University of Twente

European Institute
of
Innovation and Technology

Security and Privacy

Master Thesis

Java Card Bytecode Verification

Designing a novel verification system

Author

Alessio Parzian

Supervisors

Prof. Dr. Marieke Huisman
University of Twente

Drs. Jan Brands, PDEng
NXP Semiconductors

Reader

Prof. Dr. Massimiliano Sala
University of Trento

August 5, 2015

Contents

1 Introduction **3**

I State of the Art **7**

2 Java Card **9**
2.1 Advantages and Benefits . . . 10
2.2 Architecture . . . 10
2.2.1 Java Card Virtual Machine . . . 12
2.2.2 Java Card Runtime Environment . . . 17
2.2.3 Java Card APIs . . . 19
2.3 Programming Models . . . 21

3 Security-related Aspects **23**
3.1 Security Concepts . . . 24
3.1.1 CAP Verification . . . 24
3.1.2 Applet Loading . . . 24
3.1.3 Applet Isolation . . . 24
3.1.4 Operation Atomicity . . . 25
3.2 Security Threats . . . 25
3.3 Security Mechanisms . . . 26
3.3.1 The Java Programming Language . . . 26
3.3.2 The JCVM Architecture . . . 27
3.3.3 The JCRE Security Mechanisms . . . 28
3.4 Security Challenges . . . 35
3.4.1 Reviewing Applet Source Code and Bytecode . . . 35
3.4.2 Verification and Signing of the CAP File . . . 35
3.4.3 Securing the Applet Loading Mechanism . . . 36
3.4.4 Reinforcement of the Java Card Platform . . . 36
3.4.5 Stakeholder Awareness and Cooperation . . . 36

4 Attack Nomenclature **39**
4.1 Logical Attacks . . . 41
4.1.1 Vulnerabilities . . . 41
4.1.2 Countermeasures . . . 43
4.2 Physical Attacks . . . 44
4.2.1 Vulnerabilities . . . 44
4.2.2 Countermeasures . . . 45
4.3 Side Channel Attacks . . . 46
4.3.1 Side Channel Analysis Vulnerabilities . . . 46
4.3.2 Side Channel Analysis Countermeasures . . . 47

4.3.3 Side Channel Manipulation Vulnerabilities . . . 48
4.3.4 Side Channel Manipulation Countermeasures . . . 48

5 Attack Vectors **51**
5.1 Power Analysis and Manipulation . . . 52
5.1.1 Differential Power Analysis . . . 52
5.1.2 Fault Injection . . . 52
5.2 Applet Exploitation . . . 54
5.3 Type Confusion . . . 54
5.3.1 Obtaining rights to load code . . . 55
5.3.2 Injection of ill-formed code . . . 55
5.3.3 Running a developed attack vector . . . 63

II An Augmented Bytecode Verifier **65**

6 A Solution to the Physical Secure Element Issue **67**

7 Requirements Analysis **71**
7.1 Business Model . . . 72
7.2 Architectural Features . . . 73
7.3 Protection Techniques . . . 75
7.3.1 Data Encryption . . . 76
7.3.2 Runtime Integrity Checks . . . 76
7.3.3 Code Obfuscation and Code Flattening . . . 76
7.3.4 White-box Cryptography . . . 77
7.4 Key Points . . . 78

8 A Scenario Analysis **79**
8.1 Scenario Definition . . . 80
8.1.1 A Scenario Story . . . 80
8.1.2 Scenario Properties . . . 81
8.1.3 Stakeholder's Phases . . . 81
8.2 Card Issuers . . . 82
8.2.1 Activation . . . 82
8.2.2 Usage . . . 83
8.2.3 Distribution . . . 85
8.3 Application Developers . . . 87
8.3.1 Activation . . . 87
8.3.2 Usage . . . 88
8.3.3 Distribution . . . 89
8.4 Key Points . . . 92

9 Technical Specification and Design **93**
9.1 Design Choices . . . 94

9.2 The System Architecture . . . 96
9.2.1 The Protocol Concept . . . 98
9.2.2 Runtime Integrity Checks . . . 105
9.2.3 Code Obfuscation and Code Flattening . . . 107
9.2.4 White-box Crypto . . . 108
9.2.5 Binary Encryption . . . 108
9.3 Enforcing Edge Case Features . . . 109
9.3.1 Key Renewability . . . 109
9.3.2 License Revocation . . . 109
9.3.3 Verifier Binary Diversity . . . 109
9.4 Implementation Directions . . . 110
9.5 Key Points . . . 111

10 Conclusions and Future Work **113**
10.1 Improved Secure Element Management . . . 113
10.2 Related Work . . . 114
10.3 Future Work . . . 115

Bibliography **117**

List of Figures

I The Java Card architecture . . . 11
II The Java Card Virtual Machine . . . 13
III Finite state machine of the Java Card interpreter . . . 16
IV The Java Card applet development . . . 17
V The Java Card applet deployment . . . 17
VI The Java Card Runtime Environment architecture . . . 18

VII Example of contexts within the Java Card platform. . . . 30

VIII The Java Card attack nomenclature . . . 40

IX Attacking the checkcast check . . . 60
X Java and Java Card memory addressing . . . 62

XI The off-card verifier working principle . . . 68

XII The traditional cryptography conception . . . 74
XIII States of the software . . . 75

XIV Card Issuer Verifier Usage . . . 84
XV Legend Message Sequence Card Issuer . . . 85
XVI Message Sequence Card Issuer . . . 86
XVII Application Developer Verifier Usage . . . 88
XVIII Legend Message Sequence Application Developer . . . 90
XIX Message Sequence Application Developer Distribution . . . 90
XX Message Sequence Application Developer . . . 91

XXI The token states . . . 95
XXII The verifier states . . . 96
XXIII The *Usage* phase from a verifier perspective . . . 97
XXIV The protocol concept I . . . 103
XXV The protocol concept II . . . 104
XXVI The protocol concept legend . . . 104

List of Tables

I Comparison between Java Card and traditional Java 14

II The access level permitted by each identifier in Java 27

III Power Manipulation and Analysis security overview 52
IV Type Confusion security overview 56

V Attacks considered 68

VI Stakeholders' business needs priority 73

Chapter 1

Introduction

"The convergence of payments and mobile communications is not just logical – it is inevitable". In March 2007, John Philip Coghlan, then CEO of Visa USA, made this announcement at the CTIA Wireless Conference. Such a convergence is claimed to be inevitable for essentially three reasons [Alliance 2009]:

- *Contactless payment adoption.* Payment brands, issuers and consumers adopts contactless payment solutions due to its speed, easy of use and security, while merchants adopt it because of faster transaction time, increased spending and higher loyalty. Moreover, the contactless infrastructure is built on top of the existing financial network, therefore merchants are required only to upgrade their point-of-sale (POS) to contactless-enabled POS with negligible costs.

- *Mobile device ubiquity.* Mobile phone subscribers do not leave home without their phones. In addition, near field communication (NFC) technology has become an international communication standard to deliver simplified and robust implementation of contactless payments using mobile devices due to its secure nature [Coskun 2013]. Today, NFC is a standard functionality provided in most mobile phones.

- *Expanded mobile functionalities.* Mobile devices are powerful tools that can deliver a variety of payment and payment-related services such as proximity mobile payments, remote payments through the mobile Internet or text messaging, and person-to-person money transfers. Value-added applications can enrich the purchase experience and include account management, banking, offers, and security applications.

Many attempts of creating a secure and open mobile *proximity payment system* have been made. Typical examples are *Google Wallet* and *Apple Pay* that are trying to create an homogeneous system on top of the already existing financial circuits such as *Eurocard*, *Amex*, *Mastercard*, *Maestro* and *Visa*. However, they have been only partially successful due to the strict security requirements that the system needs to comply with, the hostile environment in which the system must run and the many business parties involved in providing it, i.e. device manufacturers, application providers and card issuers. Today, interesting technologies that could meet the strict requirements are available in the market, but designing a smooth environment that stakeholders, even competitors, can trust and participate in is the greatest problem. As a result, mobile proximity payment systems are today enabled only in specific countries and still have to gain a foothold not withstanding the huge potential.

Technically speaking, the cornerstone of security in a proximity payment system is the secure element [Alliance 2009]. Essentially, it is a protected area, independent from the application process/operating system of the device, which is capable of storing and processing sensitive information of the device holder. Authentication, encryption of private data, data integrity and non-repudiation are typical services that a secure element provides. One of the solutions provided by *NXP Semiconductors* is the use of a secure element, which consists essentially of a built-in smartcard chip embedded in the device running a *Java Card virtual machine* that communicates with an external terminal by means of NFC. The main innovating concept that has been brought into the market with the introduction of *Java Card* is the multi-application environment. That is, applications (from now on "applets") can run on the same smartcard chip and can be uploaded even after a smartcard issuance. Such a feature might be the key for creating an open, homogeneous and trusted environment, but, due to security reasons, it has never been extensively used. As a consequence, *NXP Semiconductors*, as a smartcard manufacturer, is currently strictly controlling the access to its secure elements enforcing a tight collaboration with card issuers. Usually, applets of different parties never run on the same chip and are never uploaded after the card issuance, even if it is developed by a trusted application provider related to the same card issuer. Obviously, this is a huge limitation that is obstructing the spread of a proximity payment system and is leading to the development of workaround solutions that do not need a physical secure element, such as the *Host-based Card Emulation* approach [Friedman 2004] that is starting to be used by Google. As a secure element manufacturer, *NXP Semiconductors* wants to stop this trend, proving that a secure and versatile environment using a physical secure element can be truly designed and implemented enabling in such a way the use of physical secure elements in mobile devices.

This thesis work investigates the current state of the art of Java Card and propose a verification system design to allow *NXP Semiconductors* to tackle the above-introduced problem enforcing security and trust without losing control on what is uploaded on its secure elements. The document is structured as follows:

- Chapter 2 provides an exhaustive introduction of the Java Card technology in terms of benefits, architecture and standards.
- Chapter 3 focuses on Java Card security aspects, ranging from crucial concepts to threats, from security mechanisms already in place to security challenges.
- Chapter 4 introduces a new framework to classify attack vectors on Java Card by vulnerabilities.
- Chapter 5 discusses the known techniques used to attack Java Card. The explained attack vectors are mapped back to the presented threats in Chapter 3 and the vulnerabilities defined in Chapter 4.
- Chapter 6 introduces the second part of my work introducing our proposal for solving the problem introduced in this chapter.

- Chapter 7 provides a requirements analysis introducing the most relevant aspects that characterize the verifier and impact the software architecture, forcing us to take specific design decisions.
- Chapter 8 presents a new scenario that mitigates the current inflexible context and enforces a high level of security. Firstly, it is described through a scenario story and the properties it needs to guarantee. Secondly, an explanation of how the system works in the eye of the stakeholders, involved in the process of verification, is provided by means of use cases. Thirdly, the infrastructure required to meet the scenario is presented.
- Chapter 9 focuses on the definition of the system architecture in terms of security, presenting concepts of protocols and mechanisms needed to meet the stakeholders' business requirements.
- Chapter 10 discusses the planned future work and draws conclusions on the designed system.

Part I

State of the Art

Chapter 2

Java Card

Contents

2.1 Advantages and Benefits 10
2.2 Architecture 10
2.2.1 Java Card Virtual Machine 12
2.2.2 Java Card Runtime Environment 17
2.2.3 Java Card APIs 19
2.3 Programming Models 21

In this chapter the main aspects of the Java Card technology are introduced and discussed in order to provide the reader with a general understanding of the platform. Since the Java Card technology was conceived from the traditional Java technology, several comparisons between these two technologies are made throughout the entire chapter with the purpose of catching the motivations of Java Card's particular structure and consequentially clarifying their differences and similarities. Section 2.1 explains the concept of Java Card technology along with its benefits. Section 2.2 focuses on the whole architecture, explaining its structure and the role of each component. Its behaviour at runtime is also described in terms of life-cycles, statement processing, and further tasks performed to ensure its proper functioning. Finally, the currently available APIs are introduced package by package. Finally, Section 2.3 gives a general understanding of the programming models used when developing Java Card applets.

2.1 Advantages and Benefits

Java Card is a fast growing technology that enriches the smartcard's world with a whole set of new possibilities. Before its invention all manufacturers had its "proprietary operating system" with native applications developed for specific chips. The applet development was difficult, error-prone, time-consuming and overall non-portable. Ensuring compatibility across different platform was really costly. The idea behind the Java Card technology, conceived from the traditional Java technology, is completely changing this approach, permitting developed applets to be run on any Java technology-enabled smartcard independently of the card vendor and underlying hardware. It was introduced in 1997 and essentially defines a platform on which applets can be written using a dialect of the Java programming language and run in any smartcard regardless of its hardware. Note that, only a dialect of Java is supported because of the strict hardware constraints of smartcards. Along with the "write once, run everywhere" paradigm, this technology has undoubtedly brought many benefits improving the development and deployment of applets: the Java Card API is completely compatible with international standards for smartcards such ISO7816-4 and new applications can be installed after a card issue, enabling card issuers to respond to their customer's changing needs dynamically. The Java Card architecture concept allows also multiple applets, written by different vendors, to be run on the same card without compromising their functionalities. This prepares the basis for the ideal environment where a smartcard can be used for different purposes in accordance with its user's needs. The most relevant innovations that Java Card brought into the market can be outlined as follows:

- *Interoperability*, developed applets can be run on any Java-enabled smartcard.
- *Multi-application*, multiple applets can reside on the same smartcard.
- *Dynamism*, applets can be added after smartcard issuance.
- *Enhanced security*, built-in dedicated security mechanisms are deployed in the architecture.

Today the most recent version of the Java Card technology is version *3.0* Classic and Connected Edition [Microsystems 2008]. However, version *2.2.2* is still the most widely used one. Therefore, considering the fact that version *3.0* Classic Edition is a simple extension of version *2.2.2.*, along this document all references point to features that are present in version *2.2.2*, but still apply to version *3.0* Classic Edition.

2.2 Architecture

The most relevant component of the Java Card technology is the runtime environment that provides a clear separation between the smartcard system and the applets. The runtime environment encapsulates the underlying complexity and details of the smartcard system. Applets simply request system services and resources through a well-defined high-level

programming interface. From an architectural point of view, as shown in Figure I, the Java Card platform is distributed between a smartcard and a desktop environment in both space and time, making this technology extremely flexible and modular.

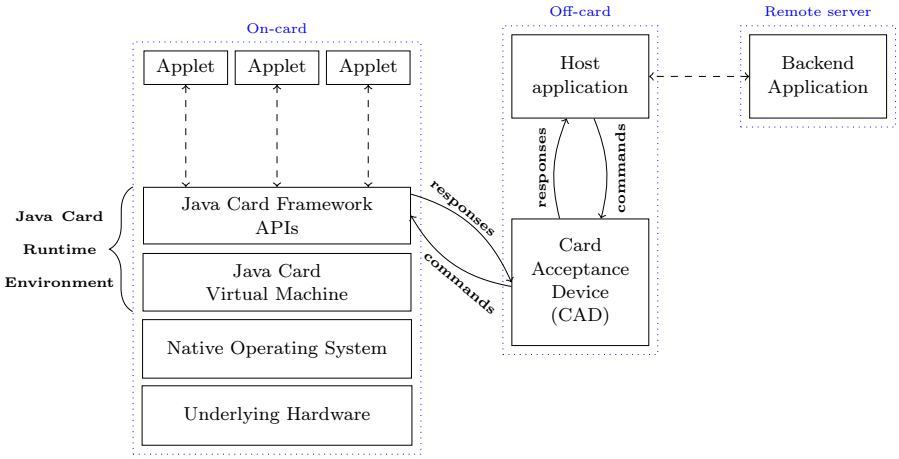

Figure I: The Java Card architecture [Ortiz 2003].

Referring to Figure I, from left to right, the Java Card platform is a multi-application environment. Many applets can reside on-card along with supporting software: the *Java Card Runtime Environment* (JCRE) and the *native smartcard operating system*. The JCRE consists of the Java Card Framework, the APIs and a Java Card virtual machine instance. A Java Card is powered up through a *card acceptance device* (CAD)[1] which provides an interface that sits between the host application and the smartcard. For the sake of clarity, a CAD might be a card reader attached to a workstation as well as an integrated piece into an electronic terminal. An host application sets up a full-duplex communication with a specific applet, that works as a server, through the *Application Data Unit Protocol* (APDU) which is a standard in the smartcard environment. The content of its messages, commands and responses are described in [ISO 2013a]. Sometimes, also a backend application is required in the architecture in order to provide support to applets. A typical example is in an electronic payment system where the back-end application provides access to sensitive payment information before authentication through in-card credentials. Runtime behaviours are described in Section 2.2.2.

Formally, the Java Card platform consists of three parts, each defined in an official specification document:

- **The Java Card Virtual Machine (JCVM)** defines a Java programming language subset and the file formats used to install applets and libraries into Java Card technology-enabled devices [Microsystems 2006d]. Its role can be better understood in the context of the process for development and deployment of applets. Further

[1] Power can be supplied to a smartcard in different ways, by contact or by induction. For further information about smartcards operating principles and their classification, refer to [Rankl 2010, Ch.2].

details can be found in Section 2.2.1.

- **The Java Card Runtime Environment (JCRE)** defines the necessary behavior of the runtime environment in any implementation of the Java Card technology. The JCRE includes the implementation of the Java Card Virtual Machine, the Java Card API classes, and runtime support services such as memory and applet management [Microsystems 2006c]. Further details can be found in Section 2.2.2.

- **The Java Card Application Programming Interface (API)** completes the JCRE APIs implementation providing a description of the Java packages and classes usable for programming smartcard applets. In other words, it reports all packages and classes definitions required to support the JCVM and JCRE [Microsystems 2006a]. Further details can be found in Section 2.2.3.

2.2.1 Java Card Virtual Machine

The Java Card Virtual Machine consists of two separate pieces: the *converter* and the *interpreter*. The converter runs on a workstation and is the off-card piece of the virtual machine. Its purpose is to load all the *.class* files of a Java package and to output an executable file suitable for the Java Card platform. This executable file has a *.CAP* extension which stands for converted applet. The need of a component which converts traditional Java executables, *.class* files, to a Java card executable, *.CAP* file, is dictated by the smartcards hardware constraints. Therefore, the goal of the CAP format is to minimize the size of a Java card executable while semantically preserving the equivalence to the Java executables it represents [Krishna 2001]. Preserving such a semantic equivalence is a requirement in order to actually enable applets written in a subset of the Java programming language to be run properly on Java Card technology-enabled smartcards. The Java Card executable size reduction was enabled by splitting the traditional *.class* Java executables into two parts. The first part, the CAP file, contains the core, that is the essential needed for execution on-card, while the second part, which is called *export file*, contains information needed for linking purpose that is never loaded into the smartcard. Together they represent the Java class package taken as input by the converter. The interpreter runs on the Java card and is the on-card piece of the virtual machine. It takes as input the *.CAP* file previously generated by the converter and executes it. Figure II illustrates the Java Card conversion process above introduced. Since the virtual machine itself does not provide a unit to load and install a CAP file into a smartcard, a tool called *installer*, which resides within the card, has been developed allowing to perform such a task. Each virtual machine's component is now explained individually to remark its tasks and to describe some relevant details. Afterwards, the operational process of an applet life cycle is illustrated before moving to the definition of Java Card Runtime Environment.

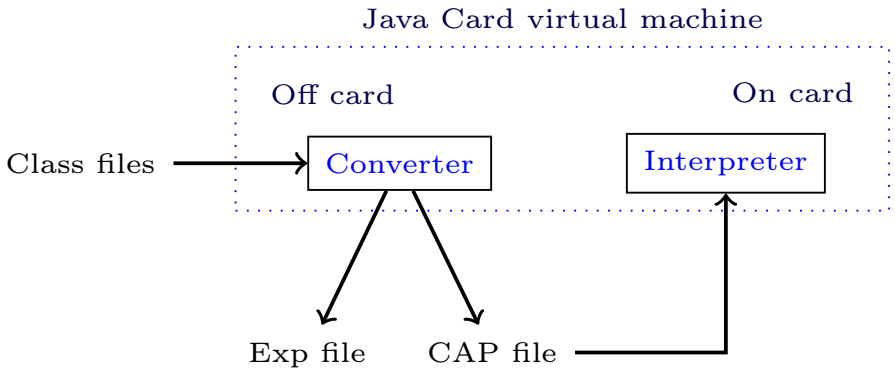

Figure II: The Java Card Virtual Machine

2.2.1.1 Converter

The main purpose of the converter is to translate source files into a format suitable for the runtime environment. Initially, all source files of a Java package are converted by a standard Java compiler into class files which, in turn, are used as input for the actual converter that produces a CAP file. This procedure uses as conversion unit a Java package, unlike the traditional Java virtual machine which processes one class file at a time. Along with the CAP file also an export file is produced with *.exp* extension in order to manage any kind of class which requires the import of other packages. In other words, an export file consists of all public API linking information of classes in a package. Each unicode name of a class, a method or a field is given a unique identifying token. Even if such a file is not loaded onto the smartcard and thus not directly used by the interpreter, an export file is critical to the operation of the on-card virtual machine since it is used for verification and linking purposes. An export file is generated when a package is converted. If another package uses a package previously converted, the information of that export file is included in the CAP file of the package currently processed. Such information is being afterwards used to link the two packages [Microsystems 2006b]. A description of the export file format can be found in [Microsystems 2006d, Ch.5]. Along with the conversion, from source files to a CAP file, the converter preprocesses the classes of the examined package with the goal of keeping the on-card Java virtual machine part as small as possible. That is, it performs some of the tasks that a Java virtual machine normally does when it loads a Java class in a desktop environment. Some of the preprocessing tasks the converter tool performs are to initialize static variables in the classes and to resolve symbolic references. In addition, the converter tool checks whether the Java classes in the package are properly formed, and whether the applets use only the subset of the Java programming language that is supported by the Java Card platform. Table I shows the main differences between the Java Card dialect and the traditional Java programming language. This code inspection is already considered a verification task and is the minimum requirement to thwart the most straightforwards attack vectors that could compromise the integrity of the JCVM implementation, and hence the installed applets. This is an important step as a Java card

does not have to implement an on-board bytecode verification mechanism for performance reasons. As a consequence, the platform trusts that only verified applets are loaded. At the end of the checks, the converter tool directs the result of its preprocessing to the standard output stream.

Functionality	Java Card	Java
Operators	all	all
Sequence control functions	yes	yes
Exception Handling	yes	yes
Data Types: boolean, byte, short	yes	yes
Data Type: int	optional	yes
Data Types: long, float, double, char	no	yes
Fields	one-dimensional	multidimensional
Objects Fields	one-dimensional	multidimensional
Cloning of Classes and Objects	no	yes
Dynamic Object Creation	yes	yes
Static and Virtual Methods	yes	yes
Dynamic Downloading of Classes	no	yes
Load Unit	Package	Class
Interfaces	yes	yes
Dynamic Memory Management (Garbage Collection)	optional	yes
Threads	no	yes

Table I: Functional comparison between Java Card and traditional Java [Rankl 2010, Ch.13].

To sum up, the Java Card converter, when transforming *.class* files of a Java package to a *.CAP* executable, performs the following tasks [Microsystems 1998, Chen 2000, Ch.3]

1. **Verification** – checks that the loaded *.class* images are well-formed with consistent symbol tables and no language violations of the Java Card specification.

2. **Preparation** – allocates the memory and creates the virtual machine data structure needed to represent the classes, create static fields and methods, and initializes static variables to default values.
3. **Resolution** – replaces symbolic references to methods or variables with direct references when possible.

2.2.1.2 Interpreter

The interpreter provides runtime support for the Java language allowing the actual hardware independence. The tasks it performs are [Chen 2000, p.34]:

- Executing bytecode instructions
- Controlling memory allocations and object creation
- Enforcing runtime security (refer to Section 3.3 for further details).

Due to the critical nature of these operations, it directly affects the applet's execution time. This aspect is extremely relevant to industry since applets often have strict timing requirements. For this reason the interpreter is frequently subject to studies for improving its performance. Ideally, a classic interpreter consists of a huge switch wrapped by a while loop [Klint 1981]. Even if this approach suffers from low speed performance, it is still the most feasible solution as its code is simple and compact in term of size. As discussed in [Microsystems 2006d], Java Card Technology 2.2.2 uses a *fetch-code-execute* loop which slightly differs from the traditional Java technology because of the JCVM's more limited support for data types. Figure III illustrates an example of such a loop along with its pseudocode. Essentially, during this loop, the interpreter retrieves a bytecode instruction from its memory, determines what actions the instruction requires, and eventually carries it out.

The design and implementation of the interpreter needs to be improved. Currently, many other approaches that reduce the applets' execution time have been proposed and could be deployed in the near future impacting not only the performance, but also the entire system security which could suffer from exploitable flaws. For the sake of clarity and completeness the most discussed approaches are quickly introduced. A first alternative, discussed in [Piumarta 1998], is the *direct threaded interpreter* (DTI) which is based on a compiler that produces a machine-depended code without losing applets' portability. It improves the performance but the executable size is still too large. Another solution, which could remarkably decrease the execution time of applets, is the use of the *just in time* compilation that is presented and discussed in [Cramer 1997]. In this solution, the bytecode, instead of being interpreted, is converted in optimized machine code. Although this mechanism extremely improves the execution time, it is still too much RAM-consuming. Finally, another interesting approach, discussed in [Zilli 2014], has been recently proposed. It introduces an hardware-software co-design solution which actually improve the interpreter's performance.

```
while(1){
   JBC = JProgram[++JPC];
   JBCFunct=JBCTable[JBC];
   JBCFunct();
}
```

Legend:
JProgram = Java Program
JBC = Java Bytecode
JPC = Java Program Counter

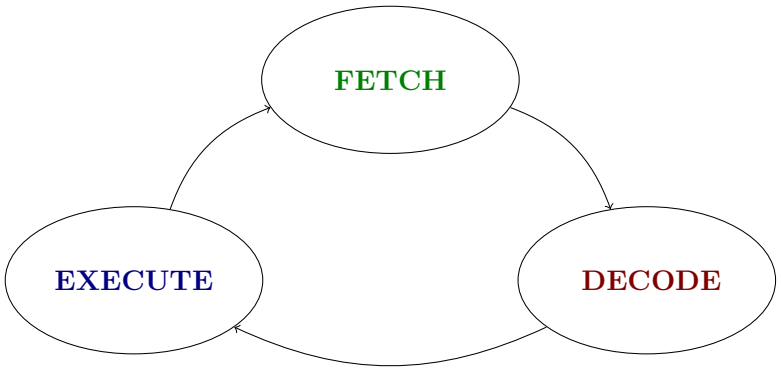

Figure III: Pseudocode and finite state machine of the Java Card interpreter [Zilli 2014].

2.2.1.3 Installer

The purpose of the installer is to load and install a CAP file into a Java card. The installer resides within the smartcard and cooperates with an off-card installation program to perform its tasks. The off-card program transmits a CAP file to the installer, through the CAD, which, in turn, checks whether the card's available memory resources are sufficient and if the optional Java Card specification features, like for example the *int* type, are supported and used by the CAP file. In case of positive responses, it writes the received file into the smartcard memory, performs any needed linking and initializes any data structures that are used later by the runtime environment. The installer has not been embedded into the interpreter in order to minimize its size and to provide more granularity and flexibility for installer implementations [Chen 2000, p.34]. An important aspect of the installation process to realize is the fact that applets are uploaded instead of downloaded. In other words, the authority that controls the smartcard providing users with services decides to move code onto a card or a batch of cards. In the standard Java environment applications are downloaded by users who have much more control over the content of his Java implementation.

The Java Card Virtual Machine operational process can now be introduced in term of applet life cycle, which is divided in development and deployment phases. Figure IV illustrates the transformations an applet undergoes after its design (details about the programming paradigms in Section 2.3) and implementation in the development phase. The Java source code is compiled by a standard Java compiler whose output is taken in by the converter that verifies whether the generated bytecode complies to the Java Card

specifications. Figure V illustrates the steps performed during the deployment phase. The verified bytecode is converted to CAP format and uploaded using the installer.

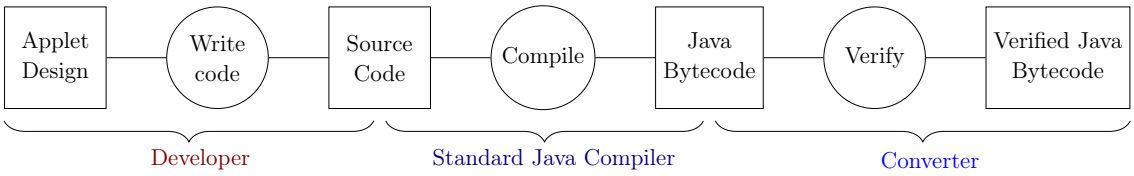

Figure IV: The Java Card applet development

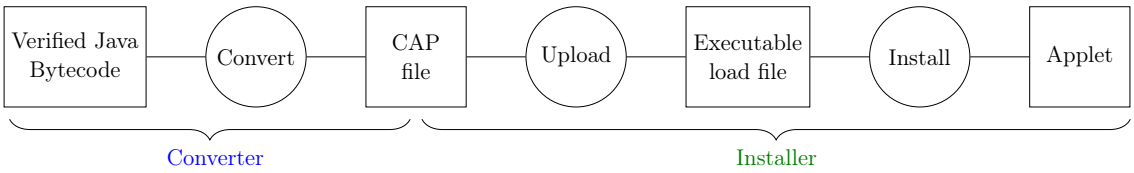

Figure V: The Java Card applet deployment

2.2.2 Java Card Runtime Environment

The Java Card Runtime Environment (JCRE) is comprised of all Java components which run in the smartcard. Figure VI shows the components the JCRE consists of. Compared to Figure I, the JCRE is presented at a lower level for the purpose of illustrating precisely every part, however, this representation can be easily mapped back. On top of the architecture there is an implementation of the Java APIs and all extensions which a manufacturer has decided to add. Through them, applets can request resources and interact with the underlying components. Along with these two parts the installer, described in Section 2.2.1, is deployed. Below that, the implementation of a set of policies and routines which allow, at running time, to manage applets and all transactions as well as any kind of input/output communication. At the bottom, the interpreter (the on-card piece of the virtual machine, refer to Section 2.2.1) that enables the actual instruction execution together with all native methods, which allow it to access and manage the smartcard hardware.

Such a component division is extremely advantageous thanks to its granularity and flexibility as it separates applets from proprietary smartcard technology and provides Java APIs for applets development. The JCRE can be considered, roughly speaking, as the smartcard's operating system; it is responsible for resource management, I/O communication, applet life cycle management and Java security model enforcement [Fort 2006].

The JCRE is initialized at card initialization time, therefore only once during the card lifetime. During this process the virtual machine is initialized and all objects needed for providing the JCRE services and managing applets are created. As a consequence, for a

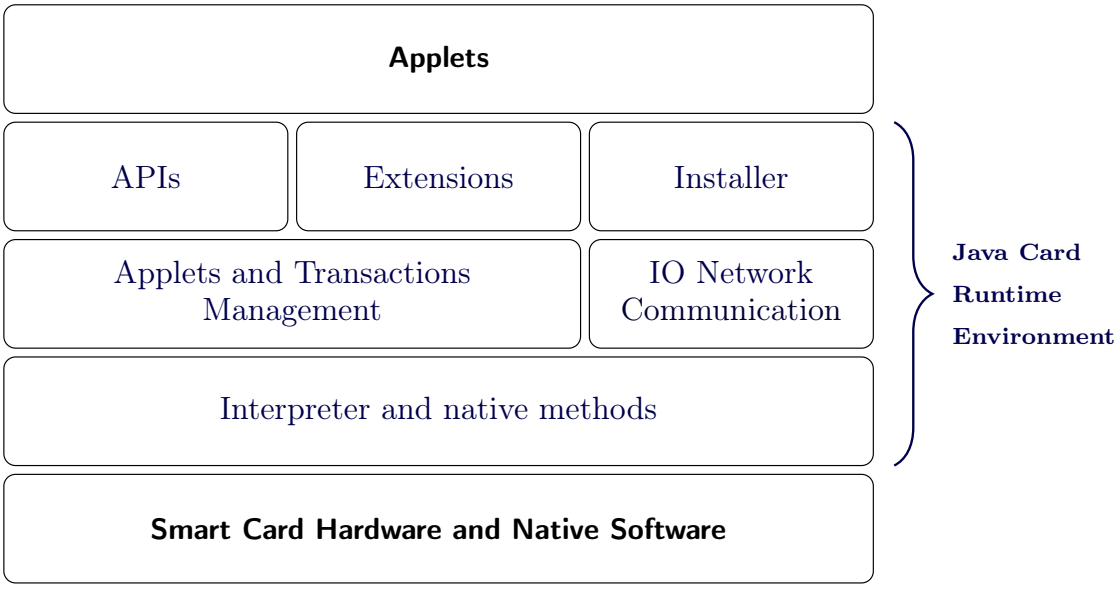

Figure VI: The Java Card Runtime Environment architecture

proper Java card operation, these objects live for all the JCRE's life cycle. As applets are installed, the JCRE takes care of creating applets instances and, in turn, applets create objects for providing services for final consumers.

From the time the card is inserted into a CAD until the time the card is removed, the JCRE operates like any smartcard supporting APDU I/O communication with a host application: the smartcard enters a loop and waits for APDU commands to arrive from the host. When the JCRE receives a command, it selects an applet based on the unique applet identifier (AID) included in the APDU packet and forward it to the applet's *select* method. This method tells the JCRE if the applet is ready to process the request, returning True or False. In accordance to the applet's response the JCRE notifies the host application. In case of a positive reply, the selected applet takes control and passes the received command to its *process* method that interprets the command. Once finished it sends a response back to the host application and gives back control to the JCRE. This process is repeated for each command received. Only when the JCRE is asked by the host application to access another applet the JCRE calls the *deselect* method of the applet currently selected which performs actions on the applet, according to the applet's programmer decisions.

Some data needs to be preserved over JCRE life cycles, therefore persistent memory technology are used to achieve such preservation. In such a way, the life time of the JCRE corresponds to the life time of the smartcard: when the card does not receive power anymore, the virtual machine is only suspended and all objects previously created in persistent memory are preserved. At the next smartcard usage, the virtual machine is reset and executes from the beginning of its life cycle but it loads all applets and objects previously created, therefore differing from the initialization phase [Chen 2000, p.38].

Along with the support of the Java Card language at runtime, the JCRE performs additional tasks and provides further services which can be grouped mainly in three categories [Chen 2000, p.39]:

- *Persistent and transient objects.* As previously described in this section, Java Card objects are by default persistent since they are stored in persistent memory, i.e. EEPROM . But, for security and performance reasons, applets can create objects into the volatile memory, i.e. RAM. Such objects take the name of transient object as they are obviously not persistent across sessions. Note that the number of physical writes are limited and writes to EEPROM cells are typically more than 30 times slower then writes to RAM, in fact a write operation on persistent memory usually takes 3/4ms.
- *Atomic operation and transactions.* The JCRE ensures that any kind of write operation is atomic. Therefore, either the new data is completely and correctly written or the old value is restored. This property is extended to the concept of transactions. In other word, many statements can be part of a single transaction which need to be atomic. During a transaction up to three write operations are typically performed, therefore writing to a variable can consume about 12ms [Rankl 2010, ch.13]. This performance deficit has always to be kept in mind.
- *Applet firewall and sharing mechanism.* The JCRE enforces security at running time. Each applet is isolated from the outside by means of sandboxes and can interact with other applets only respecting defined policies. Security mechanisms are analyzed in detail in section 3.3.

2.2.3 Java Card APIs

The Java Card APIs define only a small set of the traditional Java API for programming smartcard applets and is completely compatible with the ISO7816 model. As already mentioned at the beginning of the chapter, only a dialect of Java is supported due to smartcards' hardware constraints. As a consequence, many common Java classes like *String*, *Integer*, *Boolean*, *Thread*, *System* are not supported. Along with the APIs subset of the Java language, the Java Card environment defines its own set of core classes, specifically developed to support applets development. Hereafter, the collection of all packages defined in the Java Card platform [Fort 2006].

2.2.3.1 Java API subset packages

- **java.lang** provides the fundamental Java language support. It defines *Object* and *Throwable* classes but lacks many features compared to its traditional Java package counterpart. The *Object* class defines a root for the Java Card class hierarchy specifying only a default constructor and the *equals* method while the *Throwable* class provides a common ancestor for all exceptions. The set of defined exceptions consists of

the common *Exception* class, some runtime exceptions and the environment-related *CardException*.

- **java.rmi** defines the *Remote* interface, which identifies interfaces whose methods can be invoked from the card acceptance device (CAD) client applications. Moreover, it defines the *RemoteException* class that can be thrown to indicate an exception occurred during the execution of a remote method call.
- **java.io** defines only one exception class, *IOException*, with the goal of maintaining a hierarchy of exceptions identical to the standard Java programming language. *IOException* is the superclass of the exception *RemoteException* defined in the *java.rmi* package. None of the other traditional *java.io* classes are offered.

2.2.3.2 Javacard-specific packages

- **javacard.framework** provides a set of essential classes and interfaces which define the core functionality of a Java card applet. It defines and implements some relevant concepts such as the Java Card applet, the application protocol data unit (APDU), the personal identification number (PIN), the Java Card system and some more utilities. The *Applet* and *PIN* classes are self-explanatory, the former specifies the base methods and properties of applets (each applet must extend this base class), while the latter defines the most common form of passwords used for user authentication. The *JCSystem* class implements a collection of methods to control and manage applets, resources and transactions. In addition, this class includes methods to handle the persistence and transience of objects.
- **javacard.security** defines interfaces and classes for the Java Card security framework. This API includes many cryptographic algorithms, both symmetric like DES, AES and asymmetric such as DSA, RSA. Moreover, other base classes are defined in order to generate random data, compute message digest and signature: *RandomData*, *Signature*, *MessageDigest*.
- **javacardx.crypto** is an extension of the *javacard.security* package. It introduces two new classes, *KeyEncryption* and *Cipher*. The former defines the methods used to enable encrypted key data access to a key implementation, the latter is the abstract base class for cipher algorithms.
- **javacardx.rmi** is an extension of the *javacard.rmi* package. It introduces *CardRemoteObject* to enable and disable remote access to an object from the outside and *RMIService* which allow to process RMI requests.

2.3 Programming Models

From a programming point of view there are two models that can be chosen when developing a Java card applet [Ortiz 2003]: *the message-passing model* and *the remote method invocation.*

The message-passing model is designed around the APDU protocol which is based on the exchange of a logical data package between CAD and Java Card framework. The Java Card framework receives and forwards to the appropriate applet every APDU sent from the CAD. The applet produces a response, according to the APDU previously received, which is sent back to the CAD. Therefore, developing an applet using such a model is basically a two-step process: defining the command and response APDUs and writing the applet itself.

The Java Card remote method invocation (JCRMI) is conceptually based on the traditional Java RMI distributed model, where a server application makes objects accessible and clients can invokes methods on them remotely. In such an approach the Java Card applet is the server and the host application is the client. JCRMI is provided by the class *RMIService* which belong to the package *javacardx.rmi* previously described. It provides a distributed object model mechanism which runs on top of the APDU-based messaging model, and, through it, the server and clients communicate. Namely, each JCRMI packet is encapsulated within an APDU object and passed to the *RMIService* methods.

Chapter 3

Security-related Aspects

Contents

3.1 Security Concepts 24
3.1.1 CAP Verification 24
3.1.2 Applet Loading 24
3.1.3 Applet Isolation 24
3.1.4 Operation Atomicity 25
3.2 Security Threats 25
3.3 Security Mechanisms 26
3.3.1 The Java Programming Language 26
3.3.2 The JCVM Architecture 27
3.3.3 The JCRE Security Mechanisms 28
3.4 Security Challenges 35
3.4.1 Reviewing Applet Source Code and Bytecode 35
3.4.2 Verification and Signing of the CAP File 35
3.4.3 Securing the Applet Loading Mechanism 36
3.4.4 Reinforcement of the Java Card Platform 36
3.4.5 Stakeholder Awareness and Cooperation 36

This chapter presents the security-related aspects of the Java Card technology. Section 3.1 focuses on the security aspects that play a vital role in the platform. Section 3.2 introduces the types of threats that endanger the Java Card platform. Section 3.3 discusses all mechanisms designed to enforce and enhance security of smartcards. Finally, Section 3.4 presents the current security challenges emphasizing particularly the ones of interest for the report.

3.1 Security Concepts

In the Java Card technology four aspects play a vital role in the platform security:

- CAP verification
- Applet loading
- Applet isolation
- Operation atomicity

3.1.1 CAP Verification

The CAP file, as described in Section 2.2.1, is a compressed representation of the original Java code that contains the essential core needed for an applet execution. This small representation has lost some security features of the underlying Java language security leading to possible attacks. An attacker could easily forge an executable, or tamper an existing one, such that it violates the Java language constraints specified in the JCVM and threatens the security of the entire platform, attempting to compromise the integrity of the JCVM implementation and its applets. That is feasible because a JCVM implementation is not required by the Java Card technology specification to withstand attacks from ill-formed applets, therefore, as a consequence, different JCVM implementations might behave differently under the same attack vector. Note that an applet that has passed the verification step is called an *Verified Applet*, while an applet that has failed the verification process is an *Ill-formed Applet*. Accordingly, a JCVM implementation expects only verified applets. For such a reason, it is crucial to perform CAP verification.

3.1.2 Applet Loading

The security of the applets' loading process is paramount. In case an attacker can bypass the loading mechanism [s]he would be able to upload onto the card any kind of malicious applet, breaking the security of the platform and of the installed applets. An applet installer is already provided by the development kit, but without cryptographic support, therefore it is suitable only for testing or pre-issuance purpose. As it will be mentioned in Section 3.3.2, a manufacturer should implement a proprietary mechanism to handle any kind of post-issuance applet upload.

3.1.3 Applet Isolation

Java Card is a multi application environment, thus it is possible that applets of competing providers co-exist on the same card. Applet providers might have different security requirements depending on the type of applet they want to deploy: a financial/loyalty

applet could be developed under specific and strict security conditions whereas an entertainment applet functions well in a completely unprotected environment. Therefore, it is fundamental to enforce isolation and to allow only controlled interactions.

3.1.4 Operation Atomicity

Java Cards are handy and often operate in an dangerous environment that is outside the control of the card issuer. Unwanted conditions in terms of temperatures, voltage, humidity and vibrations might arise. Also, unexpected card tears could happen. These situations might compromise the consistency and integrity of data in case they arise when data is being manipulated. As a consequence, there would be no evidence that the interrupted operations were completed leading to an inconsistent state. This behaviour cannot be tolerated for a proper functioning of the platform.

3.2 Security Threats

Java Cards are becoming a popular target for attackers, for various reasons [Witteman 2002]:

- Successful attacks enable profitable frauds and attackers could start a business case.
- Java Cards are cheap and easy to obtain. Thus, attackers can test techniques.
- Java Cards are portable. Attackers can easily control conditions in an hostile environment.

A Java Card can be threatened in a variety of ways. Attackers could try to exploit not only logical (software), but also physical (hardware) features due to the nature of smartcards. In addition, also physical phenomenons related to the smartcard's hardware behaviour at runtime might be exploited. Note that in this report physical attacks, i.e. directs attacks on the hardware, are mentioned but they are not addressed, as they are out of scope for our work. An exhaustive attack nomenclature can be found in Chapter 4, while the specification of the attacks that are taken into account in the report is provided in Chapter 5.

At runtime, Java Card applets could manifest unwanted behaviours, which can be listed as follows in increasing order of severity [Witteman 2003]:

1. Perform harmless but annoying behaviour
2. Crash temporary or permanently the Java platform
3. Manipulate unauthorized resources external to their domain
4. Leak sensitive user data
5. Attack the platform or other installed applets

As a consequence of these undesirable behaviours four basic threats types can be identified:

1. **A verified applet leaks sensitive data**. Applets, because of developer oversights, could leak sensitive data and/or expose themselves to data tampering. As a consequence, data confidentiality of the other installed applets might be also endangered.
2. **A verified applet abuses features**. Applets abuse regular Java Card features to perform undesirable or harmful behaviours towards the platform and/or other installed applets.
3. **A verified applet exploits bugs or design oversights**. Applets identify and exploit bugs or design oversights in the platform to provoke damages to the platform and/or to the other installed applets.
4. **An ill-formed applet compromises the entire platform**. Ill-formed applets could succeed in being loaded into the card and compromise the integrity of the platform and of the other installed applets.

Three exhaustive use cases for threat 2, 3 and 4 are presented in [Witteman 2003].

3.3 Security Mechanisms

The Java Card technology provides a number of security features which are essentially derived from [Microsystems 2003]:

- The Java programming language along with specific packages
- The JCVM architecture
- The JCRE security mechanisms

It is important to realize, referring to the above list of unwanted behaviours, that only item 5, i.e. *"Attack the platform or other installed applets"* and item 4, i.e. *"Leak sensitive user data"*, are partially mitigated by these built-in security mechanisms. The above list of security features is now introduced and explained.

3.3.1 The Java Programming Language

The Java programming language integrates many features that protect the integrity of the platform. These mechanisms can be outlined as follow:

- Encapsulation
- Type checking
- No pointer manipulation
- Array bounds check

- Variables initialization
- Security by APIs

Due to its programming paradigm, Java provides programmers with the means to encapsulate data, restricting access to objects' components. This restriction can be described as a barrier that does not allow code and data being accessed by other code and is achieved using the well-known keywords *public*, *no modifier*, *protected* and *private*. Table II shows the provided access levels.

Modifier\Scope	Class	Package	Subclass	World
Public	yes	yes	yes	yes
Protected	yes	yes	yes	no
No modifier	yes	yes	no	no
Private	yes	no	no	no

Table II: The access level permitted by each identifier in Java [Rose 2001].

The Java language is a strongly typed language, therefore it generates an error refusing to compile source code if the argument passed to a function does not match the expected type. As a consequence, also the bytecode is type correct. Further security risks are prevented by avoiding pointer manipulation, variables usage before initialization and enforcing array boundaries. Moreover, the Java API, as described in Section 2.2.3, introduces specific security and crypto packages which allow to provide a secure mechanism for authenticating and downloading applets and removes potentially risky features such as threading and dynamic class loading.

3.3.2 The JCVM Architecture

The Virtual Machine architecture, as described in Section 2.2, is divided into two parts. Such a separation between the on-card part and off-card part provides more security as the off-card part asserts that a CAP file, before being loaded onto the smartcard, conforms to the Java Card specification. That provides a further assurance that the executable code will not compromise the integrity of the virtual machine. However, it might happen that the executable code is modified prior to installation on the card, as the code check is performed by the off-card part of the virtual machine whereas the actual installation is done by the installer that resides on-card (refer to Section 2.2). To avoid that, *card manufacturers* can easily implement a code loading mechanism that enforces the integrity and authenticity of the applet through Public Key or Symmetric Key cryptography. Keys for loading code are then used by the card manufacturer that sometimes plays also the role

of card issuer or directly by the card issuer; this depends on the use case. This feature can be nested without difficulty thanks to the granularity provided by the separation between installer and interpreter: a *card manufacturer* only needs to add a feature to the installer.

3.3.3 The JCRE Security Mechanisms

The JCRE enforces security through essentially two features which are already introduced in Section 2.2.2: *transaction atomicity* and the *applet firewall*. Furthermore, the JCRE might optionally perform runtime checks that are redundant with the static checks performed by the converter in order to ensure that the code does not violate the fundamental language description, for example trying to access a private variable from outside a class.

3.3.3.1 Transaction Atomicity

A transaction is a logical set of persistent data updates performed atomically. The purpose of this mechanism is to protect the integrity of data against sudden events such as power loss (overall due to card tears) and program errors which could cause data corruption. As illustrated in Listing 3.1, an applet states the beginning of an atomic set of updates with a call to *JCSystem.beginTransaction* and only when the method *JCSystem.commitTransaction* is reached all conditional updates are committed to the persistent memory. In case the method *JCSystem.abortTransaction* is called instead, the transaction is aborted and all updates within the initiated transaction are rolled back.

```
1 ...
2 private short balance;
3 ...
4 JCSystem.beginTransaction();
5 balance = (short)(balance + creditAmount1);
6 balance = (short)(balance - creditAmount2);
7 JCSystem.commitTransaction();
8 ...
```

Listing 3.1: Java Card transaction example

However, a power loss might occur before *commitTransaction* or *abortTransaction* are reached by the interpreter. As soon as the card receives power again, the updates performed before the power loss are rolled back automatically by the JCRE. To enforce such a behaviour, the system has to keep track of the data state before the transaction, logging it at the granularity of a single persistent memory access, i.e. logging the content of each persistent memory slot. Typically, large transaction systems manipulate data in RAM during the transaction and log the updates accordingly to the persistent memory. However, the necessary RAM resources for providing this transaction design are not available on current smartcard hardware. The recovery process is performed comparing the

data state before the transaction and after the unexpected event. Note also that the system throws an exception in case of any irregularity, such as buffer overflow, during the transaction. In case the thrown type of the exception is not handled by the applet, the transaction is aborted.

3.3.3.2 Applet Firewall

The Java Card platform is a multi-application environment, therefore, applets need to protect sensitive data against malicious access. In traditional Java, this goal is achieved by means of *class loaders* and *security managers*, for further details refer to [Oaks 2001]. Whereas, in Java Card, this is done via the *applet firewall* which allows to create private namespaces for applets. The firewall not only provides a first level of security against malware, but also against developer mistakes and design oversights. This security mechanism is based on essentially two concepts: **package** and **context** notions. Private namespaces for applets take the name of contexts and are enforced by the common concept of a package. Two applets are said to belong to the same context if they are defined within the same package, this information is contained in the unique applet identifier (AID) which is assigned to an applet when created. That is, all applet instances belonging to the same package share the same context and can freely access objects of other applet instances which reside in the same package. Applets belonging to different contexts generally cannot access each other's objects. In addition to applets' contexts a further "system" context is defined: the *JCRE context*. Its peculiarity is that objects belonging to this context can access any object from any other context on the card. Figure VII illustrates this context-based mechanism. According to [Éluard 2001], the set of *Contexts* can be defined as follow:

$$Contexts = \{JCRE\} \cup \{pckg \mid pckg \text{ is a legal package name on card}\}$$

Every object, once created, has an owner and a context. The owner is the applet that created the object, while the context is the context to which the applet that created the object belongs to.

At any point in time, only one context is active within the VM, such a context is called the *currently active context* [Microsystems 2006c]. This can be either the JCRE context or an applet context. When bytecodes try to access an object's method of another applet a *runtime* check against the currently active context is performed in order to determine if that access can be granted or not. In case of failure, a *java.lang.SecurityException* is thrown. Otherwise, the VM has to determine if a *context-switch* is required. If the currently active context is equal to the context of the object owner no context switch is required, in case it is not, under specific conditions, defined in Section 6.2.8 of [Microsystems 2006c], a context switch happens. The previous context and object owner information is pushed in the VM stack and the result of this switch is a new currently active context. Upon exit from the current method the previous information stored in the stack is restored popping it out from the stack. Therefore the currently active context

becomes again the context of the applet, which caused the context switch executing a method of another applet's object. Note that, if a context switch occurs during a transaction it does not interfere with the execution of the transaction anyhow, i.e. updates to persistent data continue to be conditional in the new context until the transaction is committed or aborted. Hereafter an example is presented to clarify the main points.

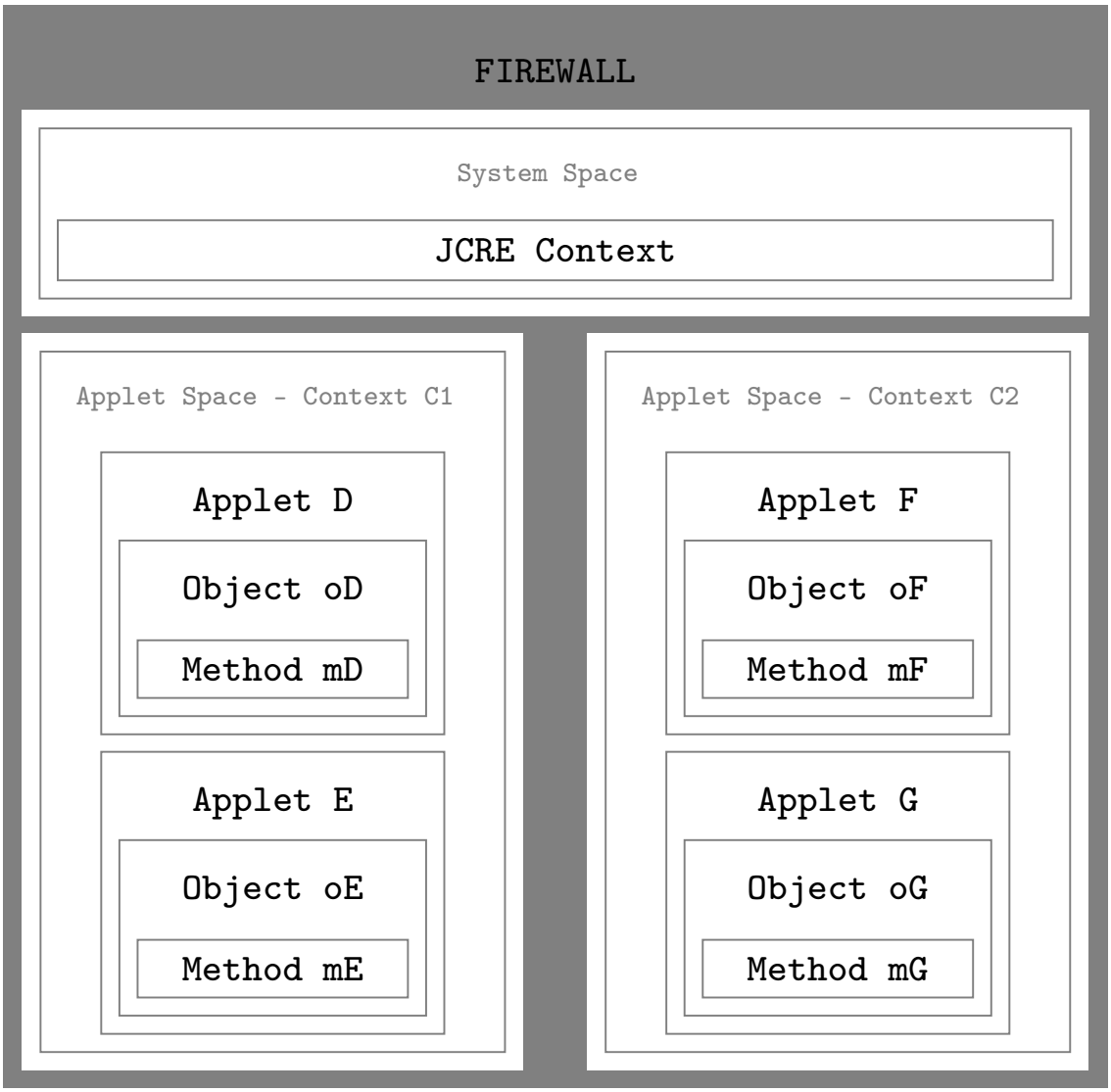

Figure VII: Example of contexts within the Java Card platform.

Example* 3.1*: Referring to Figure VII, applets D and E are in the same package, thus share the same context, whereas applet F is defined in an other package and therefore belongs to another context. In case context *C1* is the currently active context and method *mD* on object *oD* is executed by applet *D* no context switch occurs. In case method *mD* invokes, in turn, method *mE* on object *oE* owned by applet *E* no context switch occurs

as the two applet instances are defined in the same context. However, if *mD* calls method *mF* on object *oF* which is owned by applet *F* a firewall restriction applies and the VM has to determine if the access can be granted. In case it does, the VM has to decide if a context switch is required, based on specific conditions mentioned before. If the context switch is required, the currently active context becomes the context *C2*. Upon return to method *mD* from method *mF* the context *C1* is restored and becomes again the currently active context.

To sum up, we have the following rules that define the context concept and object ownership [Microsystems 2006c]:

- Every applet is assigned an AID and belongs to a context. Applets instances which are defined within the same package belong to the same context.
- Every object is owned by an applet instance and therefore it is assigned to its context. When executing a method of an object in an applet's instance, the object's owner context is the currently active context. Note that, in case of a static method call, the execution is in the caller context as only class instances, i.e. objects, have a context.

The firewall isolates each applet to its designated context for security reason, however, applets often need to communicate with other contexts in order to work properly. The following mechanisms enable an object to interact with an object belonging to a different context [Microsystems 2006c]:

- **JCRE Entry Points Objects**
- **Sharable Objects**

Java Entry Points Objects are system objects, owned by the JCRE, which allow non-privileged applet instances to request system services. These objects have been designed to be accessed by any object from any context. When a method of those objects is invoked by an applet instance a context switch occurs to the JCRE context. The service is performed after verifying that all parameters are within bounds and all object passed in as parameters can be accessed by the applet caller's context. The JCRE defines two types of entry points objects: *temporary* and *permanent*. The need of specifying two types of entry points come from the requirement that some references of those objects cannot be allowed to be stored in class variables or array components. That is the fundamental difference: temporary entry point object references cannot be stored whereas permanent entry point references can. An typical example of temporary entry point is the APDU object that contains the buffer in which commands sent to the card are stored. This object has to be considered as temporary in order to prevent unauthorized reuse. An example of a persistent entry point object is the table containing the AIDs of all installed applets on the card.

The *Sharable Objects* mechanism allows applets in different context to share information. An applet can share object's methods, but not fields, through an interface which extends *javacard.framework.Sharable*. In this interface the methods' signatures that the applet

wants to share has to be specified. The class of the object to be shared has to implement this defined interface. Moreover, the applet has to include the method *getSharableInterfaceObject* which is called every time the applet is asked to provide a shared object. This method takes as input parameter the AID of the applet requesting to access the shared object allowing, in this way, different objects to be shared with different applets. More precisely, this mechanism works in essentially three main steps [Microsystems 2006c]:

1. **Building a sharable interface object**

   - An applet *D* defines a shareable interface *SI* extending *javacard.framework.Sharable* and specifying the signature of the methods to share.

   - *D* defines a class *C* that implement the defined *SI*. *C* might also define methods that are not defined in *SI*, in that case this methods will not be sharable because they are protected by the firewall.

   - *D* instantiates an object *O* of class *C*. *O* belongs to the applet *D* and is part of its context. All objects inside that context can access any field or method of *O* in according to the class specification, for example, a private field cannot be accessed.

2. **Obtaining the sharable interface object**

   - An applet *E*, which belongs to another context, wants to access the shared methods of *D* therefore an object reference *SIO* of type *SI* is created.

   - *E* calls the method *JCSystem.getAppletSharableInterfaceObject* to request a shared interface object reference from *D*.

   - *D* receives the request along with the AID of *E* and through the method *Applet.getShareableInterfaceObject* determines if granting the access to the shared object *O*.

   - In case the access to object *O* is granted to *E* by *D*, a reference to object *O* is returned to *E*. Such a reference is of type *Shareable*, therefore none of the fields and methods of *O* are visible. In case the access to *O* is denied a *null* reference is returned.

   - *E* casts the received reference of object *O* from *Shareable* to *SI* and stores it in the variable *SIO* previously created. After the cast, only the shared methods of *O* are visible to *E*. The firewall prevents any access to fields or methods of *O* that are not shared.

3. **Requesting services through the sharable interface object**

   - *E* requests a service from applet *D* through the sharable interface object reference *SIO*. When a shared method invocation occurs a context switch happens. The currently active context, i.e. the context of *E*, is saved onto the stack and the context of the owner of *O*, i.e. *D*, becomes the currently active context.

- *D* determines the AID of the method's caller through the method *JCSystem.getPreviousContextAID*, if it does not corresponds to the expected AID the method is not performed, otherwise it is.
- As a context switch occurred, the firewall allows the called method to access all fields or methods of object *O* and any other object defined in the context of *D*. At the same time, the firewall denies access to not-shared objects in the context of *E*.
- The method, using the parameters passed in, executes and returns a result to *E*.
- A context switch occurs restoring the previous active context stored in the stack, therefore, the context of *E* becomes again the currently active context.

In accordance to [Bernardeschi 2004] and [Éluard 2001], the above-presented JCRE security mechanisms protect applet's data from unauthorized access only partially. A mechanism for knowing the context of a method caller is implemented, but there is no way to obtain the identity of all callers involved. In other words, an applet can only know the last context switch from its point of view in the stack, by calling the method *getPreviousContextAID*. As a consequence, an applet *D* which provides a service to an applet *E* does not know whether the method call was really done by *E* or was a result of *E* being called from some other applets. Neither can *D* know what the caller will do with the method result. Therefore, an applet can only signify an object as shared, without having the opportunity to truly decide with whom to share an object. Hereafter two examples are presented showing the most relevant implications of this issue: direct and passive data/object leakage.

Example* 3.2*: There are three applets in three different contexts: *D*, *E*, *F*. A shareable interface *SI* is prepared and is implemented by *D*[1]. *D* defines the *getShareableInterfaceObject* to determine upon a sharing request if granting it or not, depending on the caller. As illustrated in Listing 3.2 (Object grant), the object is shared only if the caller is *E*. *D* is now ready to share an object *SIO*. Using the method *JCSystem.getAppletShareableInterfaceObject*, *E* requests the object *SIO* from *D*. However, as shown in Listing 3.2 (Data exposure), *E* inadvertently leaks the reference of *SIO* once received by storing the reference into a *public static field*[2]. At this point, as illustrated in Listing 3.2 (exploitation), *F* can simply use all shared methods of *SIO* although they were meant to be shared only with *E*. This leads to direct data leakage. However, this leakage can still be avoided verifying each time, at applet side, the method caller as shown in Listing 3.2 (Mitigation). Thus, given the currently deployed security mechanisms, this problem can be solved only through an applet implementation that takes into account such a platform flaw.

[1] The shareable interface is directly implemented by the applet main class for keeping the example more compact.

[2] This is just one of the possible ways which could lead to data exposure.

Example* 3.3*: There are three applets in three different contexts: *D*, *E*, *F*. *D* offers a subscription service by means of of a shareable interface object. *E* subscribed to this service offered by *D* whereas *F* did not. Assume that every time *E* uses the service, on its turn, invokes a shared method of *F*. This method invocation can be employed by *F* to infer that a subscription service is currently active and when it is actually used. Thus, sensitive information is leaked. The firewall is not able to detect this unexpected information flow as the services are shared properly. A typical example of this issue is presented in [Bieber 2000]. This lead to an indirect data leakage and cannot be avoided with the security mechanism currently deployed.

```
1  // Object grant
2  public class D extends Applet implements SI {
3      ...
4      public Shareable getShareableInterfaceObject (AID client, byte param){
5          if (client.equals (E_AID, (short)0, (byte)E_AID.length) == false)
6              return null;
7          return (this);
8      }
9      ...
10 }
11 // Data exposure
12 public class E extends Applet {
13     ...
14     public static SI D_Object;
15     D_Object = (SI)JCSystem. getAppletShareableInterfaceObject(D_AID,(byte)0);
16     ...
17 }
18 // Exploitation
19 public class F extends Applet {
20     ...
21     private static SI D_Object;
22     D_Object = E.D_Object;
23     D_Object.foo(); // This method exists in SI
24     ...
25 }
26 // Mitigation
27 public class D extends Applet implements SI {
28     ...
29     public void foo() {
30         AID client = JCSystem.getPreviousContextAID (); // Is the caller E?
31         if (client.equals (E_AID, (short)0, (byte)E_AID.length) == false)
32                 ISOException.ThrowIt (SW UNAUTHORIZED CLIENT);
33         ... // Fine, the caller is E
34     }
35     ...
36 }
```

Listing 3.2: Direct data leakage

3.4 Security Challenges

Java Card technology represents a significant step forward in the world of the smartcards, but it is still young and presents weaknesses that need to be tackled. The security challenges are now investigated in order to outline the main areas where to focus on with the purpose of counteracting the four security threat types presented in Section 3.2. They can be outlined as follow:

- Reviewing applet sources code and bytecode
- Verification and signing of the CAP file
- Securing the applet loading mechanism
- Reinforcement of the Java Card platform
- Enforcing stakeholder awareness and cooperation

Hereafter, each security challenge is introduced.

3.4.1 Reviewing Applet Source Code and Bytecode

Ideally, verified applets should not be harmful neither for the platform nor for the other installed applets, but this is not the case. In reality, applets, even if verified, could [Witteman 2003]:

- crash the platform or deny services to other installed applets
- act legally but exhibit an undesirable behaviour
- try to exploit bugs in the platform
- be badly designed, leaking data and, as a consequence, damage its related business

In order to avoid these behaviours, reviews of the source code and/or bytecode should always be performed. Only in situations where security is not an issue for all installed applets and the application provider takes full liability over any kind of consequence, it could be acceptable to not perform a review.

3.4.2 Verification and Signing of the CAP File

As already mentioned in Section 3.1, CAP verification is essential to avoid that ill-formed applets compromise the integrity of the platform. Card issuers should support verification with strict administrative policies and expert professionals should be able to interpret any output of the verification algorithm and take consequentially effective choices. Applying cryptographic signatures to the CAP files along with its all related files, i.e. export files, is also fundamental to enforce the integrity of any code files when internally/externally exchanged between involved parties prior to installation on the card.

3.4.3 Securing the Applet Loading Mechanism

The loading mechanism is of primary importance and has to be continuously updated to counteract new attack vectors. As explained in Section 3.1, in case it was bypassed the entire platform along with the installed applets would be compromised by the attacker. Note that, a Java card should also protect itself against unauthorized downloads.

3.4.4 Reinforcement of the Java Card Platform

The Java Card platform is still quite young and presents weaknesses. It has to be tested and patched regularly in order to satisfy all security requirements that are needed for an effective platform operability. A lack of trust in the platform by the stakeholders would mean the collapse of the entire Java Card technology.

3.4.5 Stakeholder Awareness and Cooperation

In the Java Card environment three stakeholders play a major role:

- Applet developers
- Java Card manufacturers
- Card issuer

Security can be effectively enforced only if the involved stakeholders are conscious of the importance of security in this field and are willing to cooperate.
Applet developers need to focus on the security requirements their applets need in order to make their products not only functional but also secure. They need to realise that the security of the platform also depends on the security of their products.
Java Card manufactures need to understand that the security of the platform directly depends on the hardware used and the software deployed.
Card issuers need to become aware of the weaknesses the platform still presents and should apply any possible security countermeasure in accordance with the risk levels.

Referring to the list of threats presented in Section 3.2, the first, second, third items can be addressed, even if not completely, by means of source code and/or bytecode review. Moreover, awareness and cooperation of stakeholders can far improve the situation as much more effort would be put in testing applets for susceptibility to exploits. The fourth item can be mitigated by securing the applet loading mechanism and enforcing the CAP executable and related files verification and signing.

The report aims at contributing to the solutions of two of the above-described security challenges, namely:

- *Reviewing applet source code and bytecode*

- *Verification and signing of the CAP file*

The end-result is a software tool to analyze applets' code for the presence of potentially exploitable and malicious code. Our motivations along with our proposal are introduced and described in Chapter 6.

Chapter 4

Attack Nomenclature

Contents

4.1 Logical Attacks 41
4.1.1 Vulnerabilities 41
4.1.2 Countermeasures 43
4.2 Physical Attacks 44
4.2.1 Vulnerabilities 44
4.2.2 Countermeasures 45
4.3 Side Channel Attacks 46
4.3.1 Side Channel Analysis Vulnerabilities 46
4.3.2 Side Channel Analysis Countermeasures 47
4.3.3 Side Channel Manipulation Vulnerabilities 48
4.3.4 Side Channel Manipulation Countermeasures 48

This chapter presents an exhaustive attack nomenclature that covers all feasible classes of attacks on Java Cards, namely *logical attacks* in Section 4.1, *physical attacks* in Section 4.2 and *side channel attacks* in Section 4.3. This nomenclature has been defined taking as starting point [Witteman 2002]. Each class introduces, at a high level, feasible attack types grouped by vulnerabilities along with the common countermeasures which should be taken to counteract those attacks. Figure VIII shows graphically this division. The purpose of this chapter is providing a big picture about the variety of vulnerabilities an attacker might exploit to reach his/her goal. A technical specification of the most relevant attacks techniques is provided in Chapter 5. Note that, physical attacks, as previously mentioned, are only introduced at high-level for the sake of completeness.

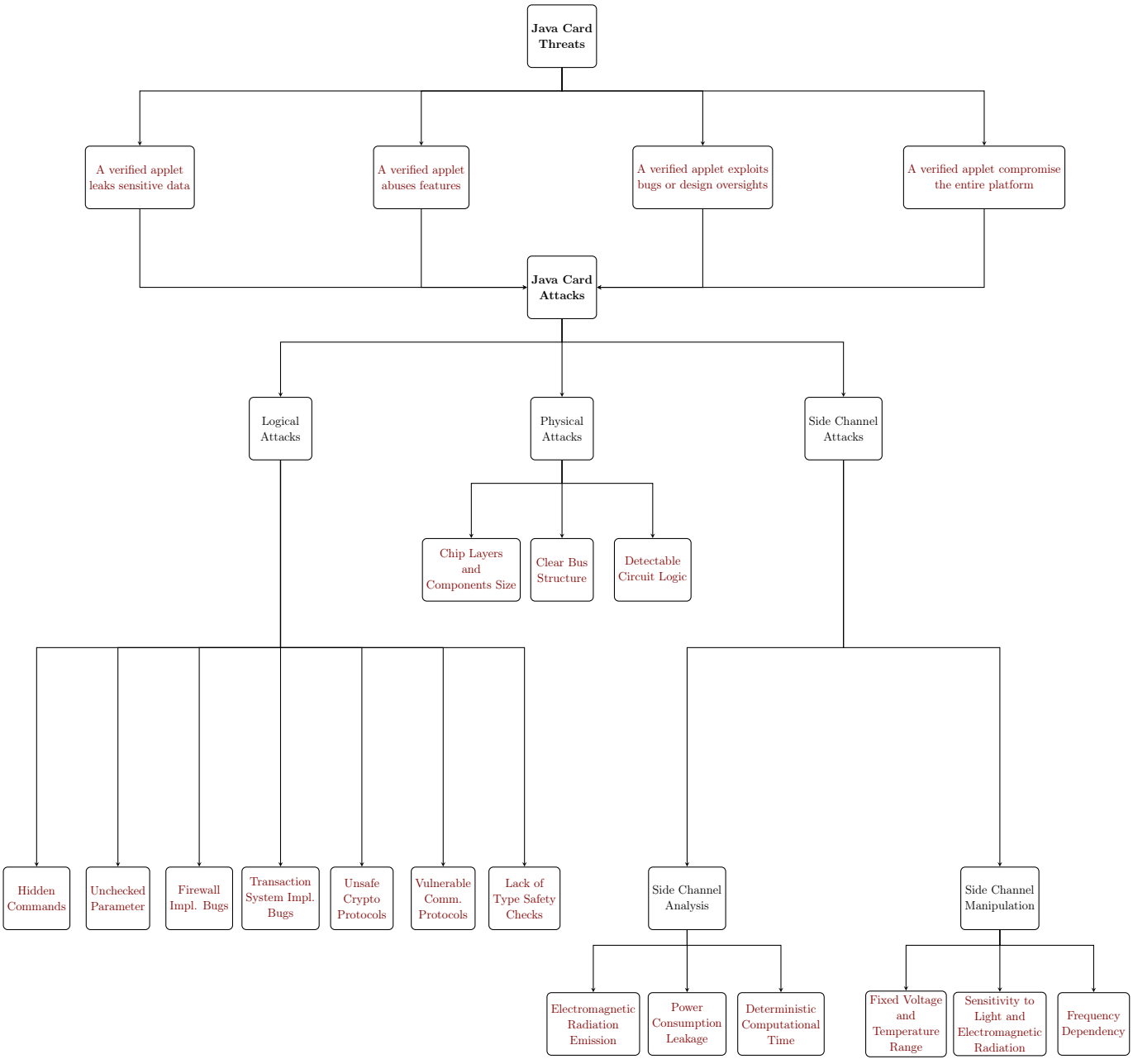

Figure VIII: The Java Card attack nomenclature

4.1 Logical Attacks

Logical attacks exploit bugs or oversights in the software implementation and abuse these flaws to gain access to confidential data, perform undesired data modification or deny services.

Logical attacks can be grouped by vulnerability as follow:

- Hidden commands
- Unchecked parameters
- Firewall implementation bugs
- Transaction system implementation bugs
- Vulnerable communication protocols
- Unsafe crypto protocols
- Lack of type safety checks

Consequences that those attacks could lead to are unpredictable because that is strongly dependent on the attack vector. For these reasons, those classes of attacks can be mapped to all the presented threats in Chapter 3.2. In other words, each attack, independently from the class it belongs to, could leak data, abuse regular platform features and/or exploit platform or applets bugs. Each category of attack is now explained.

4.1.1 Vulnerabilities

4.1.1.1 Hidden commands

The JCRE has a big set of APDU commands that allow to perform a variety of actions. A complete list can be found in [ISO 2013a]. Moreover, also applets define their own set of commands, leading to a huge collection of directives. In practical usage, only a small set of commands are required and it happens that commands that were supposed to be disabled are still active from an initialization phase or previous application. Attack vectors might abuse them to perform malicious activities.

4.1.1.2 Unchecked parameters

Commands often require one or more parameters to perform their functions. The use of disallowed or unexpected parameter values or types could be misinterpreted if not sanitized, leading to surprising results that could negatively impact not only applets, but also the entire Java Card platform. Therefore, attack vectors could inject prohibited parameters causing relevant damages.

4.1.1.3 Firewall implementation bugs

The applet firewall is quite complicated as specified in [Microsystems 2006c], this means that there is real chance of implementation bugs or firewall oversights. The applet firewall has been tested thoroughly in [Mostowski 2007] using verified code, i.e. code that complies to the Java Card specifications, and some security problems have been discovered when using sharable interfaces. A few of these issues have been already introduced in Section 3.3. In particular, an attack vector might use a shared object of the target applet to gain direct or indirect access to sensitive data.

4.1.1.4 Transaction system implementation bug

The transaction mechanism, defined in [Microsystems 2006c], is the most studied aspect of the platform due to its tricky nature. Several papers carried out investigations such as [Beckert 2003] and [Marché 2006]. As described in Section 3.3, the transaction system allows multiple statements to be atomic and, in case of failure, offers a rollback mechanism. Attacks vectors could exploit directly a bad implementation of the transaction mechanisms to gain access to or tamper with sensitive data.

4.1.1.5 Vulnerable communication protocols

The APDU protocol, defined in [ISO 2013a], handles the data flow and the error recovery between a Java card and a terminal. Sending messages outside the scope of the current protocol state could lead to unexpected events such as data leakage. Therefore, an attack vector could directly exploit the communication protocol to disclose sensitive data.

4.1.1.6 Unsafe crypto-protocols

Crypto-protocols manage transactions where cryptographic operations are required. This kind of protocols are not strongly related to the Java Card platform but to the environment applet-terminal, that means their implementation and sometimes their design depend on the card issuers. In case of design or implementation oversights, sensitive applet data could be leaked or backdoors might be inadvertently opened leading to a possible exploitation of the other installed security sensitive applets. An attack vector, such as replay-attacks, might take advantages from this negligence.

4.1.1.7 Lack of type safety checks

Java, as described in Section 3.3, is a well-typed programming language. That means, type safety is the cornerstone of its security, for further details on type safety and Java security, we refer to [McGraw 1999]. As a consequence of the hardware constraints, safety type checks are only performed at compile time, as opposed to the traditional Java platform

where they are also performed at running time. This implies that attacks vectors could exploit this lack of checks at running time to produce type confusion leading to breach the whole Java Card platform. Further explanation about type confusion can be found in [Saraswat 1997].

4.1.2 Countermeasures

The susceptibility to logical attacks is truly dependent on the complexity of the software. By definition of software engineering, the probability of bugs, oversights and flaws grows with the size of the software. Common approaches to cope with these complications are [Witteman 2002]:

- Develop a structured software design in such a way that functional blocks can be much more easily reviewed, tested and reused.
- Perform experimental tests with the purpose to validate the code's behaviour. Static or dynamic analysis might be carried out.
- Standardize interfaces and applications in order to encourage the reuse of proven software, decreasing consequentially the chances of vulnerabilities.
- Test the application externally in order to achieve a formal certificate that claims the effectiveness of the requirements.

4.2 Physical Attacks

Physical attacks exploit the raw hardware of Java cards, performing reverse engineering. This class of attacks requires an high-end laboratory equipment, but does provide significant opportunities to succeed in the exploitation. However, physical attacks are invasive and often destructive as an attacker can often not reuse the card.

Physical attacks can be grouped by vulnerability as follow:

- Chip layers and components sizes
- Clear bus structure
- Detectable circuit logic

4.2.1 Vulnerabilities

4.2.1.1 Chip Layers and components size

Modern chips contain multiple layers that can be delayered by means of etching materials and chemicals. This reveals the various building blocks in a chip making it accessible for optical and electrical analysis. The use of advanced microscopes enables optical analysis and reverse engineering permitting to reconstruct the circuit logic or the operating system logic.

4.2.1.2 Clear bus Structure

Once the chip is delayered, a probe needle can be inserted on arbitrary wires modifying the existing logic structure and even creating new channels to the outside. However, the bus structure of the chip has to be visible and clear to allow probe needles to tap all data exchanges between the CPU and the memory.

4.2.1.3 Detectable Circuit Logic

The use of scanning electron microscopes enables electrical analysis making it possible to surface wires hidden from other layers. This type of microscopes shoots ions instead of electrons not only making small details visible, but also allowing to modify an easy-to-detect chip circuitry logic.

Attack vectors can exploit the physical vulnerabilities above-described to access all data stored in the card.

4.2.2 Countermeasures

The effectiveness of attacks that exploit physical features strongly depends on the quality, in term of physical security, of the chip. Below some approaches that permit to decrease the level of success in physically exploiting a smartcard [Witteman 2002]. These techniques might be compared in spirit to software obfuscation.

- Reduce the components size to thwart the use of optical microscopes and probe needles. Note that, electron microscopes can handle sizes in the order of a few nanometer.
- Use a protecting layer to prevent the analysis of live data processing. That is possible as a protecting layer contains an active grid carrying a protecting signal.
- Use sensors to detect intolerant conditions in term of light, temperature, power supply. As a result of a detection the chip should be disabled.
- Scramble communication buses so that an attacker has to do a full reverse engineering of the logic.
- Make use of "glue" logic, i.e. mixing logic blocks together, so that an attacker is not able to identify easily the functional building blocks by analysing the chip structure.

4.3 Side Channel Attacks

Side Channel attacks exploit physical phenomenons related to the smartcard hardware, such as time, temperature, power and radiation to gather sensitive information or even manipulate the hardware behaviour of the card. Typical targets are crypto-protocols. This class of attacks can be, in turn, subdivided into *Side Channel analysis attack* and *Side Channel manipulation attack*. The former only examines the collected data to gather useful information, while the latter exploits physical phenomenon to change the card behaviour.

4.3.1 Side Channel Analysis Vulnerabilities

Side Channel analysis attacks can be grouped by vulnerability as follow:

- Power consumption leakage
- Electromagnetic radiation emission
- Deterministic computational time

4.3.1.1 Power consumption leakage

Smartcard semiconductors consume power according to the ongoing processes. Attack vectors can measure the power consumption revealing details about the information being processed. The most known and used attack vector in this category is *Differential Power Attack* (DPA). An exhaustive overview of the attack can be found in [Kocher 1999].

4.3.1.2 Electromagnetic radiation emission

Every time a smartcard transistor switches, it produces electromagnetic radiation. Attack vectors intercepting the phenomena effects can determine a complete picture of the ongoing process, leading to the leakage of sensitive information.

4.3.1.3 Deterministic computational time

Smartcard microprocessors take time to execute logical operations. Due to the deterministic nature of the smartcard microprocessors today used, precise measurements of the time for each operation can leak sensitive information such as the seed used for computing a key in a crypto system.

4.3.2 Side Channel Analysis Countermeasures

Side Channels analysis attacks try to exploit environment conditions related to the smartcard hardware to gather data. A variety of countermeasures can be applied. They can be categorized into two types depending on the logical place where the security mechanism is deployed:

- Hardware countermeasures
- Software countermeasures

The first category can only decrease the susceptibility to Side Channel analysis attacks, but not eliminate it completely. However, it is independent of the software implementation so that, even if the software is updated or completely changed, the security mechanisms are still in place. The second category can either reduce the sensitivity to this type of attacks deploying countermeasures at platform level, or implement at application-level tailored mechanisms that can eliminate the emission of useful information. Especially the application-level countermeasures can be expensive and hard to maintain due to the strict connection with the applet design.

The most typical *Hardware countermeasures* are:

- Decoupling from power source, e.g. by putting a capacitor in the IC
- Reduce the power signal and the electromagnetic emissions by means of metal shields.
- Increase the amplitude noise level deploying concurrent random processes.
- Increase the timing noise level with variable clock speeds and interrupts.

The most common *Software countermeasures* are:

- Reduce relevant signal by random process ordering.
- Add timing noises by means of random delays or alternating path.

Due to the nature of this type of attacks, in order to gain useful information from physical phenomenons an attacker needs to collect many phenomenons-related samples and know the input and/or output of the algorithm [s]he wants to exploit. Therefore, two further software countermeasures can be deployed to completely eliminate the emission of useful and enough information for the attacker:

- Make use of retry-counters that limit the number of samples an attacker can take.
- Limit the visibility of the input or output of cryptographic algorithms.
- Change or update the secret keys regularly

4.3.3 Side Channel Manipulation Vulnerabilities

Side Channel manipulation attacks can be grouped by vulnerability as follow:

- Fixed voltage and temperature range
- Sensitivity to light, X-rays and other electromagnetic radiation
- Frequency dependency

4.3.3.1 Fixed voltage and temperature range

Smartcard electronic circuits are designed to operate at a defined and constant voltage supply and at a fixed temperature range. Sudden changes in temperature or voltage outside the boundaries might change the behaviour of the chip and trigger alternative actions. The most known and used attack vector in the Side Channel manipulation class exploits the card voltage and is called *Fault Injection*. Futher details can be found in [Hsueh 1997].

4.3.3.2 Sensitivity to light and electromagnetic radiation

The circuitry of the smartcard is sensitive to light and strong electromagnetic pulses. An attack vector can make use of a direct beam of (laser) light or an other intense electromagnetic signal to damage the chip, modify data or instruction(s) in memory/on the bus or change its logical behaviour.

4.3.3.3 Frequency dependency

Similarly to voltage and temperature, semiconductors are also designed to operate at a well-defined clock frequency range. In case the frequency is forced to exceed the defined top boundary, the Java Card behaviour might be affected triggering errors in the execution of complex instructions that need more time. An attack vector could force the increase of the frequency in order to trigger errors in crucial applets' checks leading to for example an erroneously successful security check.

4.3.4 Side Channel Manipulation Countermeasures

Side Channels manipulation attacks try to exploit environment conditions related to the smartcard hardware to change the smartcard behaviour triggering software errors. A variety of countermeasures can be applied. They can be categorized into two types depending on the logical place where the security mechanism is deployed:

- Hardware countermeasures
- Software countermeasures

The only possible *Hardware countermeasure* that can be deployed in order to avoid Side Channel Manipulation attacks is a strict use of sensors for voltage, temperature and frequency. However, electronic circuits will be never completely immune to signal injections [Witteman 2002]. Furthermore, note that a too strict use of sensors affects the reliability of the card, causing possible malfunctioning in unexpected climate conditions. For this reason, this vulnerability should be mitigated at software level as an adequate complement.

With respect to *Software countermeasures*, it is important to carry out fault detection enforcing values validation of crucial program flow decision and cryptographic data. This enforcement can be achieved in several ways, the most common techniques are:

- Introduce value redundancy over the program flow and check fault injection comparing the redundant values.
- Reverse the input from the output and check if the obtained and reversed values are the same.

We refer to [Bouffard 2014, Gadellaa 2005] for a complete list of possible hardware and software countermeasures against fault injections.

Chapter 5

Attack Vectors

Contents

5.1 Power Analysis and Manipulation 52
5.1.1 Differential Power Analysis 52
5.1.2 Fault Injection 52
5.2 Applet Exploitation 54
5.3 Type Confusion 54
5.3.1 Obtaining rights to load code 55
5.3.2 Injection of ill-formed code 55
5.3.3 Running a developed attack vector 63

The previous chapter presented a complete set of vulnerabilities that could be exploited for attacking a Java card. As a further step, this chapter investigates how these vulnerabilities are exploited today and what kind of techniques attackers use. With the purpose of presenting an exhaustive state of the art on Java Card security, each presented attack vector is related with its respective vulnerabilities and threats presented in Chapter 3 and Chapter 4. Risks levels are not defined because these are strictly dependent on card issuers' requirements and applet type.

Java Cards have a common point with native smartcards: they perform sensitive operation on critical user data, which should be kept secret between the involved parties. Therefore, these cards have to be secured both from a physical and a software point of view. The attacker's goal is to disclose already stored code and confidential data. Even if Java cards provide a secure data container, by means of its security mechanisms introduced in Section 3.3, attackers have still some chances of success. Recalling the fact that a JCVM implementation is not required to withstand attacks from ill-formed applets (refer to Section 3.1), an attacker can threaten a Java card using generally three techniques:

1. Perform differential power analysis or fault injection.
2. Exploit directly bugs or oversights in the installed applets.
3. Trigger type confusion by means of ill-formed applets.

These techniques are presented respectively in Section 5.1, Section 5.2, Section 5.3.

5.1 Power Analysis and Manipulation

This technique focuses on exploiting two physical phenomenons related to electrical power: *voltage* and *power consumption*. Thus, they are part of the Side Channel Attack class. Each attack, namely differential power analysis (DPA) and fault injection, is explained in Section 5.1.1 and Section 5.1.2. Hereafter, Table III illustrates a complete overview in term of attacks, vulnerabilities, threats and countermeasures.

Attack	Vulnerability	Threat	Countermeasures
DPA	"Power consumption leakage"	*A verified applet leaks sensitive data*	Refer to Section 4.3.2
Fault Injection	"Fixed voltage"	*A verified applet leaks sensitive data* *A verified applet abuses features* *A verified applet exploits bug/oversights*	Refer to Section 4.3.4

Table III: Power Manipulation and Analysis, security overview.

5.1.1 Differential Power Analysis

Differential Power Analysis is a statistical method for analyzing sets of power measurements (power traces) to identify input/output data-dependent correlations, with the purpose of discovering critical secrets such as encryption keys. It is a topic widely studied [Joye 2005, Clavier 2000, Bevan 2003, Petrvalsky 2014] that was introduced for the first time by [Kocher 1999]. His goal was to disclose RSA keys in a weak algorithm implementation during the "Square and Multiply" step of modular exponentiation [Gopal 2009]. This technique, as introduced by Kocher, can be also used to reverse the code of an application or algorithm by means of power traces collections coming from different executions. Also electromagnetic emission can be exploited using conceptually the same technique to reach the goal [Dyrkolbotn 2011].

5.1.2 Fault Injection

Fault Injection refers to the technique of physically stressing a chip with the purpose of triggering erratic behaviours to elude in-act countermeasures. By means of fault injection, perturbations in the execution environment are injected so that values in memory cells change, different signals are sent over communication bus lines and structural elements get damaged [Bar-El 2006]. As a result different behaviours can be generated: entire instructions changed, instructions skipped, wrong or invalid branches triggered etc. Mainly, it can permit an attacker to access sensitive data or execute operation beyond his/her rights abusing platform features or exploiting bugs or oversight at operating system or applet

level. For example, an interesting attack is presented by Barbu et al. [Barbu 2012a]. Note that it is a combined attack[1], therefore this requires an attacker to have the privilege to upload onto the card its own applet. They targeted the APDU buffer itself, making it accessible to a malicious applet at any time, thus threatening the security of the platform, the hosted applications as well as the privacy of the cardholder. At first glance, this attack could be interpreted simply as a man-in-the-middle attack between the terminal and the card. However, one could observe the APDU buffer contains received, emitted data and temporary files used as buffer from applets. Using a single fault injection Barbu et al. proved that is possible for an attacker to gain permanent access to the buffer. The JCRE is responsible, as described in [Microsystems 2006d, Sec.6.2.8.3], to prevent an applet from storing in its memory area references to global arrays, and in particular to the APDU buffer array. The APDU buffer, due to its global status, is also cleared to zeroes whenever an applet is selected [Microsystems 2006d, Sec.6.2.2]. That said, the JCRE must perform some runtime checks in order to enforce these two mentioned behaviours. A possible check is the one illustrated in Listing 5.1.

```
1 // ref points to the object to store
2 if (is GlobalArray (ref)) {
3 // Handle storage attempt
4     throw Security Exception
5 }else{ ... }
```

Listing 5.1: Detection in the APDU buffer storage

Barbu et al. showed that an attacker, using fault injection, can force to jump in the *else* branch. Using his/her own applet, which tries to store the reference of the APDU buffer into a global array, the attacker is able to store the reference of the APDU buffer in a non-volatile field, thus accessing it anytime. Depending on the applet that has been uploaded, an attacker is potentially able to copy every byte written in the APDU buffer in a local applet buffer. Moreover, the attacker can also modify the content of the APDU buffer using the reference to write it back instead of simply copying. An example is illustrated in Listing 5.2.

```
1  // field Buf is the stored APDU buffer reference.
2  // BUF LEN is the assumed APDU buffer length.
3  // tmpBuf is a byte array initialized with a size of BUF LEN.
4  // os is an OutputStream used by the attacker.
5  public void run (){
6      while (true){
7      // APDU buffer is different, copy its content.
8          if (arrayCompare (field Buf,0,tmpBuf,0,BUF LEN) != 0){
9              System.arraycopy(fieldBuf,0,tmpBuf,0,BUF LEN);
10             os.write(tmpBuf);
11         }
12     }
```

[1]Combined attack refers to the technique of combining side-channel attacks with software attacks. The side-channel attack changes the environmental conditions so that the logical attacks can be performed.

```
13 }
```

Listing 5.2: Eavesdropping the APDU buffer

In the smartcard field, beside voltage manipulation, which is the most used and known technique to trigger faults, other three strategies have been widely studied and are still today used: electromagnetic spikes [Kömmerling 1999], clocks glitches [Anderson 1998] and optical attacks [Skorobogatov 2003].

5.2 Applet Exploitation

This technique is strictly dependent on a use case, i.e. an installed applet and its environment. For such a reason little literature can be found. However, an interesting tool to automatize attacks aimed at exploiting communication and crypto protocols is described in [De Koning Gans 2012].

Attacks vectors of this technique belong to the Logical Attacks class and exploit some of the vulnerabilities presented in Chapter 4, namely:

- *Hidden Commands*
- *Unchecked Parameters*
- *Unsafe Crypto Protocols*

Attacks vectors related to this technique can only lead to applet data leakage or tampering. However, note well that, in case the exploited applet communicates via the *Object Sharing Mechanism* with other applets, also useful information of these other applets might be leaked, due to the applet firewall limitation presented in Section 3.3. That said, all attack vectors related to this technique can be related to threat *"A verified applet leaks sensitive data"* presented in Section 3.2. High level countermeasures can be found in 4.1.2.

5.3 Type Confusion

This technique is the most direct, disruptive and straightforward approach that an attacker can use for exploiting Java cards, as with relatively simple code it is possible to break entirely the integrity of the platform. The attacker's goal is uploading onto the card ill-formed code so that the cornerstone of Java[Card] security, safety type checks, is endangered. The main advantage of this technique is the fact that it is not related to any installed applet, therefore, theoretically it is feasible in any use case. However, due to the deployed security mechanisms in place loading ill-formed code onto the card is not a straightforward task.

This technique can be conceptually divided in three steps as follows:

- Obtain the right to load and install application on card, refer to Section 5.3.1.

- Inject ill-typed code on card, refer to Section 5.3.2.
- Run the developed attack vector, refer to Section 5.3.3.

5.3.1 Obtaining rights to load code

The right to load an applet onto a card is not as obvious as it often looked in previous works [Iguchi-Cartigny 2009, Mostowski 2008, Witteman 2003] attempting to attacks Java Card devices.

An attacker needs to be somehow granted with the right of uploading code onto the card. There are three ways in which this can be done [Barbu 2010]:

1. The attacker obtained from the card manufacturer the card manager key set, which means he plays the role of issuer. The attacker can upload any package or install any applet without limitations. This can happen in three cases:
   - The issuer is doing malicious activity.
   - The attacker stole the card manager key set from the issuer.
   - The attacker is using a bought white card for performing an attack. However, in this case, few assets are at stake, making the reach of any attack limited.
2. The attacker can load a package or install an applet by *Delegated Management*, described in [Platform 2003]. That means the issuer, which has been given by the card manufacturer the master keys, has created a kind of loading environment, called *Security Domain*, which provide loading, installation and communication features with specific privileges. Any use of this mechanism requires ownership of some secret keys for authentication and secure communication purpose. Note well that installing an applet by *Delegated Management* requires the *install* APDU command to be signed by the card's issuer.
3. The attacker can load a package or install an applet by *Authorized Management*, described in [Platform 2006]. To do so, an issuer has to be deployed beforehand a *Security Domain* and provided the respective secret keys.

In conclusion, having the right to upload and install applets in the current situation, described in Chapter 1, is not obvious as either card manufacturers and card issuers collaborate strictly or card manufacturers play also the role of card issuers. Moreover, note that post-issuance code loading is often forbidden and therefore disabled.

5.3.2 Injection of ill-formed code

In order to inject ill-formed applets onto the card we need to assume that the attacker gained somehow, as described above, the right needed to load code onto the card.

At that moment, in order to trigger type confusion, four approaches can be used to inject ill-formed code onto a smartcard [Mostowski 2008]:

- CAP file manipulation
- Abusing the Sharable Interface Object
- Abusing the Transaction Mechanism
- Fault Injection

Each of them is below specified providing examples. Table IV outlines type, vulnerabilities, threats and countermeasures of each attack vector. The threat column is not reported in the table as all the listed attacks refer to *An ill-formed applet compromises the entire platform*.

Attack	Type	Vulnerability	Countermeasures
CAP Manipulation	Logical	"Lack of Type Checks"	Static on-card bytecode verifier Secure signing system Runtime safety type check[1]
Abusing Sharable Interface Object	Logical	"Lack of Type Checks"	Typed verification on export files Runtime safety type check[1]
Abusing Transactions	Logical	"Transaction System Bugs" "Lack of Type Checks"	Deallocate objects Reset references Runtime safety type check[1]
Disabling the Checkcast	Side-channel Logical	"Power consumption leakage"	Refer to Section 4.3.4
Confusing the Operand Stack	Side-channel Logical	"Power consumption leakage"	Refer to Section 4.3.4
Corrupting Java Card Exceptions	Side-channel Logical	"Power consumption leakage"	Refer to Section 4.3.4
Agnostic Modification	Side-channel Logical	"Power consumption leakage"	Refer to Section 4.3.4

[1] Particularly impactful on performance. Hardware constraints still do not usually allow it. However, it would be possible to turn this feature on only in particular sensitive situations as described in [Bouffard 2014, Ch.5].

Table IV: Type Confusion, security overview.

Some of the listed countermeasures are tailored for a specific attack, for this reason they have not been mentioned in the common high level countermeasures in Chapter 4.

When it comes to ill-formed code the more runtime checks a VM implementation deploys the more resilient it becomes. Hereafter a list of further proposed dynamic countermeasures [Mostowski 2008]:

- *Object Bounds Checking*, any JCVM implementation is required to perform array bound checks when accessing array elements. This check could be extended to object bounds when accessing instance fields and invoking methods on object that are not arrays. Obviously, this mechanism requires an object to record its size variable, which corresponds to its number of fields, in the same way it records the size of an array object. When converting *.class* files to *.CAP* files, field's names of the instances are replaced by sequential numbers. Therefore this checks is easy to implement.

- *Physical Bounds Checks*, array bounds checks are performed using logical sizes, this could be extend with a further check on the physical size in term of memory offsets.

- *Integrity Checks in Memory*, when performing dynamic checks for array bounds, downcasting and firewall's context switching the VM has to store some metadata in the memory representation of objects. Further meta information about references might be stored and checked in a object's memory representation in order to prevent switching or spoofing of references.

5.3.2.1 CAP File Manipulation

CAP file manipulation is the simplest way to get ill-formed code onto the card. It consists in editing a CAP file with the purpose of introducing a type flaw in the bytecode and install it. Obviously, this attack works only on cards that are not equipped with either an on-card bytecode verifier or a *secure signing system* for CAP executables and related files, as already mentioned in Section 2.2.1. In this case, *secure* means robust in term of cryptography and coherent in term of policies, i.e. it has to be executed immediately after the applet's CAP file verification procedure. A typical example that proves how straightforward it is to perform this attack can be the following one.

Example* 5.1*: The opcode *baload*, which is used to load a byte or Boolean value from an array, can be changed to *saload* opcode, which is used to load a short type from an array. This misinterpreted array type might potentially lead to accessing the memory area of another installed applet. Further details in Section 5.3.3.

5.3.2.2 Abusing the Sharable Interface Object

The Sharable Interface Object, introduced in Section 3.3, allows applets belonging to different contexts to communicate legally across the firewall. This mechanism can be exploited to trigger type confusion. Mostowski et. al succeed in this attack developing two applets, a client and a server, with the following two interfaces:

```
// Client Interface
public interface MyInterface extends Shareable {
    public byte[] giveArray ();
    public void accessArray (byte[] myArray); // Client assume byte[]
}

// Server Interface
public interface MyInterface extends Shareable {
    public byte[] giveArray ();
    public void accessArray (short[] myArray); // Server assume short[]
}
```

Listing 5.3: Abusing the Sharable Interface Object Mechanism

The key point of this approach is compiling and loading the two applets separately because an off-card verifier does not usually perform types checking over the export files. Since the two applet resides in different contexts the server cannot access an array of the client, therefore, to make the attack works the client must first request the array of the server and then give it back. In this way, the server interprets the received array as a *short* type instead of *byte* and can read a doubled memory size.

5.3.2.3 Abusing the Transaction Mechanism

The Transaction Mechanism, as described in Section 3.3, permits to perform a set of bytecode instructions atomically offering a rollback mechanism in case the operation is aborted. The rollback mechanism should deallocate any objects allocated during the aborted transaction, and reset references to such objects to *null* [Microsystems 2006c, Ch.7]. However, in the reality, this deallocation process is often performed incorrectly leading to unauthorized access to some resources [Hogenboom 2009]. An example of an aborted transaction that could lead to type confusion, and in turn, to unauthorized data is illustrated in Listing 5.4.

```
short[] arrayA; // instance field, persistent
byte[] arrayB; // instance field, persistent
...
short[] localArray = null; // local variable, transient

JCSystem.beginTransaction();
arrayA = new short[1]; // allocation to be rolled back
localArray = arrayA; // save the reference in local variable
JCSystem.abortTransaction(); // arrayA is reset to null,
// but not localArray
arrayB = new byte[5]; // arrayB gets the same reference as arrayA

// this can be tested as follow:
if((Object)arrayB == (Object)localArray) // condition is true
```

Listing 5.4: Abusing the Transaction Mechanism

This is made possible because only the persistent objects involved in the transaction are freed and reset to null. Updates to transient objects and global arrays are never undone, regardless of whether or not they were involved during a transaction. Surprisingly, this behaviour is specified in the JCRE specification of both version 2.2.2 [Microsystems 2006c, Ch.7-7] and 3.0 Classic Edition [Oracle 2011, Ch.7-7]. Card manufactures should be aware of this attack vector and deploy further security mechanism (refer to Table IV).

5.3.2.4 Fault Injection

Fault injection can be used to also introduce type flaws. Usually, fault injection is difficult to control as it does not provide high precision. However, it is possible to increase the opportunities of success by means of interesting expedients embedded in the code. For further details we refer to [Govindavajhala 2003]. That said, it has been proven that it is possible to combine fault and logical attacks to induce type confusion attacking different system's components. In this case, the applet's specifications are correct but the environmental hypothesis are false. Below, the techniques known today are introduced.

Disabling the Checkcast Barbu et al. [Barbu 2010] described for the first time this new type of combined attack. He used a laser beam to modify a correct applet's execution flow at running time. The applet was verified by an on-card static bytecode verifier and installed. The goal was forging a reference to an object by means of type confusion. In doing this three classes A,B,C were defined, as illustrated in Listing 5.5.

```
1  public class A {                public class B {                public class C {
2      byte b00,...,bFF;               short addr;                     A a;
3  }                               }                               }
```

Listing 5.5: Used classes to create type confusion

When a type cast needs to be performed, the JCRE dynamically verifies if the involved types are compatible through the *checkcast* instruction. The cast mechanism is explained in [Microsystems 2006c, Sec.6.2.8]. Barbu et al. developed an applet with an illegal cast (line 12), using the above-presented classes. The code is shown in Listing 5.6.

```
1  public class AttackExtApp extends Applet {
2      B b;
3      C c;
4      boolean classFound;
5
6      // Constructor, install method
7      public void process (APDU apdu) {
8          byte[] buffer = apdu.getBuffer() ;
9          switch (buffer[ISO7816.OFFSET_INS]) {
10             case INS_ILLEGAL_CAST :
11             try {
12                 c = (C) ((Object)b);
```

```
13                  A.a = c.a;
14                  b.addr = ADDRESS_TO_READ; // The reference of c.a equals b.addr
15                  read_area_memory [0x00] = a.b00; ... read_area_memory [0xFF] = a.bFF;
16                  return ; // Success, return the sw 0x9000
17              } catch (ClassCastException e) {
18                  // Failure, return the sw 0x6F00
19              }
20          }
21      }
22  } // more later defined instructions
```

Listing 5.6: Casting to create type confusion

Looking at the above code, a *ClassCastException* is thrown by line 12. With specific instruments (oscilloscope etc.) the thrown exception is visible in the power traces. Barbu et al. were able to prevent the checkcast routine from throwing the exception with an highly time-precise fault injection, using a laser beam. Once the cast is done, it is possible to access an actual B instance either as a B or a C object. Therefore, forging a's reference to any value, through $b.addr$ variable, allows to read and write as many bytes as declared byte fields of class A. Figure IX shows graphically this malicious connection between the three objects

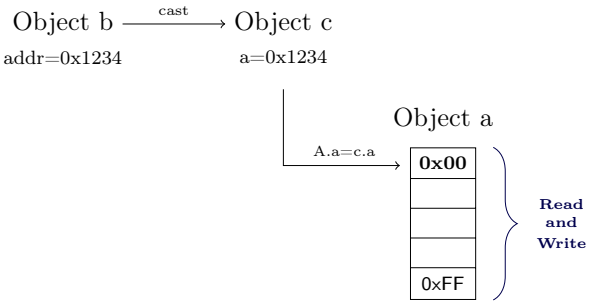

Figure IX: Attacking the checkcast check

This attack can also enable the *EMAN1*, introduced in [Iguchi-Cartigny 2010], enabling an attacker, exploiting static instructions, to read and write anywhere in the Java Card memory.

Confusing the Java Card Operand Stack The JCVM, and more generally, Java Virtual Machines are known as stack-based machines, in opposition to register-based machines[2]. Several type of stacks are described in the JVM specification, we refer to [Lindholm 1999] for further details. However, the below-explained attack focuses on a particular type of stack, the *operand stack*. A Java frame is created on each Java method *invoke* to store temporary specific data needed for the execution. The operand stack is

[2]A register-based machine is a generic class of abstract machines used in a manner similar to a Turing machine. It takes its name from the use of registers in managing instruction's execution

part of this frame and is in charge of holding the operands and results of the VM instruction. 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.

Barbu et at. [Barbu 2011] focused on manipulation and corruption of data being pushed in the operand stack using fault injection. Recalling the fact that a Java card complies with the ISO-7816 specification, it receives command through APDUs. The Java Card API, described in Section 2.2.3, provides a class to represent an APDU object and access it as an array of bytes. This array defines the behaviour of the applet. Usually the lines of code, illustrated in Listing 5.7, are executed by an applet.

```
1 byte ins = apduBuf[ISO7816.OFFSET_INS];
2
3 // push ins on the stack and execute the appropriate switch instruction
4 switch (ins) {
5     case INS_A: processInstructionA(apdu); break;
6     case INS_B: processInstructionB(apdu); break;
7     ...
8     default: ISOException.throwIt(ISO7816.SW_INS_UNKNOWN);
9 }
```

Listing 5.7: Example of applet's APDUs access

The value of *ins* is pushed onto the stack before the *switch* instruction. A manipulation of this pushed value using fault injection prior to its execution could lead to a totally changed applet's behaviour. Obviously, the outcome depends on the applet that is being attacked, a payment applet could be definitely an interesting target.

Barbu et al. proposed three types of attack on the operand stack. In the first one a *Boolean* value, involved in a conditional branch, is manipulated prior to its use in order to jump to another statement. The second one is an high probability attack to trigger type confusion, circumnavigate the firewall and gain privileges in another applet. Mainly, to succeed, an attacker should combine fault injection and a malicious applet that generate a huge amount of references from a specified class to increase the chance that a reference will be referred by type confusion. Finally, the last technique is conceptually similar to the previous one and shows how to produce *instance confusion*. Seemingly to type confusion, instance confusion refers to the fact of using an instance i^1 as if it were an instance i^2 .

Corrupting Java Card Exceptions Barbu et al. [Barbu 2012b] presented another combined attack exploiting the exception mechanism. Three variations of this attack have been introduced. In the first one, the author was able to trick the exception mechanism when searching the correct statement of the *try-catch* block, and execute an expected statement. The second one is based on the *athrow* instruction, defined in the JCRE specification [Microsystems 2006d], that throws an exception or an error based on an object reference pushed onto the stack. Based on the attack previously described [Barbu 2011],

they were able to manipulate the reference value loaded onto the stack and throw another object, even a not-throwable one. In the last one, Barbu et al. focused on the class constructor. The first statement to be executed is the *super* function that calls the constructor of the parent class. In case the *super* function throws an exception the corresponding *try-catch* block is executed. Note that in the *catch* code block the *super* method is not required to be run. Based on that, the author used fault injection to induce the VM to throw an exception from the *super* function creating an instance that was not correctly initialized.

Agnostic Modification Lancia [Lancia 2013] exploited the Java Card instance allocator using an high-precision fault injection. His attack development is based on the work [Govindavajhala 2003].

In order to understand the attack some main differences between Java and Java Card have to be pointed out. Firstly, as opposed to the traditional Java Runtime Environment, each instance's header created by the JCRE is allocated in the persistent memory. More precisely, the JCRE specification provides functions to create transient objects whose data are stored in the RAM memory, but their headers are always in the persistent memory. Secondly, traditional Java and Java Card platform use a different memory management. Figure X illustrates these mechanisms. In standard Java VM, references are represented using a direct memory addressing, which means that physical memory addresses of the referenced instances are stored in the physical memory as traditional pointers. Whereas, in JCVM references are represented using an indirect memory addressing: references are indexes that are interpreted using a global instance pool managed by the JCVM.

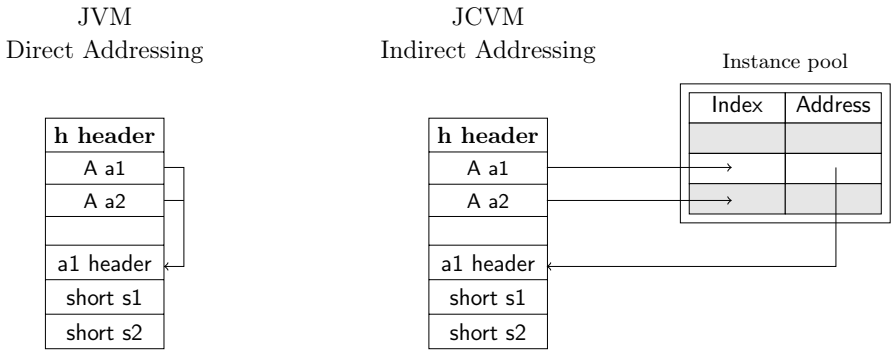

Figure X: The Java and Java Card memory addressing

Because of this difference, the attack on traditional Java VM introduced by Govindavajhala is not applicable to JCVM. For example, a type confusion between a *short* field and a *instance* field in case of direct memory addressing, is simply performed using the value of the *short* variable as an address to scan the memory. In case of indirect memory addressing, the same type confusion gives access to an index in the global index pool, which means that further steps are needed. Lancia developed an attack starting from this

point. He attacked the global instance pool mechanism (persistent memory) using a laser beam. As a result, the index referred to the bytecode was shifted. Therefore, during the applet execution, the JCRE resolves an index with the associated address using the global instance pool 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.

5.3.3 Running a developed attack vector

Section 5.3.2 described the known expedients to get ill-formed code onto the card providing some examples from the literature. The goal of this section is generalizing on what an attacker can try to do once ill-formed code is uploaded, recalling the fact that the purpose of the attacker is to disclose and/or manipulate data already on the card.

Mostowski et al. [Mostowski 2008] pointed out that the most used attacks are:

- Treating a Byte Array as a Short Array
- Treating an Object as an Array
- Treating an Array as an Object

The first two attacks are explained below . The third one is not further examined as it is conceptually quite close to the second one.

5.3.3.1 Treating a Byte Array as a Short Array

This is the most common attack that allows to read unauthorized memory areas. A typical example is treating a *byte* array as *short* array so that you can read a double memory offset. Different type of arrays could be taken as target obtaining the same outcome: global arrays, persistent context-owned arrays, transient context-owned array.

5.3.3.2 Treating an Object as an Array

An attacker might try to confuse an arbitrary object with an array. This opens the following attack opportunities.

Fabricating Arrays Witteman [Witteman 2003] proved that an attacker could confuse the VM and be able to fabricate an array of an arbitrary size. Listing 5.8 shows the class used.

```
1 public class fake{
2     short len = (short)0x7FFF;
3 }
```

Listing 5.8: Fake class to fabbricate an array

The attack relies on a specific representation of arrays and objects in memory. An attacker, to succeed, has to store the length field of an array at the same offset in physical memory as the *size* field of a *Fake* object. If one is able to confuse the VM, then the array size is set to the one predefined, i.e. *0x7FFF*, giving access to 32kb of memory. Updating the *size* field means updating also the size of the array to an arbitrary size.

Direct Object Data Access As a consequence of the previous attack, an attacker could try to treat object fields as array elements. Thus, references fields could be read and written as numerical values leading to pointer arithmetic. Listing 5.9 illustrates an example assuming that object *Example* is treated as an array *a*.

```
1 public class Example{
2     Object o1 = new object();
3     short s1 = 1;
4 }
5
6 // Treating an instance of Example as an array a gives the following values
7 a.length:  0x0E90 // Number of fields
8 a[0]:      0x010A // Reference of o1
9 a[1]:      0x020C // Reference of s1
```

Listing 5.9: Example of direct data access

By reading and writing the a[0] element it is possible to directly read and write references.

Switching References Once an attacker has direct access to the references of object's fields, he could try to replace them with others, also of incompatibles types. Mostowski et al. proved that this is perfectly acceptable if the new values, assigned through direct access, are valid references. At this point, an attacker is able to produce even more type confusion, but he cannot simply make a field of his object to point to another applet's object. In that case the firewall would prevent the access.

AID Exploitation An attacker, by means of reference manipulation (switching references), could try to tamper system-owned AID objects. An AID object is defined in [ISO 2013b] to be a sequence of bytes between 5 and 16 bytes in length. The JCRE creates instances of AID class to identify and manage every applet onto the card. These kind of objects are permanent Java Card runtime environment Entry Point Objects and can be accessed from any applet context. References to these permanent objects can be stored and re-used. An AID object has a field which points to a byte array that stores the actual AID bytes. An attacker could change this value, without being detected by the firewall, through the above described direct data access. At this point, an attacker could try to impersonate another applet. An interesting attack that makes use of this concept is described in [Montgomery 1999].

Part II

An Augmented Bytecode Verifier

Chapter 6

A Solution to the Physical Secure Element Issue

Referring to Chapter 1, the current initialization and management of physical secure elements is the main issue that is leading to the use of workaround solutions such as the well-know "host-based card emulation". As a secure element manufacturer, *NXP Semiconductors* is interested in inverting this trend. This chapter introduces the second part of my work introducing our proposal to tackle the problem.

Current situation A secure element should be capable of storing and processing sensitive information of a user securely. Authentication, encryption of data, data integrity and non-repudiation are services that a secure element provides. Currently, *NXP Semiconductors* typically establishes a strong relationship with a service provider (Card Issuer) and installs, on its behalf, all the software needed into secure elements to make the requested service(s) work. Typically, after the secure element is issued no software can be installed afterwards and JCOP, the proprietary Java Card Technology implementation of *NXP Semiconductors*, is in charge of enforcing the above-mentioned security services.

Ideal situation Ideally, a mobile device's secure element should be largely customizable based on the needs of an end user, who should be able to install or remove different kind of applets as needed. Proximity payment applets are the most interesting use case, but also other kinds of applets, which require an high level of security, might be installed such as loyalty applets. A secure element should be simply considered as an extension of a mobile device operating system that enforces higher security but with a comparable flexibility. Unfortunately, this ideal scenario is still too unrealistic because of the lack of a system with the capabilities of enforcing security while presenting also a high level of flexibility. Technically, designing a highly flexible system would mean allowing a card issuer to manage its own content inside a secure element and an end user to install applets as needed. In other words, post-issuance installation and multi-party applets on a secure element must be allowed. Two of the main benefits of Java Card technology are *multi-application*, which permits multiple applets to reside on the same card, and *dynamism*, which allows post issuance applets uploads. Thus, technically, a Java Card secure element allows to provide these features. On the one hand the use of these features would greatly improve flexibility, but, on the other hand security would be put in danger and, consequentially, trust in *NXP Semiconductors* would lower considerably. The probability for an attacker to

gain the right to install malicious code onto a secure element increases significantly, and, as a consequence, attacks that before were considered unfeasible because it was assumed that an attacker did not have any right to upload malicious code onto the card, become now possible making any kind of attack described in Chapter 5 an opportunity to break into the system and damage the entire set of applets data onto the secure element. To avoid that, **our proposal is to design an augmented bytecode verifier** that not only checks that *CAP files* comply with the Java Card specifications, but also provides a further level of security implementing specific features to tackle the current set of known attacks and adapting quickly when new ones are discovered. Applying these extra checks enables flexible installation of libraries by card issuers and applets by end users.

Technical challenges Figure XI illustrates graphically the software's working principles. The system takes as input an executable *CAP* applet, along with its export file and the export files of the already installed applets. Firstly, the consistency of the *CAP file* is checked, as performed by any common verifier. Secondly, all modules needed to check for unwanted behaviours and to enforce security, with post issuance uploads and multi-party application features enabled, are run. Thirdly, the *CAP file* and its *export file* are signed to enforce integrity and source authentication. After verification, the applet is ready to be uploaded.

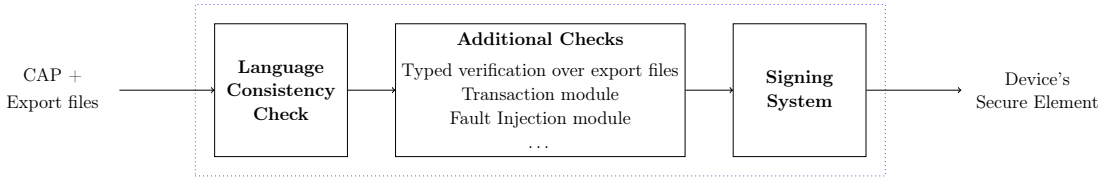

Figure XI: The Off-card verifier working principle

This solution allows to counteract any kind of attack at compile time and also at runtime using emulation technologies. The use of independent modules for each check makes the system strongly adaptable over time in case of newly discovered attack vectors. Table V shows currently known threats that can be mitigated at compile time. *CAP Manipulation* is avoided enforcing consistency checks and *CAP file* signing. *Abusing SIO* is avoided by

Attack	Type	Vulnerability	Countermeasures
CAP Manipulation	Logical	"Lack of Type Checks"	Secure signing system
Abusing Sharable Interface Object	Logical	"Lack of Type Checks"	Typed verification on export files

Table V: Attacks considered

means of a module to enforce typed verification over export files.

Further improvements Other useful checks that might be performed deal with side-channel attacks. Typically, security mechanisms against side-channel attacks and overall fault-injection should be deployed directly at JCRE or applet level in order to be effective [Barbu 2011, Bouffard 2014], therefore no verifier can prevent them. However, a module that tests how vulnerable an applet is before its upload onto a secure element might be extremely of interest and might be embedded in our verifier concept. An interesting source that might provide a starting point for the design of such a module is [Diaconescu 2013].

Research direction The verification process itself will not be analyzed any longer in this report. Any piece of software to increase security that an applet has to comply with in order to be uploaded onto a secure element can be easily added to our system as a module. However, the most relevant question at this point is: **how can we design a process such that an augmented bytecode verifier can be used to provide a flexible and highly-secure system for applet installation?** Next chapters introduce all required aspects to answer this question. More precisely:

- Chapter 7 focuses on the business requirements of our stakeholders analysing their impact on the software design. Related research questions:
  - *What are the business requirements of our stakeholders?*
  - *What is the impact of their requirements on the software design?*
  - *Why do protection techniques need to be integrated? And what security mechanisms should be deployed?*
- Chapter 8 presents a new scenario that mitigates the inflexible context presented in Chapter 1 while enforcing a high level of security. Related research questions:
  - *How do stakeholders interact to meet their own business requirements and create a trusted environment that is flexible and scalable?*
  - *What are the properties the scenario should guarantee?*
  - *How can the applet verification process be indirectly monitored by NXP Semiconductors without impacting the flexibility of the process?*
- Chapter 9 focuses on the technical details presenting the working principle of the augmented bytecode verifier in terms of protocols and security mechanisms. Related research questions:
  - *How can the system be designed to withstand to White-box attacks by design?*
  - *How NXP Semiconductors can be self-confident that the applet verification process ended successfully?*

- Chapter 10 explains our future plans and draws conclusions. Related research question:
  - *What makes the system trusted by each involved stakeholder?*
  - *Why can we consider our system design successful?*

Chapter 7

Requirements Analysis

Contents

7.1 Business Model 72
7.2 Architectural Features 73
7.3 Protection Techniques 75
7.3.1 Data Encryption 76
7.3.2 Runtime Integrity Checks 76
7.3.3 Code Obfuscation and Code Flattening 76
7.3.4 White-box Cryptography 77
7.4 Key Points 78

This chapter provides a requirements analysis on the main relevant aspects that characterize the augmented bytecode verifier and impact the software architecture. The main goal is to introduce the software from different angles to catch the requirement's complexity behind a verification system that at first glance someone could miss. More in detail, with this chapter, we want to answer the following questions:

- *What are the business requirements of our stakeholders?*
- *What is the impact of their requirements on the software design?*
- *Why do protection techniques need to be integrated? And what security mechanisms should be deployed?*

Section 7.1 introduces in detail the parties involved in the verification process to grasp their needs, with the purpose of designing a system that meets their business requirements along with the ones of *NXP Semiconductors*. Section 7.2 describes in terms of software the impact of the business requirements presented in Section 7.1 and focuses on further features the system should provide by design to tackle the edge cases[1]. Section 7.3 emphasizes the power of a malicious agent and describes protection techniques that should be deployed on the system to counteract potential attacks. Finally, Section 7.4 briefly summarizes each answer of the above-presented research questions to explicitly highlight the key points of the chapter.

[1] An edge case is a problem or situation that occurs only at an extreme (maximum or minimum) operating parameter.

7.1 Business Model

In an ideal mobile device's secure element environment three stakeholders, which act in the process of applet verification, can be identified. Namely, card manufacturer, card issuer and application developer. For the sake of clarity, hereafter, a quick description of each of them is given with the purpose of describing their role under the ideal situation introduced in Chapter 6.

A *Card Manufacturer* is an authority that fabricates the raw hardware and software. It delivers secure elements ready to be used to *Card Issuers*. Note that the same secure element might be accessed securely by several *Card Issuers* upon agreement. *NXP Semiconductors* is part of this category.

A *Card Issuer* is an authority that controls the secure element content. It has the right to add further software features to the secure element (proprietary libraries), along with applets that provide services to end users. Moreover, it can grant also other institutions to administrate their own applets. Examples of famous companies that might be part of this category are *Google*, *Apple*, *Samsung*, *Visa*, *Mastercard*.

An *Application Developer* is an authority that implements applets. It can be anybody who has been granted by a *Card Issuer* to implement on its behalf an applet and to make it ready to be distributed using predefined channels.

According to *NXP*'s experts, these stakeholders have several needs and requirements, which need to be met to ensure the effectiveness of the new product. Hereafter, a list of all relevant stakeholder's business requirements[2] in order of priority, as also illustrated in Table VI:

1. The system runs in a potentially hostile environment, the *Application Developer*'s location.

2. *NXP Semiconductors* needs to be confident that the process of verification is successfully performed prior to any upload onto its secure element.

3. *Card Issuers* want to establish relationships with *Application Providers* freely without any control by *NXP Semiconductors*. In other words, they want to manage their own *Application Providers* and be an active part in any applet upload process performed by one of their *Application Providers*.

4. *Card Issuers* do not want to share their applet source code or bytecode. *Application Providers* do not want to share their applet source code or bytecode with someone else who is not the *Card Issuer* they are working with (Sort of Intellectual Property protection). However, a *Card Issuer* might also be uninterested in controlling the source code or bytecode of an *Application Provider*. On the other hand, *NXP Semiconductors* does not want to personally look at the verified code in order to decline any kind of responsibility in case of legal issues.

[2]The system is designed for NXP Semiconductors, therefore the card manufacturer examined from which several business requirements come out is NXP.

5. *Card Issuers* want *transparency* in term of checks performed to applets' files. In other words, they want to be able to measure on their own the effectiveness of the implemented checks as needed. This would enforce trust in the system by *Card Issuers*.

6. *NXP Semiconductors* would like to be able to indirectly control any uploads onto its secure elements.

			Priority		
		Stakeholder	Major	Moderate	Minor
Requirement	1. *Hostile environment*	AD	✓		
	2. *Verification enforcement*	NXP	✓		
	3. *Flexible relationships*	CI - AD	✓		
	4. *Code confidentiality*	CI - AD		✓	
	5. *Transparency*	CI		✓	
	6. *Uploads monitoring*	NXP			✓

NXP: NXP Semiconductors, CI: Card Issuers, AD: Application Developers

Table VI: Stakeholders' business needs priority

All the mentioned business needs are taken into consideration while designing the augmented bytecode verifier architecture. I refer to my entrepreneurial thesis, which can be found at the end of this document, for further business-related information. The report focuses on the design of a framework to discover effectively the widest range of needs of these stakeholders.

7.2 Architectural Features

The functional objective of the augmented bytecode verifier is fairly clear: testing the correctness[3] of an applet and enforcing a security threshold level that has to be met prior to its upload onto a secure element. At first glance, this might look like a simple modular software, running locally in a specific location, where each module corresponds to a particular check performed on applets files. However, due to the stakeholders' business requirements discussed in Section 7.1, the software architecture become more complicated and its design is influenced tremendously.

[3]Correctness in terms of security, i.e. to guarantee that an applet is not malware that might endanger the JCRE or the installed applets.

Requirement 1 is strongly demanding and influences significantly the mechanisms required to successfully enforce requirements 2 and 6. It requires the software to be resilient to attacks where each part of the system can be inspected and exploited by a malicious agent. This type of context is called *white-box* [Brecht 2012], as an attacker has complete control of the software core and its hardware. Simple deployments of traditional cryptography mechanisms to enforce our requirements are not enough anymore since classic cryptography techniques, both symmetric and asymmetric, are designed assuming an attacker in a *black-box* scenario, which means that encryption and decryption of data is supposed to be done in a secure environment, giving an attacker access only to input and output data. Figure XII shows this concept. Our scenario requires that even the developed software and its underlying hardware is not trusted. In other words, protection techniques, which will be introduced in Section 7.3, need to be deployed with the purpose of protecting the software binary, its execution and its embedded secrets such as licenses and cryptographic keys.

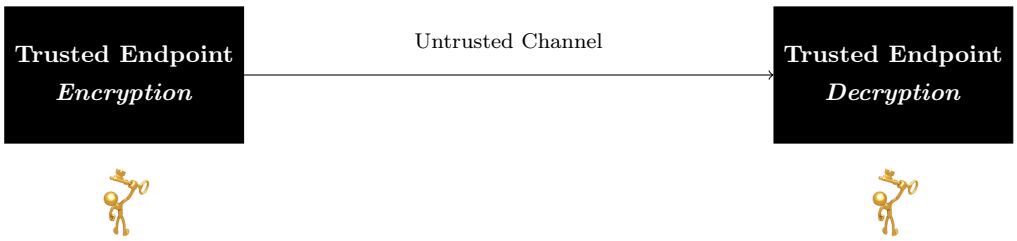

Figure XII: The traditional cryptography conception

Requirements 3 and 4 refer to the architecture of the system that needs to be somehow distributed over the involved parties. However, the interaction amongst the system's parts need to be carefully designed as *Card Issuers* want to have an active role in the whole process and have a certain level of freedom, whereas *NXP Semiconductors* wants to be able to control indirectly any applet upload.

Requirement 5 might be easily achieved in two ways. Firstly, one has to make the system strictly modular and allow *Card Issuers* to inspect the code. With this structure, *Card Issuers* might even propose further checks and *NXP Seminconductors* could perform updates much more easily. Secondly, *Card Issuers* should be able to test a verifier if requested. However, under a *white-box scenario* allowing transparency, this means also to weaken the system itself and to make it vulnerable. A trade-off between security and transparency has to be found.

In conclusion, in this hostile context it is extremely difficult to predict how harmful a flaw in the design will be to the integrity of the verifier (functionalities, secrets etc.), therefore the system has to be designed to withstand attacks, adapt to attacks and avoid catastrophic failures in case of exploitation. Following this claim, three fundamental features have to be designed:

- *Renewability*
- *Revocation*
- *Binary Diversity*

Renewability is a channel to updates secrets used along the verification process. Such secrets might be for cryptographic purpose like keys or control purpose like licenses. This feature allows to renew and restore security in case of breach. However, under specific conditions, performing a secret renewal might not be feasible and to minimize the cost of failure *Revocation* could be applied. In this case, *NXP Semiconductors* might directly revoke from an *Application Provider* or *Card Issuer* the right to use its system. Moreover, to further defend the integrity of the whole architecture (described in Chapter 8) *Binary Diversity* might be employed. It consists of changing slightly the binary of each distributed verifier so that, in case of failure, the damages will be limited as an attack is not scalable.

7.3 Protection Techniques

When designing any security system it is critical to identify the points of failure and protect them whenever possible. With regards to the context above-described, a malicious agent could perform attacks to break the integrity of the system in any software state illustrated in Figure XIII.

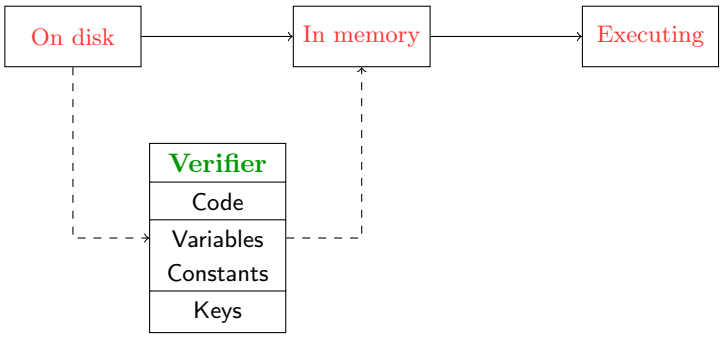

Figure XIII: States of the software

Each of these states has unique conditions that favor different protection techniques. Some of them are more robust than others, but all come with a trade-off. However, their final goal is common, defending the integrity of the system along with its secrets, both on disk and in memory. In other words, due to the sensitive process the verifier is supposed to perform, it is a primary requirement not to allow an attacker to:

1. discover secrets on disk and in memory through static or dynamic forensics analysis.
2. change the software execution flow performing static or dynamic reverse engineering.

An attacker must not be able to sign any applet's file unless the verification process is successfully completed. The next paragraphs describe the protection techniques that are deployed in the system to counteract the widest range of attacks that could be performed on any software states.

7.3.1 Data Encryption

Data encryption is one of the strongest protection measures against unauthorized access to data, if used correctly. This protection technique is strongly relevant in our context since an attacker has full access to the disk and the central memory. Firstly, disk encryption needs to be enforced, otherwise an attacker would be able to perform static reverse engineering or static forensics analysis, endangering the software integrity. These attack techniques are the most straightforward and feasible approaches to trick the verifier, thus they should be avoided at any cost. Secondly, the software should not be totally decrypted when the application runs, otherwise a malicious agent could try to perform reverse engineering or forensics analysis dynamically with unnegligible opportunities of success. Technically, it would not be straightforward, due to security mechanisms deployed at operating system level such as "address space layout randomization" (ASLR), however, assuming that the hardware and software core are under the control of an attacker we cannot rely on them. Therefore, a secure loader to perform partial decryption at runtime should be designed and implemented.

7.3.2 Runtime Integrity Checks

A Runtime Integrity Check is a runtime verification check, based on extracted information from the running system, that allows to detect and possibly react to observed behaviours satisfying or violating defined properties. The use of this protection technique allows to counteract possible modifications an attacker could try to perform to the execution flow. However, due to the totally hostile context, it is not a protection that can be easily deployed. A "remote attestation", which is a totally remote integrity check where a trusted party participates actively, is not acceptable for essentially two reasons: firstly it is not considered an effective techniques unless the hardware where the system is running is well-known; secondly in term of business, it means higher responsibility and costs for *NXP Semiconductors*. Perhaps, making use of another trusted entity, such as a smartcard, could be a good trade-off solution. A smartcard is tamper-resistant, it might be easily authenticated by the system and could perform any type of sensitive check.

7.3.3 Code Obfuscation and Code Flattening

Obfuscation is a protection technique that refers to the process of taking as input the source or machine code of a software and making it extremely difficult to understand for a human. Typically, input and output remains the same, but the complexity of the code

increases greatly for an attacker, who, ideally is not able anymore to perform static or dynamic reverse engineering. Practically, obfuscation is considered a technique to increase the time needed for an attacker to understand the code, making reverse engineering unpractical and not worthy. Common practices are to scramble variable names, split data that was originally in one variable over many, duplicate methods keeping the same name but different signatures etc. Obfuscation is also often coupled with code flattening, which refers to the technique of adding useless extra paths to the decision graph flow of the software. Its purpose is equal to obfuscation, i.e. making the code much more difficult to understand. Used together, the level of security slightly increases.

Obfuscation affects heavily transparency and, as a consequence, modularity. Therefore, in order to respect Requirement 5, this technique has to be carefully deployed finding an acceptable trade-off between security and transparency. Moreover, in our case, the system is extremely sensitive and an attacker would be motivated even if code obfuscation and code flattening are in place. Therefore, this technique cannot be considered an option to protect any secret used by the code.

7.3.4 White-box Cryptography

White-box cryptography refers to those set of techniques that are designed to securely deploy cryptography concepts into a *white-box scenario*. In other words, its primary purpose is protecting sensitive data such as keys or licenses. In our case, this set of techniques is extremely important to create a sort of "trusted island" inside the verifier completing what obfuscation and code flattening missed. Typically, look up tables are used for this goal. Lookup tables should contain sensitive data, which is made incomprehensible for an attacker by means of transformations. For further details refer to [Brecht 2012]. In this context, it is critical how this "trusted island" using lookup tables is created, transformations have to be securely applied so that an attacker is not able to understand how they work. Therefore, this set of techniques is extremely important as well as difficult and challenging to deploy. Perhaps, also in this case, the use of a trusted token, such as a smart card, might be an option to avoid the use of white-box cryptography, or at least to have an assistant for performing transformations on input/output data.

7.4 Key Points

At the beginning of the chapter, three research questions were presented. The goal of this section is to provide clear and explicit answers with the purpose of outlining and emphasizing the chapter's key points.

What are the business requirements of our stakeholders?
The stakeholders' business requirements are presented and discussed in List 1 and Table VI. Our main purpose is to design a system from which all stakeholders can benefit.

What is the impact of their requirements on the software design?
The impact of each stakeholder's business requirement on the software architecture is described in Section 7.2. The *augmented bytecode verifier* architecture is tremendously influenced by these requirements, in particular from Requirement 1 that requires our software to be run in a *white-box context*.

Why do protection techniques need to be integrated? And what security mechanisms should be deployed?
An *augmented bytecode verifier* instance performs checks on applets' code. In case of success, these applets are signed by the verifier instance and afterwards are distributed and installed on secure elements. The sensitivity of the operations a verifier has to perform is unquestionable. In case the software run on a trusted host, protection techniques would not be required. Unfortunately, Requirement 1 explicitly asks to allow a verifier instance to be run on a totally untrusted machine, i.e. in a *white-box context*. Thus, the use of protection techniques is crucial to preserve the correctness of the verifier operations and minimize the likelihood of an attacker being able to break the system. As explained in Section 7.3, a malicious agent in a *white-box context* could perform attacks in any software state: on disk, in memory or during execution. Therefore, effective protection techniques to secure each state have to be deployed. Namely, *data encryption*, *runtime integrity checks*, *code obfuscation and code flattening* and *white-box cryptography*.

Chapter 8

A Scenario Analysis

Contents

8.1 Scenario Definition 80
8.1.1 A Scenario Story 80
8.1.2 Scenario Properties 81
8.1.3 Stakeholder's Phases 81
8.2 Card Issuers 82
8.2.1 Activation 82
8.2.2 Usage 83
8.2.3 Distribution 85
8.3 Application Developers 87
8.3.1 Activation 87
8.3.2 Usage 88
8.3.3 Distribution 89
8.4 Key Points 92

Prior to introducing the technical specifications of the verifier, a scenario needs to be detailed in order to concretely define how the involved parties might act in the process of applet verification, respecting the business requirements described in Section 7.1. The goal of this chapter is to present a new scenario where all involved parties are considered and benefit by the usage of this new product. More in detail, with this chapter we want to answer the following questions:

- *How stakeholders do interact to meet their own business requirements and create a trusted environment that is flexible and scalable?*
- *What are the properties the scenario should guarantee?*
- *How the applet verification process can be indirectly monitored by NXP Semiconductors without impacting the flexibility of the process?*

Section 8.1 introduces the general aspects of the scenario pointing out its goal and phases, while Section 8.2 and 8.3 explains how the scenario changes in the eyes of stakeholders. Finally, Section 8.4 briefly summarizes each answer of the above-presented research questions to explicitly highlight the key points of the chapter.

8.1 Scenario Definition

Ideally, as described in Chapter 6, a secure element should be simply considered an extension of the mobile device's operating system, which provides a higher level of security to applications that need to store and use sensitive user data. The main goal the scenario needs to meet is to provide high flexibility in term of initialization and management of secure elements enforcing a high level of security and trust. The *augmented bytecode verifier* proposed aims at turning this ideal situation into reality. Chapter 7 introduced the three stakeholders that are part of the system along with their business requirements that, as explained, influence tremendously the design of the system. The verifier must run in a white-box scenario, which is the most extreme case to enforce high security. Failures in a process where secure elements can be accessed post issuance might lead to huge damages in term of money and reputation to both *Card Issuers* and *NXP Semiconductors*, thus they must be prevented or, at least, limited and managed. This context opens the doors for the definition of a new scenario where stakeholders interact and cooperate in order to:

1. Meet their own business requirements.
2. Create a trusted environment, i.e. a successful trade-off between transparency and security.
3. Make the distributed software architecture flexible and scalable.

The next sections define several aspects of the scenario, introducing in Section 8.1.1 a short scenario story on how the system works, in Section 8.1.2 the properties the scenario must guarantee and in Section 8.1.3 the scenario phases that must be defined.

8.1.1 A Scenario Story

A Card Issuer **C** *wants to take advantage of the service, offered by* **NXP Semiconductors**, *that allows to manage in a flexible manner secure elements both during the initialization and post issuance phase.* **NXP Semiconductors** *equips* **C** *with a verifier to able it to manage its own content on mobile devices' secure elements where it has its own security domains.* **C** *develops, verifies and uploads onto its security domains some libraries needed for the release of a proximity payment service intended to be provided in the future.* **C** *decides to outsource the actual implementation of the proximity payment applet to an Application Developer* **D**. **C** *asks* **NXP Semiconductors** *to provide* **D** *with an augmented bytecode verifier to enforce the correctness of the implemented applet prior to its distribution to end users.* **D** *implements the requested applet and verifies it. Through an agreed channel between* **C** *and* **D** *the verified applet is sent from* **D** *to* **C**. *The applet is made available for download from* **C**. *An end user, who owns a mobile device equipped with one of those secure elements, wants to make use of the payment system offered by* **C**. *He accesses the applet store and downloads the needed applet, which is then ready to be used.*

The above story presents the ideal case where maximal flexibility is offered. It simply requires *NXP Semiconductors* to initialize a set of secure elements with several security domains upon agreement with a group of *Card Issuers*. Secure elements are then embedded into mobile devices that will be bought by final users. *Card Issuer*, equipped with a verifier, can manage with maximal flexibility their own security domains even outsourcing the implementation of applets to third parties, the *Application Developers*, on which they have complete control.

8.1.2 Scenario Properties

Flexibility and *Security* are the main properties that the scenario needs to offer. Focusing on both the business requirements presented in Section 7.1 and the two properties the system must provide we decided to centralize security into *NXP Semiconductors* hands, whereas distribution is in the hands of the *Card Issuers*. This design choice was taken in the definition of the scenario because *NXP Semiconductors* is strongly interested in inverting the current trend presented in Chapter 6 without exposing itself to failures that might impact its reputation and financial aspects. Whereas, *Card Issuers* are strongly interested in managing their own content in a flexible and secure manner without caring how these properties might be achieved. However, the system needs to instill trust proving its own security and transparency. In the definition of the software infrastructure, this design choice implies that *NXP Semiconductors* is considered the only trusted entity in the system, i.e. there is no need to put trust into *Card Issuer* or *Application Developers* during the process of verification. The verifier and *NXP Semiconductors* backend are the cornerstone of security.

More in detail the scenario must guarantee that:

- the verification process is successfully completed
- CAP and export files preserve their integrity after verification
- stakeholders are always authenticated prior to any action to avoid unwanted behaviours
- revocation can be always applied to avoid relevant failures
- the augmented bytecode verifier is strongly resilient to white-box scenario attacks

8.1.3 Stakeholder's Phases

Stakeholders have different requirements, different roles, different resources and are numerically distant. Therefore, they must be treated separately to be much more effective in designing the system architecture. Referring to the scenario story presented in Section 8.1.1, three main phases can be deduced from a secure element issuance to the distribution of a new brand verified applet, namely:

- *Activation*, which refers to the moment where a new stakeholder registers into the service.
- *Usage*, which refers to the phase where content, intended to be uploaded onto one or more secure elements, is verified.
- *Distribution*, which refers to the step of uploading a verified content onto one or more secure elements as needed.

These three phases differ depending on which stakeholder, *Card Issuer* or *Application Developer*, is involved. The first two phases are interconnected and strongly relate to security. They must be designed together to develop a consistent and robust system. The third phase is related to the distribution of already verified content, thus it becomes more a business problem instead of technical. For this reason, it is only mentioned along the report for the sake of clarity.

The following sections define an actor-based scenario to illustrate how the scenario differs in the eye of the two stakeholders and to introduce some more technical details.

8.2 Card Issuers

The role of *Card Issuer*, described in Section 7.1, deals with the management of secure elements content and with the administration of *Application Developers* that can, under well-defined rights, develop further content on its behalf. Upon agreement, a secure element can be released by *NXP Semiconductors* with several *Security Domains* (refer to Section 5.3.1 for further details), one for each *Card Issuer* in the agreement. Technically, this means allowing *Card Issuers* to manage securely their own content. However, in case one of these *Card Issuers* uploaded malicious content, the integrity of the entire Java Card secure element, as extensively described in Chapter 5, would be endangered. This behaviour is considered to be highly likely because, in the worst case, contents of competitors, even sensitive, might be uploaded. To counteract these possible malicious behaviours and enforce trust, also *Card Issuers* must verify any content intended to be uploaded. The next sections describes the three above-mentioned phases for *Card Issuers*, one at a time. In conclusion, a complete message sequence diagram is presented to provide a more visual and intuitive explanation of all relevant steps performed along the three phases.

8.2.1 Activation

Use Case* 8.1*: *Card Issuers* want to be part of the architecture and make their request to *NXP Semiconductors*. Upon agreement, *NXP Semiconductors* initializes a set of Java Card secure elements with several security domains, one for each *Card Issuer* and generate public keys pairs for authentication purpose. The related public keys are used for the release of *Card Issuers*' certificates, whose issuance body is *NXP Semiconductors*, while private keys are stored into smartcard tokens (Java Card token). *NXP Semiconductors*

delivers to each *Card Issuer* an initialized augmented bytecode verifier, a Java Card token with its private key and its certificate embedded. Each pair *verifier license - certificate* is stored in a *NXP Semiconductors*' database for signatures verification and control purposes such as revocation.

During this phase the system is initialized and prepared to provide successful verification of contents and to manage properly *NXP Semiconductors*' secure elements. This phase is crucial. In case data was leaked during this phase, the entire process would be endangered. In the worst case – the attacker gains a *Card Issuer*'s private key – a malicious agent would be able to sign applets on behalf of a *Card Issuer* without verifying its code and upload them onto secure elements. The entire system would be broken and performing revocation might be not possible unless *NXP Semiconductors* equips secure elements with a system to perform remotely security domain management. Note that an attacker could make the attack faster and more dangerous by making other developers implement applets and then sign them; in this way the attack might have more impact and be more difficult to prevent even with the required security mechanisms in place. Thus, the security domain set up along with the key generation must be performed under strict confidentiality. Furthermore, channels for delivering verifiers and secure tokens must be carefully chosen, depending also on the already existing business channels of *NXP Semiconductors*.

8.2.2 Usage

***Use Case* 8.2:** *Card Issuers* verify libraries or applets that are intended to be uploaded onto secure elements. More information on the verification process can be found in Chapter 6. Figure XIV illustrates the process along with the system infrastructure required; numbers in the text refer to that figure. A *CAP* file along with its *export file* is passed to the verifier (1). Firstly, the validity of the verifier's license is checked online by *NXP Semiconductors*'s servers and all *signed export files* of the already installed applets and libraries are retrieved from a centralized database (2). Secondly, the specialized modules aimed at code verification are run (3). Thirdly, in case of successful verification, a report on the final process outcome is automatically generated, signed using the token given during the *activation* phase (further details in Chapter 9) and sent to the *NXP Semiconductors* servers (4). It is then processed to check for unwanted behaviours. Fourthly, both the *export* files and the *CAP* file are signed by the system using the token. The former is sent to *NXP Seminconductors* and stored in a centralized database (5), while the latter is returned as output (6) and stored in a *Card Issuer* database (7).

During this phase *Card Issuers* are considered untrusted entities because of the very dangerous behaviours they could adopt, as previously explained. In this phase, the cornerstone of security is the augmented bytecode verifier that must enforce specific protection techniques, as already discussed in Chapter 7, and the Java Card token that must preserve the confidentiality of the private key of the *Card Issuer*. In other words, the *Card Issuer* is given the token for signing, but it does not know the key itself. As will be explained

in Chapter 9, the process of signing is performed on a Java Card token after a successful verification. This means that the action of signing is bound to the process of verification: a *Card Issuer* cannot sign an applet without the use of a verifier. A *Card Issuer*'s private key never leaves the token and cannot be accessed digitally because no API is provided. Thus, the confidentiality of the *Card Issuer*'s private key relies on the security properties of Java cards, both logical and physical. More information about the security mechanisms provided by Java Card technology are discussed in Chapter 3.

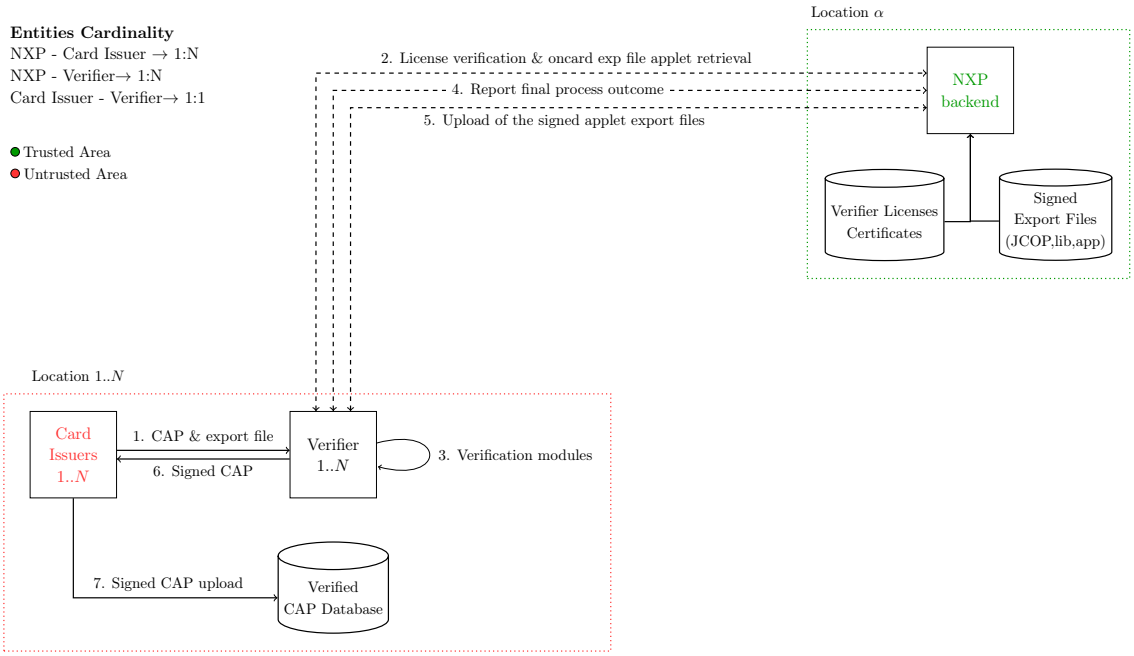

Figure XIV: Card Issuer Verifier Usage

Note that, in order to counteract Side Channel attacks, which could be performed against the token for extracting a *Card Issuer*'s key, tailored countermeasures should be deployed, preferably at application level (further information in Section 4.3). On one hand, *signed CAP files* are stored at the *Card Issuer* location with the purpose of enforcing requirement 4. In this way, once a *CAP file* is verified and signed, the *Card Issuer* has also complete power on its applets or libraries and can decide as needed when to upload them, make them available for download etc. On the other hand, *export files* are uploaded on a centralized server because they are needed for the process of verification and must always be available. However, a *Card Issuer* might have the need to keep an *export file* confidential because it is used only for testing or internal purposes and will not be distributed later. Thus, the verifier should allow to check an applet without signing it and forcing to upload the related *export file* on a centralize *NXP Semiconductors* server. Through Communication *4* in Figure XIV, *NXP Semiconductors* can detect unwanted behaviours during the entire process and apply revocation, blocking *Card Issuers*, even if they are still in possession of the signing token thanks to the forced bind between the

verifier and the signing token. This gives complete control in terms of security to *NXP Semiconductors* without affecting flexibility and scalability.

8.2.3 Distribution

***Use Case* 8.3:** A *Card Issuer* wants to install libraries or applets onto the final customer secure elements. A *signed CAP file* is sent from a *Card Issuer*, along with its certificate, to a set of secure elements' loading systems through the Internet. A loading system takes care of *Card Issuer* authentication and *CAP file* integrity verification. In case of a successful check, the loading system allows the applet installation in the related *Card Issuer* security domain.

The loading service is a Java Card applet developed by *NXP Semiconductors* to manage remotely a Java Card secure element content. It requires a certificate to authenticate a *Card Issuer* and a *signed CAP file* to check the integrity of an applet file prior to its installation. The service provided by this applet can be considered a further security layer over the *Security Domain* mechanism of a traditional Java Card that complies to the *Global Platform Standards*. Typically, access to a security domain is performed using a symmetric crypto scheme, thus, there is no real authentication but only the assumption that a file, encrypted with the right key, has been enciphered by the real owner of the key. Using this additional service, the security level highly increases and remote management of secure element contents is allowed. Note that, the loading service takes care of the communication with a *Card Issuer security domain* after a successful authentication.

Figure XVI shows a message sequence diagram to present at once the three phases above-described, making them more visual. Colors in the figure are relevant, *green* means trusted whereas *red* means untrusted. Thus, in the diagram, *NXP* is considered trusted all along with all its computation performed, whereas the *Card Issuer* is untrusted for the reasons already explained in Section 8.2. The verifier is *white* to emphasize the fact it is in a white-box scenario and its security cannot be totally guaranteed. The end-user (mobile device) is *gray* since he is not considered a stakeholder and does not influence the process of verification. Figure XV shows a complete legend for the sequence diagram.

Legend:

Req, PKI gen() → Request, Public Key Pairs Generation

Pub,Priv,Cert → Public Key, Private Key, Certificate

Ver, Lic, Token → Verifier, Verifier License, Java Card Token

Lic Lookup(), Run Modules() → Search of the license into the database, Run verification modules

Analyse Report() → Analysis of the generated report for unwanted behaviour management

CAP,exp → CAP executable and related exp file

Figure XV: Legend sequence message diagram Card Issuer

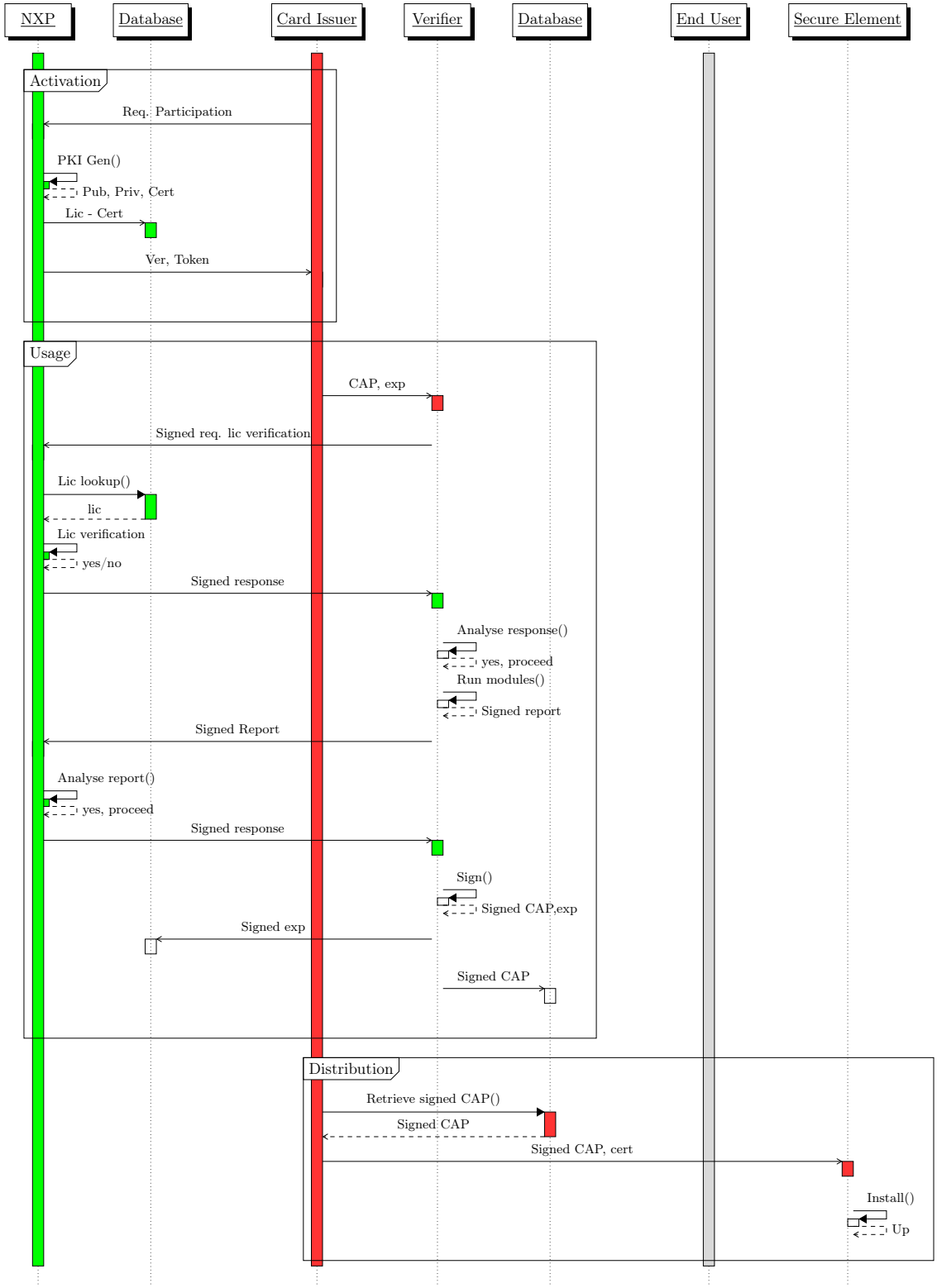

Figure XVI: Message Sequence Card Issuer

8.3 Application Developers

The role of an *Application Developer* is to implement applets on behalf of *Card Issuers*. A *Card Issuer* has complete control over its developers, in other words, it can define its own policy about their eligibility. *Application Developers* might be simply considered an extension of *Card Issuers*, whose purpose is to outsource the implementation of applets for final customers and make the environment much more open, comparable to the current mobile environments, for example Google Play, but far more secure. Similarly to *Card Issuers*, *Application Developers* might be inclined to upload malicious applets for many reasons, due to the high value of the asset they are attacking. Therefore, any applet implemented by an *Application Developer* must be verified prior to its distribution. The next sections describes the three system phases in the eye of *Application Developers*, one at a time. In conclusion, a complete message sequence diagram is presented to provide a more visual and intuitive explanation of all relevant steps performed along the three phases.

8.3.1 Activation

Use Case* 8.4*: A *Card Issuer* chooses one or more *Application Developers* and forwards a request to *NXP Semiconductors* for initializing the needed number of augmented bytecode verifiers and tokens. *NXP Semiconductors* generates public key pairs, one for each *Application Developer*, for authentication purposes. The related public keys are used for the release of certificates, while private keys are stored in Java Card tokens. The set of certificates is distributed to the *Card Issuer* along with the requested tokens and verifiers, which, in turn, are given to the chosen *Application Developers*. *NXP Semiconductors* stores in its database the pair *verifier license - certificate* for control purpose such as revocation, while the *Card Issuer* stores in its database the set of its *Application Developers*'s certificates for authentication and management.

During this phase any kind of data to perform applet verification successfully must be initialized. The sensitivity of this data is low compared to the *Card Issuer activation phase* because it is more under control, but still dangerous. In the worst case – an attacker gains an *Application Developers* private key – a malicious agent could sign applets on behalf of an *Application Developer*, but in this scenario a *Card Issuer*, as explained in the next phase, might deny the request of upload for distribution. This might preserve the integrity of the system, but for *NXP Semiconductors* would not be enough anyway, because the control on the security of the process would be lost and left in the hand of a *Card Issuer*, since the process of verification would be skipped. As explained in Section 8.1.3 the security of the system is controlled by *NXP Semiconductors* that does not want to put trust in other stakeholders. Thus, also in this activation phase, the key generation must be performed under strict confidentiality. Furthermore, channels for delivering verifiers and secure tokens must be carefully chosen depending also on the already existing business channel of NXP Semiconductors.

8.3.2 Usage

***Use Case* 8.5:** *Application Developers* verify any applets intended to be distributed to a final customer. More information on the verification process can be found in Chapter 6. Figure XVII illustrates the process along with the system infrastructure required; numbers in the text refers to that figure. An *Application Developer* inputs a *CAP file* with its *export file* into the verifier (1). Firstly, the validity of the verifier's license is checked online by *NXP Semiconductors*' servers and all *signed export files* of the already installed applets and libraries are retrieved from a centralized database (2). Secondly, the specialized modules aimed at verifying applet code are run (3). Thirdly, in case of successful verification, the respective *Card Issuer* receives a cryptographically signed request of confirmation, verifiable through certificates (4). In case the *Card Issuer* grants the verification process to proceed, a report on the final process outcome is automatically generated, signed using the token given during the activation phase (further details in Chapter 9) and sent to the *NXP Semiconductors* servers (5). It is then processed to check for unwanted behaviours. Fourthly, both the *CAP file* and the *export file* are signed by the system using the token. The *signed export file* is then sent to *NXP Semiconductors* (6). The *signed CAP file* is returned as output (7) and successively given to the *Card Issuer* using a channel agreed on beforehand between a *Card Issuer* and its *Application Developers* (8). In conclusion, the *Card Issuer* signs the already *signed CAP file* of the *Application Developer* and uploads it onto its *signed CAP file database* to make it available for distribution (9).

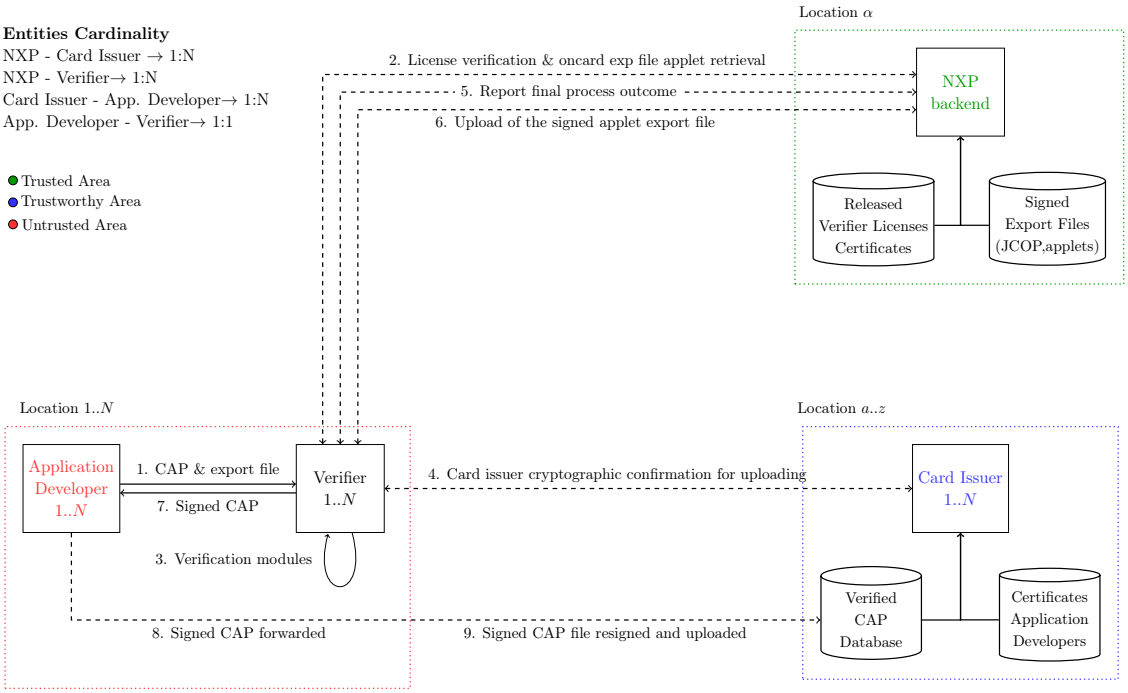

Figure XVII: Application Developer Verifier Usage

During this phase both stakeholders, *Application Developers* and *Card Issuers*, are involved. The former is considered an *untrusted* entity because of the malicious behaviours it might assume, while the latter is *trustworthy*[1] because its failure would not impact the system security. This is a major advantage for *NXP Semiconductors* since it allows to fully respect Requirement 3, permitting *Card Issuers* to control any verification process of its *Application Developers* without affecting the security of the system, which in this way is fully under control. In terms of security features this phase is equal to the *Card Issuer Usage* phase. A verifier must be resilient in a *white-box scenario* and a token must preserve the confidentiality of an *Application Developer*'s private key. The *Application Developer* is given the token for signing without knowing the key value. In this way, the process of signing is bound with the process of verification: only after a successful verification phase the token can be used for signing. The *signed CAP file* is then returned to the *Application Developer* that, in turn, forwards it to its *Card Issuer* using a channel agreed upon between the two entities. If successfully, the applet must be signed in-house by the *Card Issuer* prior to be uploaded onto its database for distribution. Regarding *export files*, they are uploaded on a centralized database at *NXP Semiconductors* location because they are needed for enforcing the process of verification. In case of detected unwanted behaviours during the whole process, revocation might be applied to *Application Developers* even if they are still in possession of the signing token. This gives complete control in terms of security to *NXP Semiconductors* making itself the only *trusted* entity in the system without affecting flexibility and scalability. Communication *5* in Figure XVII is crucial to guarantee this system feature.

8.3.3 Distribution

Use Case* 8.6*: An end user wants to install an applet onto his/her mobile device's secure element to take advantage of an offered service. The applet market is accessed and the chosen *signed CAP file* along with its related *Card Issuer*'s certificate is downloaded. The secure element loading service takes care of the communication with the mobile device operating system and of the *Card Issuer*'s authentication and applet integrity check. The *signed CAP file* is then installed onto the corresponding *Card Issuer* security domain and made available for usage.

Referring to Section 8.1.1, an end user is a customer who makes use of a verified applet. He is not considered a stakeholder in the system because he does not act in the process of applet verification. In terms of distribution, the ideal situation, as mentioned multiple times, should allow a user to download applets as needed. An approach might be the use of a mobile application that allows to access an applet market, which consists of all *Card Issuers* databases where *signed CAP files* are stored.

[1] A "trustworthy" system or component is one whose failure will not break security

Figure XIX and XX illustrate a message sequence diagram to present at once the three phases above-described, making them more visual. The former shows the *Distribution* phase, whereas the latter the *Activation* and *Usage* phase. Colors in the figure are relevant, *green* means trusted, *blue* trustworthy, whereas *red* untrusted. Thus, in the diagram, *NXP* is considered trusted along with all its computation performed, whereas the *Card Issuer* trustworthy and the *Application Developer* untrusted for the already explained reasons in Section 8.3. The verifier is *white* to emphasize the fact it is in a white-box scenario and its security cannot be totally guaranteed. The end-user (mobile device) is *gray* since he is not considered a stakeholder and does not influence the process of verification. Figure XVIII shows a complete legend for the two sequence diagrams.

Legend:

CI → Card Issuer

AD → Application Developer

Req → Request

PKI gen() → Public Key Pairs Generation

Pub,Priv,Cert → Public Key, Private Key, Certificate

Lic → Verifier License

Ver → Verifier

Token → Java Card Token

Lic lookup() → Search of the license into the database

Analyse Report() → Analysis of the generated report for unwanted behaviour management

CAP,exp → CAP executable and related exp file

Run modules() → Run verification modules

Signed report CI → Cryptographic request of confirmation with process-related data

Applet lookup() → Search of the needed applet into the database applet storage possibly through a market

Applet download → Request of download of the chosen applet onto the mobile device

Install() → Installation of the applet onto the secure element

Figure XVIII: Legend Message Sequence Diagrams of Application Developers

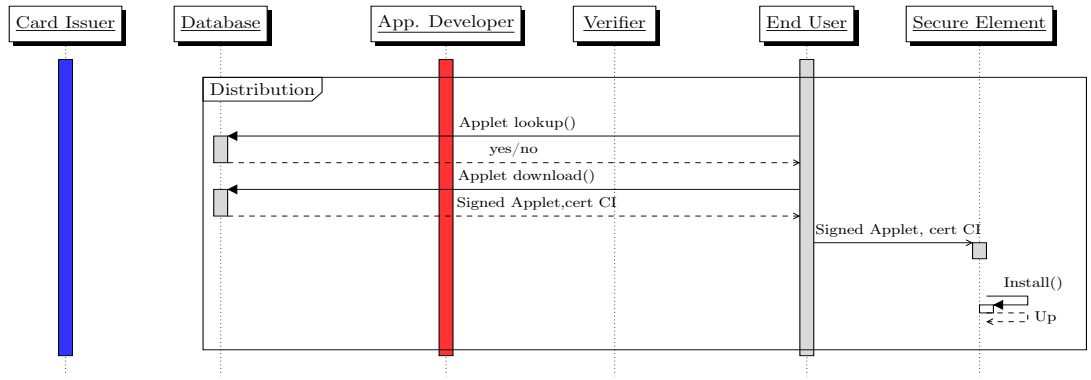

Figure XIX: Message Sequence Application Developer Distribution

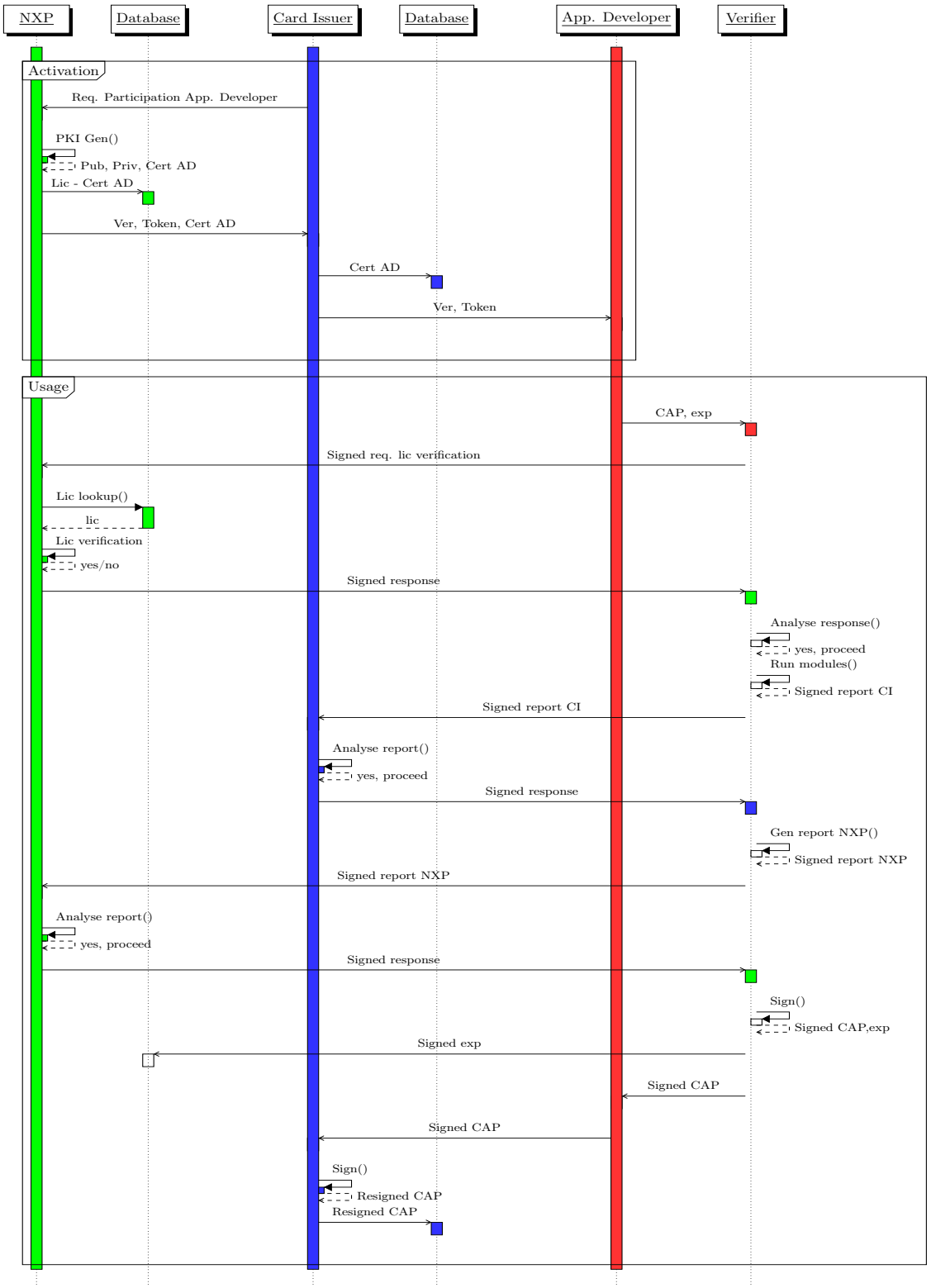

Figure XX: Message Sequence Application Developer

8.4 Key Points

At the beginning of the chapter, three research questions were presented. The goal of this section is to provide clear and explicit answers with the purpose of outlining and emphasizing the chapter's key points.

How do stakeholders interact to meet their own business requirements and create a trusted environment that is flexible and scalable?
Each stakeholder plays an active role in the verification process, but with different purposes. Focusing on the business requirements presented in Chapter 7 we identified each stakeholder's main goal and designed the system assuring those goals were met. *NXP Semiconductors* wants to enforce security onto its secure elements, *Card Issuers* want to freely control the content of secure elements where they have a security domain and to manage without limitation its *Application Developers*. *Application Developers*, which are an extension of a *Card Issuer*, ask for intellectual property protection. These requirements implicitly require flexibility and scalability. Section 8.2 and 8.3 present thoroughly stakeholders' interactions.

What are the properties the scenario should guarantee?
The scenario's properties are presented in Section 8.1.2. In a nutshell, the scenario should guarantee that:

- the verification process is successfully completed
- CAP and export files preserve their integrity after verification
- stakeholders are always authenticated prior to any action to avoid unwanted behaviours
- revocation can be always applied to avoid relevant failures
- the augmented bytecode verifier is strongly resilient to white-box scenario attacks

How can the applet verification process be indirectly monitored by NXP Semiconductors without impacting the flexibility of the process?
The designed system enables *NXP Semiconductors* to control indirectly each verification process through communication *4* in Figure XIV and communication *5* in Figure XVII. An augmented verifier has enough trusted data, more information in Chapter 9, to generate and transmit a signed report that *NXP Semiconductors* can analyze and interpret to detect unwanted behaviour or simply monitor the outcome of a verification process. In addition, *NXP Semiconductors* plays an active role along the entire process of verification by means of its back-end (see Section 9.2.1). The flexibility of the system is indirectly preserved through the distributed architecture that allows all stakeholders to still meet their own business requirements.

Chapter 9

Technical Specification and Design

Contents

9.1 Design Choices 94
9.2 The System Architecture 96
9.2.1 The Protocol Concept 98
9.2.2 Runtime Integrity Checks 105
9.2.3 Code Obfuscation and Code Flattening 107
9.2.4 White-box Crypto 108
9.2.5 Binary Encryption 108
9.3 Enforcing Edge Case Features 109
9.3.1 Key Renewability 109
9.3.2 License Revocation 109
9.3.3 Verifier Binary Diversity 109
9.4 Implementation Directions 110
9.5 Key Points 111

This chapter focuses on the technical design of the augmented bytecode verifier in terms of security. More in detail, with this chapter, we want to answer to the following questions:

- *How can the system be designed to withstand to white-box attacks by design?*
- *How NXP Semiconductors can be self-confident that the applet verification process ended successfully?*

Section 9.1 presents some design choices taken to maximize the system's security and flexibility. Section 9.2 introduces the working principle of the augmented bytecode verifier describing the protection techniques deployed. Section 9.3 describes three crucial features that the verifier should provide by design to deal with edge cases. Section 9.4 discusses a possible implementation direction. Finally, Section 9.5 briefly summarizes each answer of the above-presented research questions to explicitly highlight the key points of the chapter.

9.1 Design Choices

In designing the verification system some relevant, but needed, design choices were taken to maximize the system flexibility and security. Hereafter, an explanation of each of them.

Figure XVII in Chapter 8 illustrates the usage scenario for an *Application Developer*. In particular, there are two moments, step 4 and 5, that require a *Card Issuer* and *NXP Semiconductors* to analyse data sent over the internet from the augmented bytecode verifier to grant the process to proceed. The first design choice relates to these two moments, which might be structured in three different ways:

- The interactions are automatic, therefore a back-end, both at *Card Issuer* and *NXP Semiconductors* location, is in charge to analyse the received data and decide to whether to grant the process to proceed.
- The interactions require a manual review and only after that, an answer is sent to the verifier.
- The interactions are partially manual and automatic.

From a business requirement point of view, *Card Issuers* are interested in managing autonomously and with a good extent of flexibility their own *Application Providers* and also want to control the process of verification (Business requirement 3). If the interaction at *Card Issuer* side was performed automatically it would mean that *NXP Semiconductors* should provide each *Card Issuer* with a system extension that is able to correctly parse the data produced by the verifier and take a decision based on their own policies. Ideally, this would be the best case, but not strictly relevant in this product phase. At the beginning it is fundamental to instill trust in *Card Issuers* and emphasize flexibility, therefore we decided to leave the interaction manual, so that *Card Issuers* can take their own decisions without putting trust in other parts of the system. Technically, leaving the interaction manual means that the process of reviewing the data received takes much more time, therefore an augmented bytecode verifier, running at an *Application Developer*'s location, must preserve and secure its own state to avoid that an attacker is able to skip steps required by the verification process. At *NXP Semiconductors*' side, we decided that the process of reviewing should be done automatically as, the format of the received data is known and the details to be checked can be easily extended and modified overtime.

In Chapter 8, the use of a Java Card token is mentioned multiple time, but no explanation about its application is provided. The motivation behind its adoption is simple: enforcing high security in a *white-box scenario*, which is the context where the verifier is running, is considered not feasible and the use of a token gives us a *trusted* area to make computations that need to be done in a safe environment. The computation of the signature on applets to be verified, as already mentioned in the previous chapter, is performed onto the token, as it is the most sensitive operation of the entire process. However, the use of a token alone cannot permit the system to meet the level of security required, as there is always a moment in which the verifier itself has to compute data that the token must

trust for taking the final decision of signing the submitted applet. Thus, we decided to involve actively, during the process of verification, *NXP Semiconductors*' back-end, with the purpose of creating a secure tunnel between the two trusted components. This approach permits to monitor effectively the untrusted verifier instance while performing the checks defined (refer to Chapter 6 for further details about tests performed on executable applets), without the need of putting trust in unverifiable data. In fact, the use of a secure tunnel between *NXP Semiconductors* and the token enforces data source and data integrity.

Another solution, which would simplify the software complexity, is to substitute the use of a token with a tamper resistance hardware on which the verifier runs. This approach would permit to move from a *White-box* scenario to a *Gray-box* context, but the cost for *NXP Semiconductors* would be much higher an amount that it is not willing to spend. Otherwise, the whole system could be offered as a service in the cloud making verification happen in an almost safe environment. However, using this approach some stakeholder's business requirements would not be respected and flexibility not completely maximized.

Based on the introduced design choices, it is crucial:

1. to create a secure tunnel between *NXP Semiconductors*' back-end and the *Java Card token* with the purpose of monitoring the augmented bytecode verifier running on an hostile machine.
2. to make the augmented bytecode verifier enforce, preserve and secure its states along a session to avoid an attacker to skip key stages fundamental for the success of the process.

To facilitate and make the system design more effective, we decided to implement the verifier and the token as *deterministic automata*. Figure XXI and XXII illustrate the states of both the token and the augmented bytecode verifier.

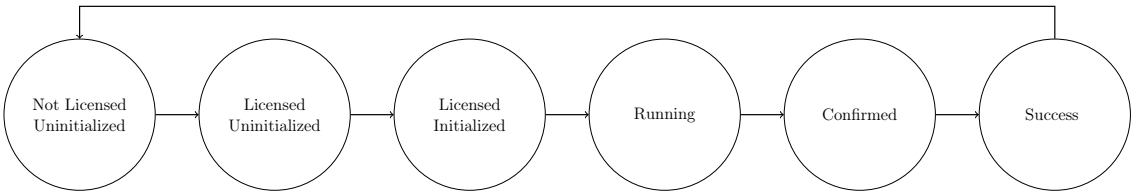

Figure XXI: The token states

Both automata are extremely straightforward, but enough for our purposes since what we need is simply a method to enforce a sequential and predefined order of operations. This might be achieved also without the use of states, but with this approach it is much easier to keep the parties of the system synchronized and aware of each other's internal state. Thus, errors and misbehaviours can be detected more effectively and strong cooperation between system's entities can be achieved by design. States change only in according with specific communications between *NXP Semiconductors* back-end, the *verifier* and

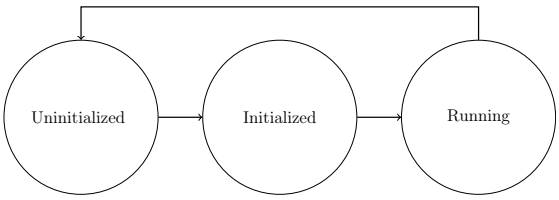

Figure XXII: The verifier states

its related *token*. Further details are presented in Section 9.2, which describes the entire protocol concept.

Many researches such as [Howard 2005] and [Manadhata 2011] pointed out that security cannot be guaranteed, but can be measured in terms of attack surface. The larger the surface, the more insecure the system. Therefore, in designing the verifier architecture, we reasoned also in terms of attack surface. Any kind of computation performed by the verifier cannot be trusted and any data given to the verifier has to be assumed leaked, as given to an attacker. Thus, to improve the global security of the process, we decided to reduce the attack window of a malicious agent giving to an augmented bytecode verifier only:

- partial information
- information strongly bound to a session
- a time limit within which the computations must be performed

In this way we increase security by design, increasing the software complexity and reducing the number of attackers with enough skills to break into the system.

The next sections complete the picture. More precisely, Section 9.2 describes the working principle of the augmented bytecode verifier. Section 9.3 explains how the software by design can handle edge cases. In conclusion, Section 9.4 suggests programming languages and technologies that might be used for its implementation.

9.2 The System Architecture

In our new scenario, introduced in Chapter 8, three phases are required by the applet verification process, namely *Activation*, *Usage*, *Distribution*. Together they cover the whole verification process, from the moment a stakeholder subscribes to the service to the stage of applets' distribution. The *Activation* phase is straightforward and does not require further technical explanation. The *Distribution* phase, as previously explained, is only mentioned for the sake of completeness and is considered out-of-scope. The phase of interest, from a security point of view, is undoubtedly the *Usage* phase, which is so far only described at a high-level. Thus, the goal of this section is to explain technically how the process of verification works from a verifier perspective. In presenting the technical

details we refer to the *Usage* phase of *Application Developers*, illustrated at a high level in Figure XVII Chapter 8, as that phase is the most complex and covers also the *Usage* phase of *Card Issuers* in term of functionalities.

Figure XXIII presents the process of verification from an augmented bytecode verifier perspective. As illustrated, the verifier, running onto an hostile machine, consists of two main parts: *the set of modules* and *the main execution*. The former refers to the collection of implemented checks that have to be run on an applet before being signed. Each module has to be considered as an independent piece of software, which is run as needed. The main execution process coordinates the entire process of verification. We decided to separate the set of modules from the main execution to find a balance between security, transparency and software flexibility/scalability. In fact, as discussed in Chapter 7, Requirement 5 asks for transparency whereas Requirement 1 for high security. The decided division permits to meet a feasible trade-off and increases also the software flexibility and scalability as, in this way, it is far easier to add new modules and to release new software versions.

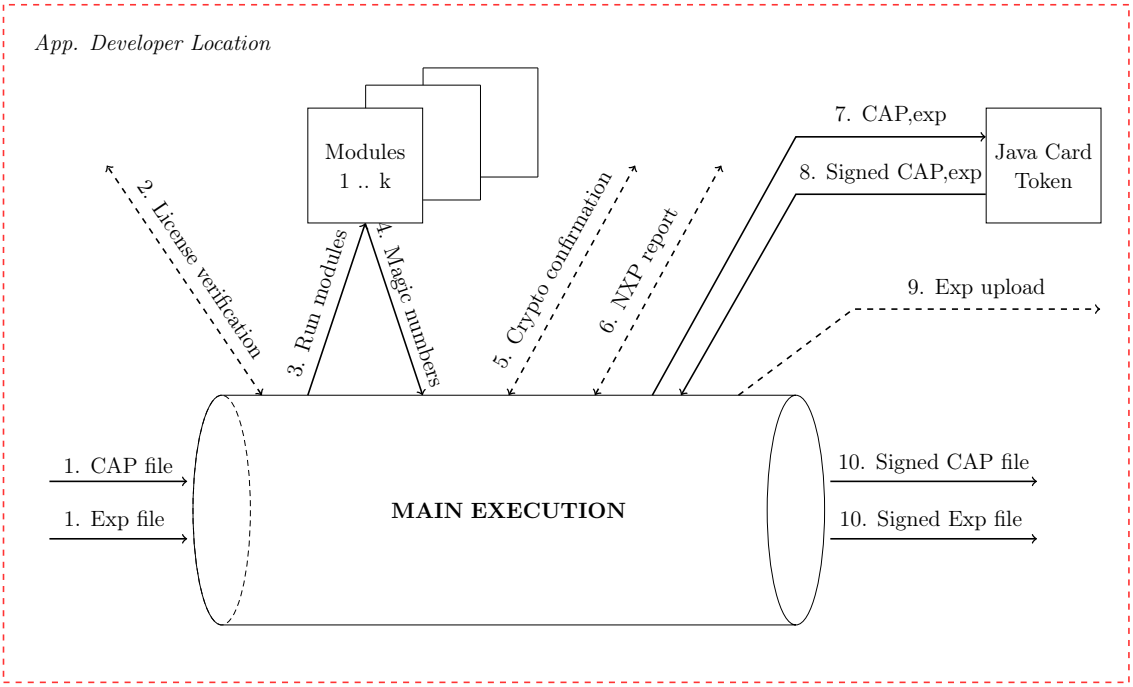

Figure XXIII: The *Usage* phase from a verifier perspective

The *verifier*, as already explained in the previous section, collaborates with a *Java Card token* and *NXP Semiconductors* back-end to enforce high security. Below, a list of *assumptions* under which, in the next section, we describe the operating principle of the system.

- The *Activation* phase of the *Application Developer* was successful, allowing certified communications between all entities involved in the process. In particular:

  - *NXP Semiconductors* has a tuple *Verifier licence - Certificate* for each stakeholder stored into its database.
  - The *Card Issuer* has the certificate of the *Application Developer*.
  - The *Application Developer* was given the augmented bytecode verifier along with its related Java Card token, which contains in terms of data the *Application Developer private key* and the *related certificate*.
- The *Application Developer* has successfully installed the verifier and a card reader is available on the machine.
- The *Application Developer* and *Card Issuer* agreed on a channel for the delivery of verified applets.
- The *Card Issuer* has initialized a database where all verified applets will be kept and a simple back-end where request of confirmations, to grant the process to proceed, are delivered.
- The *NXP Semiconductors* back-end is available and reachable from the *Application Developer* host where the *verifier application* is running.

9.2.1 The Protocol Concept

The process of verification starts at the moment an *Application Developer* wants to upload an applet for distribution. Firstly, a *CAP* file to be signed along with its *export file* is taken as input by the verifier's *main execution* that computes a hash on both. The hash, which we call *Ah* is then forwarded to the token, which takes control.

The *token* prepares a request of licence verification[1] to be sent to *NXP* back-end (1). More in detail, a nonce N_1 and a *request of license verification* are generated and signed along with *Ah* using the *Application Developer private key*. The signed data together with the *Application Developer certificate* is sent to *NXP* back-end.

$$\mathbf{T} \mapsto \mathbf{NXP} \qquad (N_1 \parallel Request \parallel Ah)_{Privk_{appdev}} \parallel Cert_{appdev} \tag{1}$$

Note that, the instantiation of the network connection between the *verifier* and *NXP* back-end is carried out by the *main execution* and not by the *token*. This detail is very important, as it allows *NXP Semiconductors* to use cheaper tokens for the purpose without lowering the level of security (further detail in the next steps). Nonces are used to counteract replay attacks.

Once the *NXP* back-end receives the request, it is verified. In case of success, a license lookup is performed using the tuple *Verifier License - Certificate* stored during the *Activation* phase. If the license turns out to be invalid *NXP* back-end answers with a failure message, otherwise the process proceeds.

[1] The request of license verification can be considered a simple predefined string.

The *NXP* back-end responds with its own nonce N_2, the token's nonce N_1, and a session key Sk_1 encrypted with the *Application Developer public key*. The message is signed using the *NXP private key* and sent to the *main execution*, which in turn, forwards it to the *token* (2). The *NXP* back-end starts also a timer within which an answer by the token is expected. In case it is not, the session expires.

$$\mathbf{NXP} \mapsto \mathbf{T} \qquad (N_2 \parallel N_1 \parallel (Sk_1)_{Pub_{appdev}})_{Privk_{NXP}} \tag{2}$$

The main idea behind this message is to set up a new session secret between *NXP Semiconductors* and the *token*, which will be used to create a secure tunnel, allowing to monitor the *main execution* and exchange trusted messages. For this reason the instantiation of the network connection can be carried out by the *main execution* without impacting the overall security. The use of a timer permits to reduce the surface of attack, in fact it decreases the time a malicious agent has at its disposal to try to leak Sk_1. Even if it is only a session secret, sensitive information about the process are enciphered using this key. In case an attacker would be able to have it before or during the applet checks the success of the process might be endangered. The duration of the timer can be easily decided as the computational power of the token is known and an estimation on the network delay can be computed.

Once the *token* receives the response, it verifies it and changes state from *Unlicensed/Uninitialized* to *Licensed/Uninitialized*. A simple response, which consists of nonce N_2 encrypted with $Sk1$ is prepared and sent to notify the *NXP* back-end about the success of the previous step (3).

$$\mathbf{T} \mapsto \mathbf{NXP} \qquad ((N_2)_{Sk_1})_{Privk_{appdev}} \parallel Cert_{appdev} \tag{3}$$

If the *NXP* back-end receives this message within the expected timer and Sk_1 is successfully verified, the process proceeds. At this point, *NXP Semiconductors* knows that the state of the token changed and the initialization of the session is ready to be finalized. To complete the session initialization the *NXP* back-end prepares a new message to send to the token with a new session key, Sk_2, a set of lookup tables *slt* and a list of hashes, *lh* (4).

$$\mathbf{NXP} \mapsto \mathbf{T} \qquad ((Sk_2 \parallel slt \parallel lh)_{Sk_1} \parallel N_1)_{Privk_{NXP}} \tag{4}$$

The former, Sk_2 is a symmetric key generated at *NXP* back-end whose purpose is to secure the communication between the *main execution* and the *token*. In other words, this key must be considered as a further security mechanism to avoid man-in-the-middle attacks between the two entities. We want to force a malicious agent to take the most difficult path for trying to compromise the system: a malicious agent must attack the *main execution* and should not be able to place himself in front of the *token* to intercept communications, as they are all encrypted. Since Sk_2 is a symmetric key whose goal is

to enforce message confidentiality between the *main execution* and the *token*, it should be securely stored at both sides. Contrary to the *token*, the *main execution* is untrusted including its storage, thus Sk_2 cannot be simply sent and kept in clear. Our proposal is to make use of "white-box cryptography", changing the data structure of Sk_2. In a nutshell, an encryption algorithm implementation, designed as a set of lookup tables, should be embedded in the verifier binary. Some of those tables should be left empty and initialized at run time, so that the key concealed is changeable over sessions (further details in Section 9.2.4).

Referring to communication 4, the *NXP* back-end sends the session key Sk_2 and a set of lookup tables *slt* to the *token*. The communication from the *NXP* back-end to the *token* is encrypted and signed, thus the distribution of the key is assumed secure. The set of lookup tables *slt* are forwarded by the *token* to the *main execution*, which initializes the whole algorithm implementation making the channel between the *token* and the *main execution* ready to be used.

The latter, *lh*, refers to a set of hashes computed at *NXP* back-end whose purpose is to provide the *token* with enough trusted information to ensure modules integrity. A complete explanation can be found in Section 9.2.2. For now it is enough to know that each module binary is broken into two parts. The first part is delivered with the augmented bytecode verifier to the *Application Developer* while the second part is kept at the *NXP* back-end and given to the verifier only at run time as needed.

Once the *main execution* completes the encryption algorithm initialization for the session, it changes its state from *Uninitialized* to *Initialized* and notifies the *token*, which in turn controls that the lookup tables initialization was successful. In case of success, the *token* changes its state from *Licensed/Uninitialized* to *Licensed/Initialized*. The *token* notifies *NXP* back-end that the initialization of the session is completed encrypting *NXP*'s nonce N_2 with both session keys and signing it (5).

$$\mathbf{T} \mapsto \mathbf{NXP} \qquad (((N_2)_{Sk_2})_{Sk1})_{Privk_{appdev}} \parallel Cert_{appdev} \tag{5}$$

NXP back-end verifies the message and checks the correctness of keys Sk_1 and Sk_2 decrypting N_2. In case of success, *NXP* back-end notifies the *token* that it is ready to release the chunk of the first module through a signed message that consists of a predefined string, which we call RM_1, and a nonce N_3 bound to it (6). RM_1 has the purpose to inform the token about the next module that will be run on the applet being processed, so that *NXP Semiconductors* can control and force the order of modules that are run on an applet and the *token* can locally monitor the outcome of each module. We generate a new nonce N_3 for each module release to strictly bind each module request with a session secret and encrypt with Sk_1 both M_1 and N_3 to increase the protocol complexity from an attacker perspective.

$$\mathbf{NXP} \mapsto \mathbf{T} \qquad ((RM_1 \parallel N_3)_{Sk1} \parallel N_1)_{Privk_{NXP}} \tag{6}$$

The token verifies the message and decrypts N_3 and RM_1. At this point the *token* knows which module will be run and acknowledge the reception of the message encrypting N_3 with session key Sk_1 and signing the whole response (7).

$$\mathbf{T} \mapsto \mathbf{NXP} \qquad ((N_3)_{Sk1})_{Privk_{appdev}} \parallel Cert_{appdev} \tag{7}$$

NXP releases the chunk of the first module CM_1 that has to be run (8). A timer is also initialized to increase the difficulty for an attacker to tamper with the module at runtime to interfere with the process of verification. If an attacker copies on its own storage CM_1 and rebuilds the module M_1 when the verification process is already concluded he cannot make a meaningful use of it as the module is strictly bound to the session.

$$\mathbf{NXP} \mapsto \mathbf{T} \qquad ((CM_1)_{Sk1})_{Privk_{NXP}} \tag{8}$$

The *token* verifies the integrity of the sent module chunk and forwards it to the *main execution* encrypted using Sk_2 (9). The token changes its state from *Licensed/Initialized* to *Running*.

$$\mathbf{T} \mapsto \mathbf{MExc} \qquad ((M_1)_{Sk2}) \tag{9}$$

The *main execution* decrypts the module chunk, build the complete binary and runs it on the *CAP* and *exp* file. The module returns as output a *magic number*, which is used along with the module binary M_1 and the hash Ah by the *main execution* to generate a verification hash, h_1. The hash is then sent to the *token* encrypted with Sk_2 (10). At this point, the *main execution* state is changes from *Initialized* to *Running*.

$$\mathbf{MExc} \mapsto \mathbf{T} \qquad ((h_1)_{Sk2}) \tag{10}$$

The *token* knows the module that was supposed to be run, thus it is able to verify the hash using the list, lh, previously sent by *NXP Semiconductors*. The *token* informs then *NXP* back-end about the outcome of the first module run sending an encrypted and signed message that consists of a predefined string, which we call OM_1 and N_3 (11).

$$\mathbf{T} \mapsto \mathbf{NXP} \qquad ((OM_1 \parallel N_3)_{Sk1})_{Privk_{appdev}} \parallel Cert_{appdev} \tag{11}$$

NXP back-end expects this message, as said at step 8, before the expiration of the started timer otherwise the process expires. Assuming that the message is correct and received within the defined time frame, *NXP* releases the second module chunk and the process from step 6 restarts. This happens for all modules that need to be run.

Once all modules checks are completed, the *token* gathered enough trusted information about the ongoing verification process to generate a complete report. Information contained in the report has to be decided by *NXP Semiconductors*, however the token owns

enough trusted data to document and prove the outcome of the checks performed during the session. The report, which we call R_1, along with a nonce N_4 is sent to the *Card Issuer* related to the *Application Developer* who started the process. The message is encrypted using the *Card Issuer* public key.

$$\mathbf{T} \mapsto \mathbf{CI} \qquad ((R_1 \parallel N_4)_{Pub_{CI}})_{Privk_{appdev}} \parallel Cert_{appdev} \tag{12}$$

The *Card Issuer* verifies the message and manually checks the report. In this case it also makes sense to define a deadline (timer) within which the *Card Issuer* has to answer to decrease the attack surface and make the process smoother. In case the *Card Issuer* agrees on signing the applet, a positive answer, which is a predefined string that we call PA_1, along with nonce N_4 is sent to the *token* encrypted using the *Application Developer* public key (13). Since only two messages are exchanged between a *token* and a *Card Issuer*, we decided not to generate session keys and use public key encryption.

$$\mathbf{CI} \mapsto \mathbf{T} \qquad ((PA_1 \parallel N_4)_{Pub_{appdev}})_{Privk_{ci}} \parallel Cert_{CI} \tag{13}$$

The *token* verifies and decrypts the answer of the *Card Issuer*. If successful, it changes the state from *Running* to *Confirmed* and a report, which we call R_2, is generated and sent to *NXP Semiconductors* (14). An answer is expected within a defined time span.

$$\mathbf{T} \mapsto \mathbf{NXP} \qquad ((R_2)_{Sk_1} \parallel N_2)_{Privk_{appdev}} \parallel Cert_{appdev} \tag{14}$$

NXP back-end verifies, decrypts and checks automatically R_2. If it complies with *NXP Semiconductors* policies a positive answer, which is a predefined string that we call PA_2, is encrypted using Sk_1 and sent signed along with nonce N_2 (15).

$$\mathbf{NXP} \mapsto \mathbf{T} \qquad ((PA_2)_{Sk1} \parallel N_2)_{Privk_{NXP}} \tag{15}$$

The *token* verifies and decrypts PA_2 and, based on that, signs the *CAP* file along with its *exp* file or invalidates the session. In conclusion, the applet *exp* file is uploaded at *NXP* back-end and the *token* changes state from *Confirmed* to *Successful* and performs all needed operations to bring itself and the *main execution* to the initial state.

Figure XXIV and XXV illustrate the protocol concept in a whole through a sequence diagram assuming that only one module is enabled. In the figure, numbers refer to the above-numbered messages and colors are relevant. *Green* means trusted, *blue* trustworthy, whereas *red* untrusted. Thus, in the diagram, *NXP* and the *token* are considered trusted all along with their computation performed, whereas the *Card Issuer* trustworthy and the verifier's *main execution* untrusted for the already explained reasons. The main point to note is the secure tunnel that *NXP* and a *token* can build over an untrusted *main execution* instance. Figure XXVI shows a complete legend for the sequence diagram.

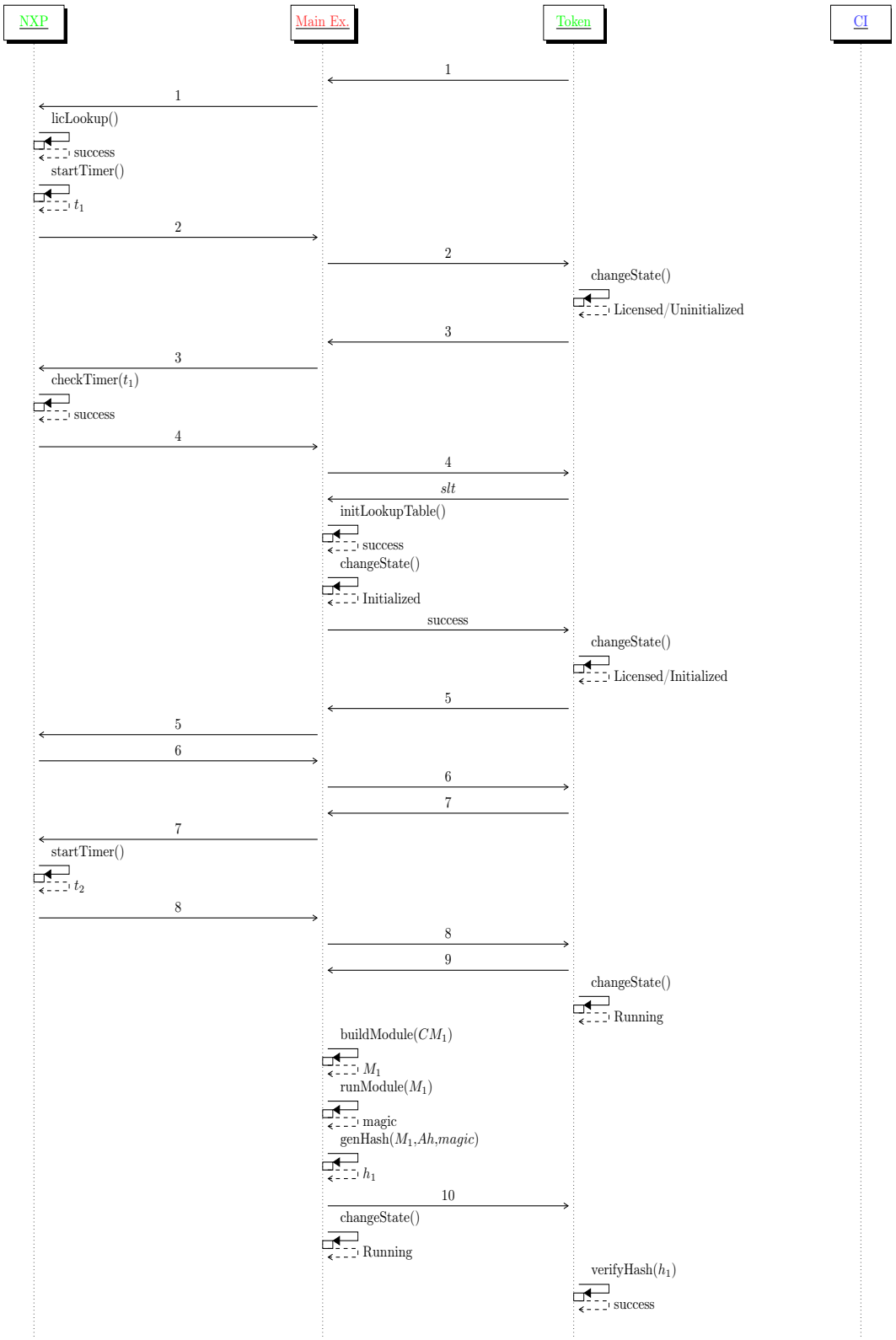

Figure XXIV: The protocol concept I

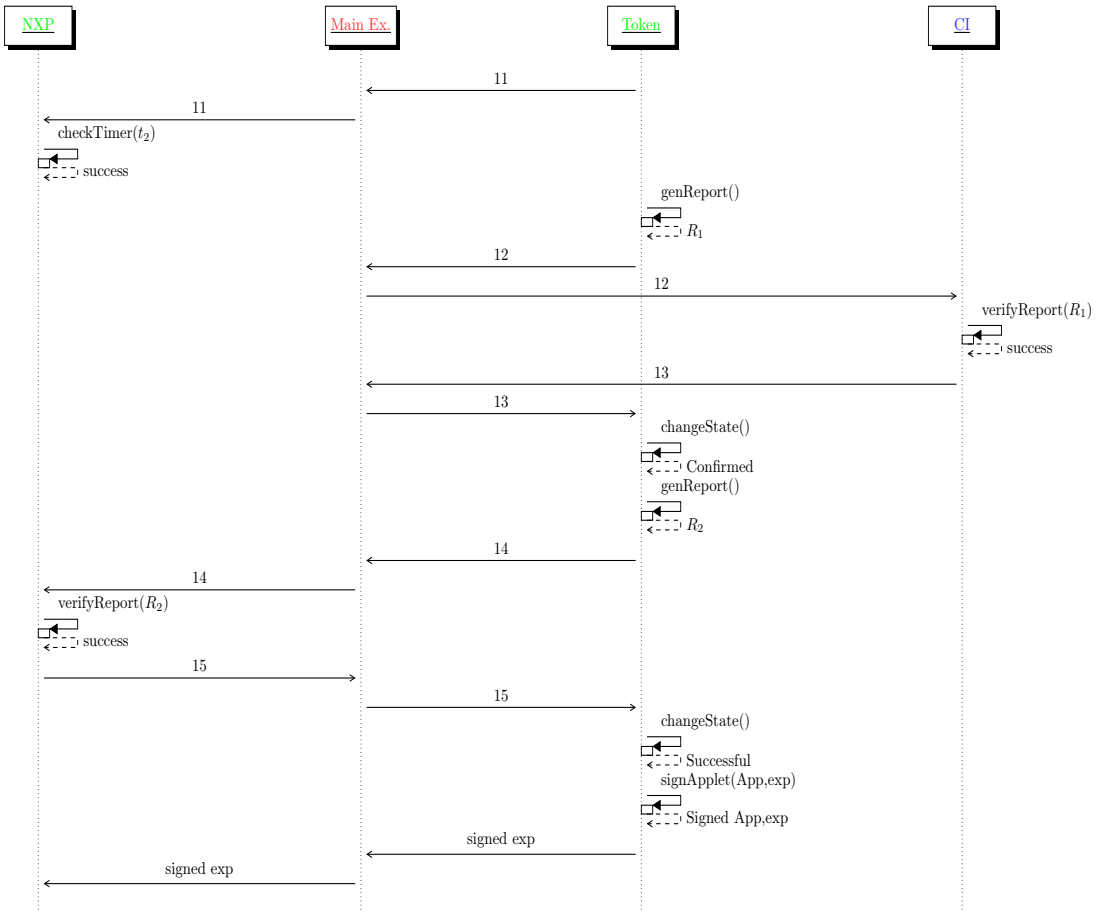

Figure XXV: The protocol concept II

Legend:

licLookup() → Search of the license and check validity

startTimer() → Start a timer

checkTimer(t) → Check if timer t was respected

changeState() → Change the instance state

initLookupTable() → Initialize *main execution* lookup tables

buildModule(CM_1) → Build module M_1 from the CM_1 and the already owned part

runModule(M_1) → Run check module M_1

genHash(M_1,Ah,magic) → Generate hash h_1

verifyHash(h_1) → Verify hash h_1

genReport() → Generate Report for monitoring

verifyReport(R_1) → Verify correctness of report R_1

Figure XXVI: The protocol concept legend

The operating principle of the protocol is explained as a flow in this section to make it easily understandable. However, some details are still missing. Therefore, the next subsections introduce more technicalities in terms of security mechanisms and explain how the system deals with edge cases. More precisely, Section 9.2.2 describes the approach used to enforce modules integrity. Section 9.2.3 explains the role of obfuscation and code flattening in the augmented bytecode verifier. Section 9.2.4 presents more in detail the white-box crypto technique proposed. Section 9.2.5 explains the role of disk encryption in the system.

9.2.2 Runtime Integrity Checks

One of the main threats is related to module tampering. In case an attacker was able to modify a check with another self-implemented module the entire process would be broken. Thus, enforcing the integrity of modules is fundamental.

Our proposal for minimizing the likelihood of this attack vector is splitting each module binary in two parts. The first part is delivered with the augmented bytecode verifier to the *Application Developer*, while the second part stays onto *NXP* back-end. As explained in Section 9.2.1, a module at a time is downloaded from *NXP* back-end and it has to be built and run by the verifier within a predefined time frame. Referring to step 5 the *token* informs *NXP* back-end that the session has been successfully initialized and *NXP Semiconductors* releases the part of the first module to be run. Once received, the module is built and run by the augmented bytecode verifier and as output a magic number is produced. The *main execution* can then compute an hash h_1 with three data:

- The complete binary of the module
- The applet hash *Ah*
- The computed magic number

Since h_1 is not trusted because computed in a *White-box scenario*, it is sent to the token, which can proceed with its verification using the list of hashes *lh* sent from *NXP Semiconductors* during step 4. The token informs *NXP* back-end about the outcome and in case of success the part of the second module is released. Otherwise, the session expires and all the process has to be performed again.

The most interesting details of this security mechanism are essentially two. Firstly, how the magic number is computed. Secondly, why those three data types are used to compute h_1. The primary purpose of the magic number, as already mentioned, is to avoid an attacker to tamper with a module. Our idea to achieve the goal is to compute a module's magic number using a slave-master approach. We can consider the first module part, delivered with the *augmented verifier*, the slave, whereas the other part, which stays at *NXP* back-end, the master. The slave is delivered with a bunch of *NOP* assembly instructions embedded that have to be initialized for the computation of the magic number. The master, before being released from *NXP* back-end, needs also to be initialized with

random data/computations that has/have to be unpredictable and brute-force resistant from an attacker perspective. These are the most important requirements. Once the *main execution* receives the initialized second part of the module, it can build the whole module and change the assembly *NOP* instructions of the first module part with some of the data/computations initialized in the second module part by *NXP* back-end. At this point, the *main execution* can run the module and, as a result, not only the applet check is performed, but also the magic number is computed and given as output. Finally, the *main execution* computes the hash h_1 and sends it to the *token*, which can verifies the correctness of the computed hash using the list of modules' hashes *lh* received previously from *NXP* back-end. The correctness of h_1 is considered as a proof that the module check ended successfully. *NXP* back-end can compute *lh* because has all the needed data beforehand. Note that the second part of a module is *initialized differently for each session* making extremely difficult for an attacker being able to retrieve the all needed data to compute h_1 and bypass the *token* check. Besides the difficulty an attacker could have at retrieving the needed data, *NXP* back-end initializes also a timer within which it expects an answer from the *token* about the outcome of the module. This increases further the complexity from an attacker perspective. For the computation of the hash h_1 we are using three data, i.e. the magic number, the hash of the whole module and the hash of the applet. In this way, we strongly bound the hash value to a session and the applet itself.

Listing 9.1 and 9.2 illustrate a dummy example to change a set of embedded *NOP* instructions with something more meaningful, which could be a number or a computation. The *C* code should be thought as the first part of a module while the *python* code as the set of operation performed by the *main execution*. Of course, the hexadecimal number *0x8345fc01* is not fixed, but should be taken from the second part of the module, which in turn, has been previously initialized for that session using a random and unpredictable source by *NXP* back-end.

```c
#include <stdio.h>

/* insert assembly NOP instructions without affecting the code functionality */
int func(int a) {
  __asm__(
    "nop\n"
    "nop\n"
    "nop\n"
    "nop\n"
    "nop\n"
  );
  return a;
}

int main() {
  printf("Result: %d\n", func(5));
  printf("MY JOB IS DONE HERE\n");
}
```

Listing 9.1: First part of a module

```
# Open and read the first part of the module binary
with open("uninitialized") as f:
    content = f.read()

# Find the offset of the NOP instructions
code_offset = content.index("\x90" * 5)

# Retrieve the data to replace with
new_code = "8345fc01".decode('hex')

# Replace NOP instructions with the retrieved data
patched_content = (
    content[:code_offset] + new_code + content[code_offset+len(new_code):]
)

# Save the binary
with open("initialized", "w") as f:
    f.write(patched_content)
```

Listing 9.2: Main Execution

This approach presents many advantages:

- The module part released from *NXP Semiconductors* is strongly bound to a session. Therefore, even if an attacker is somehow able to compute a magic number within the given time frame it is strictly related to only one module and in the next session for the same module that magic number will be useless.

- *NXP* back-end is an active part and, thanks to a strict collaboration with the *token*, a secure tunnel to exchange sensitive information about the state of the process can be initialized and data sent along it can be considered trusted.

- The process can be easily improved, changing computations of the magic number in the module parts stored at *NXP* back-end as needed.

9.2.3 Code Obfuscation and Code Flattening

Code obfuscation and code flattening have to be applied on the *main execution* to highly increase its complexity from an attacker perspective. Our purpose is to make our protocol reverse engineering unpractical and not worthy. We want to apply these techniques only on the *main execution* to maintain a balance between transparency and security as explained in Section 9.2. Proposing methods and approaches for applying effectively these techniques is out of scope because they strictly dependent on the implementation. However, some considerations can be found in Section 9.4 where possible implementation directions are discussed.

9.2.4 White-box Crypto

White-box cryptography is used in our protocol concept to secure local communication between the verifier and the token.

Contrary to the *token*, the *verifier* is in a *White-box scenario*, thus an attacker can access implementation of algorithms and observe their dynamic execution (with instantiated dynamic keys). Thus, cryptographic algorithm implementations onto the *token* and the *verifier* must be managed differently. As mentioned in Section 9.2.1, the *token* and *NXP* back-end can establish a secure tunnel where integrity, confidentiality and authentication are assumed to be achieved. Through this tunnel a session key Sk_2 is transmitted to the *token* that can use that key to perform encryption and decryption through its own cryptographic algorithm implementation, which is typically hardware-based. In the *verifier*, any key input by a cryptographic implementation is totally exposed to privileged attacks, therefore the sole remaining line of defense is the choice of implementation [Boneh 2001, Daemen 1999]. Our purpose is to apply *White-box cryptography* to our case using a *fixed key approach* [Chow 2003a]. In a nutshell, the idea is to embed the key in the implementation by partial evaluation with respect to the key, so that the key input is unnecessary. Since this key-customized implementation can be transmitted wherever bits can, embedded keys can be frequently changed making the approach feasible for our context. Chow et. al. presents in [Chow 2003b] a generator of an AES implementation that is resistant to *White-box scenario* making use of the *fixed-key* approach. The concept of this solution is particularly of interest and could be improved and adopted at *NXP* back-end to generate lookup tables that embed Sk_2 by partial evaluation. The augmented bytecode verifier AES implementation could be completely initialized only after a fixed amount of verifier life cycles, while the other times only some lookup tables could be changed. In this way, the process is less bulky, but still provides the needed level of security.

9.2.5 Binary Encryption

The verifier components have to be secured when not running to avoid an attacker to perform reverse engineering on both the *main execution* and *modules*. Since encryption and decryption are performed on an untrusted machine there is always a moment in which the attacker has the opportunity to defeat this security mechanism. Therefore, our purpose in using this mechanism is simply to increase the complexity of the process so that the attacker's motivation decreases. To achieve the goal we propose to make the *token* generate a key every time a process of verification is completed successfully or not. The *token* has to sign the key and forward it to a loader, which can be considered as an independent module in the verifier architecture. The loader receives the signed key, verifies it and encrypts the verifier binary, which includes the *main execution* and the *modules*. The key must not to be stored onto the untrusted machine, the *token* is in charge of storing it in its secure memory.

9.3 Enforcing Edge Case Features

As discussed in Section 7.2, it is not feasible to forecast the effects of a breach in a *White-box scenario*, thus security mechanisms should be designed to deal also with edge cases. Below, some considerations about the three crucial features that the augmented bytecode verifier should provide by design: key renewability, license revocation and verifier binary diversity.

9.3.1 Key Renewability

The verifier protocol concept is designed to provide flexibility and security. One of the most important aspects of our proposal is the ability to strictly bind each untrusted communication or operation to a session, to decrease the attack surface in terms of time. This design choice allows us to avoid the use of fixed keys that have to be renewed if leaked.

9.3.2 License Revocation

Performing license verification at *NXP* back-end, as described in Section 9.2.1 at step 1 and 2, permits to determine the identity of each user (*Card Issuers*, *Application Developers*) from the early start and securely controls the process of verification. In case of unwanted behaviours, detected along the process or through other ways, *NXP Semiconductors* can revoke a license even if a user is still in possession of the key for signing applets. The tuple *Verifier license* - *Certificate* of the related user, which is stored at *NXP* back-end, has to be invalidated. Revocation can so easily be applied because, as explained in Chapter 8, a user has the token for signing but does not know the key, which is strictly bound to an augmented bytecode verifier and can be used only during a process of verification, in which *NXP Semiconductors* actively participate from the beginning to the end.

9.3.3 Verifier Binary Diversity

Each verifier binary should be slightly different from others so that, in case of failure, an attack is not scalable. Our protocol concept splits modules' binaries in two parts and reassembles them for each session with different operations embedded (see Section 9.2.2). This approach allows already to provide indirectly an effective binary diversity. However, to further increase dissimilarity between binaries, different obfuscation and code flattening techniques might be applied to *main executions*. In this way, even if the functionality remains unchanged, the verifier binaries present more variations.

9.4 Implementation Directions

The system architecture, along with a description of its protocol concept, is introduced from an augmented bytecode verifier perspective in Section 9.2. In terms of implementation, several languages might be used, but we suggest the use of a combination of Java and C/C++ for both business and technical reasons.

The use of Java is favourable for the implementation of the *set of modules*, whose purpose is to make sure that the code of an applet is well typed and does not attempt to bypass JCRE security mechanisms by performing ill-typed operations at run-time, such as forging object references from integers, illegal casting of an object reference from one class to another, calling directly private methods of the API (further details in Chapter 5). We claim that Java might be a feasible choice for essentially three reasons:

- Java is a high-level language that simplifies the development of modules
- Java is portable making it easier to manage and distribute the modules
- NXP has the source of an Oracle Java Card bytecode verifier that might be used as a start

The programming language C or C++ is advantageous for the implementation of the *main execution*, whose purpose is to orchestrate the whole process of verification. We propose its use instead of Java because security mechanisms such as obfuscation and code flattening are not strongly effective on bytecode, due to its human-friendly nature. For example, obfuscated bytecode is equal in every platform and is typed, which means that a variable type cannot be changed to another one. As explained throughout the chapter, the *main execution* must be protected effectively from reverse engineering, due to the sensitive operations performed, thus we propose the use of C/C++ to increase the complexity from an attacker perspective.

The *main execution* and *modules* might communicate through the *Java Native Interface* (JNI). Even if *modules* can be considered as conceptually independent pieces of software from the *main execution*, they are strongly interconnected and controlled by means of our runtime integrity check mechanism proposed and the two involved trusted party, *NXP* and *token*, which can monitor the correctness of the operations throughout the entire process of verification.

9.5 Key Points

At the beginning of the chapter, two research questions were presented. The goal of this section is to provide clear and explicit answers with the purpose of outlining and emphasizing the chapter's key points.

How can the system be designed to withstand to white-box attacks by design?
Requirement 1 increased tremendously the design complexity, demanding the system to be run in a *white-box scenario*. To deliver the required level of security, with the primary purpose of withstanding white-box attacks, we designed several security mechanisms whose description can be found from Section 9.2.2 to Section 9.2.5. However, their use alone it is not enough as we are only increasing the complexity of the software from an attacker's perspective. What we need is a place where computation can be performed securely and considered trusted. Therefore, we made two design choices in this direction. Firstly, we decided to make use of a Java Card token to give us a trusted area, at the *Application Developer*'s location, to make computations that need to be done in a safe environment but, as explained in Section 9.1, its use alone is still not enough. Thus, our second choice was to involve actively, during the process of verification, *NXP Semiconductors'* back-end, with the purpose of creating a secure tunnel between the two trusted components. This approach allows to withstand white-box attacks by design because it permits to monitor an untrusted verifier instance without putting trust in unverifiable data. In fact, the use of a secure tunnel between a *Java Card token* and *NXP Semiconductors'* back-end enforces data source and data integrity.

How NXP Semiconductors can be self-confident that the applet verification process ended successfully?
This research question is strongly related to the previous one. *NXP Semiconductors* can be self-confident that a process of verification ended successfully because it is actively involved in the process and the final choice of whether granting an applet signature is under its control. Moreover, *NXP Semiconductors*, thanks to the defined architecture and protocol, has a high level of confidence that data received from an augmented verifier instance has not been corrupted and can be totally trusted. However, in case of detected unwanted behaviours license revocation can be easily applied as described in Section 9.3.2.

Chapter 10

Conclusions and Future Work

10.1 Improved Secure Element Management

The current limitations in the initialization and management of physical secure elements are evident. As a consequence, *Card Issuers* are looking for new virtual solutions and *end-users* have no control on the content of their mobile device's secure element. This thesis project presents a novel system concept to provide a flexible and highly-secure mechanism for applet installation and management, proposing a solution to the physical secure element issue. We worked on the whole stack, introducing new stakeholders, defining their interactions and proposing an innovative distributed system architecture based on the studied Java Card attack vectors.

The recurring theme of this thesis work is strongly related to *security*, *trust*, *flexibility* and *transparency*. Any system design choice has been taken with these interconnected properties in mind. *Security* can be considered an antonym of *flexibility*, due to their opposite nature. *Transparency* in term of functionalities can limit the effectiveness of the deployed security mechanisms because of the *white-box context* where the augmented verifier runs. *Trust* can be considered a consequence of the system *security* and business choices. Due to the complex interconnections among the required characteristics, the system comes with trade-off. *NXP Semiconductors* requested us to maximize *flexibility* while enforcing an high-level of *security* and respecting the stakeholder's business requirements. In other words, *NXP Semiconductors* asked us to analyze the most extreme scenario, which allows to verify applets in a hostile environment. With this in mind, our purpose was to design a new system concept, which provides the above-mentioned properties by design.

The adoption of our designed system would greatly maximize the potential of physical Java Card secure elements with relevant benefits for both stakeholders and users.
NXP Semiconductors might focus on providing physical secure elements ready to be used, enforcing security along the whole applet life cycle chain.
Card Issuers could concentrate on providing new services to *End-users* developing contents for secure elements. They could also easily outsource the implementation of applets granting *Application Developers* to act on its behalf.
End-users could manage the content of their mobile devices freely, installing as needed applets.

The system has many intrinsic advantages, which can be outlined as follow:

- Well-defined roles based on business activities: *NXP Semiconductors* takes care of security and physical secure elements, while *Card Issuer* can focus only on contents.
- *Flexibility* in the entire applet life cycle is maximized still providing the required level of *security* and *transparency*.
- Misbehaviours can be monitored and detected.
- *Card Issuers* have total control on *Application Developers* and can monitor their activity.
- Applets source/binary are never inspected directly by *NXP Semiconductors*, preserving *Card Issuer* intellectual property privacy and avoiding legal issues in the worst case.
- The augmented verifier is able to enforce the required level of *security* without the need of hiding functionalities of the verification modules. They could also be made available for download for *Card Issuers* to improve *transparency*.
- Applet distribution can be achieved in a variety of ways, without impacting the security behind the verification process.
- *End-users* can manage their own mobile device secure element content freely.

For these reasons we strongly believe that *trust* in the system by stakeholders could be achieved. Therefore, as all requirements presented in Chapter 7 are fulfilled, the system designed should be considered successful at this stage.

10.2 Related Work

Our thesis work tackled a problem that apparently nobody tried to solve so far. We focused on how to maximize the flexibility of a verification process without losing its security in the scenario of mobile devices. All related projects aim attention at the raw verification of bytecode with the purpose of proposing new solutions or improving the performance/effectiveness of already existing bytecode verifiers, both off-card [Posegga 1998, Barthe 2007] and on-card [Klein 2001, Casset 2002, Wang 2009, Berlach 2014]. Currently, very few Java Card secure elements are equipped with a on-card bytecode verifier because of the strict hardware-constraints. Ideally, an on-card bytecode verifier is more effective for essentially two reasons, firstly the applet code is checked immediately before installation and secondly the verifier can also include some runtime checks. As soon as the hardware-constrains of secure elements will be overcome someone might think that an off-card verifier would not be needed anymore. We strongly believe that in our scenario an off-card verifier is needed in any case as the checks that can be run on an applet are much more advanced and the architecture itself allows to strictly keep under control any applet verification process. An on-card verifier would only improve the security of the process.

10.3 Future Work

The work in this thesis could be continued and extended in a variety of directions.

The designed system concept presents enough detail to start with its implementation with the purpose of developing a prototype and testing the effectiveness of the proposed security mechanisms. The implementation is crucial for a proper functioning of the proposed functionalities, thus it has to be carefully carried out. The complexity of the task is very high, not only for the sensitivity of the functionalities to be developed, but also for the strongly heterogeneous nature of the system architecture, in fact components (back-end, token, augmented verifier) come under different assumptions and environments. In addition, further technical details should be analyzed and studied in more depth. For example, structure and format of reports sent from an *augmented verifier* instance to both *NXP Semiconductors* and *Card Issuers* (step 12 and 13 in Chapter 9) should be defined to enforce and ensure misbehaviours detections.

The verification modules to be run within an augmented verifier instance have not been studied in this report, only suggestions on possible modules were given. Therefore, proposed modules should be implemented and new ones designed based on the current attack vectors proposed in Chapter 5.

Bibliography

[Alliance 2009] Smart Card Alliance. *Security of Proximity Mobile Payments.* A Smart Card Alliance Contactless and Mobile Payments Council White Paper, Publication CPMC-09001, 2009.

[Anderson 1998] Ross Anderson and Markus Kuhn. *Low cost attacks on tamper resistant devices.* In Security Protocols, pages 125–136. Springer, 1998.

[Bar-El 2006] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall and Claire Whelan. *The sorcerer's apprentice guide to fault attacks.* Proceedings of the IEEE, vol. 94, no. 2, pages 370–382, 2006.

[Barbu 2010] Guillaume Barbu, Hugues Thiebeauld and Vincent Guerin. *Attacks on java card 3.0 combining fault and logical attacks.* In Smart Card Research and Advanced Application, pages 148–163. Springer, 2010.

[Barbu 2011] Guillaume Barbu, Guillaume Duc and Philippe Hoogvorst. *Java card operand stack: fault attacks, combined attacks and countermeasures.* In Smart Card Research and Advanced Applications, pages 297–313. Springer, 2011.

[Barbu 2012a] Guillaume Barbu, Christophe Giraud and Vincent Guerin. *Embedded Eavesdropping on Java Card.* In Information Security and Privacy Research, pages 37–48. Springer, 2012.

[Barbu 2012b] Guillaume Barbu, Philippe Hoogvorst and Guillaume Duc. *Tampering with Java Card Exceptions - The Exception Proves the Rule.* In SECRYPT 2012 - Proceedings of the International Conference on Security and Cryptography, Rome, Italy, pages 55–63, 2012.

[Barthe 2007] Gilles Barthe, David Pichardie and Tamara Rezk. *A certified lightweight non-interference java bytecode verifier.* In Programming Languages and Systems, pages 125–140. Springer, 2007.

[Beckert 2003] Bernhard Beckert and Wojciech Mostowski. *A program logic for handling Java Card's transaction mechanism.* In Fundamental Approaches to Software Engineering, pages 246–260. Springer, 2003.

[Berlach 2014] Reinhard Berlach, Michael Lackner, Christian Steger, Johannes Loinig and Ernst Haselsteiner. *Memory-efficient on-card byte code verification for Java cards.* In Proceedings of the First Workshop on Cryptography and Security in Computing Systems, pages 37–40. ACM, 2014.

[Bernardeschi 2004] Cinzia Bernardeschi and Luca Martini. *Enforcement of applet boundaries in Java Card systems.* In IASTED Conf. on Software Engineering and Applications, pages 96–101, 2004.

[Bevan 2003] Régis Bevan and Erik Knudsen. *Ways to enhance differential power analysis.* In Information Security and Cryptology—ICISC 2002, pages 327–342. Springer, 2003.

[Bieber 2000] Pierre Bieber, Jacques Cazin, Pierre Girard, J-L Lanet, Virginie Wiels and Guy Zanon. *Checking secure interactions of smart card applets.* In Computer Security-ESORICS 2000, pages 1–16. Springer, 2000.

[Boneh 2001] Dan Boneh, Richard A DeMillo and Richard J Lipton. *On the importance of eliminating errors in cryptographic computations.* Journal of cryptology, vol. 14, no. 2, pages 101–119, 2001.

[Bouffard 2014] Guillaume Bouffard. *A Generic Approach for Protecting Java Card^{TM} Smart Card Against Software Attacks.* PhD thesis, Université de Limoges, 2014.

[Brecht 2012] W Brecht. *White-box cryptography: hiding keys in software.* NAGRA Kudelski Group, 2012.

[Casset 2002] Ludovic Casset, Lilian Burdy and Antoine Requet. *Formal development of an embedded verifier for Java Card byte code.* In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pages 51–56. IEEE, 2002.

[Chen 2000] Zhiqun Chen. Java Card technology for smart cards: Architecture and programmer's guide. 2000.

[Chow 2003a] Stanley Chow, Phil Eisen, Harold Johnson and Paul C Van Oorschot. *A white-box DES implementation for DRM applications.* In Digital Rights Management, pages 1–15. Springer, 2003.

[Chow 2003b] Stanley Chow, Philip Eisen, Harold Johnson and Paul C Van Oorschot. *White-box cryptography and an AES implementation.* In Selected Areas in Cryptography, pages 250–270. Springer, 2003.

[Clavier 2000] Christophe Clavier, Jean-Sébastien Coron and Nora Dabbous. *Differential power analysis in the presence of hardware countermeasures.* In Cryptographic Hardware and Embedded Systems—CHES 2000, pages 252–263. Springer, 2000.

[Coskun 2013] Vedat Coskun, Busra Ozdenizci and Kerem Ok. *A survey on near field communication (NFC) technology.* Wireless personal communications, vol. 71, no. 3, pages 2259–2294, 2013.

[Cramer 1997] Timothy Cramer and Richard Friedman. *Compiling Java Just In Time.* IEEE, vol. 17, pages 36–43, 1997.

[Daemen 1999] Joan Daemen and Vincent Rijmen. *Resistance against implementation attacks: A comparative study of the AES proposals.* In The Second AES Candidate Conference, pages 122–132, 1999.

[De Koning Gans 2012] Gerhard De Koning Gans and Joeri de Ruiter. *The smartlogic*

tool: Analysing and testing smart card protocols. In Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, pages 864–871. IEEE, 2012.

[Diaconescu 2013] Alexandru Ionut Diaconescu. *Automated Code Review for Fault Injection.* PhD thesis, TU Delft, Delft University of Technology, 2013.

[Dyrkolbotn 2011] Geir Olav Dyrkolbotn. *Reverse Engineering Microprocessor Content Using Electromagnetic Radiation.* 2011.

[Éluard 2001] Marc Éluard, Thomas Jensen and Ewen Denney. *An operational semantics of the Java Card firewall.* In Smart Card Programming and Security, pages 95–110. Springer, 2001.

[Fort 2006] Milan Fort. *Smart Card Application Development Using the Java Card Technology.* 2006.

[Friedman 2004] Lawrence Friedman. *Method for a host based smart card*, December 29 2004. US Patent App. 11/025,495.

[Gadellaa 2005] KO Gadellaa. *Fault Attacks on Java Card: An Overview of the Vulnerabilities of Java Card Enabled Smartcards against Fault Attacks.* Master's Thesis, Technische Universiteit Eindhoven, 2005.

[Gopal 2009] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich and Martin Dixon. *Fast and constant-time implementation of modular exponentiation.* In 28th International Symposium on Reliable Distributed Systems. Niagara Falls, New York, USA, 2009.

[Govindavajhala 2003] Sudhakar Govindavajhala and Andrew W Appel. *Using memory errors to attack a virtual machine.* In Security and Privacy, 2003. Proceedings. 2003 Symposium on, pages 154–165. IEEE, 2003.

[Hogenboom 2009] Jip Hogenboom and Wojciech Mostowski. *Full memory read attack on a Java Card.* In 4th Benelux Workshop on Information and System Security Proceedings (WISSEC'09), 2009.

[Howard 2005] Michael Howard, Jon Pincus and Jeannette M Wing. Measuring relative attack surfaces. Springer, 2005.

[Hsueh 1997] Mei-Chen Hsueh, Timothy K Tsai and Ravishankar K Iyer. *Fault injection techniques and tools.* Computer, vol. 30, no. 4, pages 75–82, 1997.

[Iguchi-Cartigny 2009] Julien Iguchi-Cartigny and Jean-Louis Lanet. *Évaluation de l'injection de code malicieux dans une Java Card.* In Symposium sur la Sécurité des Technologies de l'Information et de la Communication (SSTIC'09), 2009.

[Iguchi-Cartigny 2010] Julien Iguchi-Cartigny and Jean-Louis Lanet. *Developing a Trojan applets in a smart card.* Journal in Computer Virology, vol. 6, no. 4, pages 343–351, 2010.

[ISO 2013a] ISO. *ISO/IEC 7816-4, Organization, security commands for interchange.* Technical report, International Organization for Standardization, 2013.

[ISO 2013b] ISO. *ISO/IEC 7816-5, Registration of application providers.* Technical report, International Organization for Standardization, 2013.

[Joye 2005] Marc Joye, Pascal Paillier and Berry Schoenmakers. *On second-order differential power analysis.* In Cryptographic Hardware and Embedded Systems–CHES 2005, pages 293–308. Springer, 2005.

[Klein 2001] Gerwin Klein and Tobias Nipkow. *Verified lightweight bytecode verification.* Concurrency and Computation: Practice and Experience, vol. 13, no. 13, pages 1133–1151, 2001.

[Klint 1981] Paul Klint. *Interpretation Techniques.* Software: Practice and Experience, vol. 11, pages 963–973, 1981.

[Kocher 1999] Paul Kocher, Joshua Jaffe and Benjamin Jun. *Differential power analysis.* In Advances in Cryptology—CRYPTO'99, pages 388–397. Springer, 1999.

[Kömmerling 1999] Oliver Kömmerling and Markus G Kuhn. *Design principles for tamper-resistant smartcard processors.* In USENIX workshop on Smartcard Technology, volume 12, pages 9–20, 1999.

[Krishna 2001] Ksheerabdhi Krishna and Michael Montgomery. *A Simple(r) Interface Distribution Mechanism for Java Card.* In Java on Smart Cards: Programming and Security, pages 114–120, 2001.

[Lancia 2013] Julien Lancia. Java card combined attacks with localization-agnostic fault injection. Springer, 2013.

[Lindholm 1999] Tim Lindholm and Frank Yellin. Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Manadhata 2011] Pratyusa K Manadhata and Jeannette M Wing. *An attack surface metric.* Software Engineering, IEEE Transactions on, vol. 37, no. 3, pages 371–386, 2011.

[Marché 2006] Claude Marché and Nicolas Rousset. *Verification of Java Card applets behavior with respect to transactions and card tears.* In Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on, pages 137–146. IEEE, 2006.

[McGraw 1999] Gary McGraw, Edward Felten and Ryan MacMichael. Securing Java: getting down to business with mobile code. Wiley Computer Pub., 1999.

[Microsystems 1998] Sun Microsystems. *Java Card Applet Developer's Guide.* Technical report, Sun Microsystems Inc., 1998.

[Microsystems 2003] Sun Microsystems. *Java Card Platform Security, Technical White Paper.* Technical report, Sun Microsystems Inc., 2003.

[Microsystems 2006a] Sun Microsystems. *Application Programming Interface, Java Card Platform, Version 2.2.2.* Technical report, Sun Microsystems, Inc, 2006.

[Microsystems 2006b] Sun Microsystems. Development kit user's guide. 2006.

[Microsystems 2006c] Sun Microsystems. *Runtime Environment Specification, Java Card Platform, Version 2.2.2.* Technical report, Sun Microsystems, Inc, 2006.

[Microsystems 2006d] Sun Microsystems. *Virtual Machine Specification, Java Card Platform, Version 2.2.2.* Technical report, Sun Microsystems, Inc, 2006.

[Microsystems 2008] Sun Microsystems. *The Java Card 3 Platform.* Technical report, Sun Microsystems, Inc, 2008.

[Montgomery 1999] Michael Montgomery and Ksheerabdhi Krishna. *Secure object sharing in Java Card.* In Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard'1999), Chicago, Illinois, USA, 1999.

[Mostowski 2007] Wojciech Mostowski and Erik Poll. *Testing the Java Card Applet Firewall.* Technical report, Radboud University Nijmegen, 2007.

[Mostowski 2008] Wojciech Mostowski and Erik Poll. *Malicious code on Java Card smartcards: Attacks and countermeasures.* In Smart Card Research and Advanced Applications, pages 1–16. Springer, 2008.

[Oaks 2001] Scott Oaks. Java security. O'Reilly Media, Inc., 2001.

[Oracle 2011] Oracle. *Java Card 3 Platform, Runtime Environment Specification, Classic Edition.* Technical report, Oracle America, Inc, 2011.

[Ortiz 2003] C. Enrique Ortiz. *An Introduction to Java Card Technology - Part 1.* Technical report, Sun Developer Network, 2003.

[Petrvalsky 2014] Martin Petrvalsky, Milos Drutarovsky and Michal Varchola. *Differential power analysis attack on ARM based AES implementation without explicit synchronization.* In Radioelektronika (RADIOELEKTRONIKA), 2014 24th International Conference, pages 1–4. IEEE, 2014.

[Piumarta 1998] Ian Piumarta and Fabio Riccardi. *Optimizing Direct Threaded Code by Selective Inlining.* SIGPLAN, vol. 33, pages 291–300, 1998.

[Platform 2003] Global Platform. *Global Platform Card Specification 2.1.1.* Technical report, Global Platform Inc., 2003.

[Platform 2006] Global Platform. *Global Platform Card Specification 2.2.* Technical report, Global Platform Inc., 2006.

[Posegga 1998] Joachim Posegga and Harald Vogt. *Byte code verification for Java smart cards based on model checking.* In Computer Security—ESORICS 98, pages 175–190. Springer, 1998.

[Rankl 2010] Wolfgang Rankl and Wolfgang Effing. Smart card handbook. John Wiley & Sons, 2010.

[Rose 2001] Eva Rose and Kristoffer Høgsbro Rose. *Java access protection through typing.* Concurrency and Computation: Practice and Experience, vol. 13, no. 13, pages 1125–1132, 2001.

[Saraswat 1997] Vijay Saraswat. *Java is not type-safe*, 1997.

[Skorobogatov 2003] Sergei P Skorobogatov and Ross J Anderson. *Optical fault induction attacks.* In Cryptographic Hardware and Embedded Systems-CHES 2002, pages 2–12. Springer, 2003.

[Wang 2009] Tongyang Wang, Pengfei Yu, Jun-jun Wu and Xin-long Ma. *Research on On-card Bytecode Verifier for Java Cards.* Journal of Computers, vol. 4, no. 6, pages 502–509, 2009.

[Witteman 2002] Marc Witteman. *Advances in smartcard security.* Information Security Bulletin, vol. 7, pages 11–22, 2002.

[Witteman 2003] Marc Witteman. *Java Card security.* Information Security Bulletin, vol. 8, pages 291–298, 2003.

[Zilli 2014] Massimiliano Zilli, Wolfgang Raschke, Reinhold Weiss, Johannes Loinig and Christian Steger. *A High Performance Java Card Virtual Machine Interpreter Based on an Application Specific Instruction-Set Processor.* In Digital System Design (DSD), 2014 17th Euromicro Conference on, pages 270–278. IEEE, 2014.

University of Twente

European Institute
of
Innovation and Technology

Security and Privacy

Minor Thesis

A Customers Needs Discovery Framework

Author

Alessio Parzian

Supervisors

Prof. Dr. Rainer Harms
University of Twente

August 5, 2015

Contents

1 Introduction **1**
- 1.1 The company . . . 2
- 1.2 Contextualization . . . 3
- 1.3 Research Direction . . . 4

2 Literature **5**
- 2.1 The VOC Concept . . . 6
- 2.2 Costumers Needs and Wants . . . 7
- 2.3 Identifying Costumers Needs . . . 9

3 A qualified costumer's needs analysis **13**
- 3.1 Persistent Challenges . . . 14
- 3.2 Building an Effective Voice of the Customer Programme . . . 14
  - 3.2.1 Define and Prioritize Business and Organization Goals . . . 15
  - 3.2.2 Define and Prioritize Customers Segments . . . 15
  - 3.2.3 Gather Raw Data . . . 16
  - 3.2.4 Translate Raw Data into Interpreted Needs . . . 16
  - 3.2.5 Analyze and Prioritize Interpreted Needs . . . 17
  - 3.2.6 Translate Customers Needs into Functional Requirements . . . 17

4 Conclusion **19**

Bibliography **21**

Chapter 1

Introduction

Contents

1.1 **The company** 2
1.2 **Contextualization** 3
1.3 **Research Direction** 4

The I&E Thesis focuses on addressing an Innovation and Entrepreneurship topic, motivated from key challenges documented for the thematic focus area of the Master Thesis project. Referring to the guidelines, I chose to analyze a step along a relevant business development frame-process, making this work tightly coupled to my final thesis project.

The following chapter provides some details about the company I am working with in Section 1.1 and introduces the project's context along with its scope and purpose, respectively in Section 1.2 and Section 1.3.

1.1 The company

NXP Semiconductors is a Dutch semiconductor manufacturer. It is one of the worldwide top 20 semiconductor sales leaders and was founded in 1953, when the Philips Board started a semiconductor operation in Nijmegen, Netherlands [Penning de Vries 2010]. Formerly known as Philips Semiconductors, the company was sold by Philips to a consortium of private equity investors in 2006. The new name, NXP, stands for the consumer's "next experience". According to NXP's official website, Figure I outlines quickly the main aspects of the company.

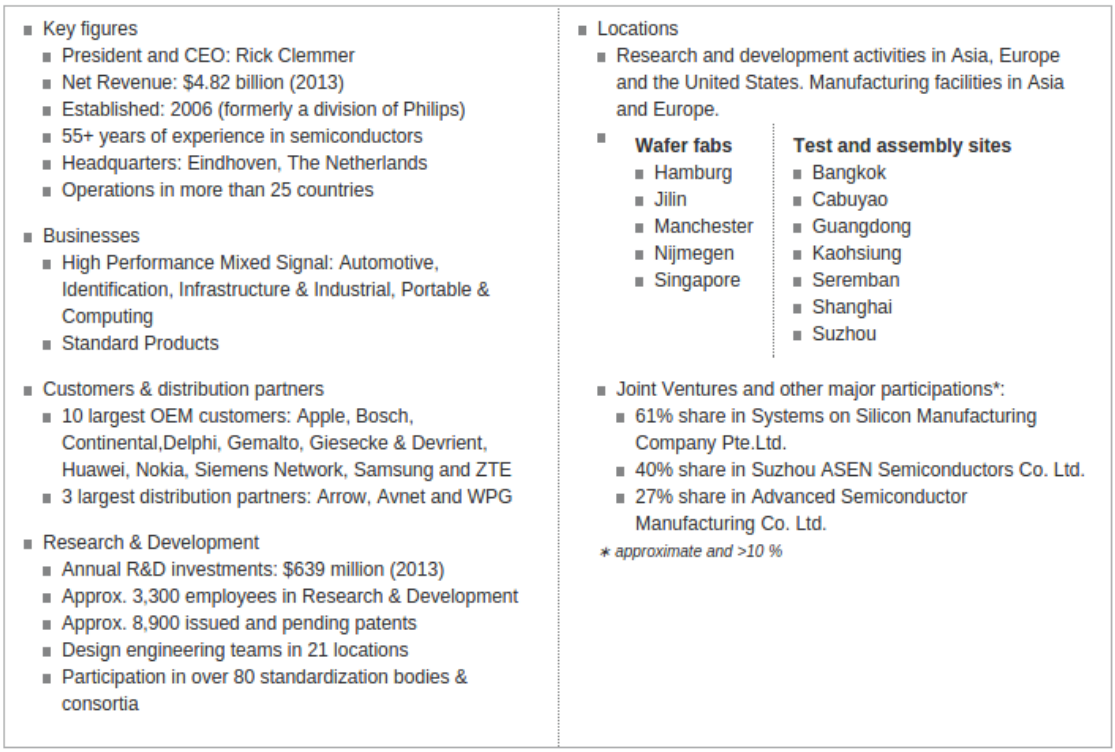

Figure I: Facts & Figures

NXP covers a wide variety of IT solutions in several areas, but always with a strong focus on security. More precisely, NXP creates solutions for Connected Cars, Cyber Security, Portable & Wearable and the Internet of Things. Currently, I am employed at the "Innovation Center of Crypto and Security" and am part of the "Security Concept" team, which consists of fifteen professionals. My thesis project focuses on secure mobile transactions in the cyber security field.

1.2 Contextualization

"The convergence of payments and mobile communications is not just logical – it is inevitable". In March 2007, John Philip Coghlan, then CEO of Visa USA, made this announcement at the CTIA Wireless Conference. Such a convergence is claimed to be inevitable for essentially three reasons [Alliance 2009]:

- *Contactless payment adoption.* Payment brands, issuers and consumers adopts contactless payment solutions due to its speed, easy of use and security, while merchants adopt it because of faster transaction time, increased spending and higher loyalty. Moreover, the contactless infrastructure is built on top of the existing financial network, therefore merchants are required only to upgrade their point-of-sale (POS) to contactless-enabled POS with negligible costs.

- *Mobile device ubiquity.* Mobile phone subscribers do not leave home without their phones. In addition, near field communication (NFC) technology has become an international communication standard to deliver simplified and robust implementation of contactless payments using mobile devices due to its secure nature [Coskun 2013]. Today, NFC is a standard functionality provided in most mobile phones.

- *Expanded mobile functionalities.* Mobile devices are powerful tools that can deliver a variety of payment and payment-related services such as proximity mobile payments, remote payments through the mobile Internet or text messaging, and person-to-person money transfers. Value-added applications can enrich the purchase experience and include account management, banking, offers, and security applications.

Many attempts of creating a secure and open mobile *proximity payment system* have been made. Typical examples are *Google Wallet* and *Apple Pay* that are trying to create an homogeneous system on top of the already existing financial circuits such as *Eurocard*, *Amex*, *Mastercard*, *Maestro* and *Visa*. However, they have been only partially successful due to the strict security requirements that the system needs to comply with, the hostile environment in which the system must run and the many business parties involved in providing it, i.e. device manufacturers, application providers and card issuers. Today, interesting technologies that could meet the strict requirements are available in the market, but designing a smooth environment that stakeholders, even competitors, can trust and participate in is the greatest problem. As a result, mobile proximity payment systems are today enabled only in specific countries and still have to gain a foothold not withstanding the huge potential.

Technically speaking, the cornerstone of security in a proximity payment system is the secure element [Alliance 2009]. Essentially, it is a protected area, independent from the application process/operating system of the device, which is capable of storing and processing sensitive information of the device holder. Authentication, encryption of private data, data integrity and non-repudiation are typical services that a secure element provides. One of the solutions provided by *NXP Semiconductors* is the use of a secure

element, which consists essentially of a built-in smartcard chip embedded in the device running a *Java Card virtual machine* that communicates with an external terminal by means of NFC. The main innovating concept that has been brought into the market with the introduction of *Java Card* is the multi-application environment. That is, applications (from now on "applets") can run on the same smartcard chip and can be uploaded even after a smartcard issuance. Such a feature might be the key for creating an open, homogeneous and trusted environment, but, due to security reasons, it has never been extensively used. As a consequence, *NXP Semiconductors*, as a smartcard manufacturer, is currently strictly controlling the access to its secure elements enforcing a tight collaboration with card issuers. Usually, applets of different parties never run on the same chip and are never uploaded after the card issuance, even if it is developed by a trusted application provider related to the same card issuer. Obviously, this is a huge limitation that is obstructing the spread of a proximity payment system and is leading to the development of workaround solutions that do not need a physical secure element, such as the *Host-based Card Emulation* approach [Friedman 2004] that is starting to be used by Google. As a secure element manufacturer, *NXP Semiconductors* wants to stop this trend, proving that a secure and versatile environment using a physical secure element can be truly designed and implemented enabling in such a way the use of physical secure elements in mobile devices.

1.3 Research Direction

My final research thesis project investigates the current state of the art of Java Card with the final purpose of pointing out how NXP Semiconductors might tackle the above-introduced problem enforcing security and trust without losing control on what is uploaded on its secure elements. That said, this paper focuses on the needs discovery of the direct and indirect NXP's consumers[1] involved in the environment I am working on. Namely, card issuers and application providers. The goal is to provide *NXP Semiconductors* with a framework to extensively find out the real problems these stakeholders encounter along the process, so that the final solution of the thesis project can be suitable not only in term of security and privacy, but also in term of consumers requirements. Thus, referring to the "Framework of an Innovation Development process" (guidelines Annex 2), this work refers to *The voice of the costumer* in the *Conceptualization* phase. The present document is structured as follow:

- Chapter 2 investigates the current state of the art about costumer's needs discovery.
- Chapter 3 proposes an effective framework, based on the presented concepts, which allows *NXP Semiconductors* to deeply investigate the real needs of its customers and translate them meaningfully into functional product requirements.
- Chapter 4 draws my final personal conclusions.

[1] Direct customer is someone dealing directly with the supplier, indirect customer is someone who deals with the supplier through an intermediary (agent, etc.)

Chapter 2

Literature

Contents

2.1 The VOC Concept 6
2.2 Costumers Needs and Wants 7
2.3 Identifying Costumers Needs 9

With respect to product development and innovation, the language used when speaking about costumers' needs differs depending from what angle this aspect is being analyzed. Marketing, Engineering and Industrial Design literatures use different terminologies and often key terms are used interchangeably [Bayus 2008]. This trend creates confusion, increases the level of uncertainty and lessens opportunities of success in any market research. The purpose of this chapter is clearly present the current state of the art about the discovery of costumers' needs and wants, using a coherent terminology and emphasizing its interdisciplinary. More in details, Section 2.1 introduces the concept of *Voice of the Costumer* explaining its scope and purpose. Section 2.2 defines formally the meaning of *need and want* and provides a quick description of several aspects that are crucial for being effective in a customer needs discovery phase, regardless of the customer segment. In conclusion Section 2.3 focuses on identifying and classifying customers needs to maximize the satisfaction of final customers and consequentially making a product successful.

2.1 The VOC Concept

Satisfying customers' unmet needs is considered the key to company growth as it allows to create more value. For such a reason, when developing a new product, understanding the real needs of future costumers is crucial. Conceptually, understanding costumers' needs leads to products that are *desirable*, *feasible* and *salable* to the target market, creating consequentially *successful innovations*. Figure II shows how these attributes creates the so called *innovation space*.

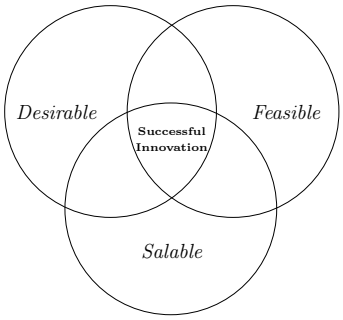

Figure II: The Innovation Space

Costumers' need discovery is an active part in the new product development process[1]. Figure III shows the major steps involved in the "fuzzy front-end" – it refers to the set of the activities employed prior to entering the formal product development process. As illustrated, understanding the costumers' needs has become known as the *Voice of the Costumer* (VOC).

Formally, VOC is defined as follow:

"The Voice of the Costumer" (VOC) is a term used in business to described the process of identifying customers' requirements [Yang 2007].

Besides identifying customer requirements, VOC includes also producing a detailed list of needs, which has to be firstly organized into a hierarchical structure and then prioritized with respect to its costumer importance [Griffin 1993]. There are several approaches that can be used for prioritization, from subjective scoring by the development team to customers rating methodologies [Pullman 2002]. Note that, prioritizing customer needs is valuable, because it allows the development team to make necessary tradeoff decisions when balancing the costs of meeting a customer need with the desirability of that need relative to the entire set of customer needs. Then, to provide the big picture, the information gathered during the VOC phase is translated into requirements and product specifications,

[1] In business and engineering, new product development (NPD) is the complete process of bringing a new product to market. It is basically divided in four phases: *Fuzzy front-end*, *Product design*, *Product Implementation*, *Fuzzy back-end*.

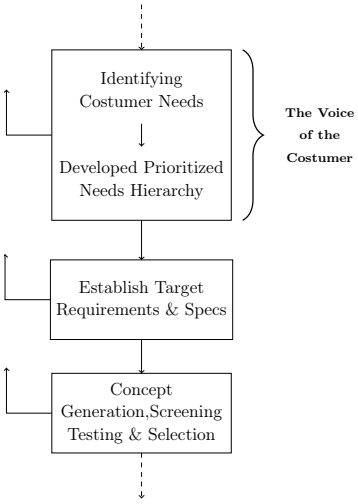

Figure III: The Voice of the Costumer in the "fuzzy front-end"

which in turn are translated into specific product attributes for prototypes [Dahan 2002].

This discovery process brings relevant benefits to a company providing:

1. an extensive understanding of costumers' requirements
2. a common language for the developers team
3. key pieces of information to design appropriately the product
4. a strong starting point to innovate the product

Typically, VOC studies are qualitative and quantitative market-research steps. They are performed during the initial design phase of a product to better understand the wants and needs of a costumer or during a product improvement phase using the resulting information as a key input for a new definition.

2.2 Costumers Needs and Wants

"A customer need is a description, in the customer's own words, of the benefit to be fulfilled by the product or service" [Gaskin 2010]. It is long-term in nature and cannot always be identified and described verbally by a costumer [Burchill 1997, Mello 2003]. On the other hand, "a customer want is a thing that a customer believes will fulfill a known need" [Bayus 2008]. It is short-term and temporary in nature, however, can be easily influenced by means of advertising, laws, norms, personal recommendations.

When designing a product, needs and wants are about "what" is wished by costumers, whereas specifications concern with "how" a need is fulfilled. More precisely, from the engineering and design literature, a *requirement* is a technical solution to meet a costumer's need whereas a *specification* is a metrics associated to a requirement [Ulrich 2004]. In

addition, in the economics literature, a *characteristic* is a property of a product that is relevant for the costumer final choice [Geistfeld 1977]. While, a product *attribute* relates to a product *characteristic* and describes the perceptual dimension that a customer uses to make purchase choices. Figure IV illustrates the relationship between these concepts in the new product development phase.

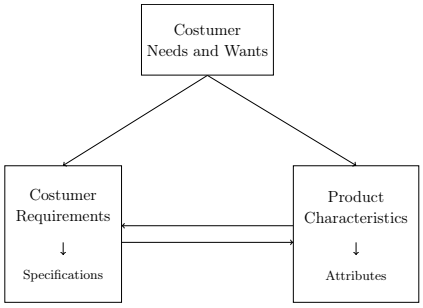

Figure IV: The language of new product development [Bayus 2008]

It has been proven that understanding how costumers take final decisions with respect to a product is one of the keys to success. Brunswick [Brunswik 1952] suggests the costumers see the world through the lens of their perception and introduces the so called "lens model". Figure V shows such a model clearly illustrating the several factors that influence a costumer final choice. Within the context of the "lens model", VOC identifies the values customers take care about (needs) and how costumers create preferences with respect to those needs (prioritization). VOC might also be useful to identify how to influence a costumer final choice, i.e. affecting his/her perceptions through advertising for example.

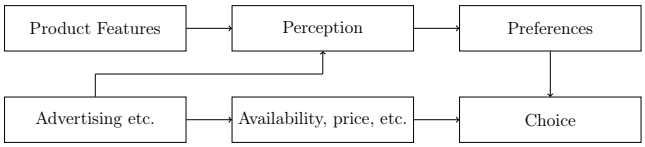

Figure V: The "lens model"

Customers communicate explicitly or implicitly using different channels and behaviours that often are not correctly interpreted by companies. Figure VI illustrates the main sources of VOC, which need to be analyzed to understand and unveil the real needs of customers or to validate the current quality of a product/service. As a result of the complexity of this information flow, costumer needs have to be *interpreted* from the raw data gathered during the first step of a VOC study. Section 2.3 presents how to identify and classify meaningfully needs and wants in order to increase the costumer satisfaction.

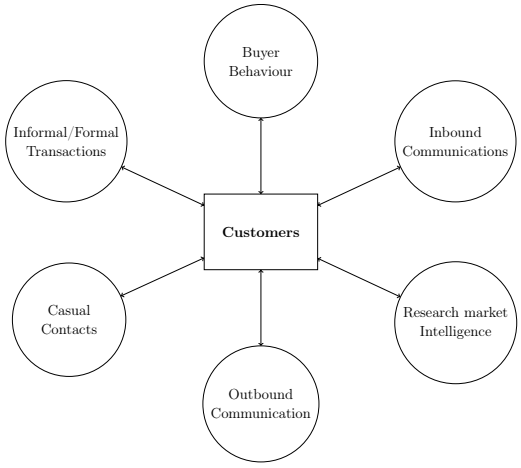

Figure VI: Source of Costumer Voices

2.3 Identifying Costumers Needs

One of the most known approach for identifying the types of consumer needs is the *"Kano Model of Customer Satisfaction"* [Kano 1984]. Kano et. al. developed this methodology adapting the ideas of Fredrik Herzberg [Herzberg 2011] on the asymmetry of the factors related to job satisfaction and dissatisfaction. In a nutshell, Herzberg claimed that job satisfaction is related to "motivators" such as achievement and recognition, whereas job dissatisfaction to "hygiene" factors such as company policies and working conditions.

The key concepts of Kano theory can be outlined using, as illustrated in Figure VII, a Cartesian plane. The horizontal axis indicates the degree to which a particular consumer need is addressed in a new or existing product, ranging from totally unmet to completely fulfilled. The vertical axis refers to the satisfaction for a specific implementation of a customer need ranging from disgusted to delighted. Using this two-dimensional plane Kano et. al. defines three types of customer needs, namely *basics needs*, *performance needs* and *exciting needs*. "Basics needs" represents needs that are taken for granted, i.e. already assumed by a customer to be met. An example might be the brakes in a car. Meeting these needs do not increases the customer satisfaction, but the absence or poor performance of these attributes in a product results in extreme customer dissatisfaction. "Performance needs" refer to those needs for which the customer satisfaction is roughly proportional to the level of performance shown by the product attribute. An example might be a car that provides a better fuel economy. This type of need is usually directly requested by customers when doing reviews. "Exciting needs" represents needs that customers do not expect to be satisfied. In case this need is completely addressed the customer is delighted, if not the customer does not really care. This kind of need is obviously the "order winner" for costumers. Other categories of needs have been proposed in literature such as "observable needs", "explicit needs", "tacit needs" and "latent needs". However, Kano model is still considered a cornerstone when it comes to understanding customer needs as it allows to

identify all those needs that factor into consumer decisions.

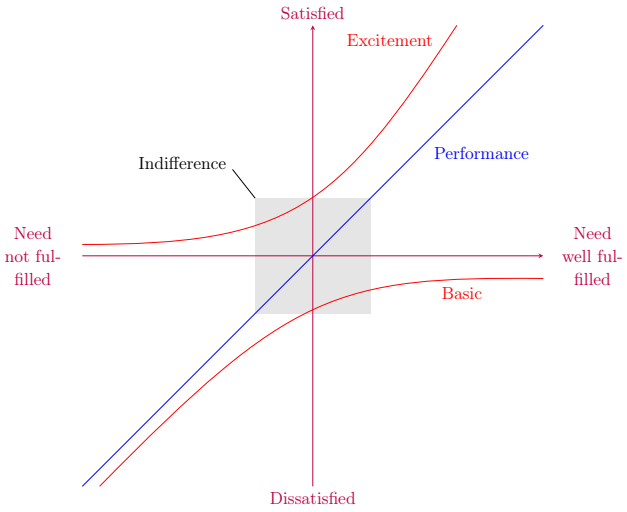

Figure VII: The Kano Model

The underlaying message that Kano wanted to convey through his model is straightforward but essential. Customers are dynamic, a need that today belongs to one category might be part of another category in the future. An example might be the air-conditioning in cars that was considered a delighter in the 1950, but a basic need today. This means that customer expectations increase over time and company to stay competitive in the market need to be flexible and strive to better understand ever-changing customer needs.

As explained in Section 2.1, during the VOC phase a company gathers raw information about customers' needs that must be interpreted, firstly defining a hierarchy and secondly creating a prioritized list. Kano model can be used for defining a hierarchy of *interpreted needs*. However, in order to gain a rich understanding of customer needs a company must be aware, as show in Figure VIII, that *interpreted needs* consists of *articulated needs* and *unarticulated needs*, which require different kind of techniques to be discovered.

The former refers to those needs that can be verbalized, if asked appropriately, whereas the latter to those that consumers cannot easily verbalize. It is important to keep in mind that there are many motivations that might stop customers from communicating his/her idea – they do not understand, they don't know how to tell, they do not remember etc. Generally speaking, "articulated needs", in order to be discovered, require to focus on "what customers say". There are several traditional market research approach that might be used such as questionnaires, surveys, interviews, focus groups [Urban 1993]. Other less traditional, but well-known, techniques include conjoint analysis, preference modeling, simulated test market [Green 2001, Kaul 1995] and category problem analysis [Swaddling 1996]. Typically, "unarticulated needs" refers to information dealing with "what customer make and do" [Sanders 1992]. In order to obtain a good understanding of this category of needs current and ideal experiences of customers should be correctly inter-

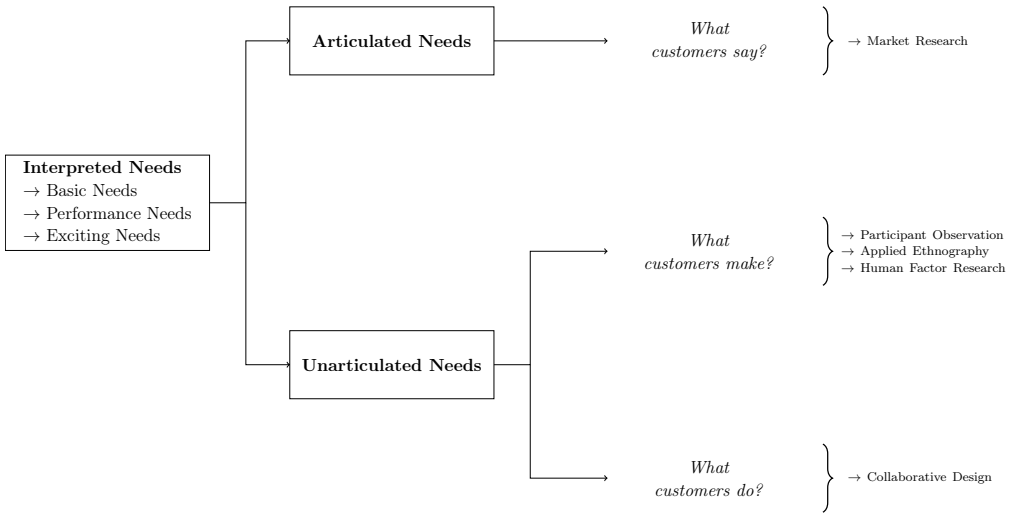

Figure VIII: Understanding Customers Needs

preted by the company. Participant observation, emphatic[2] approaches [Leonard 1995], applied ethnography [Masten 2003] and contextual inquiries [Holtzblatt 1993] are the main methods. However, the primary method is participatory and collaborative design between the development team and customer, which is used for discovering what customers know, feel, and dream through what they make. Techniques includes lead user analysis [Von Hippel 1999], customer toolkits [Franke 2004] and metaphor elicitation [Christensen 2002]. Moreover, some researches suggest to incorporate abstract attributes, such as aesthetics, emotions and experiential aspects into the identification process of these "unarticulated needs" [Schmitt 1999, Desmet 2001].

This discussion presents a variety of methods and approaches that can be used to reach a deep understanding of user needs. As suggested by Sanders [Sanders 1992], multiple methods amongst the above-mentioned should be used together to uncover the complete range of customers' needs. However, it is important to note that the degree to which needs must be *interpreted* from the raw information, gathered through these methods, increases from learning "what customer say" to "what customer make and do". Several approaches are presented in literature to make this translation from raw information to *Interpreted needs*, however, they are beyond the scope of this report. For a good coverage of this step we refer to [Otto 1998, Shillito 2000].

[2]Empathic approaches pays attention to the user's feelings toward a product.

Chapter 3

A qualified costumer's needs analysis

Contents

3.1 Persistent Challenges **14**
3.2 Building an Effective Voice of the Customer Programme **14**
3.2.1 Define and Prioritize Business and Organization Goals 15
3.2.2 Define and Prioritize Customers Segments 15
3.2.3 Gather Raw Data 16
3.2.4 Translate Raw Data into Interpreted Needs 16
3.2.5 Analyze and Prioritize Interpreted Needs 17
3.2.6 Translate Customers Needs into Functional Requirements 17

The purpose of this chapter is to define and present a framework to effectively discover, identify, understand and translate into functional product requirements customers needs, limiting any kind of waste in term of time, money and human resources. Section 3.1 points out several challenges that still are open, while Section 3.2 introduces step by step the framework, which, if applied correctly, might give significant advantages to *NXP Semiconductors*.

3.1 Persistent Challenges

Based on Chapter 2, several techniques, both qualitative and quantitative, have been proposed improving significantly the procedures for gathering and analyzing with great insights valid and reliable customers data. However, despite these advances, many challenges related to customers' thoughts, feelings and behaviours are still open. In particular, market researches need to be enhanced in a way that [Coulter 1995]:

- Provide a better understanding about customers as a basis for advertising and marketing-mix decisions.
- Improve latent and emerging needs elicitation.
- Give better guidance for capturing and engaging customers thought process.
- Enhance organization of unverbalized data.

3.2 Building an Effective Voice of the Customer Programme

Due to the huge amount of influential variables, creating an effective framework to identify and understand the complete range of customer needs is extremely challenging.

Referring to the context presented in Chapter 1, *NXP Semiconductors* is carrying out a deep research for coming up with new solutions to tackle the described problem. An interesting approach has been now identified, but it is crucial to avoid any financial, time and human resource waste. To be successful, NXP's customers, i.e. card issuer and application providers, must be given a product that satisfy their own needs.

Hereafter, a proposed framework[1] to achieve this goal with the purpose to also deal with the described persistent challenges. It consists of the following steps:

1. Define and prioritize business and organizational goals 3.2.1.
2. Define and prioritize customers segments, refer to Section 3.2.2.
3. Gather raw data, refer to Section 3.2.3.
4. Translate raw data into interpreted needs, refer to Section 3.2.4.
5. Analyze and prioritize interpreted needs, refer to Section 3.2.5.
6. Translate customer needs into function requirements, refer to Section 3.2.6.

[1] The framework proposed is based on [Mazur 2003]

3.2.1 Define and Prioritize Business and Organization Goals

Understanding customers needs requires a team with cross-disciplinary knowledge. In a large company as *NXP Semiconductors* this means that members of different departments start to work together on different aspects of the problem. Obviously, the operate of each team member is evaluated by different organizational bosses. Thus, team and business goals could differ from individual performance evaluation factors. Clarifying both team and departmental goals is crucial to avoid internal arguments, which could badly affect the project. The main goal of this step is to define and prioritize three layer of goals, namely:

- *business goal*: market share, revenue, profit, brand
- *product goal*: performance, functionality, reliability
- *project goal*: on time, on budget etc.

Their definition should be done early in development process in order to assure alignment of cross-functional activities with what matters most to the business.

3.2.2 Define and Prioritize Customers Segments

Identifying customers is an extremely important market research. However, when it comes to designing the product in order to meet the needs and wants of these customers, it is crucial to focus on characteristics. Thus, during this step customers should be defined based on characteristics of use. Usability, functionalities and appearance issues must be understood. The *"Customer Segment Table"* is strongly useful for this kind of task. Table I shows a template. During this phase, it is relevant to classify customers segments basing on their impact in making the product successful. In our case, *card issuers* are undoubtedly much more influencing than *application developers*, who are, for *NXP Semiconductors*, only indirect customers. Based on that, in case of limited resources for visiting customers a plan to best allocate them has to be done.

Who is the customer?	**What are they doing?**	**When are they doing it?**	**Where are they doing it?**	**Why are they doing it?**	**How are they doing it?**	**What is the current situation?**
Card Issuers						
App. Developers						

Table I: The Customer Segment Table

3.2.3 Gather Raw Data

The goal of this phase is collecting as much data as possible with the regards to customers needs. As introduced in Section 2.3, there are different types of customers needs, namely *articulated* and *unarticulated*. In order to identify and maximize the discovery of these needs, different techniques proposed in literature[2] might be concurrently applied.

NXP Semiconductors already had many experiences with *card issuers*, as described in Section 1.2. Therefore, firstly it is relevant to document the customers needs the company already thinks to know. Secondly, methods to discover *articulated* and *unarticulated* needs have to be applied. Deciding the most effective ones in term of outcome/cost ratio is at discretion of the company, based on the already deployed marketing channels. However, our customers are in turn supplier, providing service to final users. This means they are entities with a high technical knowledge of the scenario's issues. For *NXP Semiconductors* this aspect is undoubtedly relevant as it allows to decide accurately the methodology to use to maximize the coverage of the needs space: *card issuers* are fully or partially fully knowledgeable about all their requirements and are able to *verbalize* them. As a consequence, *NXP Semiconductors* should focus on techniques to uncover *articulated* needs, and as described in Section 2.3 it is a great advantage. Thirdly, the needs claimed initially by the company must be validated. In this way *NXP Semiconductors* would be able to identify customers needs and maximize the range.

3.2.4 Translate Raw Data into Interpreted Needs

The previous phase gathered a huge set of data with regards to costumers needs through market research methods. However, this data are still raw and need to be translated to be useful performing the following steps:

1. Document each collected data along with its context
2. With the help of customers, each documented data must be translated into a need statement. Note that it is not uncommon to derive as many as five to ten needs from one documented data. Further *unarticulated* needs will emerge.

When performing this translation, it is essential to define a set of criteria which make a documented data a true customer need with the purpose to be coherent along the whole process. The following might be feasible criteria, the documented data:

1. Defines the benefit customers receive from: their problems solved, their opportunities enabled, their image enhanced
2. Is positively stated
3. Focuses on a single issue
4. Is independent of specific products or services, features, and technologies

[2]Refer to Section 2.3 for an interesting list of the most important.

3.2.5 Analyze and Prioritize Interpreted Needs

The customer needs must be prioritized by customers in order to understand how important they are and to whom. An effective procedure is the *Analytical Hierarchy Process* [Saaty 1990] which provides ratio-based priorities based on natural language comparisons. This method breaks down decision making into sets of pair-wise comparisons allowing to determinate the relative importance of evaluation criteria[3]. Table II shows a template example.

Defining valuable and meaningful evaluation criteria is a crucial task that *NXP Semiconductors* needs to carry out carefully.

	Criteria 1	Criteria 2	Criteria 3		Sum of ratings		Weighting	
Criteria 1	1							
Criteria 2		1						
Criteria 3			1					
Total								

Table II: The analytical hierarchy process framework

This methodology provides several benefits as it:

- Converts subjective assessments of relative importance into a set of overall scores or weights
- Offers a team activity where members must decide on the relative importance of one factor against another
- Easily removes emotions and helps in making better decisions

3.2.6 Translate Customers Needs into Functional Requirements

Many methodologies for translating the interpreted needs into functional requirements have been proposed as already mentioned in Chapter 2. This step is somehow out of scope, but for the sake of completeness I want to suggest the use of an interesting technique for managing effectively this translation. The technique makes use of the so called

[3] A quick but effective explanation can be found at http://asq.org/service/body-of-knowledge/tools-analytic-hierarchy-process

Maximum Value Table (MVT), which allows to connect a particular customer need to several dimensions (customers, solution, design, project). Figure IX shows an example. The most advantage that this approach provides is to give to the reader a clear big picture, allowing in this way to create tasks that will be assigned to the development team. The MVT does not of itself kick-off the whole project, but illustrates where effort should be made to guarantee the design and delivery of a successful product.

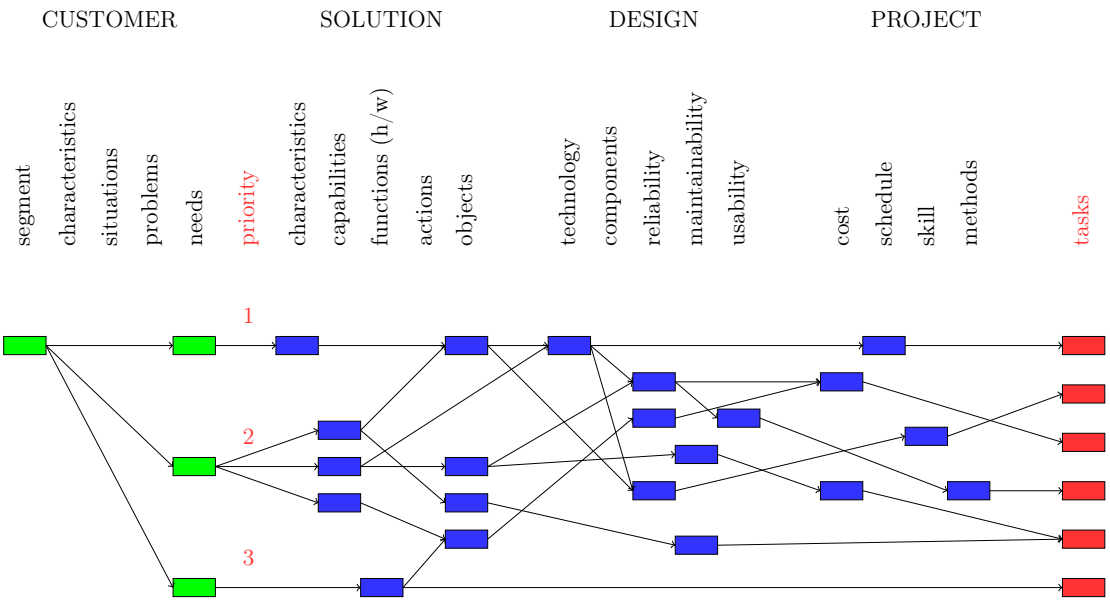

Figure IX: Minimum Viable Table

Chapter 4

Conclusion

Understanding customers needs is essential to design and implement successful products. However, at the same time, this phase is extremely challenging, as there will always the need to define tradeoff between the expediency and cost efficiency of practical methods for understanding customer needs versus methods of obtaining a deeper understanding of needs that involve more effort and resources.

Firstly, this report focused on presenting the current state of the art related to customer needs emphasizing, in Chapter 2, the interdisciplinary that their discovery requires. Secondly, a complete framework aimed at customer needs discovery, which is based on the presented literature, has been proposed in Chapter 3 with the purpose to solve the technical problem presented Chapter 1.

Bibliography

[Alliance 2009] Smart Card Alliance. *Security of Proximity Mobile Payments.* A Smart Card Alliance Contactless and Mobile Payments Council White Paper, Publication CPMC-09001, 2009.

[Bayus 2008] Barry L Bayus. *Understanding customer needs.* Handbook of Technology and Innovation Management, Edited by Scott Shame, Chp, vol. 3, pages 115–141, 2008.

[Brunswik 1952] Egon Brunswik. The conceptual framework of psychology, volume 1. Univ of Chicago Pr, 1952.

[Burchill 1997] Gary Burchill and Christina Hepner Brodie. Voices into choices: Acting on the voice of the customer. Oriel Incorporated, 1997.

[Christensen 2002] Glenn L Christensen and Jerry C Olson. *Mapping consumers' mental models with ZMET.* Psychology & Marketing, vol. 19, no. 6, pages 477–501, 2002.

[Coskun 2013] Vedat Coskun, Busra Ozdenizci and Kerem Ok. *A survey on near field communication (NFC) technology.* Wireless personal communications, vol. 71, no. 3, pages 2259–2294, 2013.

[Coulter 1995] Robin Higie Coulter and G Zaltman. *Seeing the voice of the customer: Metaphor-based advertising research.* Journal of advertising research, vol. 35, no. 4, page 35, 1995.

[Dahan 2002] Ely Dahan and John R Hauser. *The virtual customer.* Journal of Product Innovation Management, vol. 19, no. 5, pages 332–353, 2002.

[Desmet 2001] Pieter Desmet, Kees Overbeeke and Stefan Tax. *Designing products with added emotional value: Development and appllcation of an approach for research through design.* The design journal, vol. 4, no. 1, pages 32–47, 2001.

[Franke 2004] Nikolaus Franke and Frank Piller. *Value creation by toolkits for user innovation and design: The case of the watch market.* Journal of product innovation management, vol. 21, no. 6, pages 401–415, 2004.

[Friedman 2004] Lawrence Friedman. *Method for a host based smart card*, December 29 2004. US Patent App. 11/025,495.

[Gaskin 2010] Steven P Gaskin, Abbie Griffin, John R Hauser, Gerald M Katz and Robert L Klein. *Voice of the Customer.* Wiley International Encyclopedia of Marketing, 2010.

[Geistfeld 1977] Loren V Geistfeld, George B Sproles and Suzanne B Badenhop. *The concept and measurement of a hierarchy of product characteristics.* Advances in consumer research, vol. 4, no. 1, 1977.

[Green 2001] Paul E Green, Abba M Krieger and Yoram Wind. *Thirty years of conjoint analysis: Reflections and prospects.* Interfaces, vol. 31, no. 3_supplement, pages S56–S73, 2001.

[Griffin 1993] Abbie Griffin and John R Hauser. *The voice of the customer.* Marketing science, vol. 12, no. 1, pages 1–27, 1993.

[Herzberg 2011] Frederick Herzberg, Bernard Mausner and Barbara Bloch Snyderman. The motivation to work, volume 1. Transaction Publishers, 2011.

[Holtzblatt 1993] Karen Holtzblatt and Hugh Beyer. *Making customer-centered design work for teams.* Communications of the ACM, vol. 36, no. 10, pages 92–103, 1993.

[Kano 1984] N. Kano, S. Tsuji, N. Seraku and F. Takahashi. *Attractive quality and must-be quality.* Hinshitsu: The Journal of Japanese Society for Quality Control, vol. 14, pages 39–48, 1984.

[Kaul 1995] Anil Kaul and Vithala R Rao. *Research for product positioning and design decisions: An integrative review.* International Journal of research in Marketing, vol. 12, no. 4, pages 293–320, 1995.

[Leonard 1995] Dorothy Leonard. *Wellspring of knowledge.* Harvard Business School Press, Boston, MA, 1995.

[Masten 2003] Davis L Masten and Tim MP Plowman. *Digital ethnography: The next wave in understanding the consumer experience.* Design Management Journal (Former Series), vol. 14, no. 2, pages 75–81, 2003.

[Mazur 2003] Glenn Mazur. *Voice of the customer (define): QFD to define value.* In Proceedings of the 57th American Quality Congress. Kansas City, pages 1–7, 2003.

[Mello 2003] Sheila Mello. Customer-centric product definition: The key to great product. PDC Professional Publishing, 2003.

[Otto 1998] Kevin N Otto and Kristin L Wood. *Product evolution: a reverse engineering and redesign methodology.* Research in Engineering Design, vol. 10, no. 4, pages 226–243, 1998.

[Penning de Vries 2010] Rene Penning de Vries. *NXP in the making: The world's first HPMS company*, 2010.

[Pullman 2002] Madeleine E Pullman, William L Moore and Don G Wardell. *A comparison of quality function deployment and conjoint analysis in new product design.* Journal of Product Innovation Management, vol. 19, no. 5, pages 354–364, 2002.

[Saaty 1990] Thomas L Saaty. *How to make a decision: the analytic hierarchy process.* European journal of operational research, vol. 48, no. 1, pages 9–26, 1990.

[Sanders 1992] Elizabeth B-N Sanders. *Converging perspectives: product development research for the 1990s.* Design Management Journal, vol. 3, no. 4, pages 49–54, 1992.

[Schmitt 1999] Bernd Schmitt. *Experiential marketing.* Journal of marketing management, vol. 15, no. 1-3, pages 53–67, 1999.

[Shillito 2000] M Larry Shillito. Acquiring, processing, and deploying: Voice of the customer. CRC Press, 2000.

[Swaddling 1996] J Swaddling and M Zobel. *Beating the odds.* Marketing Management, vol. 4, no. 4, pages 20–33, 1996.

[Ulrich 2004] Karl T Ulrich and Steven D Eppinger. *Product design and development.*, 2004.

[Urban 1993] Glen L Urban, John R Hauser and Glen L Urban. Design and marketing of new products, volume 2. Prentice Hall Englewood Cliffs, NJ, 1993.

[Von Hippel 1999] Eric Von Hippel, Stefan Thomke and Mary Sonnack. *Creating breakthroughs at 3M.* Harvard business review, vol. 77, pages 47–57, 1999.

[Yang 2007] Kai Yang. Voice of the customer: Capture and analysis: Capture and analysis. McGraw Hill Professional, 2007.