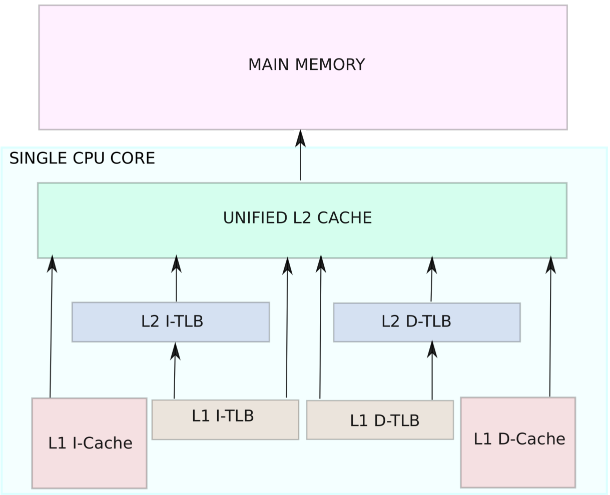


Figure 7.1: Example of memory hierarchy in an single core of a CPU

7.2 The Principle of Locality

As it has been previously studied in [Den68] by Denning *et al.*, while a program or a process is being executed by the processor, most of the memory references are not made uniformly to a small number of locations. Furthermore, most of the time [HP11], programs repeat sections of code and repeatedly access the same or nearby data. This particular characteristic has been found empirically and is embodied in the so-called *Principle of Locality.* It has two main aspects:

1. **Temporal Locality:** recently accessed items are likely to be accessed in the near future (e.g. loops)
2. **Spatial Locality:** items at addresses close to the addresses of recently accessed items are likely to be accessed in the near future (e.g. accessing sequentially an array)

This principle mainly led to the implementation of a *divide and conquer* strategy at different levels in the memory hierarchy that we are going to introduce in the next sections.

7.3 Paging Systems

In order to ensure that several processes that possibly run concurrently can be managed by the processor, to each process is associated a logical address space, namely, a range of valid addresses in memory that are available for a given program or process. Therefore, most of nowadays systems partition the main memory so to each address space is associated a specific area in the memory that is composed of *pages* that are contiguously aligned and with fixed size. Each page contains a set of Page Table Entries (PTEs). Only a subset of pages that are part of the address space of a given process are stored within the main memory. Consequently, an *address translation* is required to convert a *logical* (virtual) address (VA) to a *physical address* (PA) in order to get access to the actual physical page. Each process does not need any knowledge of the physical memory map of the system, that is, the addresses that are actually used by the hardware, or other programs that might execute at the same time. The Memory Management Unit (MMU) is responsible for performing the address translation and thus enables the system to be able to run multiple independent programs or processes in their own private virtual memory space.

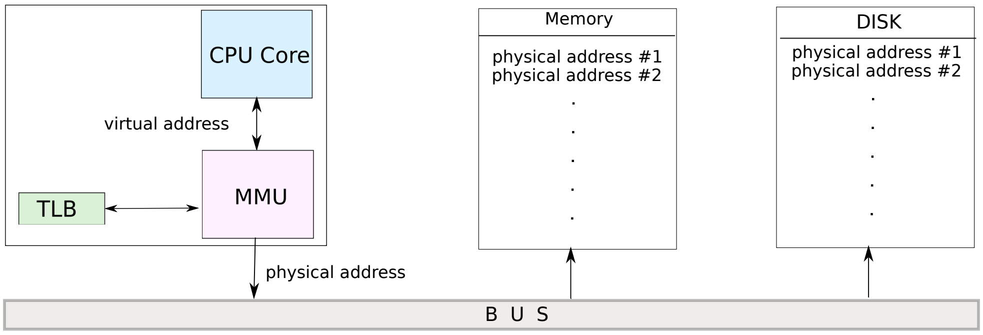


Figure 7.2: Illustration of the Virtual-to-Physical address translation through the MMU

The ARM MMU may support four different page sizes: Small pages (4KB), large pages (64KB), sections (1MB) and supersections (16MB). Furthermore, the MMU supports a two-level hierarchy for its page structure: a *first-level table* and a *second-level table.* The first-level table may contain either a pointer to the second-level table or a base address of a section or a supersection. Despite it is not mandatory for the OS to use all of these pages, using big pages such as sections or supersections enables to cover larger region of memory, such as for instance the OS.

7.3.1 The Table Lookaside Buffer (TLB)

We have seen that the MMU can manage different page sizes to provide virtual memory mappings to each process or program. For performance reasons, it is also able to store a cache of recently used mappings from the main page table (owned and managed by the operating system). This type of cache memory is called the TLB, and enables the processor to access physical addresses from their corresponding virtual addresses. The TLB is basically a simple structure that contains a page table. Each page within this table is then called a *TLB entry.* A common page is of size 4KB. The TLB contains x entries that cover several virtual memory pages of size 4KB. In the

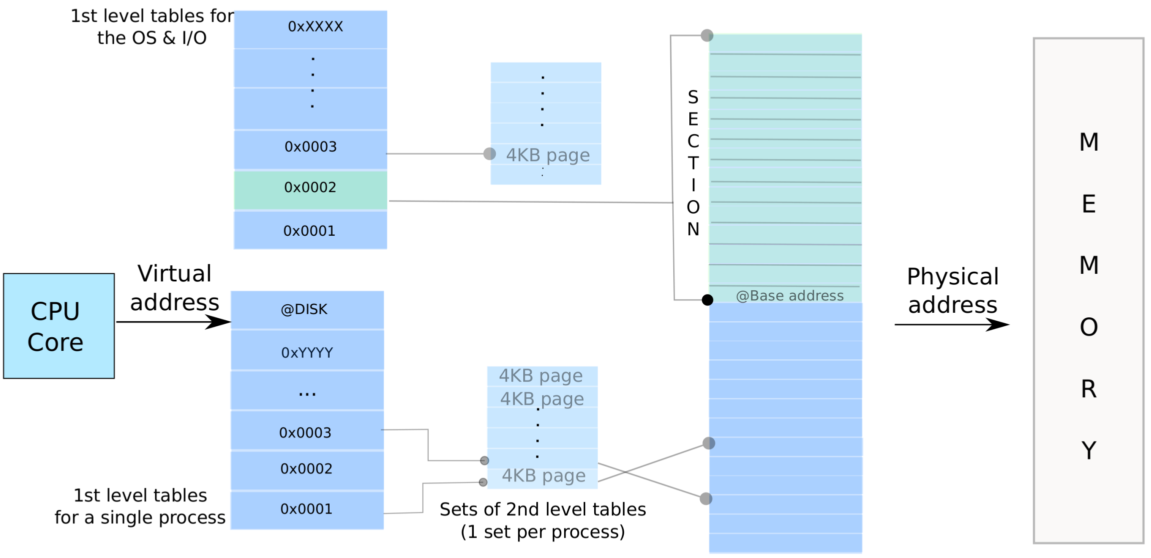


Figure 7.3: Example of multi-level page tables for the ARM memory translation. A table that contains for instance 4KB pages means that 4KB of data can be covered through each row of the corresponding table.

following example, let β the size of an array of integers where each element has a size noted α. The number of elements in the array is noted N. To fully utilize the TLB cache one has to use an array of size $x \times 4096 = \beta$ KB, thus $N = x \times 4096/\alpha$ elements.

Moreover, the same way as for the multi-level pages, ARM processors can also implement a multi-level TLB structure. ARM processors generally ahve 2 TLB levels:

- **the micro-TLBs:** which are located in the first level (L1) CPU cache, one is dedicated to the CPU instruction cache, and the other one for the CPU data cache.
- **the 2nd-level TLB:** which is located in the CPU' second-level unified cache (L2). It is mainly used while a TLB miss occurs in the micro-TLBs.

7.3.1.1 Notions about TLB misses

A TLB miss can occur if the processor did not find the information for the virtual-to-physical translation in the TLB. This involves the processor to perform a 2nd-level TLB lookup which induces a certain latency due to requiring accessing a slower level of memory hierarchy. Consequently, while a program or a process needs to access to a value at a given address, the data cache's micro-TLB is first checked. If it is not the first time the data is being accessed, then it is likely that the micro-TLB still contains the information regarding the address translation. If a TLB miss occurs, then a *table walk* is performed through the possible hierarchy of tables.

There is a last type of cache memory that is the closest to the processor and thus the fastest. The latter is described further in the next section.

7.4 The Cache Memory

The cache memory is a small high speed memory which is used by the processor to store the most recently used data or instructions from a larger but slower memory system. While the processor needs to read from or write to a location in the main memory, it first checks whether a copy of that data is in the cache. If so, it immediately reads from or writes to the cache. This is much faster than reading from or writing to the main memory. Nowadays, modern CPUs have at least three independent caches: an *Instruction cache* (I-Cache) to speed up executable instruction fetch, a *Data cache* (D-Cache) to speed up data fetch and store, and also the TLB (see previous subsection 7.3.1) that is used to speed up virtual-to-physical address translation for both executable instructions and data.

7.4.1 Cache Levels

The processor may use more than one level of cache in its memory hierarchy: from the smallest and the closest of the processor itself noted L1, to the farest and largest one Lx, where x is an index that increases according to the level of cache. As illustrated by the Figure 7.4, and assuming that the most inner cache level L1 is separated in two for data (D-L1) and instructions (I-L1), one can have three possibilities:

1. **2 levels of cache:** one on-chip (L1), and the other one, L2, off-chip and possibly optional
2. **3 levels of cache:** 2 on-chip (L1, L2) and L3 off-chip
3. **4 levels of cache:** 3 on-chip (L1, L2, L3) and one off-chip (L4)

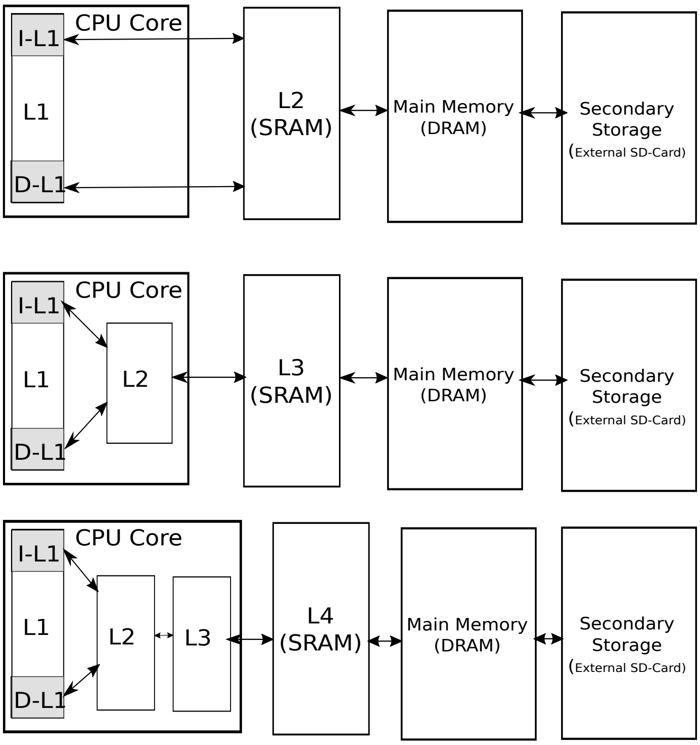


Figure 7.4: Possible Cache Levels

7.4.2 Cache Hits and Cache Misses

While the data to be processed is already in the cache, the CPU immediately uses this data: a so-called Cache Hit has occurred. On contrary, if the requested data is not yet in the cache, this is called a Cache Miss. As illustrated by the Figure 7.4, in this simplified interpretation one can notice that a cache miss costs twice more operations (refreshing the cache + providing the cached data to the CPU) than a cache hit.

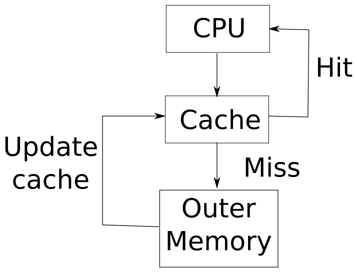


Figure 7.5: Illustration of a cache miss overhead compared to a cache hit

7.4.3 Types of Cache Misses

A cache miss occurs when the data that has to be accessed is not stored in the cache. Therefore it has to be looked up either in an higher cache level or in the worst case, it has to be loaded from the main memory. There are three well-known different types of cache misses and each of them illustrates the state of the cache prior to performing a given memory access.

Compulsory Misses (or Cold Misses). A compulsory miss occurs when the cache must miss because it does not contain the data that is requested. For instance, when a processor is first powered-on, there is no valid data in the cache and thus the first few reads will always miss. To overcome this, modern processors are able to identify patterns and proactively prefetch data a program will ask for. Listing 7.1 illustrates an example of predictable memory access pattern that causes the processor to prefetch data that will be accessed.

Listing 7.1 Example of predictable memory access

```
for(i = 0; i < N; ++i)
    tmp += data[i];
```

Conflict Misses. It occurs when the data that is requested was in the cache previously but got evicted. For instance, in a direct mapped 2-way set associative cache, it is likely that two data can be mapped to the same cache location, a cache miss occurs if the previously used data is overwritten by another data.

Capacity Misses. A capacity miss occurs when there is no more space left within the cache as the processor is trying to access too much data. A working set is the data that is currently used by a program. In case the working set size is larger than the cache, then frequent cache misses occur.

Coherence Misses. These are types of cache misses that are likely to occur due to the cache-coherence mechanism and the data sharing challenge that the processor has to face as it can handle multiple threads in parallel. e.g. a thread T_A that requires a given data must suffer a cache miss to obtain an updated version produced by a thread T_B as they are sharing the same data.

7.4.4 The Cache Associativity

Considering the fact that the CPU has to perform a great number of block reads and writes, therefore, the cache memory mapping also known as its associativity is key to ensure the speed. There are different types of mapping techniques:

Direct-mapped caches. The main idea is to associate a data block to a single cache entry. Consequently, the access time to that cache entry is very fast as there is no need to perform any cache lookup. However, the main drawback comes from the fact that, as the main memory is larger than the cache memory, this implies that it is very likely that a great number of data block will also need to use the same cache entry.

Fully-associative caches. Another mapping technique consists in having a cache where each cache line can hold a copy of any memory location. The main drawback regarding this implementation is that the cache lookup process is slower as the data can be anywhere within the cache.

Set associative caches. This technique is a compromise between the direct-mapped and the fully-associative implementation. The cache is then divided in N *sets* and an address is now linked to a *set* of cache lines instead of a single cache line.

7.4.5 Locating Data in The Cache

Data from the main memory (RAM) is transferred into the cache in blocks of fixed size instead of copying them bytes per bytes. A block is the minimum amount of data that the CPU can fetch from the main memory. The CPU can then store this whole block as a *cache line* which represents a single *cache entry*. The processor requires a memory address to lookup for the requested data within the cache. In practice, an address value can be split into three parts. For a 32-bit address it might look as in Figure 7.6. All of the three values are needed to locate the data in the cache. The *cache set* is used to determine which *cache set* the data should reside in. Then, for each block in the corresponding cache set, the tag of each data block is compared with the tag of the memory address. For the block where the data was found, considering a cache line size of 2^O, the low O bits of the memory address are used as a block offset to find the data within the cache line where the data was found.

7.4.6 Replacement Policy

On a cache miss, the cache controller must apply an heuristic to evict a cache line and refill it with the new information to cache from the main memory. As the number of memory blocks that map to a cache set is usually greater than the associativity of the cache, a so-called *replacement policy* must then decide which memory block to replace upon a cache miss. Most replacement policies uses the access history to decide which cache block to replace. ARM cached cores support two replacement policies:

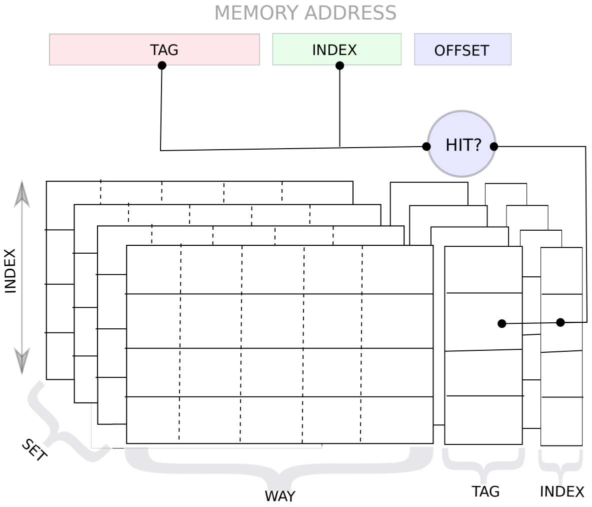


Figure 7.6: Cache Lookup within a N-associative (see 7.4.4) Cache Implementation

1. **Pseudo-Random**: randomly selects the next cache line in a set to replace. This cache line is also called the victim line. A pseudo-random counter is used for the whole cache by the cache controller.

2. **Round-Robin**: or also known as *FIFO* cache algorithm, it removes block in the order they were brought in the cache whatever their last access time.

7.4.7 Cache Writing Policies

While a write operation hits in the cache, two possible write policies exist. First of all, the data can be written both into the cache and in the main memory at the same time. This policy is called *write-through*. Another write policy called *write-back* policy involves writing the data in the main memory only when the line is removed from the cache. The cache lines are provided with a dirty bit in order to avoid useless back writings in the main memory. while it is set to 1, this means that the data has been modified in the cache, and thus it requires to be written back in the main memory on eviction.

7.4.8 Cache Allocation Policy on a Cache Miss

ARM caches may use two different strategies to reallocate a cache line while a cache miss occurs. The first one is known as *read-allocate*, and consists in allocating a cache line only during a read from the main memory. If the victim cache line contains valid data, then it is first written within the main memory before the cache line is refilled with new data. This write policy also implies that, a write operation of new data to memory does not update the contents of the cache unless a cache line was allocated on a previous read from the main memory. Moreover, if the cache line contains valid data, then a write operation would update both the cache and the main memory whether the cache write policy is *write-through* and if the data is in cache. Otherwise, only the main memory is updated. The second cache allocation policy is known as the *read-write allocate* policy, and involves refilling a cache line for either a read or write to memory. Therefore, any load or store operation performed on data that are not in the cache will induce a cache line allocation. The main difference is that, on memory reads the cache controller will use a *read-allocate* policy, otherwise, on a write, if the victim cache line contains valid data, then it is first written back to main memory before it can be refilled with new data from main memory. If the cache line is not valid, a cache line fill is directly performed. In case the cache uses *write-through* policy, it will update the main memory.

7.4.9 Inclusive versus Exclusive Multi-level Cache

On one side, we have the *inclusion* which is a cache property that implies the contents of lower level cache to be a subset of higher level cache. That is, in a two-level cache hierarchy, all the contents of L1 are a subset of L2, thus a cache hit in L1 guarantees a cache hit in L2. On the contrary, a cache miss in L1 does not imply a cache miss in L2. Therefore, whenever a block or line enters L1 cache, it must also be placed in L2 cache, and whenever the latter is removed from L2, its copy must also be removed from L1.

On the other side, we also can have exclusive (or non-inclusive) caches where there is no need to have a copy of a L1 cache line within L2. Therefore, if a cache miss or hit occurs in L1, there will be no impact in L2.

7.5 Summary

We gave the principles of ARM processor cache mechanisms and architectures and showed that a common processor implementation uses a *divide and conquer* strategy where the memory is split in different levels and takes advantage of the principle of locality (see Section 7.2) to hide the long access time to the main memory. Due to the architecture of the different cache memories, and the involved mechanisms related to cache operations, we show in the next chapter through practical experiments that it is possible to measure and especially to highlight the actual effects of the cache memories on a real mobile devices.

Chapter 8

Empirical Analysis of the Cache Memories Effects on a Mobile Device

Contents

We have seen in the previous chapter that ARM processors employ memory hierarchies that involve one or more cache memories in order to hide the large latency gap between the processor and the main memory. The details about such memory hierarchies are required in a large number of areas, such as static worst-case execution time analysis, cycle-accurate simulation, or in our case, cache timing analysis.

Unfortunately, sufficiently precise documentation of the logical organization of the memory hierarchy is sometimes not always publicly available. However, despite the documentation that is provided, ARM processors are highly configurable and the SoC (System-on-Chip) vendor and/or the system programmer may make the memory sub-systems do many different things depending on the end device features and needs. Thus to obtain a more-fine grained understanding of the micro architecture, we have decided to perform some micro benchmark measurements in order to highlight the micro architecture effects on the execution time.

In this chapter, we investigate measurement-based inference techniques to put in evidence the characteristics of the underlying micro architectures of an ARM processor embedded within an actual mobile device. The additional objective of this study is to be able to automatically characterize and calibrate parameters that are required to perform a so-called cache timing attack. Such kind of attack have been attempted and are further described in the chapter 10.

8.1 Introduction and Related Work

The execution time of many applications is mostly dominated by the cost of memory operations. Publications in the literature mainly study micro benchmarks algorithms and perform analysis to characterize the underlying micro architectures of processors in order to have detailed knowledge of their memory hierarchy parameters so they can propose more and more efficient techniques to optimize programs. Furthermore, there is a growing interest in self-tuning software that can optimize themselves for different platforms [YPS05, GDTF$^+$10, CS11, SW07, HFR13, Nus11]. One solution to this is to use micro benchmarks to determine the parameters of the memory hierarchy, such as the cache size, its associativity, the block size of data caches.

Krishnamurthy *et al.* described in [KCY98, ACK$^+$95] a "gray-box" approach to measure computers performance through the use of micro benchmarks. They specifically compared two supercomputers, the Cray T3D and T3E. They particularly tested the performance of the CPU cache memories. In the past, such kind of technique has already been used by Saavedra *et al.* and Smith *et al.* in [Smi82, SS95] to show that it is possible to characterize the execution time of any program on any machine. Saavedra and Smith papers are the most known regarding publications on memory characterization. By using a Fortran benchmark, they collect timing information in order to draw a set of curves that they interpret manually to determine the cache size, the associativity and the block size of first and second level data caches, as well as the size, the associativity and the page size of the TLB. Their technique has also been used to characterize the cache memories in GPU (Graphical Processor Unit) [VD08, PSAW09, WPSAM10, ZO11, BGDH12, MZLC14]. This technique is also known as the `P-Chase` (Pointer-Chase) micro benchmark, and the core idea is to traverse an array whose elements are initialized as the indices for the next memory access. The distance between two consecutive accessed array elements is called *stride*.

Drepper published in [Dre07] a reference article that describes fundamental concepts about memory and how the Linux kernel deals with the underlying hardware. More specifically, the author described through micro benchmarks how to reveal the behaviors of the micro architectures related to the CPU cache memories.

The experiments that have been conducted in this chapter are mainly inspired from the reference publications that have been cited previously. There are several tools that enable to perform automatic memory profiling, such as for instance `lmbench` [MS$^+$96], `perf` [pera] or `cachegrind` [cac], however we have decided to have a more fine-grained control over the test programs that we developed, in order to perform manual analysis of the results that we could collect in order to further understand the cache behaviors on the target mobile device that we have chosen.

8.2 On the Characterization of the Cache Parameters

8.2.1 Strided Memory Accesses

The most widely used micro-benchmark for such kind of measurements is Saavedra's benchmark [Smi82, SS95]. The technique basically consists in making fixed-stride accesses to the elements of a large array of size N. The iterations are performed with a different size N which is doubled every iteration. Furthermore, the accesses to the array are performed in steps of M which is also doubled every iteration. The average time per access is then measured. The kernel code that illustrates the test is as follows:

Listing 8.1 Timing measurements of strided memory accesses

```
for(N = L1_SIZE; N < (L2_SIZE + EXTRA); N *= 2)
     for(M = 1; M <= arraySize/2; M *= 2)
          for(i = 0; i < arraySize; i += M)
               LOAD A[i];
```

By measuring the average memory access latency of a great number of memory accesses, various cache parameters can be deduced from the array size and the stride value. Their simple test enabled them to characterize the cache by retrieving various values such as for instance, the size of the cache, its associativity, the size of a cache line, the penalties (in term of time) induced by cache misses, the existence of a prefetcher, and more. Furthermore, their test can also be used to retrieve the effects of the different TLBs, and to precisely characterize them by revealing for instance, the size of the memory regions that are covered by each TLBs through their virtual address spaces, the size of a memory page, the penalties due to TLB misses and so on. The Figure 8.1 illustrates an example of a graph that is composed of different traces that show the execution time variation (Y-Axis) according to the value of the stride (X-Axis).

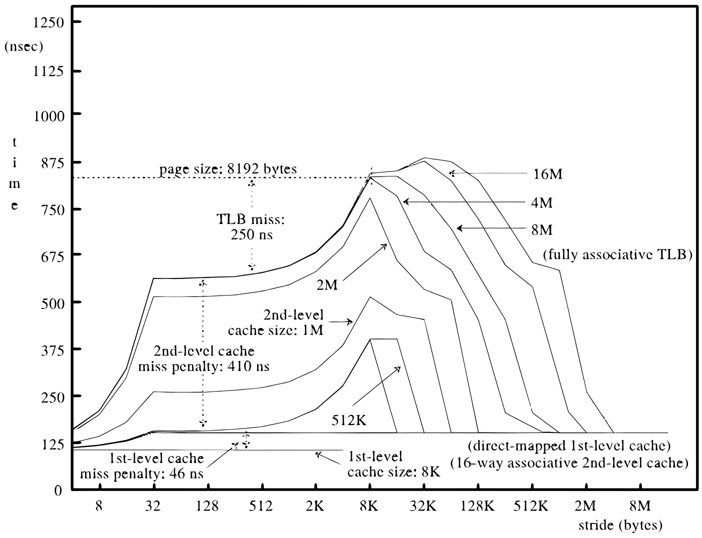


Figure 8.1: Strided memory accesses effects on the DEC ALPHA (source: Smith *et al.* [Smi82])

The effects of the strides on the memory access latencies can be explained as follows: for small strides we can observe low latencies because we have higher chances to read in the same cache block, and the prefetch mechanism is more effective in this case. Furthermore, for very large strides, we can observe huge drops in the latencies. This can be explained as the effect of the number of addresses that are being accessed which is smaller in this case. As a consequence, we have less cache misses due to the fact that we reuse data stored at less addresses and moreover we have less TLB misses as we access less pages. However, in between these two extremes, we can notice that the latency highly increases. This is mainly due to the fact that the number of memory access is high enough to make the caching mechanism more and more ineffective, and due to the strides, we have more and more TLB misses because we jump through many pages which badly increase the latencies.

Smith *et al.* describe four different scenarios that help in understanding what are the expected effects and how to produce them. The following notations are used in the next paragraphs in order to describe these scenarios:

- N, the number of elements (integers) in the array T
- W the size of an element
- $Miss$, a miss penalty
- I the number of iterations within a loop, with I_i the i^{th} iteration where $i >= 0$
- γ, the size of the stride used while accessing the array
- β the size of the cache
- θ the size of a cache line
- α the associativity of the cache

Scenario 1. In this scenario, they use an array of size N that does not exceed the size of the cache. Consequently, the array can entirely fit within the cache. Once loaded, they iterate through the array with a stride of γ. During the array access a simple operation is performed in order to cause a cache access such as for instance a simple addition. In this scenario, no cache miss will occur after the very first loading of the array within the cache regardless the value of each stride γ. The execution time T_{no_miss} per iteration will correspond to:

- reading a value within the cache
- perform an operation with this value
- write-back the value within the cache

Scenario 2.a. During this scenario, N is greater than the size of the cache β. The stride γ that is chosen must not exceed θ. We recall that due to the cache mechanism, accessing a single element within the array will induce the loading of n elements such as $n = \frac{\theta}{W}$. We have then n integers within each cache line. This implies that as long as the value of the stride γ is lower than the size of a cache line, we will have $\frac{\theta}{\gamma}$ number of accesses to the same cache line.

Scenario 2.b. In this case, the size of the array is still greater than the size of the cache. However, this time $\theta \leq \gamma < \frac{N}{\alpha}$. Every cache access will produce a cache miss as each element of T will be located at different cache line. Moreover, at the TLB level, the more γ is growing, the less we can benefit from the access to the same memory page. That is, as long γ grows, we will access less and less to the same page, which affects the execution time. Once γ reaches the size of a memory page, every access will be made on different pages.

Scenario 2.c. In this last experiment, N is still greater than β. γ will vary through the interval $\frac{N}{\alpha} \leq \gamma \leq \frac{N}{2}$. Even though N is greater than the size of the cache, we can note that $\frac{N}{\alpha} \leq \gamma$. This means that the number of address mappings will always be less or equal to the associativity of the cache. This implies that for an array of size N, only $\frac{N}{\gamma} \leq \alpha$ elements will be accessed. In the Scenario 2.b, we have seen that the execution time will vary according to the value of γ. Nevertheless, in the case of the Scenario 2.c, γ will be greater than the size of a page, therefore, we will have access to different pages each time, but as the number of access will be equal to $\frac{N}{\gamma}$, the more γ will grow, the less the number of accesses will be.

The Table 8.1 summarizes the four different scenarios:

Table 8.1

Scenarios	Array Size	Strides	Cache Misses Frequencies	Ratio Time/Iterations
1	$1 \leq N \leq \beta$	$1 \leq \gamma \leq \frac{N}{2}$	No misses	T_{no_miss}
2.a	$\beta < N$	$1 \leq \gamma \leq \theta$	1 miss every $\frac{\theta}{\gamma}$ elements	$T_{no\_miss} + \frac{(Miss \times \gamma)}{\theta}$
2.b	$\beta < N$	$\theta \leq \gamma \leq \frac{N}{\alpha}$	1 miss every elements	$T_{no\_miss} + Miss$
2.c	$\beta < N$	$\frac{N}{\alpha} \leq \gamma \leq \frac{N}{2}$	No misses	T_{no_miss}

8.2.2 Sequential and Random Memory Accesses

Another experiment has been attempted with the aim of keeping the P-Chase technique principle (see section 8.1) but without performing strided memory accesses anymore. That is, the array elements are initialized as the indices for the next memory access to produce the access pattern illustrated by the Figure 8.2.

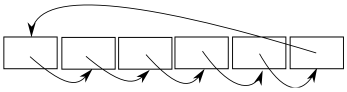


Figure 8.2: Access pattern according to the P-Chase technique

The idea is to analyse the behavior of the processor while performing sequential or random memory accesses. The experiment consists in reading sequentially elements of an integer array T of size N. This is executed I times, and each run I_i is timed. The program is run multiple times and takes as a parameter N and I which will vary such as when N is multiplied by two, I is divided by two in order to balance the work. The initial values N and I are respectively, 1 and $\lambda = \rho \times 1024 \times 1024$. This means that during the first execution of the program, a single element of the array is accessed λ times and during the very last run, ρMB of data are read sequentially only one time. What we want to observe here is the execution time evolution as we are accessing the array T with its size growing every run until it overflows L1 and L2 cache capacities. The Listing 8.2 illustrates the kernel code that is used to perform the experiment.

Listing 8.2 Timing measurements of sequential acccesses of an array

```
N = 1
I = λ

while(I >= 1)
{
    //Init the array
    for(j = 0; j < N; ++j)
        bloc[j] = j+1;
    bloc[j] = −1; //end of the array

    double time0 = get_time();
    for (j = 0; j < I; ++j) {
        i = 0;

        //perform sequential reads until we reach the end of the array
        while(i != −1) {
            i = bloc[i];
        }
    }
    TIME = get_time() - time0;
    LOG(TIME);
    N *= 2;
    M /= 2;
}
```

The second test consists in doing the same experiment but with random memory accesses to T as illustrated by the Figure 8.3. The kernel code of this test is shown in the Listing 8.3.

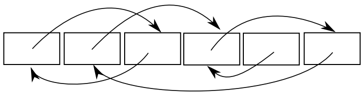


Figure 8.3: Example of Random Access pattern

Listing 8.3 Timing measurements of random acccesses of an array

```

N = 1
l = λ
int *bloc;

int idx, i, j;

while(l >= 1)
{
    idx = 0;
    i = 0;

    //allocate memory
    bloc = calloc(N, sizeof(int));

    //init the array
    for(idx = 0; idx < N; ++idx)
        bloc[idx] = -1;

    //fill the array with indexes values
    //that are randomly chosen
    //leave the N-1 as it defines the last element
    idx = 0;
    i = 0;
    while(i < N-1) {
        bloc[idx] = rand() % N;

        while(bloc[bloc[idx]] != -1) {
            bloc[idx]++;
            if(bloc[idx] == N)
                bloc[idx] = 0;
        }
        idx = bloc[idx];
```

```
            ++i;
    }

    double time0 = get_time();
    for (j = 0; j < l; ++j) {
        idx = 0;

        //perform random reads until we reach the end of the array
        while(idx != -1) {
            idx = bloc[idx];
        }
    }
    TIME = get_time() - time0
    LOG(TIME);
    N *= 2;
    l /= 2;
}
```

8.2.3 Dealing with Noise

As we do not claim to use the most accurate algorithms to perfectly highlight the cache parameters, we have identified different sources of measurement errors that we took into account in order to improve at best the clarity of the collected measurements.

Loops overhead. First of all, as we can see in these algorithms the collected timings also include loop operations. In order to overcome this issue, we carefully measured the overhead introduced by each loop structure so it is subtracted to the final execution time.

Interferences induced by the Operating System (OS). Android OS and the underlying Linux Kernel cause many interferences with user space processing due to different events such as for instance, scheduling mechanisms, interrupts, timers and other events. Consequently, it is likely that the code sees execution being delayed for no discernible reason and a variance in the execution time due to cache memories pollution. Two different strategies can be adopted to minimize these interferences:

- **Avoid being Preempted:** To minimize these interferences, we forced the programs to run on a single core by setting the CPU affinity using the system call `sched_setaffinity`, so our program will not be moved from one core to another one. This does not mean that no interrupts will occur, however the amount of interrupts will be minimized.

- **Avoid Frequency Scaling:** For saving energy, small devices like smartphones highly rely on the frequency scaling mechanism to dynamically reduce their energy consumption by changing the frequency of some of the CPU cores. We decided to take into account the possible noise it can introduce, as the scheduler may decide to badly decrease or increase the frequency of the core on which we are running our test. To overcome this we decided to set the CPU frequency scaling governor to the *performance* mode so the frequency is always at the highest rate.

Accurate Timing Function. As we wanted to get the most accurate timing in a portable way, we decided to use the `clock_gettime(CLOCK_MONOTONIC, ...)` POSIX function within our function `get_time()`, as it provides nanosecond resolution, and it is monotonic, *i.e*, it is not affected by changes in the system time-of-day clock. Furthermore, the overhead that includes invoking `clock_gettime` and returning its nonce has been also measured and subtracted from the final timing. The Figure 8.4 depicts the two timings T_1 and T_2 that are measured before and after the execution of the function X. We have then the execution time of X noted $T_i = T_2 - T_1$. However T_i is not accurate enough due to the overhead of calling `clock_gettime`. By calling the latter two times, we can compute the overhead as $T_O = T_2^{'} - T_1^{'}$. Consequently, the actual execution time of X can then be noted $T_X = T_i - T_O$.

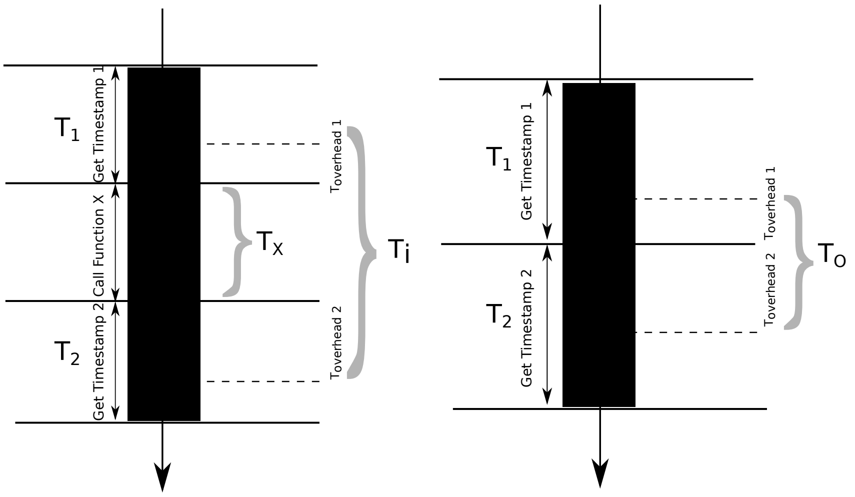


Figure 8.4: Illustration of the overhead induced by the function that is responsible of returning the timestamp to collect.

Avoiding Code Optimization. As we control the compiler (`GCC 4.9`[20]) that is used to compile our programs, we disabled code optimizations, and the assembly code was inspected in order to ensure that no code optimization was applied.

8.3 Measurement of Cache Effects on an ARM Processor

The device that has been used during these experiments was a Samsung Galaxy S3 which embeds an ARM Cortex-A9 processor.

Device	Processor	OS	Caches			
				Size	Associativity	Line Size
Samsung Galaxy S3	Cortex-A9 (Quad Core)	CyanogenMod 10.1.3	L1	32 KB	4-way	32 B
			L2	1 MB	4 to 16-way	32 B

[20] See the ARM-based toolchain provided with the Android NDK https://developer.android.com/ndk/guides/standalone:!_toolchain.html

8.3.1 Resulting Traces

The following Figures show the resulting traces of the different experiments that we have explained previously. The analysis of the various observations that can be made are presented in the section 8.3.2.

The Figure 8.5 depicts the time variation (Y-Axis) during the sequential and random reads of 2^N elements where N varies. The Figure 8.6 illustrates two examples of experiments that involve strided memory accesses with different array sizes. The Figure 8.7 and Figure 8.8 are zoomed versions of the Figure 8.6.

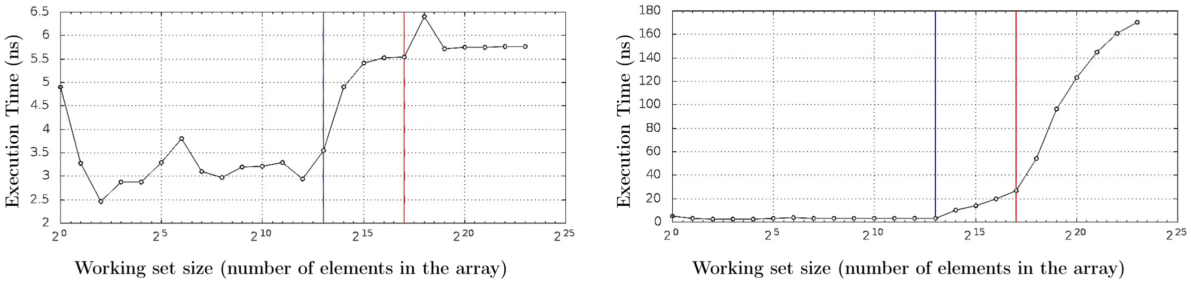


Figure 8.5: Execution time variation during the Sequential (Left) and Random (Right) Memory Accesses experiment with different working set size

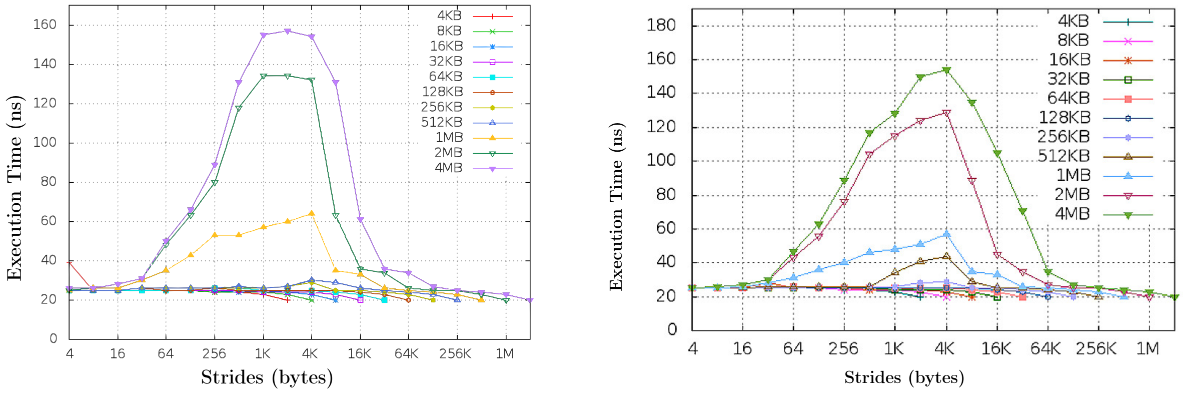


Figure 8.6: Execution time variations while performing memory accesses at different strides and array sizes

8.3.2 Observations and Analysis

Cache Misses and Hits Characterization. Observing the Figure 8.6 and 8.10, we can observe that at the very beginning the $4KB$ trace has a latency of ∼40 ns before to start benefiting from the L1 cache which can also be observed as a drop of the execution time to ∼ 25 ns. One can assume that they respectively corresponds to the execution times that characterize the penalty due to a cache miss, and the performance gain (∼15 ns) due to the cache mechanism as the data trace always fit within the L1 cache.

The Cache Level Capacity Effects. If we take into consideration the graphs in the Figure 8.5, one can divide both graphs in three different parts according to the location of the data in

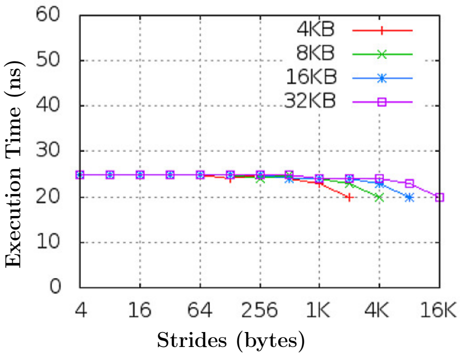


Figure 8.7: 4 traces that illustrates the Scenario 1 (see paragraph 8.2.1)

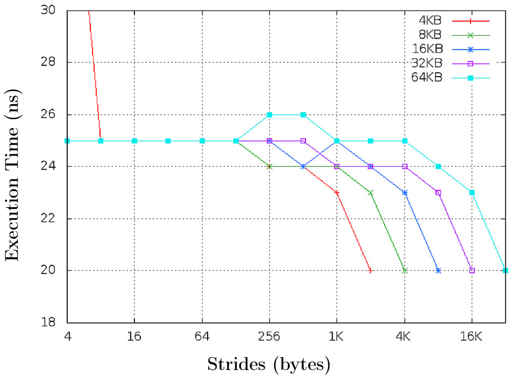


Figure 8.8: Execution time variations that are used as a complement to Figure 8.5 to highlight L1 cache size, the TLB effect, and the L1 access time during a cache hit

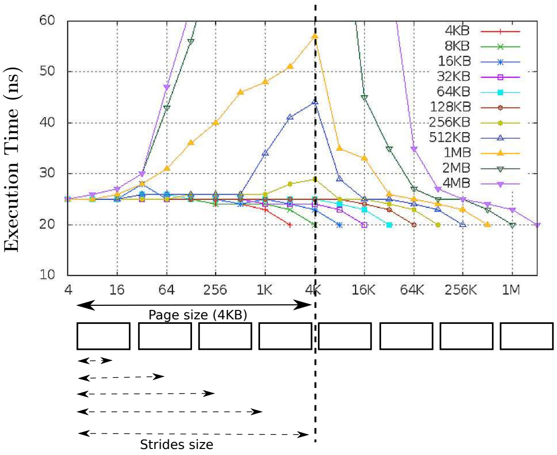


Figure 8.9: Experiment that is used as a complement to Figure 8.5 to highlight L2 cache size, the size of cache line, the TLB effect

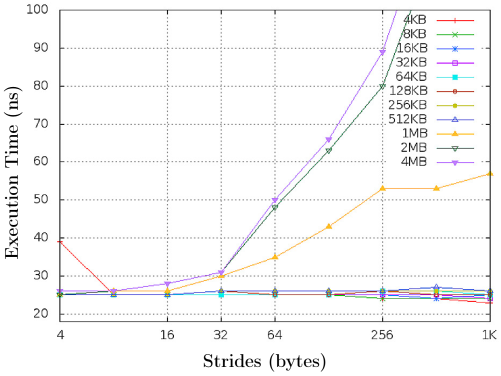


Figure 8.10: Experiment that is used as a complement to Figure 8.5 to highlight the cache line size, L2 cache size, TLB effect, the main memory access latency

memory during the processing: whether it fits within one of the different cache level, or it is in the main memory.

- **Level 1 (L1):** de 2^0 à 2^{13}
- **Level 2 (L2):** de 2^{14} à 2^{17}
- **Level 3 (main memory):** de 2^{18} à 2^{23}

The transition between two levels can be interpreted as being a sudden growth of the execution time due to the latencies induced by the underlying memory architecture in which the data is accessed from. During the transition to the Level 2 the size of the array exceeds $2^{13} = 8192$ integers. An integer on the current architecture is encoded with 4 bytes, we have then $32KB$ of data which is the size of L1. The same logic applies to the transition to the Level 3 while the size of the array exceeds the L2 cache size, *i.e*, $2^{17} = 1MB$. If we consider now the experiments with strided memory accesses, we can confirm this with the Figure 8.10. Indeed, we can see that the very first trace that has an increasing access time (above 25 ns) is the trace whose array size that is used for the experiment is $1MB$. This mainly highlights the limit of L2's capacity. Moreover, we also can observe that a group of traces are always flat until the latencies benefit of a sudden decrease (as any other trace at the end). By isolating this group of traces (see Figure8.7) we have as a result, 4 traces that never exceed about 25 ns of latency. As the 4 arrays that are used during these tests can fit within the L1 cache, we can assume that we are in the Scenario 1. Among other things, this also highlights the size of L1 as above the $32KB$ trace, the very first increase in latency occurs for the $64KB$ trace (see Figure 8.8). However as we can see in the Figure 8.8, despite the slight increase for the trace $64KB$ while the stride size is equal to $256B$, we also can observe that even with an array of size bigger than the L1 cache capacity, the latency is still low compared to traces above the $256KB$ one (see Figure 8.5, graph on the right). Indeed, this can be observed for the traces starting from $4KB$ to $256KB$. This behavior can be explained as the processor's prefetch mechanism and the TLB effect. This is explained further in the next paragraphs.

The Prefetcher Mechanism Effects. The execution time is higher in the second experiment (see Figure 8.5). Even though we are loading data from the main memory, the measured latency l is $5.5 \leq l \leq 6.5$ ns. However in the Figure 8.5 the latency is always growing, and moreover the latency of the main memory access is $30 \leq l \leq 160+$ ns which is an increase of $\sim 445\%$ at least. This behavior is mainly due to the prefetching mechanism which is working to a disadvantage in the case of the experiment with random memory accesses. Indeed during a sequential memory access, in anticipation of using consecutive memory regions, the processor is able to predict and prefetch the next cache (32 bytes) line. Consequently, when the next cache line is actually used, it is already halfway loaded. According to the Cortex-A9 Technical Reference Manual[21] the prefetcher is able to monitor cache misses done by the processor and to prefetch two independent data streams at once. Therefore, as we can see this prefetch mechanism significantly helps in reducing memory access latencies.

The Cache Line Size Effects. While the size of the array is equal to 2^3 there is an increase in the memory access latency (see Figure 8.5, left graph). This is mainly due to the size of the L1 cache line which is equal to 32 bytes. Therefore, this illustrates the penalty induced by the fact that during every run that involves a data set of size s such as $s \geq \frac{\beta}{\theta}$ where θ is the size of a single cache line, and W the size of an integer, then, accessing the i^{th} integer that does not fit in the current prefetched L1 cache line will induce a penalty of about ~ 2.8 ns. This can also be observed in the Figures 8.6, 8.9 and more particularly the Figure 8.10 as we reach a stride size equals to $32B$, there is a sudden increase of the access time.

The TLB Effects. We have already observed that in the Figures 8.5, 8.8, 8.9, 8.10 that traces from 64KB to 256KB benefit of low latencies similar to latencies of traces from 4KB to 32KB. This can mainly be explained as the result of the Scenario 2.b (see section 8.2.1). Above the trace 256KB the other ones are properly distinguishable. As the stride size is increasing, we will access less and less to the same page, and we can see that the access time is maximum for a stride size equal to 4KB which is actually the page size (see Figure 8.9). To some extent, the overhead of TLB misses that occur for smaller stride sizes is amortized, as we have more than one data points that are accessed per page. Finally, we can see that above a stride greater than

[21] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/CHDIGCEB.html

4KB, we have a huge drop in the latencies. This can also be explained as the effects induced by the TLB usage by the algorithm (see Figure 8.1). Indeed, due to the algorithm that is used, each run will not benefit the same number of page accesses. In the innermost loop, we will have in general $arraySize/M$ accesses. Moreover, we will also have $arraySize/pageSize$ unique pages that will be accessed. Therefore, we will have different unique pages that will be accessed for different runs. For instance, for the run $arraySize$ = 2M and $stride$ = 2KB, we will have 1024 accesses in the innermost loop, but only 512 × 2KB accesses to the main TLB. However, for a 4KB stride, we will still have 512 unique pages that are accessed, but 512 × 4KB main TLB accesses. Now, for $arraySize$ = 2M and $stride$ = 16KB, we will only have 128 unique pages accesses. This explains why the more the stride grows, the less we will access to pages. TLB misses will still occur, but as we access less and less pages during these runs, the latency will greatly drop compared to the other runs.

8.4 Summary

We have investigated measurement-based inference techniques to put in evidence the characteristics of the underlying micro architectures of an ARM processor embedded within an actual mobile device. We have also briefly showed that memory profiling tools already exist to automatically retrieve other parameters such as for instance, memory or interprocess communication bandwidths and more. However, we wanted to perform this study as a preliminary one to understand through practical experiments the particular effects of the cache memories on the execution time on an actual mobile device. In addition to the available ARM documentation, the visual and manual approach to infer the cache parameters turned out to be a good exercise to have a more fine-grained understanding of the actual cache behaviors. With distance and hindsight, as a future work, we still have to quantify and characterize the system noise in order to understand and minimize their effects on the resulting timing measurements.

Chapter 9

Introduction to micro architectural Attacks

Contents

We have seen in the previous chapter that the processors use different kind of cache memories to reduce the access time to data from the main memory. We also showed that due to the implementation specificities of those cache memories, it is possible to highlight their characteristics and their effects on the execution time of a given program by measuring the timing variations. A type of attacks known as Cache Timing Attacks or also known as Cache Attacks have been designed to exploit the cache effects on the execution time.

In this chapter we are going to present the different types of Cache Attacks (Section 9.2), and the different techniques that are used in the literature to extract and analyse the timing information that can be gathered (Section 9.3). However, besides cache memories, other resources and design techniques also known as the micro architectures of the processor, exist and can also be exploited to gain information through attacks known as Micro Architectural Attacks. We will introduce them in the Section 9.1 before to fully focus on Cache Attacks.

9.1 Micro architectural Attacks

Micro architectural Attacks (MAA) can be considered as a special form of Side-Channel Analysis. By definition, Side-Channel Analysis consists of any analysis/attack based on information gathered from the physical implementation of a target operation. It can be for instance, timing information, power consumption, electromagnetic emanations, sound, etc.

Indeed some software implementations can leak sensitive information because of the physical properties and requirements of the implementation under analysis, and also the computational environments.

In the case of MAA, it mainly exploits micro architectural functionalities of processor implementations to compromise the security of computational environments even though sophisticated protection mechanisms are present (e.g. as sandboxing or virtualization).

According to the hardware components used to obtain a side-channel information, MAA actually exploits timing and access variations caused by those components. Basically there are currently four types of MAA in the literature.

9.1.1 Data Cache Timing Attack (DCTA)

Also known as Cache Attack, it is the most mature type of attack in MAA as numerous attacks have already been proposed by researchers. Consequently, this type of attack is going to be particularly studied in this thesis, in the context of mobile devices enviromnent. As a summary, Cache Attacks tend to infer information from the execution time variation of a program or a subpart of it, by analysing the cache effects on the execution time of the running code. By analysing those timing variations it is possible to reveal for instance, the data access patterns or the execution flow of a given algorithm. Further investigation on this type of micro architectural attack is made in the Section 9.2.

9.1.2 Instruction Cache Analysis Attack (ICAA)

Aciiçmez proposed [Aci07] and succeeded [ABG10] in performing this type of attack that exploits the functionalities of the instruction cache (I-cache). The I-cache is very useful for the processor to store recently used instruction codes in order to reduce the average time to read them from the main memory. The main idea is to run a spy process and a target one, more or less simultaneously. The spy process (noted S) must evict instructions of the target (noted F) function from I-cache. For this purpose, it has first to know which I-cache sets F is using. Consequently, S must also know the logical address mapping of the whole code of F in order to deduce the I-cache sets to target. Let A be the number of associativity of the I-cache, a certain amount N (with $N \geq A$) of "dummy" instructions are then executed in order to fill these I-cache sets. The I-cache would be set within a known and controlled state. F is executed in order to replace some of S's cached instructions. S is then run again, but this time, the overall execution time (noted T) of all of the dummy instructions is measured. Hence, the following heuristic enables to determine if F has been executed or not: $T \geq Threshold$ then it means that some instructions of S have been evicted.

9.1.3 Branch Prediction Analysis

A Branch Predictor is also a key-component in the design of microprocessors as it also enables to enhance the execution time of the code that is being run. It is a mechanism that resolves hazard by predicting which path in the code will be taken while the processor has to handle a *branch instruction* (e.g. if-then-else, goto, for or while loops, etc.). It can use two special types of cache memories: a Branch Prediction Buffer (BPB), and a Branch Target Buffer-BTB). The main differences is that the BPB only contains prediction about whether the next branch will be taken or not. However it does not store the actual predicted Program Counter (PC) that points to a branch instruction. The BTB stores both the most recent used branch target addresses and the predicted PC. Consequently, such mechanism will also potentially introduce some latencies in the execution time in the case the prediction is not correct. If a *misprediction* occurred, then more clock cycles are consumed for additional operations so the processor can execute the right path. Aciiçmez demonstrated [AKS06] the first *Simple Branch Predicion Analysis* (SBPA). He proposed various examples of possible attacks and he attempted two of them on a modified[22] version of the RSA cryptographic algorithm from the OpenSSL library (version 0.9.7e) as Proof-of-Concepts to reveal the execution flow of the running algorithm by observing the BTB state transitions.

9.1.4 Shared Function Units Attack (SFUA)

It is common for processors nowadays to have different hardware *functional units* (FUs) (e.g. Arithmetic Logic Unit, Floating-Point Unit, Multiplier Unit, etc.) that may be used in parallel and potentially shared by different threads to perform operations and calculations as instructed by the latter. Let a pool of FUs shared by different threads noted Shared Function Units (SFUs). Aciiçmez *et al.* explained [AS07a] that by contending for the SFUs, a spy process can interfere with another one and induce a side-channel information leakage. The spy process is run in parallel with the target process and it would continuously measure the execution time T of a number of dummy multiplication instructions until $T > Threshold$. Indeed, as both processes race to occupy the shared multiplier unit, latencies may occur. The latency induced by this behavior can then be characterized with a $Threshold$ so it can be detected which thus leads to side-channel information.

9.2 Cache-based Attacks

Cache-based attacks basically exploits the cache behavior of a cryptosystem by getting the execution time and/or the generated power consumption variations due to cache hits and misses. In the literature there are three different categories of Cache Attacks: *time-driven attacks*, *trace-driven attacks*and *access-driven attacks* .

9.2.1 Time-Driven Cache Attacks

During a Time-Driven attack, an adversary is able to observe the overall time needed to perform certain computations, such as whole encryptions. From these timings he can make inferences about the overall number of cache hits and misses during an encryption.

9.2.2 Trace-Driven Cache Attacks

In Trace-Driven attack, the adversary is able to obtain the traces of cache hits and misses for a sample of encryptions and recovers the secret key of a cryptosystem using this data. A "trace"

[22] Simple RSA implementation with Square-and-Multiply exponentiation and Montgomery Multiplication implementation with the followings modifications: CRT mode (Chinese Remainder Theorem), blinding have been turned off, Window size has been changed 5 → 1

is a sequence of cache hists and misses. e.g. M H H M, H M M H, etc. The attacker has to be able to profile the cache activity during a single encryption. Furthermore he has to know which memory access of the encryption algorithm causes a cache-hit.

9.2.3 Access-Driven Cache Attacks

In Access-Driven attack the adversary is able to get more fine grained information about the cache behaviour. He should be able to determine the cache sets accessed by the cipher. For instance let a spy process S and a cryptographic victim process V run concurrently. They use the same cache. After letting V run for some amount of time and potentially letting it change the state of the cache, S observes the timings of its own memory accesses, which depend on the state of the cache.

Access-driven and Trace-driven cache attacks rely on more sophisticated knowledge about the implementation and the underlying hardware architecture. However, access-driven and trace-driven attacks require far less measurement samples than time-driven attacks. It seems that Trace and Access-driven attacks are highly platform dependent while Time-driven attacks are portable to different platforms.

9.3 Information Extraction Through Cache Attacks

9.3.1 Cache information Extraction Techniques

9.3.1.1 Evict+Time

This technique was first introduced [OST06] by Osvik et al. and aimed at retrieving the private key that is used during an AES encryption. This technique thrives on the fact that:

1. the attacked AES encryption is based on the T-table implementation
2. the attacker must have the ability to trigger the encryption with his own plaintexts
3. he also has to know when the encryption started and when it ended
4. he should have the ability to get knowledge of the virtual memory address of each table he wants access to

In the case of (1), the leakage model is based on the relation between the index used to access the lookup tables, and the key. Let a lookup table noted T_l, and i an index used to access a value within T_l. In the T-table implementation of AES, we have the following relation that depicts the information leakage: $i = Plaintext \oplus Key$. Since the table index depends on the key, this leaks information about the key. Let $V(T_l)$ be the virtual address of T_l and W is a given memory address. Let S, B, $SizeOf(T_l(i))$ and δ are respectively the number of cache sets, the number of bytes a cache line can hold, the size of an element of T_l and the number of element that can be stored within a single line such as $\delta = B/\ SizeOf(T_l(i))$.

The Evict+Time technique consists of 3 steps:

- Trigger a victim process that would fill a cache set with its data
- The attacker Evicts data from some memory addresses congruent to $V(T_l) + i \times B/\delta \mod S \times B$, $i.e$, from a given cache set
- Time the total execution time of the victim process while executed a second time

9.3.1.2 Prime+Probe

As the previous technique, this one was also presented by Osvik *et al.* It aims at discovering the set of memory blocks read by the encryption a posteriori, by examining the state of the cache after encryption. By performing a single encryption on a given plaintext P, they gather measurement scores (timing) for each elements of each tables.

This technique is split in three steps, but first one needs to allocate a contiguous byte array $A[0,...,(S \times W \times B - 1)]$ with start address congruent $\mod S \times B$ to the start address of T_0.

- Read a value from every memory block in A (Prime)
- Trigger an encryption of P
- Time the memory access of A such as for each table l and element indexes y respectively varying such as $l = 0,...,3$ and $y = 0, \delta, 2\delta, ..., 256 - \delta$, the memory addresses of $A[1024l + 4y + tSB]$ are read with $t = 0,...,W - 1$.

9.3.1.3 Flush+Reload

It was first proposed [YF14] by Yarom *et al.* and is an extension of Gullash et al.'s work [GBK11]. They demonstrate that due to a weakness in the Intel x86 processors, page sharing could expose processes to information leaks. Their attack aims at exploiting this weakness to monitor access to memory lines in shared pages by targeting the Last Level Cache (LLC).

Actually they noticed that the processor's instruction $cflush$ evicts the memory line from all the cache levels, including from the shared LLC.

Unlike the prior attacks that target the First Level Cache, their attack does not require the attack program and the victim to share the same execution core.

They run their attack between two unrelated processes in two different environment configuration: in a single operating system and then in separate virtual machines.

The attack is split in three phases:

- The spy process flushes one or more desired memory locations from the cache
- then it waits for the victim to access the evicted memory lines
- finally, the spy reloads the memory line then measures the time to load it

9.3.1.4 Prime+Trigger+Probe

Ristenpart et al introduced in [RTSS09] this cache usage measurement technique variant of the Prime+Probe that has been described previously in 9.3.1.2.

Their technique supports the setting of time-shared virtual machines, $i.e$ the shared computing resources among the different virtual machines instances.

To perform their cache usage measurement, one should first allocate a contiguous buffer $buff$ that should be large enough to fill a significant portion of the cache. Let B be the cache line size. The three-steps-attack is as follows:

- $buff$ is read at B-byte offsets in order to ensure it is cached
- a loop must keep the CPU busy until its cycle counter jumps by a large value
- finally, the time it takes to read $buff$ at B-byte offset is measured

The challenge for this type of attack is the ability to target a specific set without any shared memory.

9.3.1.5 Evict + Reload

Recently, Moritz *et al.* described in [LGSM15] a new access-based cache attack that aims at targeting the Last-Level Cache for enabling a cross-core attack on Android mobile devices. Their technique is derived from the *Flush+Reload* 9.3.1.3 and also inspired from the *Evict+Time* 9.3.1.1 and *Prime+Probe* 9.3.1.2 techniques. The main challenge on ARM-based device, is that there is no specific instruction for flushing an arbitrary cache line. Therefore, the authors had to rely on memory accesses for evicting data from the cache.

This techniques consists of the following steps:

- map the memory of a shared library used by the victim process
- get an address A from the mapped memory and evict it from the cache
- wait for the victim process to be scheduled and, hopefully, it would load again A in cache
- access to A and determine if a cache hit occured or not by measuring the access time

9.3.1.6 S$A Attack

This technique [AES15] is a derivative of the *Prime+Probe* 9.3.1.2 attack and can also be noted as *Prime+Prime+Probe*. It mainly takes advantage of the usage of huge size pages for getting information about the accessed cache set, and the fact that the targeted CPU implements an inclusive Last Level Cache (LLC).

It consists of the following steps:

- allocate huge size pages, and get access to them for analysing the lower k bits of the virtual address
- (Prime) fill a given cache set A in the LLC and subsequently, evict data from the upper level caches, in a given cache set B. As the attacker can control the virtual address, he can target a specific cache set in the LLC and in the upper level caches.
- (Prime) target and fill a different cache set $Corr$ in the LLC, and ensure it evicts data from the same cache set B in the upper level caches
- run the victim process and monitor the access to A. If it is used, it will be evicted and thus stored in memory, otherwise they will remain in the LLC. Time variations can then be measured in either cases.
- (Probe) measure the time to access to the cache lines in the cache set A.

9.3.1.7 Flush + Flush

In [GMW15], Gruss *et al.* introduced this new technique in response of cache attacks detection. Indeed, they show that it is possible to detect the *Flush+Reload* and *Prime+Probe* technique by monitoring the performance counters, the cache references and cache misses of the last-level cache. Their technique mainly relies on the execution time of the *clfush* instruction, which is shorter if the data is not cached, otherwise it is higher because it has to evict a cache line from all the cache hierarchy, which obviously takes more time. By contrast with the previous attack, their technique is not affected by the prefetecher as there is no data reload step. However, the attack has a lower accuracy than the *Flush+Reload* technique, there is a lower timing difference between a hit and a miss.

9.3.2 Cache Attacks Applied to ARM platforms

In the literature, Weiss *et al.*were the first to experiment [WHS12] time-driven cache attack on an ARM platform. The attack targeted an AES running inside a L4Re VM on an ARM Cortex-A8 single-core CPU with a `Fiasco.OC` microkernel. Their work particularly shows the possibility to extract finer grain information from a co-located VM.

Spreitzer et al. evaluated time-driven cache attacks on mobile devices [SP13b], and they particularly show that T-table based implementations of the AES leak enough timing information on these devices so it is possible to recover parts of the secret key. Instead of using lab testbeds, they run their attack on state-of-the-art Android-based mobile devices. Their work shows that it is possible to gather some possible key candidates using Bernstein's correlation phase though. Therefore one clearly observes that timing information is leaking. However as they claimed, the number of remaining key bits is still too large for an exhaustive key search. Indeed the remaining key-search phase corresponds to 2^{58} AES encryptions which is impractical. They tried to consider a first-round attack but the AES key space could not be reduced enough in order to perform an efficient exhaustive key search. The authors tried to mount their attack in a realistic scenario where applications (videos, slide-show) were launched on the mobile devices in order to try to affect the cache. But it did not help in leaking more information and hence did not further reduce the key space. The authors improved [SG14] the key enumeration phase in order to reduce the key-search complexity. They also demonstrated [SP13a] that an access-driven cache attack that focus on the memory access patterns of disaligned T-tables is feasible, and enables to reveal the secret key without a single brute-force computation.

Recently, Lipp *et.al* [LGSM15] experimented access-driven cache attacks on an ARM Cortex-A53, and they particularly focused on exploiting the inclusive property of the shared last-level cache to perform a cross-core cache attack.

9.4 Summary

In this chapter we have explained the concepts behind micro architectural attacks, and we have also reviewed the different types of cache-based attacks, and the different techniques that exist to extract information through the analysis of the cache behavior. Most of the attacks in the literature are related to `x86` platforms. Since a couple of years, researchers started to also investigate the applicability of this kind of attack on ARM platforms, and more particularly on mobile devices. While access-driven cache attack enables to extract more fine-grained information, it appears that time-driven cache attacks are also practical on mobile devices.

In the next chapter, we are going to focus more on this type of cache attack in order to investigate their applicability.

Chapter 10

Experimenting Time-Driven Cache Attack on Real Mobile Devices

Contents

We have seen that cache attacks can be divided into three different categories: time-driven, access-driven and trace-driven cache attacks. While access-driven and trace-driven cache attacks require fine-grained information leakage such as for instance, memory accesses, time-driven cache attacks are more generic and do not necessarily rely on a precise knowledge about the attacked software or the underlying hardware micro architectures.

A work of reference in the area of time-driven Cache attack is Bernstein's cache timing attack targeting the AES [Ber05]. Neve *et al.* experimented Bernstein's work in [NSW06] on different processors: Pentium III E, Pentium 4, Pentium M735, Opteron 246. Salembier [Sal] attempted the attack on different AMD CPUs in order to have an evaluation of the total amount of time to run the entire attack. Weiß et al. adapted in [WHS12] Bernstein's attack in a ARM-based platform running an L4 microkernel as virtualization layer. Spreitzer *et al.* were the first who attempted in [SP13b] Bernstein's attack on three different actual mobile devices: Acer Iconia A510 tablet computer, a Google Nexus S, and a Samsung Galaxy S2.

Up to now the algorithm that has been the most targeted to demonstrate the feasibility of time-driven cache attacks is the AES (Advanced Encryption Standard). Nevertheless, from a general point of view, cache attacks are not limited to cryptographic algorithms as long as the targeted program implementation makes use of structures in memory (e.g. arrays) that involve the cache mechanism. Consequently, we have also decided to take an implementation of the AES that makes use of arrays in order to investigate further on the feasibility of time-driven cache attacks on mobile devices. By means of statistical analysis, the best result regarding the reduction of the brute-force effort to recover an AES-128 private key is about $\sim 2^{46}$ tests [SP13b]. From the study of the literature, we have noticed that little efforts have been put in showing from a practical point of view, the means to collect the most precise timing information and with the least privileges possible. Moreover, nobody concretely show the device configurations an attacker can set up to reduce at best the sampling noise while collecting the necessary timing information to carry out a time-driven cache attack on a real device.

Considering that we have targeted other devices (Samsung Galaxy S3 and S4) than those that have been already studied in the literature, our main purpose is to highlight through practical experiments, whether or not Bernstein's time-driven cache attack is able to make information leak enough.

10.1 Introduction to AES

In this chapter, we are going to mainly focus on the AES cryptographic algorithm. The AES, originally known as Rijndael, is a symmetric-key block cipher algorithm that is divided in different parts. The `key schedule` that is in charge of generating a succession of keys, that will be used within successive repeated round transformations. The plaintext, $p = \{p_0, \cdots, p_{15}\}$, the ciphertext $c = \{c_0, \cdots, c_{15}\}$ and the initial key k, can be 128-bit, 192-bit or 256-bit long. In this thesis we only consider 128-bit key, hence $k = (k_0, \cdots, k_{15})$. The latter is expanded into 11 round keys, such as a single round key $K_r = (K_r^0, \cdots, K_r^{15})$ where $r \in \{0, \cdots, 10\}$ with $k = K_0$. As depicted by the Figure 10.1, an encryption with AES is computed as follows. The initial round key is first added during an `AddRoundKey` operation, then 10 successive rounds composed of the operations `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` are applied to an intermediate 16-bytes *state* s_r which is defined as $s_r^x = (s_r^0, \cdots, s_r^{15})$.

The initial state s_0^x is the result of the very first `AddRoundKey` such as we have

$$s_0^x = p \oplus k_0 \tag{10.1}$$

During the i^{th} round, s is then computed as follows:

$$s_i^x = p_i^x \oplus K_i^x \tag{10.2}$$

On 32-bit systems (or larger), it is possible to speed up its execution by combining the `SubBytes` and `ShiftRows` steps with the `MixColumns` step by transforming them into a sequence of table lookups noted `T-Tables`. High performance implementation such as OpenSSL uses only four 256-entry 32-bit `T-tables` T_0, T_1, T_2, T_3 during the encryption, and a separate table T_4 is used during the last round. Those tables are also known as `S-Box` tables. To access them indexes are required. For instance, in the first round of AES the indexes to the lookup tables are computed as illustrated by the Equation (10.2).

A round can then be done with sixteen table lookups and twelve 32-bit exclusive-or operations, followed by four 32-bit exclusive-or operations in the `AddRoundKey` step. It is possible to totally avoid lookup tables, however, it means that the software must perform complex Galois Field [Ben12] operations that badly induce performance penalties. The memory accesses while performing table lookups will induce some execution time variations that we will further explain in the next sections.

10.2 Typical Time-Driven Cache Attack Flow

Usually, a Time-Driven Cache Attack includes three main phases: sampling phase, analysis phase and a key search phase. The sampling phase consists in collecting the timing information. The plaintext or the ciphertext is known and the attack can be performed in (quasi) parallel with the encryption on the same processor. The latter can be triggered by different means according to the privileges of the attacker and the protections that are involved to protect the target process. At some point, while the attacker is able to collect timing information of the targeted operation, given a certain amount of measurement samples, he can start the analysis phase which consists in performing some statistical analysis in order to guess some parts of the secret key. Finally, considering the results of the analysis phase and the potential correct key byte candidates, a key-search phase can be performed.

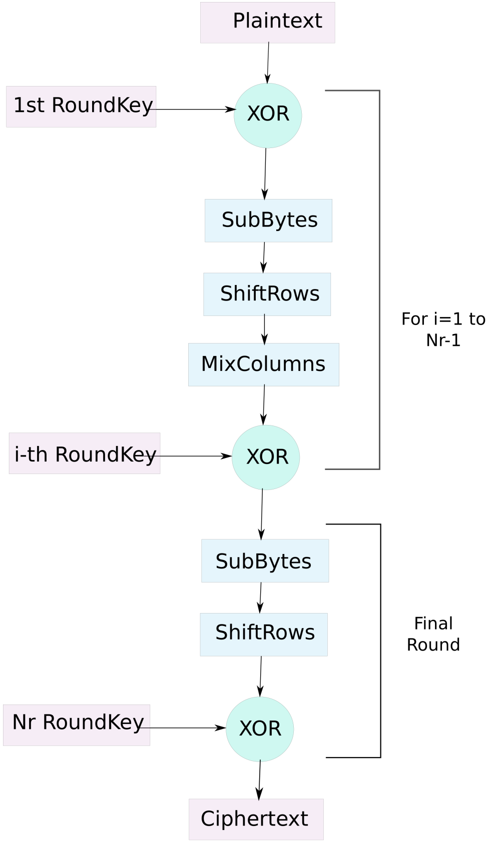


Figure 10.1: Illustration of an AES encryption

10.3 Attack Scenarios

According to the platform on which the spy process is running, we can define two types of attack configurations that can affect the technique(s) used during both sampling and analysis phase:

Local Attacks. In this case, the attacker owns the device and he can perform as much as tests he wants to. The timing information will contain noise related to the operating system activities (e.g. interrupts, scheduling events, etc.) and the behavior of the underlying hardware. However, as he has the device at his disposal, it is likely that he can design a more favorable attack setup.

Remote Attacks. Two cases exist: either the target process that owns the code to monitor sends by itself the timing information (which is unlikely), or the attacker is able to remotely install a malicious application that would tamper with the target application or process. In the first case, data are sent by the target platform through the network. This will produce additional sampling noise is related to the network delay. The second case is similar to a local attack.

10.4 The Threat Model

The targeted AES implementation relies on the use of a secret key and inputs/outputs to determine the indices to perform the table lookups. During these memory accesses, according to the indices, it is likely that some cache misses occur, which consequently lead to different execution times. Let N the number of memory accesses such as $N = N_h + N_m$ with N_h and N_m respectively the number of cache hits and cache misses. Let us also note, $\rho = \frac{N_m}{N}$ the miss rate, and T_h and T_m the access times for a cache hit and a cache miss. A cache miss latency can then be noted $T_m - T_h$. Finally, we have α which refers to the possible additional latencies induced by other events. Therefore, the total access time T during a memory access can be noted as follows:

$$\begin{aligned} T &= T_h \times N_h + T_m \times N_m + \alpha \\ &= (T_h \times (N_h + N_m) + (T_m - T_h) \times N_m + \alpha) \\ &= (T_h \times N + (T_m - T_h) \times \rho \times N + \alpha) \\ &= (T_h \times N + (T_m - T_h) \times N_m + \alpha) \end{aligned} \tag{10.3}$$

On an ARM Cortex-A9, each L1 cache line l_i can hold 32 bytes. Considering that a word is of size 4 bytes, a single memory access at a given address $addr_i$ implies the word W_i to be loaded into l_i as well as the next seven consecutive words. Consequently, while two different memory accesses induce two cache hits in the same l_i, we will have a so-called *cache collision*. The first access causes a *cold* cache miss, and the second one will inevitably cause a cache hit. Therefore, the higher the number of cache collisions, the smaller should be the number of cache misses.

During the analysis phase (introduced in section 10.2), considering that the attacker has the same device as the actual victim device, before attempting a remote attack, he can perform a learning phase (further described in section 10.5.1) where he can determine the leakage model. Since he knows both the plaintext and the key bytes, he can draw a timing profile related to the indexes (see Equation (10.2)) that will produce time variations during the different table lookups due to the cache collisions that can occur. The latter can also be caused by "natural" cache evictions induced by other hardware mechanisms related to the additional underlying micro architectures behavior, and the logical mechanisms induced by the operating system (e.g. the memory management unit, or MMU).

The analysis of the execution time variation is the basis to carry out a timing attack. The one [Ber05] proposed by Bernstein has been studied and is explained further in the next section.

10.5 Overview of Bernstein's Time-Driven Cache Attack

The author's attack uses timing analysis to reconstruct an AES key based on cache timing information by attacking the first round of the algorithm. His attack consists of four phases: a learning phase, an attack phase, a correlation phase, and a key recovery phase.

10.5.1 Learning phase

It involves measuring the encryption time of multiple random plaintexts P under a known zero-key K. The sum of all encryption times observed for plaintexts P_i whose j^{th} byte is $P_i[j] = b$ is stored in $Sum_T[j][b]$, where $j \in \{0, \cdots, 15\}$and $b \in \{0, \cdots, 255\}$, such as we have the following:

$$Sum_T[j][b] += time(AES(P_i, k)) \tag{10.4}$$

Furthermore the number of encrypted plaintexts is stored within $n_P[j][b]$. In this configuration, the attacker knows the key K, which involves that either he has a full control over the server, or he has the exact environment under study (the same device with the same attacked

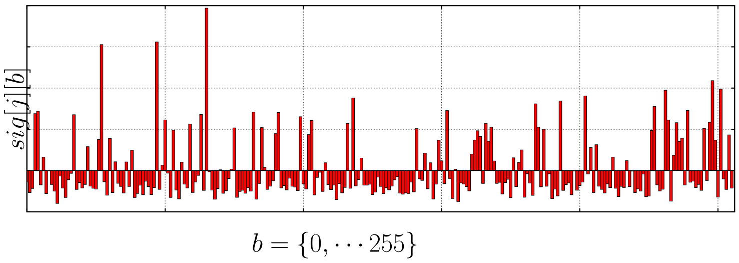


Figure 10.2: Signature chart that characterizes the timing information of each value an individual byte can take during the encryption. X-axis: every values a byte can take, Y-axis: cycles number with respect to the average (over all bytes).

software running on the same processor under the same OS environment). During this step, the attacker submits to the server a set of N plaintexts $P = \{P_0, \ldots, P_{N-1}\}$. The server receives them one by one and encrypt them. During the encryption, the operation is timed and the measurement is sent back, so one can collect it.

Therefore, through this learning phase, a signature chart can be plotted that would show the average computing time (in processor cycles) for each value a byte can take. A more formal description of the latter is depicted by the Equation (10.5).

$$sig[j][b] = \frac{Sum_T[j][b]}{n_P[j][b]} - \frac{\sum_{j=0}^{15}\sum_{b=0}^{255} Sum_T[j][b]}{\sum_{j=0}^{15}\sum_{b=0}^{255} n_P[j][b]} \tag{10.5}$$

According to j and $sig[j][b]$, one can then plot the following signature chart, where the X-axis refers to all the possible values that a byte can take ($0 \leq b \leq 255$), and the Y-axis refers to the previous $sig[j][b]$.

The whole learning phase enables to produce 16 signature charts that depict a learning phase **timing profile** for randomly chosen plaintexts and a known key. This is depicted by the Figure 10.3. This timing profile will be noted σ throughout the next sections.

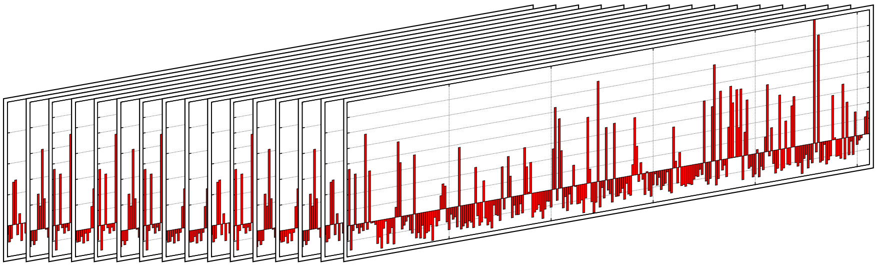


Figure 10.3: Illustration of the profile of a learning phase composed of 16 signature charts of each key byte

10.5.2 Attack phase

This phase consists in collecting the exact same information as previously gathered but under an unknown key $K^{'}$. The whole procedure is the same as the learning phase, with another set of N random plaintexts $P^{'} = \{P_0^{'}, \cdots, P_{N-1}^{'}\}$, the sum of all encryption times is saved in $Sum_T{}^{'}[j][b]$ and the number of each encrypted plaintext is stored within $n_P{}^{'}[j][b]$.

Also, the exact same way as the learning phase, the profile of the attack phase can be depicted by the different produced signature charts. The timing profile of the attack phase will be noted σ_2.

10.5.3 Correlation phase

$sig^{'}[j][b]$ of the learning phase and the attack phase are analogously computed, to produce a correlation table $Corr[16][255]$, such as, for $j \in \{0, \cdots, 15\}$ and $x \in \{0, \cdots, 255\}$, we have:

$$Corr[j][b] = \sum_{x=0}^{255} sig[j][x] \times sig^{'}[j][b \oplus x] \tag{10.6}$$

This function is the correlation of two timing signatures sig and $sig^{'}$ that searches for the value of b that produces the maximum value for the sum. Thus, at some point, sig may potentially match $sig^{'}$. The values stored in $Corr$ depict the correlation scores of each candidate key byte. The latter are sorted in a decreasing order according to their correlation score so the very first key byte candidates $Corr[j][0]$ are now the ones with the highest correlation scores.

10.5.3.1 Filtering Out Key Byte Candidates by Means of a Deviation Threshold

The correlation scores that are stored in $Corr$ are then filtered according to a deviation threshold such as $Corr[j][b]$ is kept in the list of potential candidate only if $Corr[j][0] - Corr[j][b] < y \times \sqrt{Variance[j][b]}$. This means that if the difference between the key byte candidate that has the highest correlation score and the one that has to be tested is less than the latter's standard deviation, then it can be kept in the list of potential candidates. We also note that in Bernstein's method, the deviation threshold uses an empirically chosen factor noted y.

The output of the correlation phase is depicted by the Table 10.2. This was a test performed on the Samsung Galaxy S3, where the key k to retrieve was equal to 0. The underlined values represent the candidate key bytes that resulted in the highest correlation value during the correlation computation. The first column is the total number of key byte candidates. The second column is the key byte index. The remaining values is the list of key byte candidates. This depicts the ideal case where the real key bytes are ranked first in terms of position in the list of key byte candidates.

Total Number of Key Byte Candidates	Byte Index	List of Key Byte Candidates
191	0	00 5d 13 aa 59 39 b4 fe ...
185	1	00 8e 01 43 33 25 a1 66 ...
211	2	00 78 4e 61 21 f8 39 47 ...
199	3	00 dc 69 56 60 3f f2 c5 ...
208	4	00 b5 33 d7 2a f1 45 7c ...
233	5	00 52 f4 9c 38 f1 b2 fd ...
229	6	00 9e 07 84 7c e6 1a f8 ...
161	7	00 20 2e 15 c1 da 0c f3 ...
238	8	00 59 bf 2d 0e cb 71 92 ...
175	9	00 f6 26 4f bd f2 79 36 ...

192	10	00 0e fa be b1 4d 30 73 ...
178	11	00 51 6d 14 c2 86 0d 92 ...
175	12	00 bd c1 a7 5e 1a 66 1c ...
219	13	00 30 2e 13 a7 e6 81 92 ...
213	14	00 ea a8 37 ac 62 76 6d ...
193	15	00 75 f9 3e 49 ac e2 40 ...

Table 10.2: Correlation phase output.

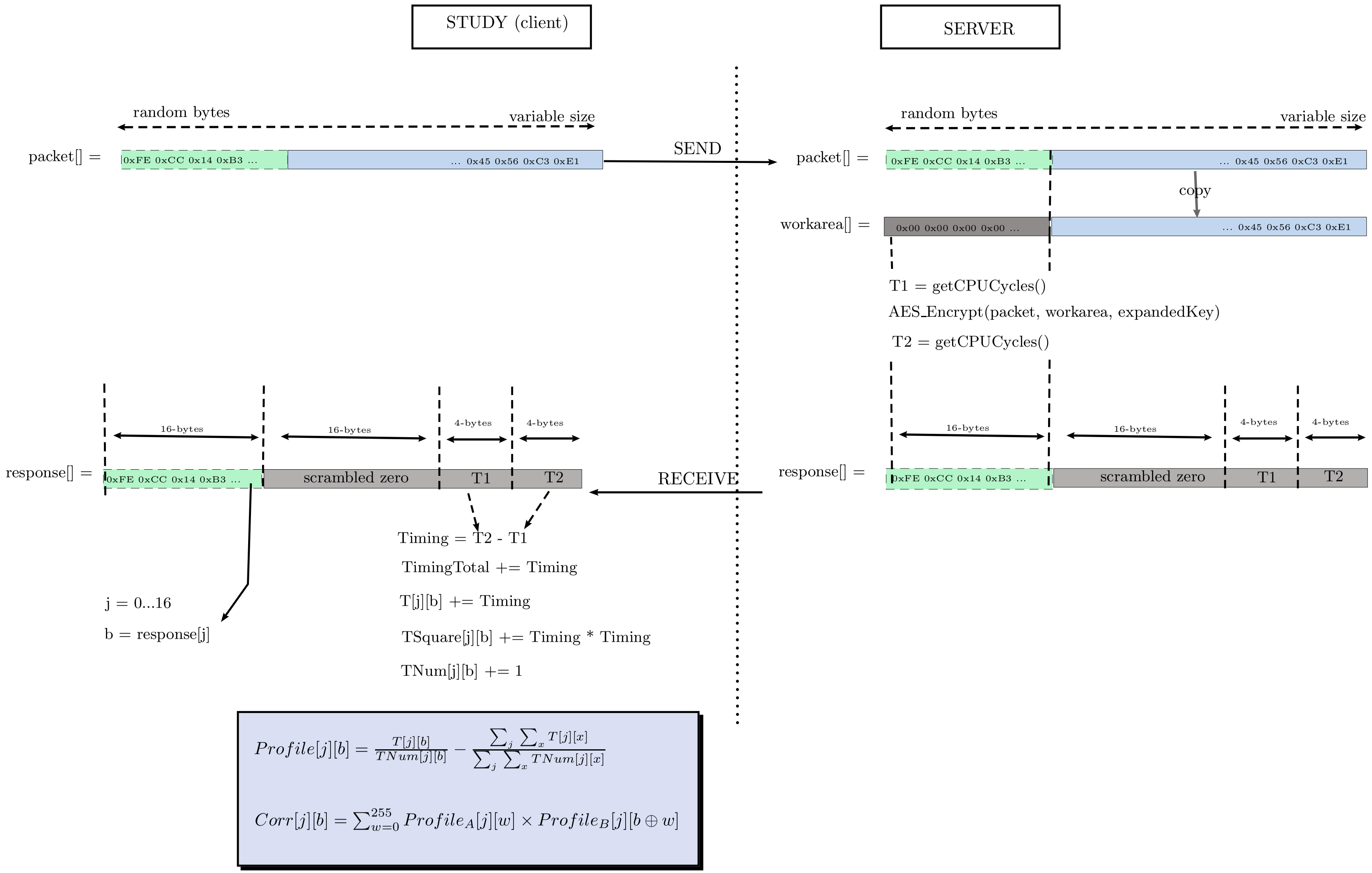
10.5.4 Full key recovery phase

Finally, the **full key recovery** phase which simply consists in brute forcing the remaining uncovered bytes. The time spent during this phase depends on the results of the correlation phase. If one takes a look at the lists of key byte candidates in the Table 10.2, the right key byte candidates (underlined) are located at the very first position of each list. Therefore, the better rank a key byte candidate has in its list, the less time it will take to test it. Consequently different cases exist:

1. **The Best Case:** each key byte candidate k_i is ranked 1^{st}
2. **The Impractical Case:** each key byte candidate k_i is ranked 255^{th}, thus it will take an effort of 2^{128} computations, which is not practical at all
3. **The Very Worst Case:** some key bytes candidate k_i are not even in their corresponding list

More generally, throughout the next sections, for a given correlation phase result, the estimation of the computation effort that remains for searching the right key bytes will be defined as being the product of all the total number of possible values for each k_i. (*e.g.* see Table 10.2, first column, `ComputationEffort` $= 191 \times \cdots \times 193 = 2^{123}$)

A summary of Bernstein's attack is depicted by the figure 10.4. For a complete description of Bernstein's source code, further reading can be found in Neve's PhD thesis [Nev06], *chapter IV*, *section 4.2*, page 52-55.



$$Profile[j][b] = \frac{T[j][b]}{TNum[j][b]} - \frac{\sum_j \sum_x T[j][x]}{\sum_j \sum_x TNum[j][x]}$$

$$Corr[j][b] = \sum_{w=0}^{255} Profile_A[j][w] \times Profile_B[j][b \oplus w]$$

Figure 10.4: Overview of Bernstein's Cache Timing Attack

10.5.5 Identification of the Right Key Byte Candidates

In his PhD thesis [Nev06], Neve gave an interesting analysis of why Bernstein's attack (theoretically) works, and enables to understand, how likely it is to retrieve some possible key byte candidates through the correlation of two timing profiles σ_1 and σ_2 (described in section 10.5).

If one takes back the equation 10.1 in the section 10.1, during the AES computation, the very first input of the algorithm is $P_0 \oplus k_0$. Furthermore, if we consider Bernstein's method during the signature computation (see Equation (10.5)), after the learning and the attack phase, we have two matrices $sig[j][b]$ and $sig^{'}[j][b]$ that contains the running times of each possible value $b \in \{0, \cdots, 255\}$, a single byte $j \in \{0, \cdots, 15\}$ can take. Consequently, according to the key k_1 and k_2 used during the learning and attack phase, individual time profiles can arise out of random plaintext encryptions.

As Bernstein's attack targets the very first round of the AES, the table lookup indices $idx_{\beta}^{r=0}$, with β the β^{th} indice, that are used to access the lookup tables are each related to only one key byte $k_j^{r=0}$ and one plaintext byte P_j such as

$$\begin{aligned} idx_{\beta}^{r=0} &= P_j \oplus k_j^{r=0} \\ with j &\in \{0, \cdots, 15\} \end{aligned} \tag{10.7}$$

Therefore, small overall timing differences may arise between the learning and attack phases while $P_{j,i} \oplus k_j$ and $P^{'}_{j,i} \oplus k_j$ are being computed with i the i^{th} plaintext. Indeed, between two runs of AES, some other tasks are performed that may lead to the eviction of some cache lines. Thus, during the next run, either some accesses to the `S-Box` tables are already resolved in the cache, then cache hits will occur and access time will be shorter, otherwise, cache misses will occur, and the encryption will take longer time.

As it has been highlighted by Neve, one can note the following heuristic that will determine if a given candidate key byte $k^{'}_j$ from the attack phase is a possible key byte candidate.

$$P_{j,i} \oplus k_j = P^{'}_{j,i} \oplus k^{'}_j \Longleftrightarrow \sigma_1 \approx \sigma_2 \tag{10.8}$$

with – we recall – j the j^{th} byte of P_i, i the i^{th} plaintext.

As a consequence, while the timing profiles σ_1 and σ_2 will approximately be the same we will have

$$k^{'}_j = P_{j,i} \oplus k_j \oplus P^{'}_j \Longleftrightarrow \sigma_1 \approx \sigma_2 \tag{10.9}$$

during the very first *AddRoundKey* operation.

Later on, during the correlation phase, the correlation scores that are defined as the good ones are kept according to a deviation threshold, where the variance for each byte values are computed, and the correlation score is kept only if it is close to the maximum correlation value, with respect to its variance.

10.6 Differences and Improvements

10.6.1 Local Attack Scenario

While initially Bernstein's attack was a remote attack scenario, in our case we rather attempted a Local Attack (see section 10.3) where both the client and the server processes are run on the same device. However, they are forced to run on different logical core by setting their processor affinity. The strategy behind this is to avoid the scheduler to move our processes from a core to another one. It will not prevent the CPU to handle interrupts however their effects should be minimized.

10.6.2 Threshold Determination to Minimize the Noise

In Bernstein's code, during the time measurement gathering, a threshold is defined to filter out some potential noise in the measurements. This was in order to empirically minimize the noise produced by events such as for instance, interrupts. As illustrated by the Figure 10.5, this very first threshold value is quite important as it can either include too much noise or on the contrary, remove too much information in the final timing measurements.

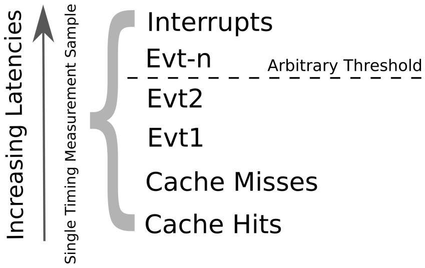


Figure 10.5: Illustration of the possible events (Evt) that define a single time measurement of the execution of a whole algorithm

However, no one in the literature has detailed how they defined this threshold value, instead, they rather focused more on refining the statistical analysis approach after collecting enough measurements during the attack phase. We have seen previously in our experiments in Section 8.3 that it is possible to characterize the latencies, noted T_1 and T_2, that define respectively a cache hit and a cache miss. Taking T_1 as a threshold would remove T_2, but T_2 is a useful information to gather during an encryption. However this T_2 cannot be taken as the threshold value because it does not reflect the actual latency during an AES encryption. More fine-grained cache timing attacks such as Access-Driven or Trace-Driven Cache attacks generally define the threshold as being the minimum value `MIN_CACHE_MISSES` of the latency induced by a cache miss on a single memory access. However, as a Time-Driven cache attack observes different runs of a whole function (and possibly a whole process), the number of cache misses may vary a lot and thus cannot be restricted to this `MIN_CACHE_MISSES`. Furthermore, as illustrated by the Figure 10.6, if we compare the encryption of different plaintexts, differences can be observed between the distribution of their execution time (cycles) and it corroborates the assumption that a dynamic threshold has to be computed in order to minimize the noise and to avoid the least information loss as possible from the very beginning of the timing information gathering. The longer the execution time T_i, more misses has occurred. However, the shorter T_i is, more cache collisions has occurred, that is, more cache hits can be observed. The execution time that is the most represented is likely to be related to the AES encryption that is being measured which can then be used as a threshold.

Consequently, the learning phase (see section 10.5.1) has been modified to add an additional step. As a reminder, the initial learning phase involved performing the following steps in a loop:

- generating a random plaintext
- triggering the AES by sending the plaintext to the server

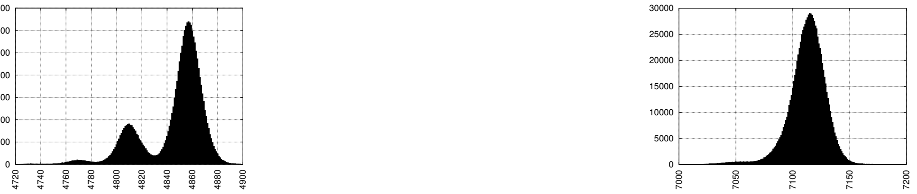


Figure 10.6: Illustration of two examples of distributions of the measured number of cycles (X-axis) of two different plaintexts during N runs of the AES encryption. The Y-axis refers to the number of times a given number of cycles has been represented among the N runs. The figure on the right has an execution time 1.75 times higher than the left one.

- receiving the execution time T_i in response
- computing the timing profile with T_i

The modification involves adding another spy process that is running on the same core as the target process. This process also acts as a server, and can only execute one operation, that is `FillAllL1()`, to fill the whole L1 data cache and thus to evict data from the cache. The reason behind this is to start the AES encryption with a "cleared" cache and thus to be able to observe the evolution of the execution time without to be polluted by preloaded data. Therefore, the new procedure is as follows:

1. generate a random plaintext
2. trigger N times the AES with the same plaintext
   - collect the execution time for each run
   - trigger `FillAllL1()` in between each run
3. compute and sort in decreasing order the frequency histogram that illustrates how many times for the current plaintext each different execution time has been measured
4. pick the one that is the most represented, and use it as a threshold
5. trigger `FillAllL1()`
6. trigger one time the AES
7. receive the execution time T_i in response
8. If $T_i < Threshold$, compute the timing profile with T_i
9. otherwise, start over from the bullet `5`

10.6.3 Timing Measurement on ARM processor

Initially, Bernstein used the x86 instruction `RDTSC` to get the number of cycles. On ARM-A processors, the most accurate way to get a reliable cycles number is through the performance-monitor control register [23], which is implemented within a special coprocessor. These registers

[23] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0433c/CIHJGICA.html

include a so called `Cycle Count Register`, which is a 32-bit register that is used to count clock cycles.

At the very beginning, a kernel module has been developed in order to trigger the following operations: (1) enable the performance monitor counter, (2) read the cycle counter. Indeed, as the ARM instructions to use in the code for performing those operations can only be made in a privileged mode, the only way to overcome this issue was through a kernel module that has enough privileges. Consequently, in a first place, the device under analysis had to be rooted in order to load the module. However, there is a solution to this and it does not require any root privileges, but it still has a disadvantage as it will be explained later.

10.6.3.1 Timing Measurement From the Kernel Space

The kernel module that has been developed was of type "character module". The communication between the module and the user space can be made through a "character device" which is located under the `/dev` directory on the device's filesystem, once the module is loaded. A character device can be accessed as a stream of bytes, like a file. Therefore, like any other file, such device can then be accessed through file operations. The kernel module is able to handle read and write events that occur on the device, hence, custom commands can be issued from the user space to the kernel space.

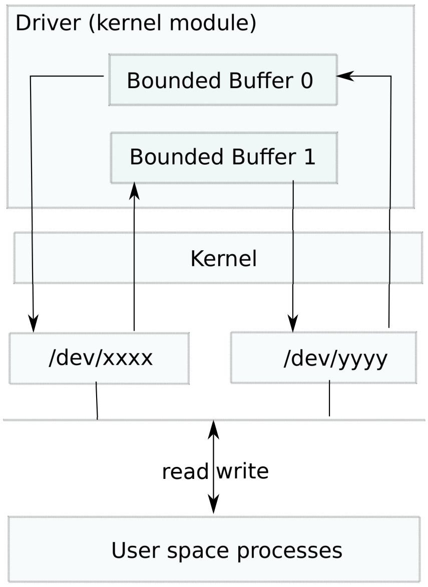


Figure 10.7: Communication while using a kernel module

Before being able to access the performance monitor counters, one should first enable a user-mode access and then the counters. This is depicted by the Listing 10.1.

Listing 10.1 Kenrel Module Code for enabling PMU counters

```
static void enable_all_pmu_counters(void* data)
{
        /* Enable user-mode access to counters. */
        asm volatile("mcr p15, 0, %0, c9, c14, 0" :: "r"(1));

        /* Enable all counters */
        asm volatile("mcr p15, 0, %0, c9, c12, 0" :: "r"((1 | 16)));
        asm volatile("mcr p15, 0, %0, c9, c12, 1" :: "r"(0x8000000f));
}
[...]
int init_module(void)
{
    printk(KERN_INFO, "[" DRVR_NAME "] Module init\n");
    /* Init PMU counters on each cpu */
    on_each_cpu(enable_all_pmu_counters, NULL,1);

    /* Bind with the character device + Init the bounded buffer*/
    [...]
}
[...]
/* Read the counter*/
static inline u32 get_cycle_number(void)
{
    u32 r = 0;
    asm volatile("mrc p15, 0, %0, c9, c13, 0": "=r"(r));
    return r;
}
```

Therefore, on the user space, in the server application every time it receives a new plaintext from the client, the part that measures the execution time of the AES does the following operations:

1. Enable counters so they can start counting
2. Get the cycle counter, noted C_1
3. Encrypt the plaintext
4. Disable counters so they stop counting
5. Get the cycle counter, noted C_2
6. Return $Timing = (C_2 - C_1) - \theta$, where θ is the average number of cycles between two calls of `get_cycle_number()`, namely, the overhead induced by itself.

Drawbacks – One can note that in the case the attacker has the device at his disposal, he can obviously permanently root the targeted device for the training phase. However, in the case of a remote attack, there is no need to have a permanently rooted device, as the program may embed a root exploit that would enable the spy program to get temporary root privileges. However, the kernel will prohibit loading a module compiled against a different kernel version, thus, a malicious application will have to foresee a kernel module compiled against the right kernel version.

Consequently, another technique has been used to overcome the need of root privilege.

10.6.3.2 Timing Measurement From the User Space

As Android's kernel is based on a Linux kernel version, it inherits various interfaces from it. Among others, there is the `perf_event_open` system call which is an abstraction layer for PMU (Performance Monitor Unit) management. It is capable of keeping track of various hardware and software events such as for instance, the number of page faults, context switches, CPU clock, branch misses and so on. However, they are not all available/implemented on all platform, but only the number of CPU cycles is required. On the Cortex-A9 and Cortex-A15, the hardware event `PERF_COUNT_HW_CPU_CYCLES` is available, but it has a significant disadvantage according to the manual [perb].

Drawbacks – it is sensible to CPU frequency scaling, however, still according to the manual, another event type exists and also enables to count the CPU cycles which is `PERF_COUNT_HW_REF_CPU_CYCLES` and is not affected by the frequency scaling. Unfortunately, this event type was not implemented within the kernels of the two devices under test.

10.7 Practical Experiments

Two devices were used for the practical experiments: the Samsung Galaxy S3 and the Samsung Galaxy S4.

Device	Processor	OS	First-Level Cache		
			Size	Associativity	Line Size
Samsung Galaxy S3	Cortex-A9 (Quad Core)	CyanogenMod 10.1.3	32 KB	4-way	32 bytes
Samsung Galaxy S4	Qualcomm Krait 300 (Quad Core)	Android 4.4.2	4 KB	Direct-mapped	64 bytes

10.7.1 Profiling

As it has been described in the section 10.5, the learning phase and the attack phase produce 16 timing profiles that illustrate the behavior of the underlying hardware while testing the first 16 bytes of the first AES round that takes as an input `plaintext` $\oplus$ `key`. These two profiles are depicted by the Figure 10.8 for the learning phase, and Figure 10.9 for the attack phase (for the Cortex-A9, namely the Samsung Galaxy S3). Each profile refers to the key byte under study during the analysis. It starts from the key byte 0 (uppermost on the left column) to the 16^{th} key byte (bottomost on the far right column). The X-Axis refers to the values a byte can take, thus $x \in 0, \cdots, 255$. The Y-Axis refers to the number of cycles with respect to the average (over all bytes).

10.7.2 Learning and Attack phase set up

The following are the different values that has been used to carry out the experiments:

- k is the known key that is used during the learning phase, and its value is arbitrarily fixed to
  `k = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00.`

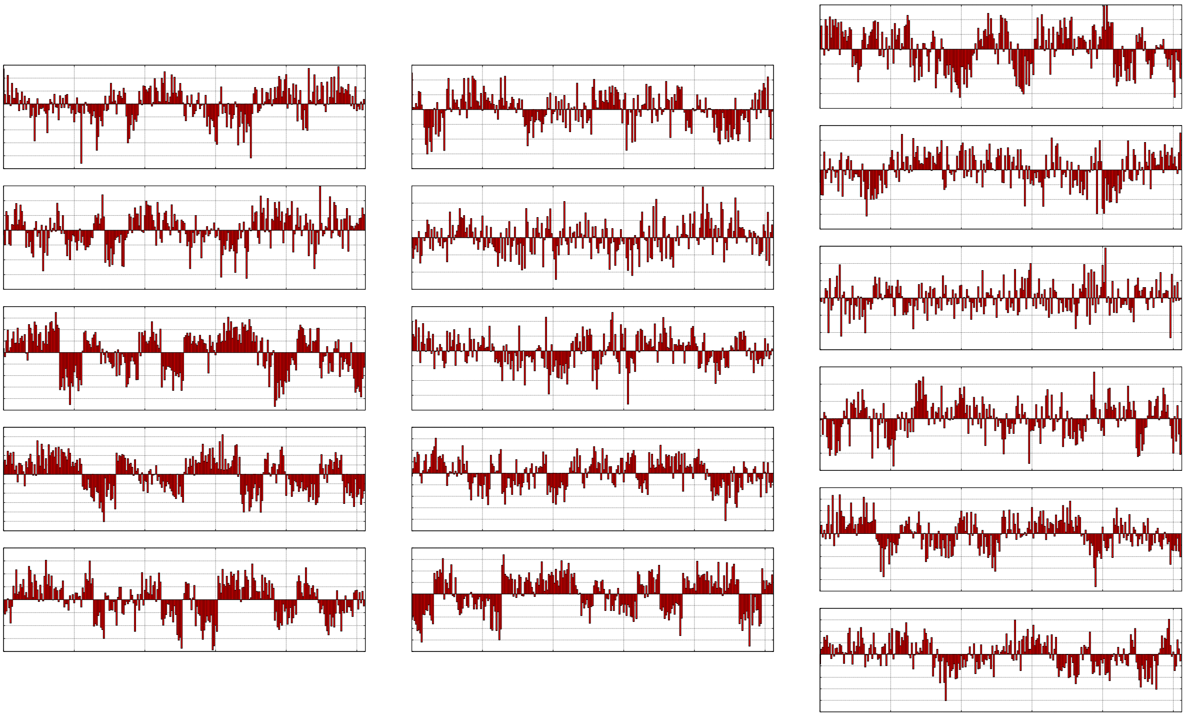


Figure 10.8: Profiling during the study phase on the Cortex-A9: reading from top to the bottom, the first column (left) highlights the profile of the key bytes 0, 1, 2, 3, 4. The middle column corresponds to key bytes 5, 6, 7, 8, 9, and the last column to 10, 11, 12, 13, 14, 15

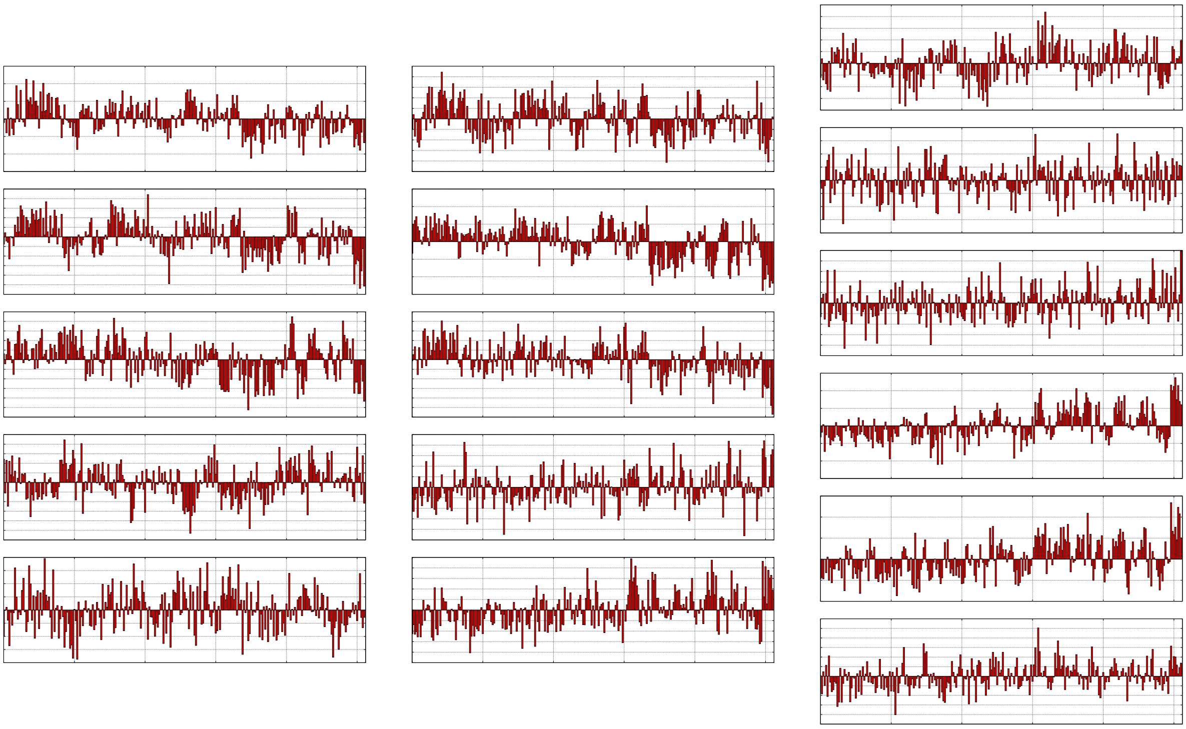


Figure 10.9: Profiling during the attack phase on the Cortex-A9

- $k^{'}$ is the unknown key to recover and it has been randomly generated such as $k^{'}$ `= 38 ca 1d 8e 98 a1 3b 63 72 02 19 c0 0f 42 f3 b6`
- about 2^{30} measurements are performed for both the study and the attack phase

10.7.3 The Correlation Phase Results

An example of result of the correlation phase is depicted by the following Table 10.3.

Table 10.3: Result from the correlation phase

Number of Key Byte Candidates	Byte Number	List of Key Byte Candidates
103	0	a7 90 b8 8e 83 . . .**38** de 01 72 93 a7 cc da fe a3 7b . . .
115	1	d4 b5 81 2b 75 71 . . .**ca** a3 b5 e2 41 32 10 81 8e 1a 4b . . .
111	2	97 c8 e4 09 01 dc . . .**1d** 47 0e c3 7f 48 77 cb b8 d4 a2 . . .
98	3	b6 5c 24 4f 61 52 . . .**8e** bc d9 fd b2 0f 08 d4 b8 50 7c . . .
69	4	a3 24 f7 c9 ca 61 . . .**98** 79 bc 1f 02 33 2d 78 d2 36 8c . . .
171	5	aa 5e 5b cd e2 91 . . .**a1** 43 a9 39 dd bc eb 47 f7 9f 21 . . .
113	6	71 20 51 bf 7e 53 . . . – **missing 3b** –
134	7	fd 71 37 22 3a 4f . . .**63** b4 88 c4 f8 91 56 d4 a3 5c 3c . . .
123	8	39 b4 f1 44 ff 45 . . .**72** 32 09 93 40 2a b6 cf ba f4 d9 . . .
121	9	e3 d8 38 7c ed 94 . . .**02** a4 99 bb 80 2b 7e 5d e0 ef d3 . . .
156	10	14 56 30 51 54 d9 . . .**19** d2 84 1d 31 3c 0d ef 58 2e 44 . . .
103	11	c7 ab 67 60 59 29 . . .**c0** 66 49 e5 d7 2e e7 47 d1 97 f3 . . .
135	12	bd c0 9f a5 bb ad . . .**0f** 64 7f 37 ef 45 fb 62 9c 15 65 . . .
107	13	d0 b2 ba 52 d4 49 . . .**42** 6d 9c e9 46 b9 ff bf 33 b2 ea . . .
123	14	c5 25 ef 55 c1 ad . . . – **missing f3** –
97	15	08 bf 1f 36 63 54 . . .**b6** 5d ec d7 6a a0 e1 db 87 7a a5 . . .

10.7.4 Observations

Similarities between patterns: In his experiment, Neve noticed similarities in the patterns between different profiles while evaluating the Pentium III.

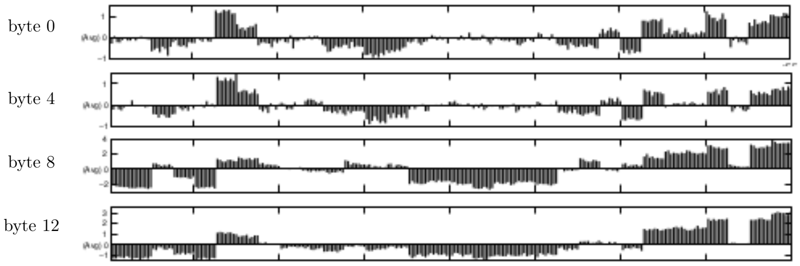


Figure 10.10: Similarities between patterns for byte 0/byte 4 and byte 8/byte 12 during the learning phase (src: Neve's PhD thesis, page 56)

Neve explains this as being the effect of the cache architecture, however, in our case, as one can see from the Figure 10.8, during the learning phase, one can notice that there is no significant patterns that can be matched from a given profile to another one. Even though, the key bytes have the same values during this phase (*i.e* the hexadecimal value `x00`), there is no specific assumption that can be made from the observation of the shape and patterns. However, from the study phase (Figure 10.9), one can notice that patterns are better defined, and similarities between profiles can be distinguished regarding their shapes. The profiles of byte 0 and byte 1 are roughly the same. The same observation can be made for the byte 13 and byte 14 that are also roughly similar.

Missing Correct Key Bytes: the key bytes 6 and 14 are totally missing from their respective list in this particular test (Figure 10.3). This illustrates one case, where the deviation threshold (see 10.5.3) is not permissive enough so some actual key bytes could be removed from the list of potential key byte candidates.

Low-ranked Key Byte Candidates: In the Table 10.3 one can notice that the right value of the key byte 0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13 and 15 are present in the list of candidates. However, in term of rank, they are respectively, 71^{st}, 98^{th}, 53^{rd}, 48^{th}, 57^{th}, 84^{th}, 117^{th}, 107^{th}, 94^{th}, 73^{rd}, 63^{rd}, 47^{th}, 85^{th} and 61^{st} in their respective list of key byte candidates. We wanted to observe how much these ranks (thus the correlation phase) varies. Therefore, we have run the same experiment (same key + the same set of 2^{30} plaintexts) in the same configuration (see sections 10.6.1, 10.6.2, 10.6.3.2) ten times (1 run = study + attack + correlation phases, about 2 non-stop days of measurement). Every test that yielded with a key byte candidate that is not contained in its corresponding list is just discarded and another test is run right after until the right k_i is present in its corresponding list. Consequently there will be a total of $10 + x$ runs where x refers to the possible ones that ended with a case where the correct value of a given key byte was not present in its corresponding list of potential key byte candidates.

Therefore, the Figure 10.11 illustrates the variability of the resulting ranks (Y-Axis) for each key bytes k_i (X-Axis), namely we want to observe the of spread between the extremes of ranks for each key bytes. The columns denote the data mean and the bars show the standard deviation.

What we observe is that from the rank point of view the variance is weak for a given key byte except for the key byte 5, however, it is very high from a key byte to another one. What we learn is that the larger is the column, the higher is the rank for the corresponding key byte in average. Moreover, we also can say that the larger is the bar, the less confidence we can have on the reported average rank for a given key byte. Consequently, we can assume that there is a relation between the position of each key byte and the rank.

The Frequency Scaling Annoyance: We have performed a simple test that highlights a behavior that we suppose to be caused by the frequency scaling induced by the current power saving policy (OnDemand[24]). We have run N times the AES encryption with a set of N different plaintexts, in a given process on a single core. The same test is replayed once more with the same set of plaintexts. The execution times have been collected, and an example of result is illustrated by the Figure 10.12. The X-axis refers to a single run of AES, and the Y-axis refers to the execution time (cycles). What we can observe is that, between the test one (left), and the second test (right), there is a noticeable difference of shape between the two tests. The first one looks noisy, and the second is flatter but this is due to the high peaks. As we are collecting

[24] http://ajgupta.github.io/android/2015/01/28/CPU-and-GPU-governors/

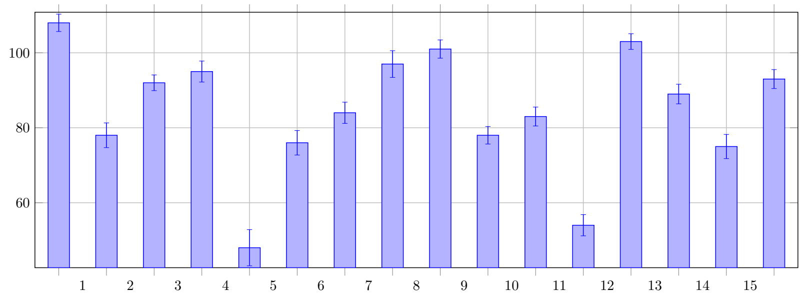


Figure 10.11: Descriptive Error Bars that illustrate the variability of the ranks (Y-Axis) for each key bytes k_i (X-Axis) that have to be guessed.

a great amount of samples ($\sim 2^{30}$), it takes few hours before each test finishes. Consequently, as we have decided to let the device to lock itself after some time of inactivity, it is very likely that the processor adjusted its frequency for power saving purpose. However, what has been observed is that even while the screen is let off during a whole test, this bouncy behavior can still be observed but at a less frequency rate. The dotted rectangles highlights two area. The first vertical one depicts at the very beginning of each graph a very high peak. They do not occur during the same run, and we assume that an interruption has occured for a long time. Compared to the dotted horizontal rectangle on the right, the one on the left illustrates the amount of noise.

These observations clearly suggest that the frequency scaling has to be fixed to be less prone to such noisy behavior.

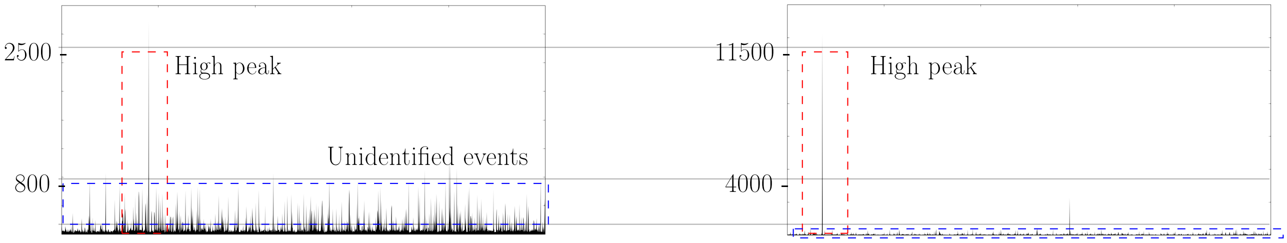


Figure 10.12: Illustration of two tests while AES is run N times during each test. On the first (left) test, noise are well visible, and a very high peak which is probably an interrupt that occurred. On the second test (right), one can note that there are less noise, however, an high peak is still visible.

The Asymptotic Limitation of the Profiling Phase: Neve already stated during his experiments that beyond a certain amount of encryptions, any additional measurements does not improve further the results of the correlation phase, thus, the brute force phase either. In our case, it appears that using a number of samples noted ns such as $2^{29} \leq ns \leq 2^{30}$ produces better results than going above 2^{30} measurements during the correlation phase. This is illustrated by the Figure 10.13 where for each number of samples 2^n, ten runs (1 run = study + attack + Correlation phases) are performed ten times (X-Axis), and the results of the correlation phase are used to compute the estimation of the number of operations (Y-Axis) to perform, if brute-force is considered to guess each key bytes. The latter will be noted ϑ. We are still in the same configuration as the one described in sections 10.6.1, 10.6.2 and 10.6.3.2.

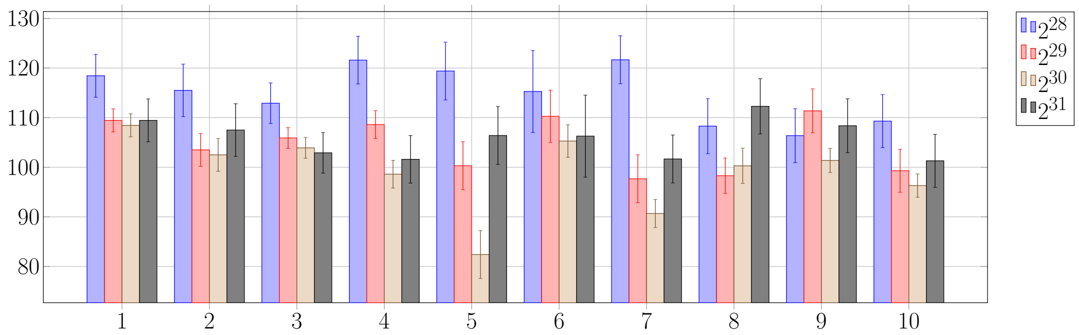


Figure 10.13: Error bars illustrating the variability of ϑ according to the number of samples used and the key byte position. X-Axis = runs, Y-Axis = ϑ, the estimated computation effort for the key search phase.

10.8 Results Summary

As a summary, we provide the results and observations highlighted by the Table 10.5. Among our several tests, we tried different configurations:

- either the performance counters were started and stopped from a kernel module, or using the `perf_event_open` system call from the user space
- either a fixed threshold (cycles) is empirically set to filter out the noise while collecting the timing information, or the dynamic threshold computation is used
- either we let the default frequency scaling policy (OnDemand), or we set its value to "Performance"

In this Table, we particularly highlight the best results we could reach. Different values are shown: the brute-force estimation (BFE), the time spent (TS) for this best run and the fixed threshold (TH) that was used during this run.

Timing Measurement		Kernel Space						User Space					
Samples		2^{29}			2^{30}			2^{29}			2^{30}		
		BFE	TS	TH	BFE	TS	TH	BFE	TS	TH	BFE	TS	TH
Fixed Threshold (cycles)	OnDemand	121	5h	450	119	9h	450	120	5h	450	119	9h	450
	Performance	116	5h	450	114	8h30	450	117	5h	450	95	9h	450
Dynamic Threshold (cycles)	OnDemand	117	12h	-	113	26h	-	109	13h	-	110	26h	-
	Performance	96	12h	-	88	25h	-	98	12h	-	95	25h	-

Table 10.5: Summary of the best runs

We can then formulate the following assumptions that summarize what configuration improve the overall attack:

1. Setting the frequency scaling policy to "Performance" improve the measurement reliability
2. Using a dynamic thresholds also contributes to the noise reduction, despite the additional amount of time that is badly increased
3. 450 depicts the best fixed threshold, that is, for all of the tests that have been performed, a threshold of 450 cycles enabled to filter out enough noise to get the current best result

4. there is no significant differences between getting the performance counters values from the kernel space, nor using the `perf_event_open` system call from the user space. However, using the second solution enables to run our processes without root privileges, which is a significant benefit.

10.9 Future Investigations

10.9.1 Performance Counters Inaccuracy

As we count on the values returned by the performance counters to get the number of cycles, it turns out that according to [Wea10, Wea13], performance counters values might not represent the actual values that occurred as to the user-space events, kernel events are also always counted. This adds non-negligible noises and must then be considered.

10.9.2 On the Quantification of System Noise

In the literature, one of the most used techniques for measuring the impact of both, hardware and software noise that will also be considered in our future investigations are the followings:

1. Fixed Work Quantum (FWQ) performs a fixed amount of work several times, and it records the time spent for each run. The collected measurements can then be used to compute useful statistics of scaled noise.

2. Fixed Time Quantum (FTQ) micro benchmark which was proposed by Sottile and Minnich in [SM04]. The authors described in their paper the limitations of FWQ, and proposed FTQ. It measures a small work quanta that is performed until a fixed time quanta is reached. For each iteration it records how many workload iterations have been carried out.

3. Selfish Detour (SD) was proposed in [BIYC06]. This micro benchmark runs in a tight loop and measures the time for each iteration. The idea is, if an iteration takes longer than the minimum times a particular threshold, then the time (detour) is recorded. SD runs until it records a predefined number of detours.

10.9.3 On a More Robust Key-Search Complexity Evaluation

As this study is still an on-going research work, an important step has not yet been fully studied: the evaluation of the key-search complexity to enhance the last step of our time-driven cache attack, namely, the full key recovery phase. To estimate the brute-force effort after each correlation phase, we just computed the product of the number of possible values for each k_i (see section 10.5.4). Up to now, two techniques have been identified to reduce the complexity of the key-search. From their results, it appears that those techniques greatly decrease the key space.

10.9.3.1 Choosing The Relevant Combination Functions

In Berstein's approach, as it as been explained in 10.5.3,*Correlation phase*, choosing the potential key byte candidates was performed via a threshold-based technique. The threshold is based on the standard error of the mean computation. In their work, [SP13b], Spreitzer *et al.* succeeded in improving Bernstein's time-driven cache attack, and consequently, reducing the complexity of the remaining key-search phase, by computing the probability of potential keys to be the correct key. To do so, they modified Bernstein's scoring function (see equation 10.6), into a Pearson correlation function

$$Corr[j][x] = \frac{\sum_{b=0}^{255} sig[j][b] \times sig^{'}[j][b \oplus x]}{\sqrt{\sum_{b=0}^{S_{kc}-1} sig[j][b]^2} \times \sqrt{\sum_{b=0}^{S_{kc}-1} sig^{'}[j][b \oplus x]^2}} \tag{10.10}$$

with S_{kc} the size of an arbitrary key-chunk. Then, they compute the probabilities of sub-key using a Bayesian extension from the correlation evaluations $Corr$, initially proposed in [GS13], by Gérard and Standaert:

$$\begin{aligned} p[j][x] &= exp[\frac{(S_{kc}-3)(arctanh(sig[j][x]))^2}{2} - \frac{(S_{kc}-3)(arctanh(sig^{'}[j][x])-1)^2}{2}] \\ &= exp[(S_{kc}-3)(arctanh(sig^{'}[j][x])) - 0.5] \end{aligned} \tag{10.11}$$

While this Bayesian extension has not been fully explained in [GS13], it has already been used on a practical side-channel attack in [RL13] by Roche and Lomné.

10.9.3.2 Using Artificial Neural Networks

Gullasch *et al.* performed an Access-Driven cache attack [GBK11] against AES that first uses an empirically chosen threshold value that defines if a cache hit has occurred or not. However, this very first step introduced noise so they decided to use Artificial Neural Networks (ANNs) [JB96, Ger03, Ger03] to filter out the noise that have been collected while gathering the timing information associated with the accessed memory address and the point in time it occurred. As a result, in their proof-of-concept they claim to successfully recover an AES-128 secret key, therefore, this technique is worth further investigation to refine the measurements in a Time-Driven cache attack context.

10.10 Conclusion

In this chapter, after providing the reader the basics related to the Rijndael (AES) algorithm and the typical time-driven cache attack flow, we have briefly described the possible attack scenarios and the threat model induced by such kind of attack. Furthermore, we have given an overview of Bernstein's time-driven cache attack in order to explain his technique. We also have highlighted the differences and improvements that we have made to adapt the attack in this new environment: a mobile device. We have analysed successive attacks on the same key with the same set of plaintexts, and we have highlighted the fact that there is a relation between the position of a key byte and its rank after the correlation phase. We have particularly shown that the arbitrary fixed threshold that is used while collecting timing information to minimize the sampling noise is not appropriated at all due to the noise distribution. Consequently we have proposed to dynamically compute this threshold to adapt it to the current plaintext that will be involved in the AES encryption we want to observe. Additionally, we have also shown that measuring the execution time from the user space does not produce signification overhead compared to the method with the kernel module. However, it provides the non-negligible ability to perform the measurements without root privileges. Finally, our final results show that using a dynamic threshold and setting the frequency scaling policy to "Performance" improves the clearness of our measurements. The number of samples is also important, and we showed that the best one is comprised between 2^{29} and 2^{30}. To fully recover the secret key, our work misses a crucial step: the evaluation of the key-search complexity. While for now, according to our results (see section 10.8) the key-search step is still impractical, however, techniques exist (see section 10.9) and they seem to greatly reduce the key space. Therefore, though it is very likely that we can still improve our results.

Conclusion and Future Work

In this thesis, different aspects of the hardware and software security of three different microcontrollers embedded in mobile devices have been studied: the (U)SIM, an evolved version of the commonly known SIM card which is basically a smart card. The baseband processor and the application processor (the main CPU) have also been considered. Their security mainly depends on the hardware implementation of the latter and the software implementation of the operating system that manages them. Hence, two categories of attacks have been considered: software attacks and physical attacks.

We first focused on the security of the most implemented operating system for smart cards, which is the Java Card platform. Our research in this area led us to develop a generic framework that allows to test Java Card platforms by designing attack scenarios, and to evaluate Java Card applications, either through test automation or by performing a manual static code review. An important outcome of our investigations is the finding of a new software attack technique that thrives on the fact that improper bounds check while performing memory accesses (read/write) can endanger the other applications and the entire platform. We particularly show that exploiting the Java frame creation mechanism allows to trick the Java Card runtime environment while performing attacks that involve out-of-bounds memory accesses.

One of the consequences of being able to perform out-of-bounds memory accesses is the ability to extract from the device either the whole content of the internal memory, or some parts. Due to the fact that smart cards are considered as very secure vaults, being able to extract to the outside world some information is a powerful capability because it gives the possibility to read the actual values of the targeted sensitive assets. One of the most valuable sensitive asset is the code of other applications. Hence, we also investigated on the forensic analysis of a raw Java Card memory dump. The so-called Index of Coincidence have been already considered in the literature as an heuristic to use in order to identify the code within the latter. However, we also showed that Pearson's correlation coefficient could also be used as an heuristic.

As a second major research topic, we also investigated on the security of baseband processors, the microcontroller that manages the actual "phone" part of mobile devices. We summarized the attack surface and the related work regarding actual local and remote attacks. We also provided our findings related to the techniques to perform security assessment of a baseband processor, and we summarized our findings regarding the functional implementation of the (U)SIM interface within one of the most popular baseband operating system vendors, namely Qualcomm.

Finally, we also studied the security of the application processor against a side-channel attack technique that exploits the variation of the execution time of a whole program due to the hardware implementation of the cache mechanism whose aim is to hide the latency gap between the processor and the main memory. The main objective was to determine the applicability of such kind of attack on actual mobile devices with the least privileges possible. We particularly showed that different parameters involved in the attack implementation have to be considered to face the sampling noise challenge.

Our future work are threefold. First of all by combining our findings related to the secure elements and the baseband processors security, we will focus more on the study of the security impacts of a proprietary baseband operating system that has high privileges, over the secure elements. Furthermore, a new trend has emerged in the area of contactless payment through mobile devices thanks to the NFC technology, namely, the Host Card Emulation (HCE). Such kind of payment system can use two solutions, either the secure element is involved in storing sensitive assets, or they are stored in the cloud. An interesting topic that we already started to investigate is the security risks of real HCE solutions. Finally, we still need to investigate more regarding Time-Driven Cache Attack applicability, as we think that we still can reduce the sampling noise before the key enumeration phase. More importantly, we did not focus yet on that last important phase which may decrease significantly the effort that has to be put in the exhaustive search phase of the key.

Publications

[AJLM^{+}12] Savary Aymerick, Lanet Jean-Louis, Frappier Marc, Razafindralambo Tiana, and Dolhen Josselin. Vtg - vulnerability test generator, a plug-in for rodin. In *Workshop Deploy 2012*, 2012.

[RBICL12] Tiana Razafindralambo, Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Smart card attacks: Enter the matrix. *Sécurité des systèmes Embarqués du GDR SoC-SiP*, May 2012.

[RBL12] Tiana Razafindralambo, Guillaume Bouffard, and Jean-Louis Lanet. A friendly framework for hidding fault enabled virus for Java based smartcard. In Joaquin Garcia-Alfaro Nora Cuppens-Boulahia, Frédéric Cuppens, editor, *26th Annual IFIP WG 11.3 Conference, DBSec 2012.*, volume 7371 of *Lecture Notes in Computer Science*, pages 122–128, Paris, France, July 2012. Springer-Verlag Berlin, Heidelberg.

[TB15] Razafindralambo Tiana and Laugier Benoit. Misuse of Frame Creation to Exploit Stack Underflow Attacks. In *Cardis 2015*, November 2015.

[TBTL12] Razafindralambon Tiana, Guillaume Bouffard, BhagyalekshmyN Thampi, and Jean-Louis Lanet. A dynamic sytntax interpretation for java based smart card to mitigate logical attacks. In *Recent Trends in Computer Networks and Distributed Systems Security*, volume 335, pages 185–194. 2012.

Bibliography

[3gpa] Ts 11.11 specification, "specification of the subscriber identity module - mobile equipment interface", http://www.3gpp.org/dynareport/1111.htm.

[3GPb] 3GPP. 3g ts 23.038, technical specification, alphabets and language-specific information.

[3GPc] 3GPP. Etsi ts 123 040, technical realization of the short message service (sms).

[ABG10] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 110–124. Springer, 2010.

[Aci07] Onur Aciiçmez. Yet another microarchitectural attack:: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, pages 11–18, New York, NY, USA, 2007. ACM.

[ACK$^+$95] Remzi H Arpaci, David E Culler, Arvind Krishnamurthy, Steve G Steinberg, and Katherine Yelick. Empirical evaluation of the cray-t3d: A compiler perspective. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 320–331. ACM, 1995.

[ACL03a] June Andronick, Boutheina Chetali, and Olivier Ly. Using coq to verify java card applet isolation properties. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 335–351. Springer, 2003.

[ACL03b] June Andronick, Boutheina Chetali, and Olivier Ly. Using Coq to Verify Java Card Applet Isolation Properties. In D. A. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 335–351, Rome, Italy, sep 2003. Springer.

[AES15] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 591–604, 2015.

[AJLM$^+$12] Savary Aymerick, Lanet Jean-Louis, Frappier Marc, Razafindralambo Tiana, and Dolhen Josselin. Vtg - vulnerability test generator, a plug-in for rodin. In *Workshop Deploy 2012*, 2012.

[AK96] Ross Anderson and Markus Kuhn. Tamper resistance: A cautionary note. In *Proceedings of the 2Nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[AKS06] Onur Aciicmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'07, pages 225–242, Berlin, Heidelberg, 2006. Springer-Verlag.

[Ale] Bogdan Alecu. Sms fuzzing–sim toolkit attack.

[AS07a] Onur Aciicmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 80–91. IEEE, 2007.

[AS07b] Jonathan Afek and Adi Sharabani. Dangling pointer, 2007.

[Bar12] Guillaume Barbu. *On the security of Java Card platforms against hardware attacks.* PhD thesis, Telecom ParisTech, 2012.

[BBK] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of gsm encrypted communication. In *Advances in Cryptology, proceedings of CRYPTO 2003, Lecture Notes in Computer Science 2729*, pages 600–616. Springer.

[BBKN12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[BDH11] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java Card Operand Stack : Fault Attacks, Combined Attacks and Countermeasures. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications: 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011*, volume 7079 of *Lecture Notes in Computer Science / Security & Cryptology*, pages 297–313, Leuven, Belgium, September 2011. Springer.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of java card programs. In *Revised Papers from the First International Workshop on Java on Smart Cards: Programming and Security*, JavaCard '00, pages 6–24, London, UK, UK, 2001. Springer-Verlag.

[Ben12] Christoforus Juan Benvenuto. Galois field in cryptography. 2012.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[BGDH12] Sara S Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-mei W Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *ACM SIGPLAN Notices*, volume 47, pages 23–34. ACM, 2012.

[BICL11] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In *Smart Card Research and Advanced Applications*, pages 283–296. Springer, 2011.

[BIL] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, volume 7079 of *Lecture Notes in Computer Science*, pages 283–296. Springer.

[Bir11] Alex Biryukov. Codebook attack. In *Encyclopedia of Cryptography and Security*, pages 216–216. Springer, 2011.

[BIYC06] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–12. IEEE, 2006.

[BKKS13] Guillaume Bouffard, Tom Khefif, Ismael Kane, and Sergio Casanova Salvia. Accessing Secure Information using Export file Fraudulence. In *CRiSIS*, pages 1–5, La Rochelle, France, October 2013.

[BL15] Guillaume Bouffard and Jean-Louis Lanet. The ultimate control flow transfer in a java based smart card. *Computers & Security*, 50:33–46, 2015.

[BLLL14] Guillaume Bouffard, Michael Lackner, Jean-Louis Lanet, and Johannes Loinig. Heap ... hop! heap is also vulnerable. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 18–31. Springer, 2014.

[Bou14] Guillaume Bouffard. *A Generic Approach for Protecting Java Card™ Smart Card Against Software Attacks.* PhD thesis, Université de Limoges, 2014.

[Bra04] Gilad Bracha. Generics in the java programming language. 2004.

[BTG10] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on java card 3.0 combining fault and logical attacks. In *Proceedings of the 9th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Application*, CARDIS'10, pages 148–163, Berlin, Heidelberg, 2010. Springer-Verlag.

[cac] Cachegrind: a cache and branch-prediction profiler, http://valgrind.org/docs/manual/cg-manual.html.

[Cal13] Andrew Calafato. An analysis of the vulnerabilities introduced with java card 3 connected edition, 2013.

[CCD00] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In *Cryptographic Hardware and Embedded Systems—CHES 2000*, pages 252–263. Springer, 2000.

[CDSS08] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster. Visual reverse engineering of binary and data files. In *Proceedings of the 5th International Workshop on Visualization for Computer Security*, VizSec '08, pages 1–17, Berlin, Heidelberg, 2008. Springer-Verlag.

[cod] Full disclosure forum, "preferred roaming list zero intercept attack", http://seclists.org/fulldisclosure/2014/aug/14.

[cod11] coderman. Full disclosure forum, "cdma and 4g android hacking", http://seclists.org/fulldisclosure/2011/aug/89, 2011.

[CS11] Keith Cooper and Jeffrey Sandoval. Portable techniques to find effective memory hierarchy parameters. *Computer Science Department, Rice University, Technical Report CS TR11-06*, 2011.

[DBBL13] Jean Dubreuil, Guillaume Bouffard, N. Thampi Bhagyalekshmy, and Jean-Louis Lanet. Mitigating Type Confusion on Java Card. *International Journal of Secure Software Engineering*, 4(2):19–39, 2013.

[Del11] Guillaume Delugré. Reverse engineering a qualcomm baseband, 2011.

[Del12] Guillaume Delugré. qcombbdbg, debugger for the qualcomm baseband chip msm6280, 2012.

[Den68] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[DG02] Damien Deville and Gilles Grimaud. Building an "impossible" verifier on a java card. In *Proceedings of the 2Nd Conference on Industrial Experiences with Systems Software - Volume 2*, WIESS'02, pages 2–2, Berkeley, CA, USA, 2002. USENIX Association.

[DMPH05] Werner Dietl, Peter Müller, and Arnd Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and analysis of safe, secure, and interoperable smart devices : international workshop : revised selected papers / CASSIS 2004, Marseille, France, March 10-14, 2004*, volume 3362, pages 129–150, Berlin, 2005. Springer.

[Dre07] Ulrich Drepper. What every programmer should know about memory. 2007.

[ETSa] ETSI. Etsi ts 136 331 technical specification v13.0.0 release 13, 2016-01, "lte: Radio resource control (rrc)".

[ETSb] ETSI. Etsi ts 136 355 technical specification v13.0.0 release 13, 2016-01, "lte positionning protocol (lpp)".

[ETSc] ETSI. Etsi ts 144 031 technical specification v13.0.0 release 13, 2016-01, "radio resource lcs protocol (rrlp)".

[ETSd] ETSI. Etsi ts 22.101, version 8.7.0 release 8, http://www.etsi.org/deliver/etsi_ts /122100_122199/122101/08.07.00_60/ts_122101v080700p.pdf.

[Fau13] Emilie Faugeron. Manipulate frame information with an underflow attack undetected by the off-card verifie. 2013.

[fm] Event-B formal method. http://www.event-b.org.

[Fre] FreeDesktop. Modemmanager, https://www.freedesktop.org/wiki/software/modemmanager/.

[Fri87] William F. Friedman. *The Index of coincidence and its applications in cryptanalysis.* Aegean park, Laguna Hills, CA, 1987.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 154–, Washington, DC, USA, 2003. IEEE Computer Society.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

[GDTF$^+$10] Jorge González-Domínguez, Guillermo L Taboada, Basilio B Fraguela, María J Martín, and Juan Touri. Servet: A benchmark suite for autotuning on multicore clusters. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–9. IEEE, 2010.

[Ger03] Carlos Gershenson. Artificial neural networks for beginners. *arXiv preprint cs/0308031*, 2003.

[GF11] Nico Golde and Anja Feldmann. *SMS Vulnerability Analysis on Feature Phones.* PhD thesis, Citeseer, 2011.

[GMW15] Daniel Gruss, Clémentine Maurice, and Klaus Wagner. Flush+Flush: A stealthier last-level cache attack. In *Submitted on ArXiv, 14 November 2015*, 11 2015.

[Gol97] Jovan Dj. Golic. Cryptanalysis of alleged a5 stream cipher. In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 1997.

[GS13] Benoît Gérard and François-Xavier Standaert. Unified and optimized linear collision attacks and their application in a non-profiled setting: extended version. *J. Cryptographic Engineering*, 3(1):45–58, 2013.

[gsd] Github repository with some documentation related to msm8960, https://github.com/yangsongx/telecomm.

[gst] Patent: "method and device for obtaining network state information",https://www.google.com/patents/ep2519041b1.

[Gui12] Barbu Guillaume. On the security of java card platforms against hardware attacks. ph.d. thesis, telecom paristech, 2012.

[HBL$^+$12] Samiya Hamadouche, Guillaume Bouffard, Jean-Louis Lanet, Bruno Dorsemaine, Bastien Nouhant, Alexandre Magloire, and Arnaud Reygnaud. Subverting byte code linker service to characterize java card api. In *Seventh Conference on Network and Information Systems Security (SAR-SSI)*, pages 75–81, May 2012.

[HFR13] Thomas Heller, Dietmar Fey, and Markus Rehak. An auto-tuning approach for optimizing base operators for non-destructive testing applications on heterogeneous multi-core architectures. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, pages 1–9. IEEE, 2013.

[HM] Jip Hogenboom and Wojciech Mostowski. Full memory read attack on a java card.

[HOM06] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 175–185, New York, NY, USA, 2006. ACM.

[HP04] E Hubbers and E Poll. Transactions and non-atomic api calls in java card: specification ambiguity and strange implementation behaviours. *Radboud University Nijmegen, Dept. of Computer Science NIII-R0438*, 2004.

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[Hyp03] Konstantin Hyppönen. Use of cryptographic codes for bytecode verification in smartcard environment. *Mémoire de master, Université de Kuopio*, 2003.

[ICL10a] Julien Iguchi-Cartigny and Jean-Louis Lanet. Developing a trojan applets in a smart card. *Journal in Computer Virology*, 6(4):343–351, 2010.

[ICL10b] Julien Iguchi-Cartigny and Jean-Louis Lanet. Developing a trojan applets in a smart card. *J. Comput. Virol.*, 6(4):343–351, November 2010.

[JB96] Michael I. Jordan and Christopher M. Bishop. Neural networks. *ACM Comput. Surv.*, 28(1):73–75, March 1996.

[KCY98] Arvind Krishnamurthy, David E Culler, and Katherine Yelick. Empirical evaluation of global memory support on the cray-t3d and cray-t3e. Technical report, DTIC Document, 1998.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer, 1999.

[Koc96] Paul C. Kocher. *Advances in Cryptology — CRYPTO '96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings*, chapter Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, pages 104–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[KZZ06] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 18–24, New York, NY, USA, 2006. ACM.

[Lan] Julien Lancia. Jacarta: un framework dâ€™exploit java card.

[LB15] Julien Lancia and Guillaume Bouffard. Java Card Virtual Machine Compromising from a Bytecode Verified Applet. In *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany*, November 2015.

[LBB$^+$07] Timothy Robert Leek, Graham Z Baker, Ruben Edward Brown, Michael A Zhivich, and Richard P Lippmann. Coverage maximization using dynamic taint tracing. Technical report, DTIC Document, 2007.

[LBL$^+$13] Michael Lackner, Reinhard Berlach, Johannes Loinig, Reinhold Weiss, and Christian Steger. Towards the hardware accelerated defensive virtual machine–type and bound protection. In *Smart Card Research and Advanced Applications*, pages 1–15. Springer, 2013.

[LBL$^+$14] Jean-Louis Lanet, Guillaume Bouffard, Rokia Lamrani, Ranim Chakra, Afef Mestiri, Mohammed Monsif, and Abdellatif Fandi. Memory forensics of a java card dump. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2014.

[LGSM15] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Armageddon: Last-level cache attacks on mobile devices. *CoRR*, abs/1511.04897, 2015.

[LJL05] Jung Youp Lee, Seok Won Jung, and Jongin Lim. *Testing of Communicating Systems: 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, May 31 - June, 2005. Proceedings*, chapter Detecting Trapdoors in Smart Cards Using Timing and Power Analysis, pages 275–288. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[Man00] Charles C Mann. The end of moores law. *Technology Review*, 103(3):42–48, 2000.

[Man02] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *Information Security and Cryptology—ICISC 2002*, pages 343–358. Springer, 2002.

[Mar03] Witteman Marc. Java card security, 2003.

[MGS11] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. Sms of death: From analyzing to attacking mobile phones on a large scale. In *USENIX Security Symposium*, 2011.

[Mic14] Benoit Michau. Analyse de la sécurité des modems des mobiles. In *Symposium sur la sécurité des technologies de l'information et des communications SSTIC*, 2014.

[Mir11] Luis Miras. Baseband playground. *Ekoparty,*, 2011.

[MK99] Michael Montgomery and Ksheerabdhi Krishna. Secure object sharing in java card. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, WOST'99, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[MM09a] Collin Mulliner and Charlie Miller. Fuzzing the phone in your phone. *Black Hat USA*, 25, 2009.

[MM09b] Collin Mulliner and Charlie Miller. Injecting sms messages into smart phones for security analysis. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[Mos07] Wojciech Mostowski. Fully verified java card api reference implementation. In *In Proceedings, 4th International Verification Workshop (VERIFY'07), Workshop at CADE-21*, 2007.

[MP07] Wojciech Mostowski and Erik Poll. Testing the java card applet firewall. Technical report, 2007.

[MP08a] Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards: Attacks and countermeasures. In *Proceedings of the 8th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications*, CARDIS '08, pages 1–16, Berlin, Heidelberg, 2008. Springer-Verlag.

[MP08b] Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards: Attacks and countermeasures. In *Smart Card Research and Advanced Applications*, pages 1–16. Springer, 2008.

[MPL04] Wes Masri, Andy Podgurski, and David Leon. Detecting and debugging insecure information flows. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 198–209. IEEE, 2004.

[MS$^+$96] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[MZLC14] Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. Benchmarking the memory hierarchy of modern gpus. In *Network and Parallel Computing*, pages 144–156. Springer, 2014.

[Nev06] Michael Neve. Cache-based vulnerabilities and spam analysis. 2006.

[Noh] Karsten Nohl. Attacking phone privacy.

[Noh13] Karsten Nohl. Rooting sim cards, security research labsi, black hat 2013, July 22 2013.

[NS05] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[NSW06] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at bernstein's aes side-channel analysis. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 369–369, New York, NY, USA, 2006. ACM.

[Nus11] Michael Nussbaumer. *Improving reliability and performance of telecommunications systems by using autonomic, self-learning and self-adaptive systems.* PhD thesis, uniwien, 2011.

[oDRG02] The Last Stage of Delirium Research Group. Java and java virtual machine security vulnerabilities and their exploitation techniques, 2002.

[OMA] OMA. Supl 3.0 specifications, http://technical.openmobilealliance.org/technical/technical-information/release-program/current-releases/secure-user-plane-location-v3-0.

[Ora15] Oracle. Java card 3.0.5 platform , virtual machine specification, classic edition, June 2015.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[PCB14] André Pereira, Manuel Correia, and Pedro Brandão. Usb connection vulnerabilities on android smartphones: default and vendors' customizations. In *Communications and Multimedia Security*, pages 19–32. Springer, 2014.

[pera] perf: Linux profiling with performance counters, https://perf.wiki.kernel.org/index.php/main_page.

[perb] Perf_event_open syscall manual page, http://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[Per02] Daniel Perovich. Secure object sharing development kit for java card, 2002.

[PPL14] Benjamin Putt, Ethan Putt, and Jean-Louis Lanet. Using side channel information for improving data partitionning strategy to test smart cards. 2014.

[PSAW09] Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Henry Wong. Micro-benchmarking the gt200 gpu. *Computer Group, ECE, University of Toronto, Tech. Rep*, 2009.

[qem] Qemu, open source processor emulator.

[Qua11] Qualcomm. Qualcomm cdma technologies announces development of gpsone global position location technology solution, https://www.qualcomm.com/news/releases/1999/10/11/qualcomm-cdma-technologies-announces-development-gpsone-global-position, 1999-10-11.

[QWL$^+$06] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[RBL12] Tiana Razafindralambo, Guillaume Bouffard, and Jean-Louis Lanet. A friendly framework for hidding fault enabled virus for Java based smartcard. In Joaquin Garcia-Alfaro Nora Cuppens-Boulahia, Frédéric Cuppens, editor, *26th Annual IFIP WG 11.3 Conference, DBSec 2012.*, volume 7371 of *Lecture Notes in Computer Science*, pages 122–128, Paris, France, July 2012. Springer-Verlag Berlin, Heidelberg.

[RL13] Thomas Roche and Victor Lomné. Collision-correlation attack against some 1st-order boolean masking schemes in the context of secure devices. In *Constructive Side-Channel Analysis and Secure Design*, pages 114–136. Springer, 2013.

[RLS12] Michael Roland, Josef Langer, and Josef Scharinger. Practical attack scenarios on secure element-enabled mobile devices. In *Near Field Communication (NFC), 2012 4th International Workshop on*, pages 19–24. IEEE, 2012.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[SAB10] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010.

[Sal] Robert G Salembier. Analysis of cache timing attacks against aes.

[SFLL15] Aymerick Savary, Marc Frappier, Michael Leuschel, and Jean-Louis Lanet. Model-based robustness testing in event-b using mutation. In Radu Calinescu and Bernhard Rumpe, editors, *SEFM*, volume 9276 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2015.

[SG14] Raphael Spreitzer and Benoit Gérard. Towards more practical time-driven cache attacks. In David Naccache and Damien Sauveron, editors, *Information Security Theory and Practice. Securing the Internet of Things - 8th IFIP WG 11.2 International Workshop, WISTP 2014, Heraklion, Crete, Greece, June 30 - July 2, 2014. Proceedings.*, volume 8501 of *Lecture Notes in Computer Science*, pages 24 – 39. Springer, 2014.

[SHP09] Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos. Optical fault attacks on aes: A threat in violet. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*, pages 13–22. IEEE, 2009.

[Sko05] Sergei Petrovich Skorobogatov. *Semi-invasive attacks: a new approach to hardware security analysis.* PhD thesis, Citeseer, 2005.

[Sko10] Sergei Skorobogatov. Optical fault masking attacks. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 23–29. IEEE, 2010.

[SM04] M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *Cluster Computing, 2004 IEEE International Conference on*, pages 371–377, Sept 2004.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[SP13a] Raphael Spreitzer and Thomas Plos. Cache-access pattern attack on disaligned aes t-tables. In *Constructive Side-Channel Analysis and Secure Design*, pages 200–214. Springer, 2013.

[SP13b] Raphael Spreitzer and Thomas Plos. On the applicability of time-driven cache attacks on mobile devices (extended version). *IACR Cryptology ePrint Archive*, 2013:172, 2013.

[SS95] Rafael H Saavedra and Alan Jay Smith. Measuring cache and tlb performance and their effect on benchmark runtimes. *Computers, IEEE Transactions on*, 44(10):1223–1235, 1995.

[ST] Xlim SSD Team. Capmap, https://bitbucket.org/ssd/capmap-free.

[Svea] Petr Svenda. Jcalgtest.

[Sveb] Petr Svenda. Jcalgtest - database of supported javacard algorithms.

[SW07] Karan Singh and Vincent Weaver. Learning models in self-optimizing systems. *COM S612—Software Design for High Performance Architectures*, 2007.

[TB15] Razafindralambo Tiana and Laugier Benoit. Misuse of Frame Creation to Exploit Stack Underflow Attacks. In *Cardis 2015*, November 2015.

[TBTL12] Razafindralambon Tiana, Guillaume Bouffard, BhagyalekshmyN Thampi, and Jean-Louis Lanet. A dynamic sytntax interpretation for java based smart card to mitigate logical attacks. In *Recent Trends in Computer Networks and Distributed Systems Security*, volume 335, pages 185–194. 2012.

[tia] C.s0022-0 v3.0, version 3.0, february 16, 2001, "position determination service standards for dual mode spread spectrum systems".

[ts1] Ts 11.14 specification, "specification of the sim application toolkit for the subscriber identity module", http://www.etsi.org/deliver/etsi_gts/11/1114/05.04.00_60/gsmts_1114v050400p.pdf.

[uim] Patent: User identity module protocol switch, https://www.google.fr/patents/us9172418?dq=user+identity+module+qualcomm&hl=en&sa= ylmahxewrqkha6gbzsq6aeijdab.

[VD08] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[vdBHT14] Fabian van den Broek, Brinio Hond, and Arturo Cedillo Torres. Security testing of gsm implementations. In *Engineering Secure Software and Systems*, pages 179–195. Springer, 2014.

[Wea10] Vincent M Weaver. Can hardware performance counters produce expected, deterministic results. 2010.

[Wea13] Vincent M Weaver. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, page 80, 2013.

[Wei12a] Ralf-Philipp Weinmann. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In Elie Bursztein and Thomas Dullien, editors, *WOOT*, pages 12–21. USENIX Association, 2012.

[Wei12b] Ralf-Philipp Weinmann. Security issues with supl implementations, blackhat us, 2012.

[Wei13] Ralf-Philipp Weinmann. Baseband exploitation in 2013: Hexagon challenges, pacsec 2013, 2013.

[Wel10] Harald Welte. Anatomy of contemporary gsm cellphone hardware. 2010.

[WHS12] Michael Weiss, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on aes in virtualization environments. In *14th International Conference on Financial Cryptography and Data Security (Financial Crypto 2012)*, Lecture Notes in Computer Science. Springer, 2012.

[Wit02] Marc Witteman. Advances in smartcard security. *Information Security Bulletin*, 7(2002):11–22, 2002.

[WPSAM10] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.

[YF14] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.

[YPS05] Kamen Yotov, Keshav Pingali, and Paul Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 168–177. IEEE, 2005.

[ZBBF14] Jonas Zaddach, Luca Bruno, Davide Balzarotti, and Aurelien Francillon. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security (NDSS) Symposium*, 2014.

[ZO11] Yao Zhang and John D Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. IEEE, 2011.

List of Figures

Résumé

Afin de pouvoir profiter de services sécurisés, efficaces et rapides (ex: paiement mobile, agenda, télécommunications, vidéos, jeux, etc.), de nos jours nos téléphones embarquent trois différents micro-controlleurs. Du plus sécurisé vers le plus générique nous avons, la carte SIM qui n'est autre qu'une carte à puce sécurisé chargée de garder de manière sûr au sein de sa mémoire des données sensibles. Ensuite, nous avons le processeur à bande de base qui est le seul à pouvoir discuter avec la carte SIM, et s'occupe de se charger des fonctions radio du téléphone (ex: le réseau GSM/3G/4G/LTE). Et enfin, nous avons le processeur applicatif, qui se charge d'exécuter tous les autres programmes sur le téléphone.

Ce qui rend ces micro-controlleurs plus particuliers, c'est le fait qu'ils sont chacun contrôlés par un système d'exploitation totalement indépendant. Néanmoins, chacun peut avoir son influence, direct ou indirect sur l'autre/les autres.

La sécurité de ces trois plateformes dépendent non seulement de leur implémentations matérielles, mais aussi de l'implémentation logicielle de leur système d'exploitation. Cette thèse s'intéresse à la sécurité logicielle, et en partie, matérielle de ces trois plateformes, afin de comprendre dans quelle mesure, une carte à puce telle que la carte SIM, est-elle résistante aux attaques logicielles dans le contexte d'un environnement multi-applicatif offert par les appareils mobiles. Nous nous intéressons aussi, à la sécurité du processeur applicatif face à une famille particulière d'attaque qui exploite le mécanisme de mémoire cache.

Nous partons alors de l'étude et de l'application en pratique des attaques logiques sur carte à puce. Après avoir étudié les différents moyens qui permettent d'atteindre la carte SIM dans un mobile et ainsi d'étudier la surface d'attaque, nous poursuivons vers une étude par rétro-conception de l'implémentation de l'interface logicielle qui communique directement avec la SIM au niveau du processeur de bande de base. Ceci afin de comprendre le fonctionnement de cette partie très peu documentée. Finalement, nous étudions les effets du mécanisme de cache sur l'exécution d'un programme dans un téléphone mobile. Enfin, nous avons commencé à étudier l'attaque de Bernstein, et plus particulièrement, par une mise en pratique, nous essayons de déterminer ce qui exacerbe ou non la réalisation de sa technique dans le contexte d'un téléphone mobile réel.

Abstract

Nowadays, in order to provide secure, reliable and performant services (e.g: mobile payments, agenda, telecommunication, videos, games, etc.), smartphones embed three different micro-controllers. From the most secure to the most general purpose one, we have the SIM card which is a secure smart card that has to prevent anyone by any means to exfiltrate sensitive assets from its internal memories. Furthermore, we also have the baseband processor, which is the only one that directly talks with the SIM card. It essentially manages all the "phone" parts (e.g: GSM/3G/4G/LTE networks) inside a mobile device. Finally, we have the application processor which runs all the general user applications.

What is interesting to note for those three micro-controllers is that they are controlled by different and independent operating systems. However, one may affect the behavior of the other(s).

The security of these three platforms depend on their hardware and software implementations. This thesis is concerned with the security of these three microcontrollers that are managed by independent OSs within mobile devices. We particularly focused on understanding to what extent a smart card such as SIM cards can be resistant to software attacks in the context of a multi-application environment provided by mobile devices. We were also interested in a specific family of, so-called cache attacks, namely time-driven one, as this kind of technique essentially exploits the hardware implementation of the different cache memories and the mechanisms that enable to manage them.

We decided to first study and experimentally perform so-called logical attacks on smart cards. In a second step, in order to understand the attack surface, we have studied the different means to reach the SIM card from both the baseband processor and the application processor. Then, by means of reverse engineering, we tried to understand how was implemented the SIM interface from the baseband side. Finally, we have studied the cache effects on the execution speed of a program on real mobile devices, and we experimentally studied Bernstein's time-driven cache attack in order to understand what possible events/mechanisms exacerbate (or not) the achievement of the latter on an actual mobile device.