Fault attacks on Java Card

an overview of the vulnerabilities of Java Card enabled Smart Cards against fault attacks

Gadellaa, K.O.

Award date:
2005

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER'S THESIS

Fault Attacks on Java Card

An overview of the vulnerabilities of Java Card enabled
Smart Cards against fault attacks

by
K.O. Gadellaa

Supervisors:

dr. E.P. de Vink

Eindhoven, August 2005

Abstract

This thesis gives a wide overview of the problems of the Java Card technology regarding fault attacks. It uses fault attacks on RSA as an example, and shows how RSA can be broken in various ways. It then gives an overview of the different defense strategies against fault attacks, and how these are applicable against the various means of fault injection.

Preface

This document presents my master thesis for Technische Informatica at the Eindhoven University of Technology. The research and work was done between november 2004 and august 2005 within the Formal Methods group of the Department of Mathematics and Computer Science. My supervisor was Dr. Erik de Vink. Other members of the committee were Dr. Jerry den Hartog, and Dr. Rudolf Mak.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter discusses the goals as were set out when starting this research, and the structure of the document.

1.1 Goal definition

The goal of this project is to identify vulnerabilities of Java Card applications with respect to differential fault attacks. This concerns both the reconstruction of fault attacks reported in the literature, as well as the simulation of differential fault attacks. The project results in a Master thesis and a demonstrator.

The planning of the project distinguishes five phases:

- Obtaining hands-on experience with the OCF Framework. Deliverable: working Internet Broker Demo (or similar).
- Getting acquainted with Java Card and the JCOP Toolset by implementing small Java Card programs. Deliverable: running generic purse application (or similar).
- Literature study on fault attacks, including differential fault attacks, electromagnetic side-channels and fault injection, power glitching. Deliverable: discussion of fault attacks in general and description of fault attack vulnerabilities of Smart Card in particular.
- Vulnerability analysis of Java Card applications regarding differential fault attacks based on the reconstruction of known fault attacks. Deliverable: report on the reconstruction and simulation of known fault attacks for Java Card.
- Identification and validation of countermeasures and robustness of Java Card regarding differential fault attacks. Deliverable: report on possibilities for DFA hardening.

Apart from a discussion of the above issues the final report will include an overview of the fault attacks that are relevant to Java Card and the implications thereof for the Java Card platform. Also a demonstrator or simulation will be built that illustrates the fault attack vulnerabilities discussed. Java Cards, and fault analysis, and a discussion whether it is actually viable to do fault injection on a Java Card will be presented

1.2 Structure

To apply to the stated goals, this document document is set up as follows: First, in chapter 2, an introduction to Smart Cards is given which flows into the general principles of programming software which uses a Smart Card, and then into programming software which can reside on a Smart Card. From there, it discusses the vulnerabilities Java Cards have with respect to fault injection in chapter 3, and then shows a few practical fault attacks on RSA in chapter 4. The next chapter then discusses the countermeasures and their quality, and the document finishes with a small chapter in which the experiences with Java Card are discussed, and the security of Java Card.

Chapter 2

Smart Cards

Smart Cards are said to be used for new, advanced appliances. Everyone should have one, it is said. This chapter digs into what Smart Cards essentially are, how they function, how it is possible to program applications using them, and what options exist for writing applications which can run on Smart Cards.

2.1 Smart Card technology

A Smart Card is a chip put on a piece of plastic. At first these were only "memory cards", holding only data, but with the ongoing miniaturization of electronic devices they quickly matured into small computers. Although developed and patented in 1970, even nowadays Smart Cards have limited capability. For example, some 25 years later, Smart Cards still only worked at 3 MegaHerz, and had 4 KiloBytes of memory. Around 2000, this had risen to 50 MegaHerz and 500 KiloBytes. Compare that to the personal computer which ran at 1000 MegaHerz at that time and it is clear that these machines are working on a different scale then normal personal computers.

As a portable computer, these Smart Cards can have a multitude of applications. And since they are designed to withstand tampering, they can be used as a secure token (similar to a key, but then digital), or as a signing mechanism, similar to a (digital) seal like an autograph. They are thus used in authenticating a person, authorizing access to places, authorizing payments, and also as temporary means to do these things ("you only have access x times", ticketing purposes, etc.). Of course, Smart Cards have been around for a while and most people have seen (or have) one. A large number of creditcards and bankcards have one. European SIM cards, used in mobile cellphones, are Smart Cards. A lot of Smart Cards are used for authentication, replacing the magnetic "sweep-through" card by simply holding it in or near a digital lock. All of these are examples of Smart Cards used in today's society.

Typically, the computer on a Smart Card itself is very small and the chip has a recognizable, gold-colored area which is shown in figure 2.1. This gold area is actually the entire chip.

To (physically) connect to the chip, there is some terminal or Card Holder Device, which is the device which actually connects to the Smart Card. Any communication with the Smart Card by a program thus first needs to connect

Figure 2.1: Sample Smart Card

to the terminal, and from there have the terminal relay the data to the Smart Card.

Important to note is that the chips themselves do not necessarily have a single uniform command set for the on-card instructions. Different cards have different commands implemented, with different names, and in different ways. It is thus in a way similar to the early years of the PC, when there were a multitude of vendors, each with different chips.

2.2 Writing programs which use Smart Cards

The usual flow of information for a program which uses a Smart Card is from program, to computer, to terminal, to Smart Card, to on-card application. All of these steps require communication. Program to computer, and Smart Card to on-card application are not a programming issue as these are already available. And as the communication between terminal and Smart Card is standardized (by ISO7816), this section describes the communication between the Smart Card and the terminal, and the terminal and the program, which can then be used to write programs which use a Smart Card.

Initially, when writing a program which used a Smart Card, the first hitch to overcome was the terminal: there are a multitude of terminal vendors, and each had different standards. Thus, programming for a Smart Card required first deciding which terminals would be supported. This lasted till 1997, when the PC/SC workgroup [8] (a consortium of terminal vendors led by Microsoft) set up the PC/SC standard. This standard then gave a universal interface for communication between the off-card program and the terminal.

The second hitch which programmers faced was the Smart Card itself: each and every vendor sold different types of Smart Cards with different possibilities, and, moreover, different standards. This was amended as well by the OpenCard consortium, which created the Open Card Framework, or OCF. This framework implemented an intermediate communication layer between on-card application and off-card program to which card vendors could write plugins (services) defin-

ing the card applications which the card implemented, and also laid down a basic filesystem, a means to access applications, and signature systems for signing, key creation and the import/export of the keys. Similarly, it also allowed terminal vendors to write plugins. A PC/SC compatible Terminal Service(plugin) was available from the start. Now writing a program which used a Smart Card was a lot easier — only a check whether the terminal was OCF or PC/SC compatible, and using an OpenCard compatible Smart Card would ensure that the OpenCard Framework would work as a dependable platform to program on.

2.3 Writing programs on Smart Cards

Initially, Smart Cards were not "programmable": only applications which were added by the card manufacturer could be used. No new application could be added. Of course, this limitation was later overcome by having some means of loading and executing arbitrary applications on Smart Cards, but then it meant having to program for a specific Smart Card. Programming for a Smart Card was similar to systems programming, i.e. it was processor-dependent. This would mean programming for a single Smart Card Processor. It thus created a dependency on the processor, which meant shutting out a host of Smart Cards, and possibly giving issues when supply of the specific Smart Card was a problem. Enter two platforms: Java Card, and MultOS. Similar in architecture but designed by different groups, both delivered some sort of platform for which applications could be written. Smart Cards which were JavaCard or MultOS capable would then be a uniform platform; programming an application for Java Card and any Java Card enabled device would suffice to execute the application on.

2.3.1 Java Card

Java Card is a type of Smart Card which supports a limited set of the Java Virtual Machine. The idea is to be able to run Java bytecode on a Java Card. For an in-depth working on what a Java Card is, see [2]. Java Card brings all the advantages of the Java runtime environment to the Smart Card, in particular:

portability The code is abstracted from any Smart Card-dependent features.

security The Java Runtime Environment ensures that the Java security model is upheld.

development tools The on-card application can be developed in already known environments for the Java language.

It is a very limited subset of the Java Runtime Environment, however. In particular, there are only a few features supported and quite a few which are not, thus forcing optimization by the programmer:
Supported:

- a few primitive data types: `boolean`, `byte`, `short`
- one dimensional arrays
- Java packages, classes, interfaces, and exceptions structure

- object oriented features: inheritance, virtual methods, overloading and dynamic object creation, access scope and binding rules

Optionally supported:

- the `int` datatype. For portability it thus depends if this is required. Different cards may or may not support the `int` datatype.
- garbage collection and finalization

Unsupported:

- large primitive data types: `long`, `double`, `float` this means that any big calculation should not be done on the card, which is as it should be.
- characters and strings: as the card has no direct output it is not necessary to work with strings, or characters. This, again, necessitates the use of these types to the off-card application.
- multidimensional arrays, for optimization purposes and keeping the instruction set small,
- dynamic class loading, also for optimization purposes,
- threads, as Smart Cards are not multi-tasking,
- object serialization, as there are no threads, this is logical
- object cloning
- the security manager object, left out as the JCRE Firewall should give adequate protection.

In particular, the above means that, if `int` is not supported, almost every calculation should be cast to `short`'s (continually) as arithmetic calculations are done in `int`'s (or 32-bit registers) . This *is* a bit of a fuss, as can be seen in the code example in Appendix C, where in figure C.1 on line 15, 16, and 20 a typecast to `short` is done. Also, look at figure 4.2 where any use of number literals needs to be preceeded by typecasting to `short`.

A sample picture of the Java Card Architecture as presented by the developers can be seen in figure 2.2. It shows applets being compartmentalized by the firewall. These then use API's added to the Smart Card by the vendor and the Java Card Framework API. It all runs on the Java Card Virtual Machine. The Java Card Virtual machine then runs on the Java Card Open Platform Operating System, thus creating a secure runtime environment. The Java Card Runtime Environment then runs the Smart Card. Also, it usually does not have any filesystem, so as to ensure that data is kept inside the application only.

To have programs executed by the (embedded) Java Card Virtual Machine the normal .class files need to be converted to .CAP (converted applet) files, so they can be run through the bytecode interpreter. The converter optimizes the code for the Smart Card in a number of ways:

- verifying that the load images of the classes are well-formed
- checking for Java card language violations

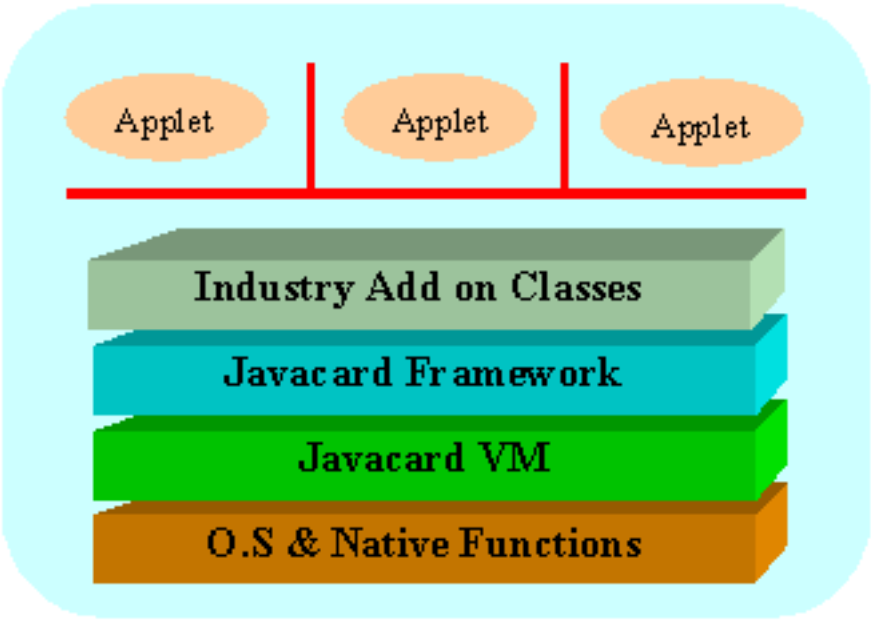


Figure 2.2: Java Card Architecture

- initializes static variables
- resolving symbolic references to classes, methods an fields to put them in a more conpact form for efficiency
- optimizing bytecode with the gained information by classloading and link-time
- allocating storage and creating data structures for classes.

After the converter is finished, it leaves a file in .cap file format for each package.

.cap file format explained

The generic .cap file is actually a .jar file bundling of several .cap files (or components) together. There are 12 components:

Header Component contains information of the entire package.

Directory Component contains the sizes of each of the components

Import Component describing the set of packages required for this package

Export Component contains a mapping from all public methods and fields to their concrete implementations

Method Component has the bytecode for each method in this .cap file.

Class Component has all the information for class checking.

Descriptor Component describes all variables so that they can be parsed and verified. It thus has the access flags(`public`, `private`, `final`, etc.) for all the variables in the package.

Static Field Component contains everything for static fields and initialization procedures for classes.

Reference Location Component contains the relative offsets in the Method.cap between calls by the Method.cap's instructions (i.e. function calls) to items in the constant pool.

Constant Pool Component contains a mapping from every reference in the Methods Component to a class, method, or field (as appropriate).

Applet Component describing the applets in this package (if any)

Debug Component for helping out with debugging — normally this is not uploaded to the card.

As can be seen, the .cap file keeps different components, each with its separate function. Quite a few are interlinked with each other.

Next to creating a .cap file, the converter also creates an export file which contains the linking information. This is not needed for the execution of the program, though. It is only used as a header with linking information, to be used when linking files together. For example, when using a library from another package, the export file is necessary to complete the linking.

After the converter is finished, the .cap file can then be loaded on to the Java Card and after installation the application can be run on the JCRE.

The Java Card Runtime Environment Explained

The Java Card Runtime Environment is the environment which should ensure that the code is executed appropriately, and any other command such as `select` and `deselect` for the selection and deselection of applets are handled in such a way that there is no room for malicious intent. To ensure that the context in which applets run are separate, it is required of the JCRE that it implements the isolation of applets . This means that one applet can not access the fields or objects of an applet in another context unless the other applet *explicitly* provides an interface for access. This is done in two steps:

The Applet Firewall ensures that all applets execute in isolation of each other.

Object Access across Contexts Enabling is done to ensure that interoperability between different applets is possible.

Applet Firewall

Every Java Card applet is run in a specific context. Its context is the package it is in. When an applet calls methods from other packets (or contexts), a context switch occurs and a Remote Method Invocation is done. In this way, all the applets are "boxed in" and should not be able to interfere with each other. A simple check for each instruction if it is within the correct context can then ensure that the firewall is upheld.

Object Sharing across Contexts

An object can only be accessed by its owning context, as the firewall prevents access by another applet in a different context. To have some means of passing data from one applet to another there are three methods to have data between contexts. One, the JCRE Entry points, is for system communication, so either Global Arrays or Shareable Interfaces need to be used.

Global Arrays need to be defined by the vendor, and the only ones which need to be defined are the APDU buffer and the byte array input parameter to the applet's `install` method. Normally, no other Global Arrays are provided.

The `Shareable` Interface defines a set of shared interface methods. When Applet A wants to access a Shareable Interface Object from context B, it requests it from B. B then should have code to see if applet A is to be granted permission or not, and if so, the object is shared.

Lastly, as said, there are JCRE Entry Point Objects, and JCRE Privileges: The JCRE can invoke a method of any object on the card, and any "system call" is handled by a JCRE Entry Point Object which verifies if the the call to the Object's method is correct.

Tools

To use the Java Card platform efficiently, a number of tools are available. The Java Card toolset provided by SUN includes the ability to convert .class files to .cap files, and the programming API for Java Card, thus enabling simple programs to be created. For testing and programming, IBM has developed the JCOP toolset, which includes a terminal emulator, a Java Card emulator, and an easy way to upload applications to Java Cards. The IBM JCOP toolset requires the use of Eclipse (an open source development environment). For the setup of these, see Appendix B.

2.3.2 MultOS

MultOS was the first multi-application Smart Card. Similarly to the Java Card, it has a virtual machine. However, where Java Card has as a virtual machine a subset of the Java Virtual Machine, MultOS has its own language, the MultOs Emulator Language, or MEL. MEL is similar to the Java bytecode in a sense that both are low-level, and are executed by a virtual machine (called the MultOS Application Abstract Machine on a MultOS card). MultOS also has a filesystem, which is a subset of ISO7816, and keeps applications separate as well with some sort of firewall. More on MultOS can be found here [36]. However, the website itself seems not too well maintained from time to time as not all links work properly.

The architecture for the MultOS card is similar to that of Java Card: again, applets are compartmentalized by a firewall. The applets run on a virtual machine, which in turn runs on the Operating System which has the loading capabilities, and runs on the Smart Card.

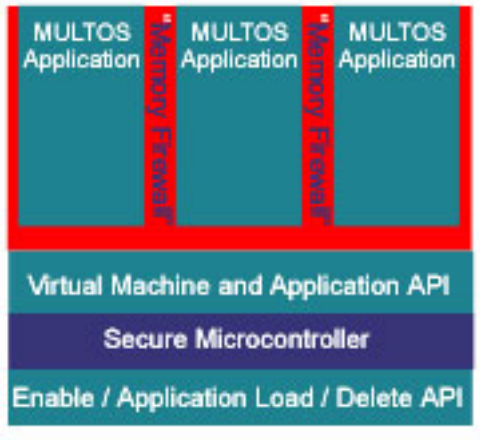


Figure 2.3: MultOS Architecture

It can be seen that this architecture is very similar to the one for Java Card presented in figure 2.2.

Tools

Similar to the toolset for Java Card, MultOS has compilers (compiling C or Java to MEL), a MultOS simulator to simulate a MultOS Card, terminal simulator to simulate a terminal with a MultOS card, and a loading facility to load Application Load Units, the format in which MultOS applications can be loaded on a MultOS compatible card.

2.4 Summary

We now have seen the technologies involved: what are Smart Cards, how do you make programs using a Smart Card, and how do you program for a Smart Card. The next two chapters look at the weaknesses these Smart Cards have.

Chapter 3

Weakness: Inducing Faults

A Smart Card needs to protect against numerous things. Where a traditional computer needs secure protocols to ensure that no data is leaked, a Smart Card is a computer in a hostile environment. Similar to a safe, it needs to ensure that no secret data can be learned by for example putting an ear to the safe and listening what happens inside when you turn the lock. These so called "side channels" will not be discussed here.

Next to protocol attacks, Smart Cards add an extra dimension to protect: the computer needs to be physically safe from tampering. Again, comparing to a safe, it should not be possible to open the safe and look what's inside the safe. It has to withstand a number of techniques which directly attack the processor or its memory. For example, by freezing the chip so that it retains its memory information and then reading memory somehow is something which normal PC's not commonly have to protect against. Similarly, by placing a voltage meter on the memory bus, it might be possible to learn the value of private keys whilst they are moved in memory. A Smart Card, however, is at the mercy of its attacker if not properly protected.

However, it is also important that when an error occurs during the execution of the program, no program requirements are violated — i.e. no secret information is exposed, no permissions are unduly granted, etc. An attack requiring a fault to occur is called a *fault attacks*. Errors almost never occur naturally, as the program are usually rigorously tested and hardware is expected to be "perfect". Because of this, programs are often not protected against errors. Assuming however that such an error exists, it needs to be controlled: when there is a chance of one in a billion that a fault occurs, it might be infeasible for an attack to use the protocol until such an error occurs. For a Smart Card, however, the computer can be in the hands of the attacker. The attacker could of course try to inject a fault himself. A real-life example of this were "unloopers", in which Smart Cards which granted access to pay-TV channels were injected a fault so that a single branching statement was effected: instead of denying access, the branch giving access was chosen [15].

Thus, this chapter looks into the means to inject a fault in a Java Card applet. To structure this, we model a Java Card enabled Smart Card as a number of layers. The following figure gives a simple view of the three considered layers of the Java Card enabled Smart Card and are thus a bit of an abstraction of the picture in figure 2.2.

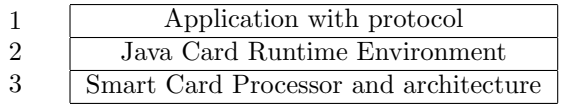


Figure 3.1: layers on a Java Card enabled Smart Card

An attack on any of these layers might potentially create a malfunctioning Smart Card. We here thus discuss each of the three layers.

3.1 Attacking the Application

This section concerns the possibilities of injecting a fault through the application on itself. We first show how the security on the application level is handled. Any possible attacks due to interleaving should be handled by the JCRE.

A quick glance at Application code shows that code generated for Java Cards goes through a number of steps:

1. Conversion to .class files
2. Conversion to .cap files
3. loading onto the card
4. execution on the card

All of these depend on an amount of trust. The original files should be programmed correctly, else the compilation to class files leaves untrustworthy code. Similarly, the class files should remain correct and unmodified when being converted to a cap file, and it should be exactly these files which are loaded onto the card. These should then be the same files which are executed. Throughout this process, the files should remain trusted. To establish this are various methods: signed certificates implying the trust of the one delivering the class or cap files, conforming to GlobalPlatform standards (see also Chapter 5.4) for loading code onto the card securely, and lastly, hardware measures to ensure that the execution of the code is tamperproof.

However, per the specification [19] it is not required to have any verification on the .cap file during upload to assure it is of the proper format, as this could be too heavy a task for a Smart Card (or any other Java Card enabled small device). It is suggested however, that at least some checking should be implemented. Assuming however that in some way a piece of (untrusted) code was uploaded to the card, the execution of the code could then only be performed on the card when the card is running. When untrusted code is executed, however, it still has to pass the runtime checks done by the Java Card Runtime Environment. There are numerous papers describing runtime checking specifically for Java Cards [20, 21, 23, 22, 24], so it can be assumed that recent cards have such checking implemented even though this is not mandatory.

Even so, it is clear that there is a distinction between attacking the .class file, the .cap file, or when it is actually 'on-card'. We assume that attacks on a .class file are not very often to occur as these files are kept in a very trusted environment. However, the .cap file needs to be uploaded in a possibly

hostile environment: during the upload it might be modified by modulating the current, when using wireless Smart Cards the transmission might be disrupted, or, when receiving the file from an application provider or card issuer it can be changed directly. We thus make a distinction between on-card attacks and off-card attacks: the one modifies the installed file, the other the .cap file. Furthermore, when modifying the application to actually produce a fault, the attack is not transient: any further use of the program continues to behave erroneously as the error persists. Yet when the ability is there to continually upload (correct) versions of the application, the problem can be circumvented by replacing the code. So the injected errors are not transient, but neither are they persistent. We will thus call them *semi-persistent*.

3.1.1 Off-card Attacks

By manipulating (or introducing errors) into the file which contains the bytecode before it is uploaded to the card, a protocol can be circumvented or broken. To do this, one needs both knowledge of the format of the file, knowledge of the program in question, and the ability to modify the file. The format is publicly available (see also chapter 2.3.1). As can be seen from this is that the size of an individual component cannot change without also having to change the dictionary. Secondly, as a few components work with offsets of each other this means that changing one component can have side effects, meaning that large modifications in one part can require modifications in the rest of the .cap file as well.

Modifying the .cap file

The .cap file format does not enforce any encryption. It is thus possible to modify the .cap file directly. This, then, can lead to incorrect code being executed on the Java Card. However, even simple checks can ensure that not a lot of modification is possible. Actually inserting code can only be done by both inserting the code, and adjusting the rest of the .cap file so all the Components match correctly. This is thus more then a simple modification. Small modifications are still possible, however. For example, a simple piece of code calculating 4+5 can be easily modified to calculate 4/5, or 4+9, as long as the instructions remain properly executable. Such small modifications have no impact on the rest of the .cap file and can thus be done without any complication.

Other uses

In [25], a method is given to pinpoint specific function calls, by having known output both before and after the call. By using these it is much easier to set up the correct timing which is might be required for fault attacks. Thus modifying the .cap file allows an attacker to gain better knowledge as to *when* a transient fault should be injected.

Feasibility

It is noteworthy that modifying the .cap file requires very precise control. This thus means that the attack is harder to implement although, again, if it is

possible to re-load the application it becomes much easier as "trial and error" should do the trick.

3.1.2 Modifying the on-card code

As it is unspecified where exactly the code on the card resides this means that wanting to modify the code includes having to have an entire memory mapping of the card. At least knowledge of how data is stored in memory and knowing where the applications resides are mandatory. If this is possible then probably much better attacks are suitable. Yet it can be assumed that the same attacks can be mounted as were done in 3.1.1 as the capabilities are similar; only the means are slightly different depending upon how the JCRE is implemented.

3.2 Attacking the Java Card Runtime Environment

This section concerns the Java Card Runtime Environment and the possibilities of using it to inject a fault. For a quick overview of the Java Card Runtime Environment, see chapter 2.3.1.

3.2.1 Java Card Runtime Environment issues

This subsection gives some pointers to other articles in which issues with the JCRE are discussed.

The firewall

In [28] a means is given to modify an arbitrary piece of memory. It roughly goes as follows:

Assume a class A with a reference to class B. Now let there be a fault injected in the reference in A, so instead of pointing to B, it points to another instance of A. Now the static type checking is circumvented, and there is a type flaw. To exploit this, consider the following piece of code in figure 3.2.

Now let there be an error in the reference to b1 such that it points to the (value) of the reference to b2 in a2. a2 now no longer has an exact reference to b2. It is now possible to access and write to any value in the memory by having a1 define the offset, and a2 write the data by doing `a1->ref_b = value` for the offset, and similarly, `a2->ref_b = value` for the value it should have.

This method has been tested on some JRE's and found working. However, to work well on a JCRE, it requires a lot of memory as the easiest way to ensure that such an exact fault occurs is by having the memory full of such structures, ensuring that there is a high chance of actually a faulty reference pointing to the value of the reference to another class. To be able to aquire a lot of memory it is thus only really viable if there is garbage collection available which is not required for Java Cards (as said, it is optional). It can then be used to modify the memory, thereby circumventing the firewall and writing into the memory of another applet.

```
01 Class A{
02   B ref_b;
03 }
04
05 Class B {
06   short val;
07 }
08
09 a1 = new A();
10 a2 = new A();
11 b1 = new B();
12 b2 = new B();
13 a1->ref_b = b1;
14 a2->ref_b = b2;
```

Figure 3.2: Sample code which can circumvent the firewall

Object Sharing

In [27] the Object Sharing across Contexts method is examined. It shows a number of issues:

AID impersonation It is possible to create and upload an applet which has the exact same AID as an existing applet, thus making a mix-up of the two possible where the incorrect applet is used.

Access to all Interface Methods of a Class By having knowledge of the class, it is possible to typecast and receive access to all interface methods, even if they are not granted access to. For example, if a class implements different interfaces, access to the class through one interface can then be typecast to get access to the other implementing interfaces.

Especially AID impersonation could be used to impersonate applications, thereby possibly using a user-uploaded version of applications instead of those already installed. This, then can break access to other applications as these use AID checks for access control.

3.3 Attacking the Smart Card Processor

This section discusses the various means to attack a Smart Card and its architecture to inject a fault. All of the attacks below are hardware attacks, and as such have not been actually performed. It was mostly retrieved from various research articles, although [12] was the primary source. Some were performed in a controlled environment in [29], so they are not theoretical at all. Moreover, Smart Cards used by PayTV were regularly broken using fault attacks [26].

To get an idea of how faults can be introduced, this section gives an overview of the various types of attacks. Attacks are also given some measure of "invasiveness" indicating the amount of tampering necessary. Sometimes direct access and contact with the chip is necessary. These are considered invasive. If no contact or direct access is necessary, the attack is considered as non-invasive.

Similarly important is the amount of control available. Some attacks allow complete control where the fault injection happens whereas others do not.

3.3.1 Spike Attacks

By varying the power fed to the chip it is possible to disrupt a computation. In some cases, this can be enough to introduce a fault [15]. A power spike can vary in some 9 different variables, including height, shape, build up and power down, duration, etc. As a spike works by simply connecting to the power led to the chip, they do not need direct access to the chip and are thus non-invasive. Equipment requirements are totally dependent on the type of the spike to be generated, but are not necessarily expensive.

3.3.2 Glitch Attacks

Similar to spikes, it is sometimes possible to disturb the clock speed. For example, by doing an update cycle at double speed some instructions will be effected where others are not, so it is possible that old data is used as the new data hasn't arrived yet [14]. The introduced "glitch" can be used to influence conditional jumps(by not performing them, or performing them when it is unwanted, etc. [16]), a shift register shifts twice (instead of once), or not at all, etc. In general it is a wide class which can create a change in the program, and depending on that change an attack can be mounted. Similar to Spike Attacks, Glitch attacks are thus noninvasive and do not require any expensive equipment.

3.3.3 Optical Attacks

By using focused light with specific wavelengths, it is possible to change the flipflops in a memory cell. By doing this it is thus possible to change or modify the memory by using the photoelectric effect. These attacks require light to be able to reach the chip, and thus that any protective coating needs to be removed. However, as no contact is needed these attacks are generally seen as semi-invasive. As for equipment, in [17] it was shown that attacks can be done relatively cheaply with simple equipment, and also that they can be very precise, effecting only a single bit. However, they need not be precise at all.

3.3.4 Electromagnetic Perturbation Attacks

By creating a strong electromagnetic source near memory the ions representing the states in the memory are moved around, and thereby the memory is disturbed. It is claimed in [18] that a so called "eddy current" can be created using an active coil with sufficient strength. This then gives a fine control of exactly what bit needs to be controlled. With that claim also comes the claim that it can be done relatively cheap. As this attack only requires to be near the processor, it need not be opened up as it can be done from outside. It is thus a non-invasive attack.

3.4 Classification of different attacks

To get some feeling for what the different types of attack can accomplish, the attacks are classified in the following categories:

Control : The amount of control the attack has on where exactly the fault occurs

Fault : The fault model which it can produce. This can be either be:

- **bf** : for Bit Flip model, in which a specific bit can be flipped,
- **saf** : for Stuck at Fault model, in which (a number of) connections can be permanently disabled,
- **bsr** : for Bit Set and Reset model, in which a specific bit can be set or reset
- **random** : indicating that it is unknown what happens but that something went wrong
- **depending** : for the different glitch attacks

#faults : indicating the number of faults generated by a single attempt.

The attacks then lead to table 3.1.

Attack	Control	Fault	# faults
.cap modification	complete	saf, bsr, bf	as required
on-card modification	complete	saf, bsr, bf	as required
Firewall circumvention	random	random	random
Spike Attack	none	random	random
Glitch Attack	depends	depends	depends
Optical Attack	complete	bsr, bf	as required
Eddy Current Attack	complete	saf, bsr, bf	as required

Table 3.1: Characterization of the different forms of physical attacks

Both AID impersonation and the ability to access all interface methods to not necessarily inject a fault in the program. It might cause inconsistency which might lead to knowledge being learned, however.

Next to seeing what can be accomplished by each different attack, it is also important to look at what the requirements are to inject a fault. In table 3.2 is a table specifying what requirements must be met to be able to implement the attack, and the type of fault produced: semi-persistent, transient, and in the case of Firewall circumvention this is random as it is unknown what is modified — it could be code, but it could also be a variable.

Attack	Requirements	Type
.cap file modification	ability to modify specific bytes access to .cap file	semi-persistent faults
on-card modification	ability to modify specific bytes access to on-card memory	semi-persistent faults
Firewall circumvention	JCRE with problems garbage collection fault in object	random error in memory, although possibly very precise
Spike Attack	ability to manipulate power	transient fault
Glitch Attack	ability to modify clock speed	transient fault
Optical Attack	unprotected access to the chip	transient fault
Eddy Current Attack	physical closeness to chip	transient fault

Table 3.2: Overview of requirements and impact of attacks

Chapter 4

Using Faults: Fault Analysis

This chapter looks into the possibilities an attacker has to retrieve secret data from the Smart Card. However, as this field is broad (ranging from reading memory during runtime to Power Analysis), this chapter (and research) will only relate to Fault Analysis, a means to get data from chips by having a chip malfunction in some way. Quite a few of these have actually been tested in laboratory conditions and found working as predicted [29], which shows that Fault Attacks are real and should not be dismissed as theoretical.

The way Fault Analysis works is often protocol related. For this, the research done was on RSA. Thus first a short introduction into the working of RSA is given, and then several examples of Fault Analysis.

4.1 RSA

RSA is a protocol devised in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adleman in [10]. The protocol is an asymmetric key protocol, so there is a private and a public key, and any message encrypted with the private key can only be decrypted with the public key (and any key encrypted with the public key can only be decrypted with the private key). The structure between these keys is as follows:

1. Take two (large and relatively equal size) primes p and q, and let $n = pq$
2. Choose an encryption key e and a decryption key d such that $ed = 1 \bmod ((p-1)(q-1))$
3. To encrypt a message m, the sender can now compute $c = m^e \pmod{n}$. To decrypt a ciphertext c, the receiver computes $c^d \bmod n$. Due to the mathematical structure, this is then again the original message m.

For sufficient security p and q should be something like 2048 or more bits for now [40]. This shows that the calculations for both encryption and decryption are very slow and can take a long time. Thus typically two optimizations are used to speed up the exponentiation. Both of them can be used to do an effective fault attack on RSA.

4.2 Fault Analysis

In 1996, three researchers at Bellcore labs [9] wrote about a method to retrieve information from protocols which should supposedly be secure. They did this not by attacking the way the protocol interacts, but by attacking the calculations done by the protocol. This would be possible by having the protocol work under conditions where a fault may be injected in the calculations for the protocol (which later was found to indeed be possible). By doing this, various information can be learned which should stay hidden. All of their attacks assume the possibility to "flip" a bit: either from zero to one, or from one to zero, possibly with conditions such as "it only flips to zero if it is one. If it is not, then nothing happens". Depending on the attack, various assumptions are then made on where this flip occurs. Fault Attacks in general rely on being able to manipulate the program, either by modifying values (bits, or bytes), or by changing the flow of the program (by making it skip certain instructions). Most of the time these concern a certain specific variable at certain specific positions in the protocol. Analyzing the returned value and knowledge of the fault can then lead to knowledge of the (hidden) parts of the protocol. By doing so the strength of the protocol is weakened to the point that it can be broken without considerable effort.

There are a number of fault attacks on RSA using Fault Analysis. In the original article [9] two were given. These work on optimizations of the RSA protocol. These will be given here, and were implemented for this thesis work on a Java Card. We thus first give an overview of the RSA protocol.

4.2.1 Fault Attack on RSA + CRT

By using the Chinese Remainder Theorem the calculation in RSA can be divided in smaller calculations, as follows:
$S = x^d \pmod{pq}$ is done by using

$$x^d = CRT(S_1, S_2) \pmod{pq} \text{ where}$$

$$S_1 = x^d = x^{(d \bmod (p-1))} \pmod{p}, \text{ and}$$

$$S_2 = x^d = x^{(d \bmod (q-1))} \pmod{q} \text{ accordingly.}$$

The function $CRT(\alpha, \beta)$ can be defined as $a\alpha + b\beta$ with $a = 1 - p_q^{-1}p$ and $b = p_q^{-1}p$

p_q^{-1} here is p under a field of size q, such that $p \cdot p^{-1} = 1 \pmod{q}$
This thus roughly reduces the number of multiplications necessary for the exponentiation. However, as this requires knowledge of both p and q, only the owner of the secret key can use this as both values are private knowledge of the key which is not shared.

The described attack goes as follows: Assume the attacker can disturb the value of S_1 before it is used to calculate S. Only S_1 is affected, and S_2 is calculated correctly. Let $\widehat{S_1}$ be the disturbed value of S_1, and $\widehat{S}$ similarly the incorrect value thus calculated for S. The following then holds: as $a = 1 - p_q^{-1}p = 0 \pmod{q}$, thus $S = \widehat{S} \pmod{q}$, but $S \neq \widehat{S} \pmod{p}$. Thus $S - \widehat{S} = aS_1 + bS_2 - a\widehat{S_1} - bS_2 = aS_1 - a\widehat{S_1} = a(S - \widehat{S})$. Yet, from $a = 1 - p_q^{-1}p = 0$

(mod q) follows that a is a multiple of q, thus $a = rq$, so the Greatest Common Divisor of $S - \widehat{S}, N$ becomes:

$$\text{GCD}(S - \widehat{S}, N) = \text{GCD}(a(S_1 - \widehat{S_1}), pq) = \text{GCD}(qr(S_1 - \widehat{S_1}), qp) = q$$

By knowing q, the attacker can then easily construct p as $N = pq$, and then calculate d as well (as this is similar to the original creation of the keys). This attack was then further refined [11] by noting that $S^e = x$, and $\widehat{S}^e = x$ (mod q). Thus then the entire line can be redone to see that $\text{GCD}(x - \widehat{S}^e, N) = q$. Thus, to use this attack, it requires either a correctly signed message and a faulty signed message, or the original message and the faulty signed message.

4.2.2 Fault Attack on RSA with Squaring

Another way to speed up the multiplication process is by squaring and multiplying. As $m^{(a+b)} = m^a \cdot m^b$, it thus follows that m can be exponentiated as $m^r = m^{2^a} \cdot m^{2^b} \cdot m^{2^c} \ldots$, so we see that the exponentiation can be written as a number of multiplications and squares of the original message, such that $m^a = m^{a \bmod 2^r} \cdot m^{a-(a \bmod 2^r)}$. Now, let $z = m^{a \bmod 2^r}$ be the the invariant, initializing z for $r = 0$ means that $z = m^{a \bmod 2^0} = m^{a \bmod 1} = m^0 = 1$. On an increase of r by 1 it follows that:

$$m^{a \bmod 2^r}$$

$$= \{r := r + 1\}$$

$$m^{a \bmod 2^{r+1}}$$

$$= \{a \bmod b^{r+1} = a \bmod b^r + b^r \cdot (a/b^r \bmod b)\}$$

$$m^{a \bmod 2^r + 2^r \cdot (a/2^r \bmod 2)}$$

$$= \{\text{split off } a \bmod 2^r\}$$

$$m^{a \bmod 2^r} \cdot m^{2^r \cdot (a/2^r \bmod 2)}$$

$$= \{\text{split off cases}\}$$

$$\begin{cases} m^{a \bmod 2^r} & \text{if } a/2^r \bmod 2 = 0 \\ m^{a \bmod 2^r} \cdot m^{2^r} & \text{if } a/2^r \bmod 2 = 1 \end{cases}$$

$$= \{z = m^{a \bmod 2^r}, \text{define additional invariant } y = m^{2^r}\}$$

$$\begin{cases} z & \text{if } a/2^r \bmod 2 = 0 \\ z \cdot y & \text{if } a/2^r \bmod 2 = 1 \end{cases}$$

$$= \{\text{define } a/2^r \bmod 2 = a_r, \text{ or the } r\text{'th least significant bit of } a\}$$

$$\begin{cases} z & \text{if } a_r = 0 \\ z \cdot y & \text{if } a_r = 1 \end{cases}$$

The initialization of additional invariant $y = m^{2^0} = m^1 = m$, and increase r by one:

$$m^{2^r}$$

$$= \{r := r + 1\}$$

$$m^{2^{r+1}}$$

$$= \{a^{b+1} = a^b \cdot a\}$$

$$m^{2^r \cdot 2}$$

$$= \{a^{2b} = a^{b^2}\}$$

$$m^{2^{r^2}}$$

$$= \{\text{definition of } y = m^{2^r}\}$$

$$y^2$$

In [9], it is assumed the following algorithm is used, where d_k is the k'th significant bit of d, so d_0 is the most significant bit, and d_{n-1} is the least significant bit of d. Invariants are $z = m^{a \bmod 2^r}$ and $y = m^{2^r}$. This algorithm ensures that after iteration t the value of $z = m^r$, with r being the t least significant bits of the private key d.

init
$y \leftarrow m; z \leftarrow 1$.
main
For $k = n-1, \ldots, 0$.
    if $d_k = 1$ then $z \leftarrow z \cdot y \pmod N$
    $y \leftarrow y^2 \pmod N$
Output z.

Figure 4.1: Exponentiation by squaring and multiplying

The article then shows the following attack: assume that the attacker can modify the value z at some point t in the calculation by flipping a single bit in z from either zero to one, or from one to zero. Define in the iteration t where the attack happens the (correct) value of z as z_t. z_t is modified to $\widehat{z}$, such that $\widehat{z} = z_t \pm 2^b$ for some value of b. Define u as the most significant bits of d which haven't been used yet in the calculation at the time of the attack, and v the ones which have, such that d is the concatenation of u and v, and define $w = d - v$. Then instead of producing the correct signed message S, the program now produces $\widehat{S} = \widehat{z}m^w$. Then $S = m^d = m^{v+w} = m^v m^w = z_t m^w = (\widehat{z} \pm 2^b)m^w = \widehat{z}m^w \pm 2^b m^w = \widehat{S} \pm 2^b m^w$. Thus, since $S^e = (m^d)^e = m$, it follows that $m = (\widehat{S} \pm 2^b m^w)^e$ (of course, all modulo N).

Now let the attacker encrypt a number of messages m containing arbitrary data, thus giving the attacker a collection of tuples $V =< m, \widehat{S} >$. The attacker can now guess the values of w and b, using variable w' and b', starting at the most significant bits, by trying to find a message i such that $m_i = (\widehat{S}_i \pm 2^{b'} m_i^{w'})^e \pmod N$ for some value of b. As there are only $\log_2(N)$ options for b', this can be done straightforward. If this holds, then two cases exist:

1. $v = w$ and the guess is correct

2. $v \neq w$ and apparently b' is guessed incorrectly such that $2^{b'} m^{w'} = \pm m^w 2^b$. The chance of this occurrence happening is dependent on b, as b has $\log_2 N$ possibilities to spread out correctly in N possibilities. As there's a $\pm$, this chance doubles to $2 \log_2 N/N$. For large N of over 2000 bits such a chance is very small indeed. But, assuming such an erroneous guess occurs, this can be detected with good probability as follows: let I be the number of collected messages. Assuming the errors are distributed equally over all iterations P, then the chance that a specific fault injected an error in this iteration is $1/P$, and thus a chance of $P - 1/P$ of being in another iteration. For all injected faults I, the chance that there is no message in which an error occurred in iteration t is $(P - 1/P)^I$. The chance that no errors occurred in r (consecutive) iterations is thus $(P - 1/P)^{I^r}$. So when r becomes too big (and thus too many bits have to be found in one program run), we can assume that apparently the guessed w' is wrong as we would have expected to find a message fitting the requirements. For example, with $P = 1000$, $I = 500$, setting this limit to $r = 20$ (i.e. up to 20 bits can be found at a time) means that the chance of actually having 20 consecutive iterations without a fault be at 0.000045. By assuming this chance is small enough and doesn't occur, we can then undo the previous (erroneous) guess, and redo the process from there on. In the case that it is actually the case that so many consecutive iterations do not have a fault the program will not terminate. Stopping it and raising r , or collecting more messages, should then ensure that it does terminate properly.

In either case, the message which was used to find w is no longer necessary: either it produces faulty guesses, or it is of no further use, so it can be removed from V.

4.3 Differential Fault Analysis

Another way to have information leak out of the protocol is by using Differential Fault Analysis, or, the knowledge between a number of separately injected faults

4.3.1 DFA attack on a simple signing protocol

In [13], the following attack is sketched: assume that there can be faults injected in the key itself by setting bits to zero, such that either one bit is set from one to zero, or none at all. For example, imagine there being gentle pressure each time as an attack which can possibly deplete a bit so it becomes 0. Now, continually feed messages to the signing instance. This then calculates m^k. Then inject a fault until m^k changes value. This thus implies that one bit in k has flipped. Continue this till there are no more changes and collect all intermittently signed messages S_i. After a number of steps r no more changes occur, as it follows that k is then equal to the bitstring consisting of only 0's, and this can be verified that this state is reached by comparing against m^0. Now k can be reconstructed as follows: the last signed message was encrypted with the bitstring consisting of only 0's. Then S_{k-1} was encrypted with a bitstring which had one bit at non-zero. If the key has length n, it thus means that this bit was in one of n locations, and thus there are n possibilities to find a key matching this message. Simply try all of these possibilities until one of them compares to S_{k-1}. Then,

consider the S_{k-2} — there are now $n-1$ places for a bit to be non-zero. Again, try all these possibilities. By repeating the process, the key can be found.

4.3.2 A bit of both — A different attack

This section describes a slightly different attack which is inspired by the DFA attack on a simple signing protocol and the attack on squaring. This time, assume the attacker has the ability to possibly disrupt the calculation by modifying the key temporarily in a single bit. For example, disrupting the read process of the key, or, when calculating in a similar squaring algorithm, having a check whether a bit equals 1 fail. In general, we assume that a single bit is flipped in the key. We now show that it is possible to find out which bit was flipped, and what its value was.

This attack was first mentioned in [30].

Calculation

Whatever the cause, the assumed effect is that instead of calculating $S = m^k$, instead it produces $\widehat{S} = m^{k \pm 2^t}$ for some t. It thus follows that

$$\widehat{S} = m^{k \pm 2^t} = \begin{cases} m^{k+2^t} & \text{if the bit was flipped from '0' to '1'} \\ m^{k-2^t} & \text{if the bit was flipped from '1' to '0'} \end{cases}$$

$$= \begin{cases} S \cdot m^{2^t} & \text{if the bit was flipped from '0' to '1'} \\ S/m^{2^t} & \text{if the bit was flipped from '1' to '0'} \end{cases}$$

Thus it is possible when both an original and a faulty sign are available to try all n positions and see whether $S \cdot m^{2^t} = \widehat{S} \pmod N$ holds for some t. If so, then s_t, or the t'th bit in s, must be 0. Similarly, a check $S/m^{2^t} = \widehat{S} \pmod N$ can reveal a t showing s_t to be 1. This then reveals the value of a single bit of the key. By having multiple faulty signs, more bits can be exposed. By having enough faulty signs the entire key can be retrieved. If not enough bits are exposed, it even then can be used as a stepping stone for a brute force attack as a number of bits are already known.

4.4 How to use Fault Injection

This section shortly describes how the methods used to injecte faults in Chapter 3 can generate faults for the three given attacks.

When examining issues for attacks, we will look with respect to the RSA algorithm. As previously seen, there are some optimizations for the RSA algorithm. A sample working implementation can be seen in figure 4.2. This implementation only works for keys up to 8 bits as Java Card does not necessarily support integers. It uses CRT for speedup, and square and multiply for the exponentiation. It is code which can be run (and was tested on) a Java Card

To do a Fault Attack on this, we aim to accomplish such a fault which produces an attack described in chapter 4.

```
 1 private void multiplyCRT(byte[] input, short offset, short len,
 2                          short d1, short d2) {
 3   for (short i = offset; i < offset + len; i++) {
 4     short s1, s2;
 5     jctools.Util.setShort(temp, (short) 0, d1);
 6     temp[2] = input[i];
 7     square_mult(temp, (short) 2, (short) 1, temp, (short) 0,
 8                 (short) 2, p);
 9     s1 = temp[2];
10     jctools.Util.setShort(temp, (short) 0, d2);
11     temp[2] = input[i];
12     square_mult(temp, (short) 2, (short) 1, temp, (short) 0,
13                 (short) 2, q);
14     s2 = temp[2];
15     input[i] = (byte) ((s1+((((s2-s1)*c2)%q)*p))% super.rsamod);
16 }
17
18 protected static void square_mult(byte[]input, short offset,
19                                   short inlen, byte[] nr,
20                                   short nroffset, short nrlen,
21                                   short mod) {
22   for (short i = offset; i < offset + inlen; i++) {
23     short c = (short) (nrlen + nroffset - 1);
24     byte t = nr[c]; //t = lowest byte
25     short z = 1;
26     short y = input[i];
27     short j = 0;
28     while (c >= 0) {
29       if( (t & (1 << (j - ( (nr.length - 1 - c) * 8)))) != 0)
30          z = (short) ( (z * y) % mod);
31       y = (short) ( (y * y) % mod);
32       if ( (j++) % 8 == 0) {
33         c--;
34         t = nr[c];
35       }
36     }
37   input[i] = (byte) z;
38   }
39 }
```

Figure 4.2: Sample simple RSA implementation

4.4.1 Attacking the Application

The .cap file can be modified in several places to inject a fault which can allow an attack to happen. For example, in the code in figure 4.2 at line 9, by saving the wrong variable to `s1`, or in line 12, by changing the "-" into a "+". In the .cap file this equals to changing the instruction with hexadecimal $0x41$, `sadd` to $0x43$, `ssub`. It is thus equal to changing a single bit. This, then, leads to an attack by using the RSA+CRT method. However, this also ensures that any encoding or decoding operation is done incorrectly from that point onward. Of further note is that it is not viable to attack the calculation of `s1` or `s2`

separately by modifying the `square_mult` method as it would then influence both `s1` and `s1`. Yet even other methods exist. Instead of calculating with `d1`, a modification could be made so that instead it calculates with `d2`, thus producing an incorrect value for S_2. As the actual value of S_2 is irrelevant to the equation, as long as it is not correct, it is even possible to let the calculation occur with an incorrect m.

4.4.2 Other methods

All the other methods were not actually performed. They either required physical tampering with the hardware, or were not immediately feasible. Even so, it is easy to see them actually perform.

Chapter 5

Prevention and hardening

This chapter concerns the actions which can be taken to either prevent or at least make fault injection harder. There are three lines which will be discussed here. One are hardware measures, which harden the ability to modify on-card programs and program flows. The second regards software measures, in which software is used to check for faults. Other software measures include ensuring that no valuable information can be learned from injecting faults. Finally, it is possible by putting down standards that any other means of injection can be prevented. For this, the Global Platform initiative is discussed which sets a standard and deals with the security of multi-application Smart Cards.

5.1 Hardware Measures

This section includes hardware measures which can be implemented by the industry which provides tamperproof chips. This section is based on [29].

5.1.1 Passive protection

Passive protection encompasses protections that increase the difficulty of a successful attack.

random dummy cycles are a means to ensure that any side channels are harder to use as timing is disturbed by random garbage being injected. By having these dummy cycles at random a timed attack can not easily be well-timed.

Bus and memory encryption to prevent laser or glitch targeting against a specific memory cell. This is done by having a temporary key created upon power-up and encrypting and decrypting data, and saving it at different addresses. Instead of saving it at address a, values are now stored in $h(k, a)$, where h is a hash function. Thus data is continually kept in a different position and has different values in-memory.

Passive Shield is a full metal layer that covers sensitive chip parts. To do attacks with light or electromagnetic beams this shield then needs to be removed first.

Unstable internal frequency generators can help by having the frequency at which calculations are done are inherently unstable, thereby making attacks which require synchronized states impossible.

5.1.2 Active Protection

Active protection encompasses mechanisms that check whether tampering occurs and take countermeasures (possibly by locking the card). A sample of active protection mechanisms are:

Light detectors to detect changes in the gradient of light against optical attacks,

Supply voltage detectors which can react to variations in the supply voltage and can ascertain that only a tolerable voltage is used to protect against spike attacks,

Frequency detectors to ensure the operation speed is constant to protect against glitch attacks,

Active shields are similar to passive shields, but these have (unrelated) data passing through them. When this data is modified, apparently there is tampering going on and protection measures can be taken. This is even better protection against electromagnetic beams as it is harder to remove, and putting an electromagnetic beam through it also disrupts the data passing through the shield.

Hardware redundancy to recalculate a number of times, by splitting the calculation, using checksums, or or by doing the calculation in different ways and afterwards verifying the correctness. A myriad of options exist here, depending on the allowed performance hit and transistors used. These protect against injected faults by trying to ensure that only meaningful data is output.

5.2 Software Measures

To protect the RSA algorithm it is also possible to use different or modified algorithms to ensure that any fault injected into the algorithm is detected — either by having a module which checks the calculation and decides upon its correctness in some way, or ensuring that any injected fault can not easily be analyzed such that private key information can be learned. This section concerns itself only with the possibilities of ensuring the correctness of the outcome (or ensuring that faults are detected).

These method protect against fault analysis in general by ensuring that an injected fault does not allow viable information to be learned.

5.2.1 Checking the outcome

Of course, an easy solution is to simply verify that the output matches the input. By decrypting the encrypted message the input should match the decrypted output. Then either an error is issued when the output doesn't match, or the

calculation is done again. Possibly some tamper-resistant architecture ensures that after a number of failures, the card locks itself as it assumes there is actual excessive tampering going on.

Many of the hardware redundancy measures can also be implemented in software. However, this might give another performance hit as parallelism is harder on the software level.

This method requires that not only the RSA calculation is done, but also a decryption. Although this means that theoretically the RSA calculation takes twice as long, it should be remembered that the public key usually is of a small order thus ensuring that the calculation goes relatively quickly. It is not odd to have a 16-bit public key and a 2000 bit private key, ensuring that decryption takes roughly a tenth of the time it takes to encrypt. However, for the generic case this cannot be assumed and such solutions thus might suffer a big performance hit.

5.2.2 Shamir's countermeasure

Richard Shamir offered and patented a measure to check whether an RSA calculation done with CRT is correct. A random integer r is selected and the following two numbers are computed:

$$s_p = m^{d \bmod \phi(pr)} \pmod{pr}$$

$$s_q = m^{d \bmod \phi(qr)} \pmod{qr}$$

Where $\phi(n)$ is the Euler phi function. Then, if $s_p = s_q \pmod{r}$ the calculation can be considered error free, and the actual combination part of the CRT algorithm can be done with s_p and s_q.

This method has a drawback. First, the chance that $s_p = s_q \pmod{r}$ is $1/r$ for any two numbers, and thus this still leaves the possibility of leaking the key.

It is an example of a mix: an extra, different calculation is done to check the correctness of the original calculation. However, it is also possible to try and ensure that no valuable information can be learned from disturbing the algorithm all together. This is explored in the next section.

5.3 Fault Cryptanalysis hardened algorithms

Instead of trying to devise methods which ensure that no fault can take place, it is also possible to consider a defense on the algorithm side: assume that it is possible to inject faults and try to have an algorithm in which no data can be learned from this fact. When designing such defensive algorithms, it is important to have some idea of the capabilities an attacker has, as this influences what the algorithm needs to protect against.

5.3.1 Threat Classification

To discuss algorithmic countermeasures, it is first nice to have some sort of framework of threats which the countermeasures protect against. This classification was first made in [33], and can be seen in table 5.1.

1	Precise Bit Errors	The attacker has precise control on both timing and location, and knows the attacked bit as well as its operation. Thus, a single bit can be effected.
2	Precise Byte Errors	The attacker has precise control on timing, but location is loose. By assumption this effects no more then a single byte.
3	Unknown Byte Errors	Similar to Precise Byte Errors, but now timing is loose as well, which although there is knowledge of which variable is targeted, it is in a small time frame in which the variable is used and it is not clear at exactly which point the variable is modified — it could be before a certain set instructions, during the execution of the set, or after it.
4	Unknown Byte Errors in Unknown Variables	Similar to Unknown Byte Errors, but now it is impossible to target a specific variable and thus it is unknown which exact variable is modified. However, only one variable is modified.
5	Random Errors	The attacker has only a loose timing and has only the ability to inject a fault which modifies any number of variables

Table 5.1: Overview of different types of errors

Comparing the categories with the physical means of fault injection from Chapter 3, an unprotected card could be under attacks of category one and two if Optical or Eddy Current attacks are used, where a spike attack would probably be in category three, four, or five.

Furthermore, when having created a defensive algorithm, it is possible to consider the computation and see what knowledge might be learned when a combination of faults under the assumed model are injected, thus leading to at least a step-wise methodology to expose flaws.

5.3.2 Infective Computation

The primary reason the computation of RSA with CRT is vulnerable to the attack is because of the break-up in separate calculations. However, it is possible to ensure that any injected fault propagates through the calculation in such a way that no information is leaked. This is called infective computation and was first described in [32]. Furthermore, infective computation tries to avoid any decision making points as decisions can be manipulated as well by an attacker — the attacker could then attack the algorithm, and any checking of the validity of the outcome could be modified as well, ensuring that improper information is released. Attacks on branches are not uncommon and therefore are something which should be avoided.

The following algorithm from [32] depends on the equivalence which can be set up between variables in which one is expressed in the other.

1. Compute $\lfloor k_p = m/p \rfloor$ and $\lfloor k_q = m/q \rfloor$
2. Let there also be another key pair (e_r, d_r) such that $d_r = d - r$ with r

relatively prime to n and $e_r = d_r^{-1} \pmod{\phi(n)}$.

3. Compute:

$$s_p = m^{d_r \bmod \phi(p)} \pmod{p}$$

$$m_1 = s_p^{e_r} \bmod p + k_p \cdot p \pmod{q}$$

$$s_q = m_1^{d_r \bmod \phi(q)} \pmod{q}$$

4. And finally, compute:

$$m_2 = s_q^{e_r} \bmod q + k_q \cdot$$

$$s = CRT(s_p, s_q) \cdot m_2^r \bmod n$$

Where CRT is the combinational algorithm for the Chinese Remainder Theorem.

This algorithm ensures that any modification to s_p propagates through the entire calculation, thereby making it impossible to use the attack described in chapter 4.2.1. It works by splitting the calculation of m^d to $m^{d_r} \cdot m^r$; m_r^d is calculated through the usual CRT speedup, and $m_2 = m$ when no errors occur:

$$m_2 = m_1^{d_r^{e_r}} \bmod q + m/q =$$

$$m_1^1 \bmod q + m/q = \{\text{similar for } m_1 \text{ to } m\}m$$

However, this algorithm puts an additional requirement on public/private key pairs, which can leave the keys vulnerable to simple "guessing" in an intelligent way, viz. small secret exponent attacks [31].

Other Algorithms

Other Algorithms have been proposed [33] and subsequently broken [35]. However, the direction taken seems sound: ensure that the CRT Speedup is safe to tampering by using a different number system. Although there is a performance hit compared to the speedup delivered by straight CRT, it is still quicker then a straightforward square-and-multiply algorithm.

5.4 Global Platform

As a final means to establish protection, the entire process of loading applications on a Smart Card needs to be addressed. Next to that there are additional security issues which a multi-application Smart Card such as a Java Card has to deal with. This section discusses the standardization of multi-application Smart Cards, as done in the Global Platform standard. By no means does it encompass the entire standard or delves into it too deeply as the standard itself is somewhat complex.

5.4.1 Structure

The Global Platform, much like the Open Platform, is a consortium of card vendors which aims to solve the issues with multi-application Smart Cards. This is done by formalizing the behavior of multi-application Smart Cards and by defining additional security constraints.

In Global Platform terms, there is a clear distinction in the different roles: there is a *Card Holder* — the actual user who has the card, a *Card Issuer* — the authority which issues the card and is essentially responsible for its security, and one or more *Application Producers* — which deliver the applications which are used on the Smart Card.

To ensure security, the on-card representative of the Card Issuer is the *Card Manager*, an application which provides an API to other applications, application selection and execution, and card content management. Furthermore, it can provide services for Card Holder Verification, and is also a *Security Domain*.

Security Domains are on-card representatives of the application provider (or Controlling Authority). Security Domains support security services (key handling, encryption, decryption, signature, verification) for their owners applications, and can authorize the loading of applications. It also governs the access to applications — much like the `Shareable` interface from Chapter 2.3.1. Each Security Domain implements its own *Secure Channel* which can be used when any off-card application wants to use an application managed by this Security Domain.

A Secure Channel can be anything from an encrypted communication with a specific key or no encryption at all — it is dependent upon the security the application requires. Both a Security Domain and an application can implement a Secure Channel: either the application has its own keyset, or it uses its Security Domain and let all communication go through there.

To check whether applications, security domains, and the actual card itself should be accessible, there is the notion of a *Lifetime Cycle*. Roughly it means that they can be in different states and when once a new state has been reached it is impossible to go to the previous state. The states themselves concern around initialization, execution, and eventually, termination. If this is for the Card, then the card is no longer usable. If this is for an Application, it could mean that the application is actually removed, or, if in ROM, is simply "disabled".

Lifetime Cycles thus provide a quick and clear way to check for accessibility, and allow for a clear distinction between the different states.

5.4.2 Additional Security

To ensure data integrity, not only Secure Channels are available, but also the loading of applications is secured by also sending a hash of the data to check integrity. Installing an application thus starts with opening a secure channel, then transmitting the data and hashes, and finally, when the Security Domain considers the data valid and the hashes are correct, the data will be committed to (persistent) memory.

In short, the Global Platform ensures that applications remain separate and cannot interfere with each other's execution (unless the Security Domain grants access). Furthermore, it handles the loading and deletion of applications after the card is issued, and ensures that during this data integrity is kept. When

using Global Platform, it is thus not easy to modify the `.cap` file during transmission as described in Chapter 3.1.1.

5.5 Overview

We have now seen a selection of measures which can make fault attacks harder. For ease, table 5.2 gives a quick overview which defense mechanism works against which attack.

Defense mechanism	.cap modification	on-card modification	Firewall circumvention	Spike Attack	Glitch Attack	Optical Attack	Eddy Current Attack	RSA with fault
Dummy cycles					±	±	±	±
bus-memory encryption		+						
passive shield						+	+	
unstable frequency generators					±			
light detectors						+		
supply voltage detectors				+				
frequency detectors					+			
active shields						+	+	
recalculation related								+
Global Platform	+	+	±					
infective computing	±	±						+

Table 5.2: Overview of the protection given by various defenses

"Recalculation related" encompasses both the hard- and software abilities to recalculate the computation and check for errors, as well as the solution presented by Shamir.

5.6 Quality of protection

This section gives a few points of thought on the quality of the protection. To view the quality of protection however, we first have to put it into a framework. IBM made a general classification [37] often cited, as it gives a generic classification of capabilities attackers can expect to have.

The attackers were classified in three groups depending upon their expected capabilities and attack strengths:

Class 1 (Clever outsiders): In this class are considered intelligent attackers without sufficient knowledge of the system who only use moderately sophisticated equipment. These attackers usually try to take advantage of the weaknesses in a system rather then actually creating one.

Class 2 (Knowledgeble insiders): In this class are attackers who have had a substantially specialized technical education and/or experience. They can have quite sophisticated tools available for analysis.

Class 3 (Funded organizations): These are attackers which are able to actually assemble teams of specialists (possibly themselves Class 2 attackers) with varying skills and are backed by great funding resources. These can thus design sophisticated attacks, and use the most sophisticated analysis tools.

5.6.1 Software protection

Both the "software measures" and the "hardware redundancy" protection methods have at some point to decide whether a computation is faulty or not. However, these decision points are natural points of attack in the first place for attackers. A little bit of sophistication would first give an attack against the algorithm, and then bypassing the decision point which should decide the (infected) computation was flawed. It is thus a defense which in itself is flawed. Class 2 attackers should be able to do this, and possibly Class 1 attackers as well if the how becomes available to them.

5.6.2 Hardware Protection

Three things can be said about the quality of the Hardware Protection:

Shields

Both the Active and Passive shield can be removed with sufficient expertise and time. Especially passive shields can be removed with some chemical processes. Of course, this does mean that it is clear that tampering has been going on with the chip, and it might take quite some time. Thus Class 1 attackers might have problems with this if done properly. Class 2 attackers will have sufficient knowledge to at least be moderately successful in circumventing these.

Voltage Detectors

The problem with voltage detectors is that the spike can be modified in a large number of ways. Especially short and strong spikes might be over before it is detected, whilst it could have done damage. For Class 2 and Class 3 attackers, it need not be adequate protection.

Dummy cycles and unstable frequency generators

It should be noted that these not so much disable attacks, but make them harder as the timing which some attacks require might be harder to do. Even so, when the gap is sufficiently wide (for example, when trying to attack RSA+CRT there it is roughly half the calculation which is susceptible to even a single fault) it might not give adequate protection.

5.6.3 Hardened Algorithms

Currently, the known algorithms have weaknesses themselves. Type 1 and 2 errors are very hard if not impossible to defend against. If an attacker can modify a single bit or byte then it is possible to do oracle attacks by simply setting a bit to a fixed value and seeing if the result is modified. If not, then apparently the guessed value of the bit or byte is correct (such an attack was proposed in [34]; the attack given in 4.3.2 is also an example of this).

Furthermore, it is very hard to see all possibilities when examining a hardened algorithm. So even with the 5 classes of errors, it is still very hard to decide whether the algorithm actually protects against a certain class of errors.

But for sufficiently hardened algorithms, it means that considerable expertise is required. It thus ranks in class 2 if not class 3 attackers.

5.6.4 Global Platform

No issues could be found within the Global Platform. It is something for further research — moreover as the standard itself seems to lack documents giving a good overview of the structure (as they instead dive into the technical requirements).

Chapter 6

Conclusion

This chapter concludes the thesis and answers the remaining questions:

1. What are the experiences with programming in Java Card?

2. What is the security of the Java Card enabled Smart Card?

6.1 Programming in Java Card

Programming in Java Card is, simply said, horrible. If there is no garbage collector available then Java Card is about the worst choice one can have from a language-feature point of view. However, considering that Java, and Java Card, has as one of its strong points the inherent security it is not hard to see why Java Card is available. As is, Java Card is a subset of Java, but with some additions in the the API. However, the pure subset is so limited that most Java programs will not work properly in Java Card without some (serious) rewriting.

Further more, it might be the case that because no garbage collection is available it would ensure that every object be well-considered to ensure that no big calculations are done on data which is kept in EEPROM — reading and writing permanently degrades EEPROM.

Overall, this causes a learning curve which is easily underestimated: it is proclaimed to be easy, yet there are quite a lot of hurdles — most importantly the lack of garbage collection (it might be optional on the card, but having to program card-independent would mean working without), and the difficulty of writing applications which work both on- and off-card (for example, one might want to have one cryptographic module written which works on Java Card, and then, as it works, also use it in the off-card program which connects to the Java Card enabled Smart Card). Because of the differences in architecture (JCRE vs JRE), cross-architecture programming requires some thought and tricks. Without garbage collection, the use of objects means that all objects must be static, which is not the (common) practice in Object-Oriented Programming.

6.2 Java Card Security

6.2.1 Open vs. closed source security

Normally, in security, everything but the key is out in the open. Kerckhoff's law (or Shannon's Maxim) is the starting point in cryptosystems. However, in Smart Cards, the security of the Smart Card is not in the open. Anderson [38] claims that until as recently as 1995 (so 25 years after the first Smart Cards were introduced!) security of Smart Cards relied upon the small size, the obscurity of its design and the (relative) unavailability of tools with which chips could be tested and probed. Even now often no indication is given with respect to the actual security measures incorporated into Smart Cards to prevent the host of attacks an a Smart Card has to defend itself to function securely. It is surprising that the Smart Card industry has not developed some sort of standard similar to a safe [39], displaying the amount of time a Smart Card is resistant to tampering. Although this might be something when Smart Cards mature and the populace in general considers them to be similar to safes.

The consequence of the closedness of Smart Cards is that it is very hard to test Smart Card security and has thus been a mostly theoretical exercise thus far.

6.2.2 Fault attack risk analysis

The major risk presented by Fault Analysis is the fact that even a single faulty bit can expose the secret key. It is thus vital for Smart Cards that there is both protection against fault injection, and awareness in the community that this protection might be inadequate. As said before, under laboratory conditions these could be reproduced [29], and other fault attacks have been seen and used by actual attackers [15], so it is not so much the question *if* fault attacks are possible, but whether there is adequate protection to make them inviable.

To ensure that security is adequate, the following things are important:

1. Smart Card vendors should ensure that the cards have tamper evidence to make it clear that tampering has been going on. This is especially true for multi-application Smart Cards such as those with Java Card, as these Smart Cards might not carry hardened algorithms.

2. Those employing Java Card enabled Smart Cards should be aware of the issues with Java Card, and should make use of Global Platform for the additional security.

3. Java Card enabled Smart Cards should only be employed under the assumption that they can be compromised within sufficient time. There thus should be a security policy in place to make clear that the Smart Card is simply a link: loss of Smart Card should be seen as possible loss of the key and thus loss of the Smart Card should not be disastrous for the overall security of the system.

4. GlobalPlatform cards should be used as these are step forward for security

5. Protocols and applets should not be written assuming that nothing can go wrong. Infective computation is nice, but even using decision points

as extra checks is better then no check. As it is, the more security, the better.

Appendix A

Installing OCF and the Chipdrive

In this appendix I show how to install OCF in windows environments, and to use the Towitoko CHIPDRIVE micro 130. The first thing to do is to install the chipdrive, then work on OCF.

A.1 Installing the chipdrive

The towitoko chipdrive needs drivers from the towitoko site.
Go to http://www.towitoko.de, pick "download", then "drivers", and install the towitoko chipdrive. Check if the diagnosis tools work, they should be installed under windows in "Start", "Programs", "SCM Microsystems CHIPDRIVE tools", "PCSCDiag"

A.2 Getting OCF and using it

Getting OCF is easy.

1. Go to http://www.opencard.org
2. Choose "download"
3. Download "OpenCard Framework All-in-One", which will download the file `installOCF.class`
4. Run `Java -classpath <saveddir> installOCF`, where saveddir is the directory where it is saved. follow the instructions, and if using Windows 2000 or XP, do not choose to setup an opencard.properties file as this checks for the Microsoft Smart Card Base Components, something which is automatically installed using Windows 2000 and XP. Installing this when they are already available *will* give issues using the software.
5. (optionally) if not using Windows 2000 or XP, then you need to install the Microsoft Smart Card Base Components. I found them at [5], but a search on "Microsoft Smart Card Base" should lead you to the right page as well.

6. Set up the opencard.properties file. For a sample PC/SC terminal (which the Towitoko driver conforms to) the following should suffice:

```
OpenCard.services = com.ibm.opencard.factory\
  MFCCardServiceFactory.opencard.opt.util.PassThruCardServiceFactory
OpenCard.terminals = com.ibm.opencard.terminal\
  pcsc10.Pcsc10CardTerminalFactory
OpenCard.trace = opencard:5"
```

7. (optionally) Check whether the demo examples work properly. These should work without a hitch.

To use it, the appropriate classes need to be in your path. In JBuilder, this is accomplished by adding the OCF directory as a required library for the project.

A.2.1 Error codes

There are multiple errors which can occur, both with installing OCF and using OCF. A few common errors include:

properties file not found The properties file is not in the required directories. It should be in the `JAVA HOME` directory, usually something like `..Java/jre/lib`

Exception, error code 8000xxxxx This is an exception raised during the runtime of the program. Most of the time these are from the PCSC environment. To know what the exception means, look in `OCF DIR/OCF1.2/components/pcscwrapper/src/src/com...` `/ibm/opencard/terminal/pcsc10/natives/MSWin32/scarderr.h` This contains a listing of error codes and their meaning.

A.3 Communication between components

This section describes the communication between the Smart Card and the terminal, and the means commonly used for data transfer. Although physical communication is done using electric pulses, this section will only detail in which format the data units need to be delivered.

A.3.1 APDU's

Communication between the terminal and the Smart Card is done according to ISO7816 with Application Package Data Units, or APDU's. Command APDU's are send to the card, which responds with Response APDU's. Both have a fixed structure. The Command APDU consists of a Header, which contains 4 bytes, in order:

1. CLA or Class byte, specifying the class of instruction of this command.
2. INS or Instruction byte, specifying the specific command.
3. P1 or Parameter 1, a byte for specific parameters to the command.

4. P2 or Parameter 2, also a byte for specific parameters to the command.

It then has a Body, constructed of:

1. LC or Length Byte, specifying the length(in bytes) of the optional data which is following.
2. Optional Data – anything that needs to be sent to the card.
3. LE or Length Expected, a byte specifying the length of the data to be returned by the return command. If this is 0, then *all* data available to the command is returned.

In total, a Command APDU thus looks like the following:

Figure A.1: Schematic display of a Command APDU

The smart card can then decode the Command APDU, execute the command, and return a Response APDU which consists of the following:

1. Optional Data: the data returned by the execution of the command. This should be equal to the length specified in the LE byte.
2. SW1 or Status Word 1: A byte containing the status of the execution according to ISO7816-4
3. SW2 or Status Word 2: similar to Status Word 1.

Or, as a schematic:

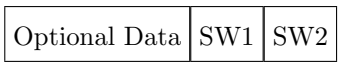


Figure A.2: Schematic display of a Response APDU

In case any errors occur during execution, the smart card is expected to answer properly in the Status Words. For example, if an error occurred, the type of error (if specified in ISO7816-4) should be specified in the returned Status Words, thus indicating that something went wrong.

A.3.2 TLV's

To be able to send data, Tag-Length-Value structures are recommended. These consist of a field which denotes the type (or Tag), the length of the value, and the value itself. It is possible to have the value be constructed of multiple values, which themselves then should be TLV's again to disable any ambiguity. Thus, a sample TLV would be a triple (T,4,HELLO), where "T" is the indicator of the type that follows is TEXT, "4" is the length of the type TEXT, and "HELLO" is the actual data, a 4-byte array of type TEXT.

A.4 Working with Smart Cards and OCF

This section describes on how to work with Smart Cards and OCF. First, it has a subsection regarding how the communication with a Smart Card works. Second, it describes using this to implement the Stock-Broker example from [1, Chapter 12]

A.4.1 Communicating with the Smart Card

In Appendix A.2 the installation of OCF is discussed. Following this should lead to a trouble-free installation. Not following this can give a multitude of problems. Problems the writer encountered were OS issues (requiring a reinstall of the OS) and Java version collisions (leading to non operating installations).

Then, following [1] gives a nice introduction on how OCF works with Smart Cards. The sample code can also be downloaded [6], but a short version is given here for reference as well in figure A.3.

```
 1 try{
 2   SmartCard.start();
 3   CardRequest cr = new CardRequest(CardRequest.ANYCARD, null,
 4     FileAccessCardService.class);
 5   cr.setTimeout(10);
 6   SmartCard sc = SmartCard.waitForCard(cr);
 7   if(sc == null)
 8     return; // no smartcard in time with required properties.
 9   FileAccessCardService facs = (FileAccessCardService)
10     sc.getCardService( FileAccessCardService.class, true);
11   CardFile root = new CardFile(facs);
12   CardFile file = new CardFile(root,":c009");
13   try{
14     String entry=new String(facs.read(file.getPath(),0,
16         file.getLength() ));
17     System.out.println(entry.trim()); //trim the end.
18   } //end try
19   catch (CardServiceInvalidCredentialException cse){
20     //handle invalid password.
21   }// end of catch invalidpassword
22 } //end try main init
23 catch (Exception ex)
24 {//multitude of exceptions possible here }
```

Figure A.3: Sample code to get data from a smartcard

As can be seen, the actual program performs numerous steps. First, on line 2, it initializes OCF with the static call to `Smartcard.start()`. This initialization also checks and tries to load software for communicating with a Terminal. If anything fails, it will throw appropriate exceptions. Then, a CardRequest is created, resembling the fact that we would like to have a card in the reader which has the FileAccessService implemented, and it is given a timeout of 10 seconds. Following that, a concrete `SmartCard` object is created if the card was inserted (or already in place) when we actually wait for a card which fullfills the request with the command `Smartcard.waitForCard(cr)`. It is here that

the terminal is actually communicated with, and thus if it has any problems, the program will halt here (by throwing an exception). If the returned value is not `null` (and thus an acceptable SmartCard was found), we continue to line 9. From the `SmartCard` Object we try and get the FileAccessService which we know it should have, and ask for its rootfile. Since the filesystem is hexadecimal, we ask for the file with name `C0:09` which is located in the root directory. We then try to access the file, and since the file might be protected by password, this is done in a separate `try...catch` block. Finally, on line 16 after the file is read, we then output it but first we remove any trailing spaces (as the file might have been bigger, it is normal to pad the text with spaces at the end).

A.4.2 Implementation of the stock-broker example

The stock broker example is an application which signs stock orders which can then be sent over the internet without being (unrecognizably) tampered with. Sadly, with the given Smart Card (an IBM MFC) which was used the stockbroker example cannot be executed [1, page 214], as it misses the required cryptographic libraries. Since this functionality was not shipped with the card, it would mean that to be able to execute a similar example it would require implementing the missing functionality and "uploading" it to the card. As this would require knowledge of both the card itself and knowing how to upload to this specific card, it was decided not to try to implement this functionality. Especially since the research should revolve around Java Card, and not an MFC card. It was thus decided to not implement this example at all.

Appendix B

Installing Eclipse and the Java Card API

This chapter concerns the Eclipse programming environment, and how to set it up for use of the JCOP toolset. Actually programming with Java Card is shown in appendix C.

B.1 Eclipse

Eclipse is a freely distributable programming environment. It also has an extension which allows the JCOP toolset be added for easy development of on-card applications. Eclipse can be downloaded from:
http://www.eclipse.org but JCOP currently does not support eclipse 3.01 yet. It only works for Eclipse 2.1. So:

1. go to http://www.eclipse.org
2. choose "downloads"
3. choose a site regionally close for quick downloads (pick a http site!)
4. choose the 2.1.3 release
5. download the file "eclipse-SDK-2.1.3-win32.zip", and unzip it to the directory where you want eclipse to be

Eclipse should now work. For working in eclipse, please consult the various documents online; [3, 4] are definitely worth looking into. Both can be found at http://www.eclipse.org/eclipse/index.html

B.2 JCOP

JCOP toolset is the Java Card Open Platform Operating System Toolset, a toolset for creating on-card applications. It can be found at:
http://www.zurich.ibm.com/Java Card.

The toolset is not free, though. Most useful is the JCOP Shell, which allows both easy communication, uploading, deleting etc. with Java Cards, as well as

allows testing on a virtual card. Other applications can then connect to the shell as if it were a terminal with a card inserted. To install the JCOP toolset:

1. Open eclipse
2. select the "Update Manager" from the "Help" menu
3. Create a new bookmark, with URL
   http://www.zurich.ibm.com/jcop/download/eclipse/
4. Expand the newly created bookmark to expose the JCOP Tools feature.
5. In the "Preview" view, click on Install. It will then install JCOP tools for Eclipse

To be able to use the JCOP toolset, you need to register. Either put a CD in which registers (which, in my case, didn't work), or point to an installed directory (which did work in my case. So then first install the "old" JCOP software from the CD)

When all is installed, consult the help on
http://www.zurich.ibm.com/jcop/download/eclipse/doc/index.html; Check the "Quick Start", "Concepts" and "JCOP Shell".

To let a program communicate with the JCOP simulation, ensure that the simulation is running, and the shell is not connected. The connecting program should use the `RemoteJC` terminal, or simply use
`Terminal.getInstance("Remote",null)`.

To connect to the card, use the `PCSCJCTerminal` or, again,
`Terminal.getInstance("PCSC",null)` should work fine as well.

B.3 Error codes

B.3.1 Error codes in Eclipse

First, there are all sorts of error codes which can be thrown in Eclipse. Keep in mind that the Java Card subset of Java has different base classes and thus all sorts of methods may not work (for example, the `Exception.getMessage()`.

Next to all the error codes created by the Java compiler (which should be known to any Java programmer), an additional disadvantage of the way things work is if an error is caught by the converter. No real error message is given, only an error message of "" (i.e., nothing) can occur. Another error which I encountered was "invalid local variable" which happened when an integer was used and the program was being converted (although this is not always apparent that the error lies in the converting). To solve other issues, I turned to the Java Card forum [7] as a source of information as well as a forum to ask any questions.

B.3.2 Debuggin JavaCard in Eclipse

When running a simulation, the debugger will halt several times on "bad data" issues. Simply stepping over these should lead to a successful installation of the application on the simulator. From there on "normal" debugging ensues with breakpoints and watches, etc.

B.3.3 Error codes in the Simulation

When running a simulation, the program is loaded automatically on the (simulated) card. This may give errors as well. If the program did not compile properly (and thus no .cap file was properly made) there should be errors in loading it. Another problem which can occur is after loading the applet, it will automatically initialize the applet (and thus run the `install` method. This creates an object of the chosen type and any errors with this will give `6a20 Bad Data` as error.

B.3.4 Error codes communicating with the on-card program

Another error occurring frequently is "**6F00**", which indicates that a runtime error occurred while executing the on-card program. This means that an exception was thrown and was not caught. Debugging can be done by throwing exceptions with the `ISOException.throwIt(<code>)` which will return `<code>` as an error code so it is known where the error occurs, or by using the debugger of Eclipse.

Appendix C

Java Card

This appendix has a small introduction on how to program an on-card application for a Java Card. For more information about Java Card, see Chapter 2.3.1

C.1 Working with Java Cards

Java Card programs can be most easily created with the JCOP toolset and Eclipse, a free, open source programming environment. For installation instructions, see Appendix B. As said, to use programs the Java Card needs .cap files. The eclipse environment creates those automatically and allows easy testing with the JCOP shell, thereby eliminating the need to do all the "converting" by calling the commands manually, or uploading it to the card. Instead, it is put in the JCOP shell, which emulates Java Cards. This speeds up development, and also development can be done without any Java Card (or terminal) being available.

C.1.1 On-card Applications

As the JCOP Toolset and the Java Card technology are mainly to allow easy access to programming on-card applications, we here give a quick rundown of a sample applet for an on-card applet.

As can be seen in figure C.1, the program has two methods, both of which are mandatory. The first is an `install` method, which is called when the applet is installed on the Java Card. This method then initializes the object. The other method is the `process` method, which is called when this applet is selected and when any APDU is sent to this applet. The process method thus handles any data sent to the applet, which can be seen as it gets the buffer from the received APDU in line 8, and checks its instruction byte (which should be there according to the ISO7816 specification of CAPDU's from Appendix A.3.1). In case it is the select statement self (`0xb1`) it doesn't need to do anything. In the case that it matches our self-defined byte `0xff` we get the length of the command, the expected return data, check if they are appropriate and if the command matches, and, if so, copy the array into the APDU and send it back out again. If anything goes wrong, we throw an exception with id 42.

```
 1 private byte[] arr = {(byte)72, (byte) 105, (byte) 33};// Hi!
 2
 3 public static void install(byte[] bArray,short bOffset,byte bLength){
 4   (new AppletName()).register(bArray,(short)(bOffset + 1), bArray[bOffset]);
 5 }
 6
 7 public void process(APDU APDU){
 8   byte[] buf = APDU.getBuffer();
 9   switch(buf[ISO7816.OFFSET_INS]){
10     case (byte)0xb1:   break;
11     case (byte)0xff:
12       short lc = APDU.setIncomingAndReceive();//receive data & get LC
13       short le = APDU.setOutgoing();//set direction to outgoing and get LE
14       if(lc == 1 && buf[5] == 0xff && le ==3) {//5 == OFFSET_CDATA for APDU
15         Util.arrayCopy(arr, (short)0, buf, (short)0, (short)2);
16         APDU.setOutgoingLength((short)2);
17         APDU.sendBytes((short)0, (short)2);
18         return;
19       }//end if
20     default:
21       ISOException.throwIt((short)42);
22   }//end switch
23 }//end void process()
```

Figure C.1: Sample Java Card On-Card Application

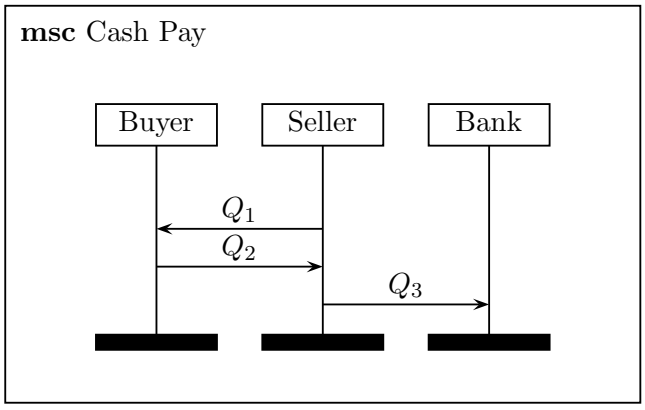
Noteworthy are the `arrayCopy`, which copies one array to the other. Since there is no garbage collection, any **new** statement would allocate memory permanently. Thus, copying arrays around by using new temporary arrays is unacceptable. This is also seen in the fact that the APDU buffer is reused. Also, in lines 15 and 16, there is continuous typecasting to shorts, as in Java, only typing "0" is always an **int** instead of a **short**

C.1.2 Cash Pay

To get an indication of the programming required for programs on a Java Card, a simple application was built which models a digital purse application. We will call this "Cash Pay". Cash Pay is a program which signs bills: the seller submits an amount, which is then signed and returned by the buyer. The seller can then 'cash' this amount by connecting to a bank where the amount can be transferred between the two agents. This simple protocol was used to demonstrate a working example on and with Java Card.

A sample run would be as follows:

1. Buyer B wants to buy an item from Seller S.
2. S signs the amount with his private key from RSA and sends both his public key and the signed amount towards the card of B.
3. B's card decrypts the amount, verifies that the amount is correct, and then signs the original message and some counter (to ensure that duplication



Q_1: $(Amount, Time)_{SK(S)}, PK(S)$

Q_2: $((Counter, Amount, Time, PK(S))_{SK(B)}, PK(B))$

Q_3: $((Counter, Amount, Time, PK(S))_{SK(B)}, PK(B))$

Figure C.2: The Cash Pay protocol

is impossible) with its own private key. He then sends the message back to Seller S.

4. S can now contact the bank, which can verify that both parties agree on the amount, and that both parties are known. Thus the transfer between the accounts of S and B can be accomplished.

Every data object is packed nicely into TLV's (see Appendix A.3.1) as it should be to ensure no typing errors occur. The implementation both on the Smart Card and the server side are relatively straightforward. To get some idea on how things work, the encryption is done using RSA (for the workings of RSA, see Chapter 4.1), with very small blocksizes. This limits the application: as everything is calculated modulus the RSA's modulus, it means that there can only be a limited size of numbers, which should be smaller then the modulus of both the Smart Card and the Seller. In appendix C.2 all the code can be found of this implementation. Again, the on-card application suffers from the architecture as typecasts to **short** are commonplace.

C.2 Cash Pay Code

The code for Cash Pay, both on and off-card, can be found in the directory on the CD, under `/cashpay/oncard` and `/cashpay/offcard` respectively.

Appendix D

Implementation of fault attacks

This chapter contains the code which was used to implement the fault attacks in Chapter 4. To get insight into the protocol of RSA and how the given four attacks work, the four were implemented. This section describes the four separately. To simulate the attack, all implementations of RSA were given a `disturb` boolean, describing whether or not its execution should be disturbed. If it is disturbed, this is done 'appropriately', i.e. no disturbance in the calculation is made at the wrong spot.

For implementation of the RSA protocols on Java Card, see the CD, under `/code/jctools`.

D.1 Implementation of DFA on RSA+CRT

This subsection discusses the implementation of the attack described in section 4.2.1. Sample code is thus as follows:

```
01   short origval = 7;
02   byte tosend[] = {(byte)origval};
03   crypto.encrypt(tosend,(short)1,(short)0);//array, size, offset
04   short scorrect = tosend[0];
05   if(s<0) scorrect += crypto.getMod();
06   tosend[0] = (byte)origval;
07   crypto.disturb();
08   crypto.encrypt(tosend,(short)1,(short)0);
09   short serr = tosend[0];
10   if(serr<0) serr+= crypto.getMod();
11   long sdiff = s-serr;
12   if(se<0) sdiff+= crypto.getMod();
13   long xse = (origval-serr)%crypto.getMod();
14   int val1 = (int)EGCD.GCD(se, crypto.getMod())[EGCD.GCD_VAL];
15   //returns Extended GCD: (xmult, ymult, gcdval) such that
16   // xmult*arg1 + ymult*arg2 = gcdval
17   int val2 = crypto.getMod() / val1;
```

D.2 Implementation of DFA on RSA with squaring

This subsection discusses the implementation of the attack described in section 4.2.2.

```
001   BigInteger crackRSA()
002   {
003     Vector v = new Vector();
004     BigInteger mod = new BigInteger((new Short(crypto.getMod())).toString());
005     for(int i = 5; i < 5*40; i=i+5)
006     {
007       byte[] b    = {new Integer(i).byteValue()};
008       byte[] orig = {new Integer(i).byteValue()};
009       crypto.disturb();
010       crypto.encrypt(b, (short)1, (short)0);
011       crypto.encrypt(orig, (short)1,(short)0);
012       BigInteger b1 = new BigInteger(b);
013       BigInteger benc = new BigInteger(orig);
014       MsgData m = new MsgData(new BigInteger((new Integer(i)).toString()), b1);
015       v.add(m);
016     }
017     //now we have a vector with stuff :)
018     byte[] key = {0};
019     int len = 0;
020     SearchData dt = new SearchData(key, mod.bitLength(),0, v);
021     while(dt.getKnownLen()!= dt.getKeyLen())
022     {
023       dt = findbits(dt);
024       if(dt==null)
025         return null;
026     }
027     return new BigInteger(dt.getKey());
028   }
029
030
031   /**
032    *
033    * <p>Title: Searchdata data holder</p>
034    * <p>Description: holds data for searching through faulty signed messages</p>
035    * <p>Copyright: Copyright (c) 2005</p>
036    * <p>Company: </p>
037    * @author Koos Gadellaa
038    * @version 1.0
039    */
040   class SearchData
041   {
042     byte[] key; //the key which is known.
043     int foundlen;//the number of bits which are known in the key
044     int keylen;//the length of the key in bits???
045     Vector messages;
046     public SearchData(byte[] k, int keylength, int fndlen, Vector msg)
047     {
048       if(k.length<2)
```

```
049       {
050         key = new byte[2];
051         key[1] = k[0];
052       }
053       else
054         key = k;
055       keylen = keylength;
056       foundlen = fndlen;
057       messages = msg;
058     }
059     public void setMessages(Vector v)
060     {
061       messages = v;
062     }
063     public void setKey(byte[]k, int l)
064     {
065       key = k;
066       keylen = l;
067     }
068     public byte[] getKey()
069     {
070       return key;
071     }
072     public int getKeyLen()
073     {
074       return keylen;
075     }
076     public int getKnownLen()
077     {
078       return foundlen;
079     }
080     public Vector getMsgs()
081     {
082       return messages;
083     }
084   }
085
086   class MsgData
087   {
088     BigInteger original;
089     BigInteger encoded;
090     public MsgData(BigInteger orig, BigInteger coded)
091     {
092       original = orig;
093       encoded = coded;
094     }
095     public BigInteger getOrig()
096     {
097       return original;
098     }
099     public BigInteger getCoded()
100     {
101       return encoded;
102     }
```

```
103
104
105   }
106
107   /**
108    * find bits from d where maximum keysize is given.
109    * @param d the data to use for searching
110    * @param keysize the maximum size of the key to be found from the data.
111    * @return
112    */
113   SearchData findbits(SearchData d)
114   {
115     int keysize = d.getKeyLen()-d.getKnownLen();
116     BigInteger mod = new BigInteger(new Short(crypto.getMod()).toString());
117     BigInteger pub = new BigInteger(new Short(crypto.getPub()).toString());
118     for(int r=1; r<keysize+1; r++)
119     {
120       byte[][] w = generate_append(d.getKey(), keysize, r);
121       for(int i = 0; i < w.length; i++)
122       {
123         Object[] v = d.getMsgs().toArray();
124         BigInteger wi = new BigInteger(w[i]);
125         BigInteger two = new BigInteger("2");
126         for(int p = 0; p<v.length; p++)
127         {
128           MsgData dt = (MsgData)(v[p]);
129           BigInteger c = dt.getOrig().modPow(wi, mod);
130           BigInteger origmsg = dt.getOrig();
131           BigInteger sgnmsg = dt.getCoded();
132           for(int t = 0; t < d.getKeyLen(); t++)
133           {
134             // 2 vars: one for S_j + 2^t M_j^w, and one for S_j - 2^t M_j^w
135             // we then 'encrypt' these and check if they're equal to the original
136             // so var-orig == 0 <==> var == orig
137             BigInteger added = c.add(sgnmsg);
138             BigInteger subst = c.subtract(sgnmsg);
139             subst = subst.modPow(pub, mod);
140             added = added.modPow(pub, mod);
141             subst = subst.subtract(dt.getOrig());
142             added = added.subtract(dt.getOrig());
143             subst = subst.mod(mod);
144             added = added.mod(mod);
145             //either subst or added needs to be 0 now to go have success
146             if (added.compareTo(BigInteger.ZERO)==0||
147                 subst.compareTo(BigInteger.ZERO)==0) {
148               //success!
149               Vector m = d.getMsgs();
150               m.remove(v[p]); //this was a success so no need to use it anymore
151               return new SearchData(w[i],d.getKeyLen(), d.getKnownLen() + r, m);
152             }
153             //else
154             c = c.multiply(two);
155             c = c.mod(mod);
156           }//end for loop for t.
```

```
157         }
158       }
159       //nothing found with this r...
160       if(r > 3 )//arbitrary size here...
161       {
162         // we assume we've made a mistake and backtrack!
163         System.out.println("backtracking...");
164         byte[] k = d.getKey();
165         BigInteger b = new BigInteger(k);
166         // we now need to remove the previous key, so
167         int ln = d.getKnownLen();
168         b=b.clearBit(d.getKeyLen() - ln);
169         ln--;
170         return findbits(new SearchData(b.toByteArray(), d.getKeyLen(),
              ln, d.getMsgs()));
171       }
172     }
173     System.out.println("r was too small so we couldn't continue");
174     System.out.println("backtracking...");
175     byte[] k = d.getKey();
176     BigInteger b = new BigInteger(k);
177     // we now need to remove the previous key, so
178     int ln = d.getKnownLen(); // this is thus one too much
179     b=b.clearBit(d.getKeyLen() - ln);
180     ln--;
181     //this works since the actual key isn't modified
182     // it's not the nicest code, but it works.
183     //we thus might want to revise this...
184     return findbits(new SearchData(b.toByteArray(), d.getKeyLen(),
          ln, d.getMsgs()));
185   }
186
187   /**
188    * generates bitstrings of w (so possible new keys).
189    * @param key the original key to which the strings are appended to
190    * @param ln the length of the original key
191    * @param r the length of the bitstrings appended
192    * @return an array of possible keys.
193    */
194   byte[][] generate_append(byte[]key, int ln, int r)
195   {
196     int i = 1<<r;
197     byte w[][] = new byte[i][];
198     int off = ln -r;
199     BigInteger bkey = new BigInteger(key);
200     //off is the place where the various bits need to be inserted
201     // as bitstring is 1,2, 3, ... which is 01,10, 11, ... in binary we can
202     // then simply append this starting at position off, such that LSB = off.
203     for(int j = 0; j < i; j++)
204     {
205       BigInteger b = BigInteger.valueOf(j);
206       b = b.shiftLeft(off);
207       b = b.add(bkey);
208       if(b.toByteArray().length <2) //since it can be smaller then one...
```

```
209       {
210         byte[] b2 = new byte[2];
211         b2[1] = b.toByteArray()[0];
212         w[j] = b2;
213       }
214       else
215         w[j] = b.toByteArray();
216     }
217     return w;
218   }
```

D.3 Implementation of DFA on a simple signing protocol

This subsection discusses the implementation of the attack described in section 4.3.1.

```
01   /**
02    * Attack on an asymmetric cryptosystem according to Biham and Shamir
03    * from Advances in Cryptology, 1997
04    * Works by disturbing until the key is 0,
05    * thus having m^0=m, and then work backwards.
06    */
07   int maxsize = 256;
08   byte[][] mem = new byte[maxsize][]; //arbitrary size here.
09   int i;
10   for(i =2; i < maxsize; i++)
11   {
12     //we here calculate the crypto of i, and save i,signed(i)
13     byte[] b = new byte[2];
14     b[0] = (byte)i;
15     b[1] = b[0];
16     crypto.encrypt(b,(short)1,(short)1);
17     if(b[1]==1 || b[1] == b[0])
18       break;
19     //b0 !/ b1 so encryption is with non-null key.
20     crypto.disturb();
21     mem[i-2] = b;
22   }
23   if(mem[253] != null)
24     return; // it took too many tries decrypting
25   //we're not at maxsize so fault attack succeeded
26   // so now we need to start decrypting.
27   int key = 0;
28   //we need to find a t such that key|2^t = M^i-1
29   for(i=i-3; i>=0; i--)
30   {
31     int q=1;
32     for(int t=0; t<15; t++)
33     {
34       int t2 = 1<<t;
35       t2 |= key;
36       if(check_message(mem[i],t2))
```

```
37      {
38        key = t2;
39        break;
40      }
41    }
42  }
```

D.4 Implementation of DFA on RSA by flipping bits

This section describes the attack from chapter 4.3.2.

```
01    short thekey = 0;
02    for(i=0; i < maxsize; i++)
03    {
04      //for each item b in mem[]
05      //b[0] is the input, b[1] is the output.
06      byte[] b = mem[i];
07      // now we need to check if something was changed at all,
08      //so we try to sign the original and see if it matches
09      //but first we save the origanal value...
10      int base = b[0];
11      int err = b[1];
12      crypto.sign(b, (short)0, (short)1);
13      int signed = b[0];
14      if(b[0] == b[1])
15        continue;
16      // now we calculate:
17      // err == base^FS
18      // FS == S - 2^t
19      // S == FS + 2^t
20      // base^S == base^(FS+2^t)
21      // base^S == base^FS * base^2^t
22      // so we can find a correct value which is base^2^t such that
23      // the equation holds. This t then is probably the bit where it
24      // went wrong.
25      int basesq = (base * base) % crypto.getMod();
26      if(signed < 0) signed += 256; // different since it's an issue
27                                    // due to char manipulation...
28      int tocheck = err * base % crypto.getMod(); // case where t = 0...
29      if(tocheck < 0) tocheck += crypto.getMod();
30      if( signed == tocheck)
31      {
32        //correct value found with i = 0 so i_0 = 1 in key...
33        thekey |= 1;
34        continue;
35      }
36      int j;
37      for( j=1; j < 16; j++)
38      {
39        tocheck = err * basesq % crypto.getMod();
40        if(tocheck < 0) tocheck += crypto.getMod();
41        if(signed == tocheck)
```

```
42          {
43            thekey |= (1<<j);
44            break;
45          }
46          basesq = (basesq * basesq) % crypto.getMod();
47          if (basesq < 0) basesq += crypto.getMod();
48        }
49      }
```

Appendix E

Java Card Bytecode Converter

This concerns a small program which was used to convert Java Card Bytecode to the more readable instruction set. It is something which I did not find on the internet, and thus have written myself to get insight into the Java Card Bytecode, and the structure of the .cap file. It was written in PHP and simply outputs three columns: the HEX value, the value in decimals, and the possible instruction code it might represent. Whether this actually is an instruction is dependent upon the structure of the .cap file and the instructions themselves as some of them take arguments.

E.1 PHP code

```
001  <HTML>
002  <head>Java Bytecode Converter</head>
003  <br>
004  <form enctype="multipart/form-data"
005        action="index.php"
006        method="post"><br>
007  Type (or select) Filename:
008  <input type="file"
009         name="thefile">
010  <input type="submit"
011         value="Upload File">
012  <br>
013  <?
014
015  if($_FILES['thefile']['error'] == 0)
016  {
017  echo "file uploaded is:";
018    echo $_FILES['thefile']['name'];
019  echo "<br>\n" ;
020  }
021  else
022  {
023    echo "No file uploaded or file upload had an issue!!<br>\n";
```

```
024   return;
025 }
026
027 if(!move_uploaded_file($_FILES['thefile']['tmp_name'], "temp.cap"))
028 {
029   echo "error processing file!<br>\n";
030   return;
031 }
032
033 echo "Now displaying file with stuff moved around...<br>\n";
034
035 function key_in_array($needle, $haystack) {
036    $haystack = array_flip($haystack);
037    if (in_array($needle, $haystack))
038    {
039        return true;
040    } else {
041        return false;
042    }
043 }
044
045 $toparse = "
046 0  00     nop  47  2F     sstore_0
047 1  01     aconst_null  48  30     sstore_1
048 2  02     sconst_m1  49  31     sstore_2
049 3  03     sconst_0  50  32     sstore_3
050 4  04     sconst_1  51  33     istore_0
051 5  05     sconst_2  52  34     istore_1
052 6  06     sconst_3  53  35     istore_2
053 7  07     sconst_4  54  36     istore_3
054 8  08     sconst_5  55  37     aastore
055 9  09     iconst_m1  56  38     bastore
056 10  0A     iconst_0  57  39     sastore
057 11  0B     iconst_1  58  3A     iastore
058 12  0C     iconst_2  59  3B     pop
059 13  0D     iconst_3  60  3C     pop2
060 14  0E     iconst_4  61  3D     dup
061 15  0F     iconst_5  62  3E     dup2
062 16  10     bspush  63  3F     dup_x
063 17  11     sspush  64  40     swap_x
064 18  12     bipush  65  41     sadd
065 19  13     sipush  66  42     iadd
066 20  14     iipush  67  43     ssub
067 21  15     aload  68  44     isub
068 22  16     sload  69  45     smul
069 23  17     iload  70  46     imul
070 24  18     aload_0  71  47     sdiv
071 25  19     aload_1  72  48     idiv
072 26  1A     aload_2  73  49     srem
073 27  1B     aload_3  74  4A     irem
074 28  1C     sload_0  75  4B     sneg
075 29  1D     sload_1  76  4C     ineg
076 30  1E     sload_2  77  4D     sshl
077 31  1F     sload_3  78  4E     ishl
```

```
078  32  20     iload_0  79  4F     sshr
079  33  21     iload_1  80  50     ishr
080  34  22     iload_2  81  51     sushr
081  35  23     iload_3  82  52     iushr
082  36  24     aaload  83  53     sand
083  37  25     baload  84  54     iand
084  38  26     saload  85  55     sor
085  39  27     iaload  86  56     ior
086  40  28     astore  87  57     sxor
087  41  29     sstore  88  58     ixor
088  42  2A     istore  89  59     sinc
089  43  2B     astore_0  90  5A     iinc
090  44  2C     astore_1  91  5B     s2b
091  45  2D     astore_2  92  5C     s2i
092  46  2E     astore_3  93  5D     i2b
093  94  5E     i2s  141  8D     invokestatic
094  95  5F     icmp  142  8E     invokeinterface
095  96  60     ifeq  143  8F     new
096  97  61     ifne  144  90     newarray
097  98  62     iflt  145  91     anewarray
098  99  63     ifge  146  92     arraylength
099  100  64     ifgt  147  93     athrow
100  101  65     ifle  148  94     checkcast
101  102  66     ifnull  149  95     instanceof
102  103  67     ifnonnull  150  96     sinc_w
103  104  68     if_acmpeq  151  97     iinc_w
104  105  69     if_acmpne  152  98     ifeq_w
105  106  6A     if_scmpeq  153  99     ifne_w
106  107  6B     if_scmpne  154  9A     iflt_w
107  108  6C     if_scmplt  155  9B     ifge_w
108  109  6D     if_scmpge  156  9C     ifgt_w
109  110  6E     if_scmpgt  157  9D     ifle_w
110  111  6F     if_scmple  158  9E     ifnull_w
111  112  70     goto  159  9F     ifnonnull_w
112  113  71     jsr  160  A0     if_acmpeq_w
113  114  72     ret  161  A1     if_acmpne_w
114  115  73     stableswitch  162  A2     if_scmpeq_w
115  116  74     itableswitch  163  A3     if_scmpne_w
116  117  75     slookupswitch  164  A4     if_scmplt_w
117  118  76     ilookupswitch  165  A5     if_scmpge_w
118  119  77     areturn  166  A6     if_scmpgt_w
119  120  78     sreturn  167  A7     if_scmple_w
120  121  79     ireturn  168  A8     goto_w
121  122  7A     return  169  A9     getfield_a_w
122  123  7B     getstatic_a  170  AA     getfield_b_w
123  124  7C     getstatic_b  171  AB     getfield_s_w
124  125  7D     getstatic_s  172  AC     getfield_i_w
125  126  7E     getstatic_i  173  AD     getfield_a_this
126  127  7F     putstatic_a  174  AE     getfield_b_this
127  128  80     putstatic_b  175  AF     getfield_s_this
128  129  81     putstatic_s  176  B0     getfield_i_this
129  130  82     putstatic_i  177  B1     putfield_a_w
130  131  83     getfield_a  178  B2     putfield_b_w
131  132  84     getfield_b  179  B3     putfield_s_w
```

```
132  133  85     getfield_s  180  B4     putfield_i_w
133  134  86     getfield_i  181  B5     putfield_a_this
134  135  87     putfield_a  182  B6     putfield_b_this
135  136  88     putfield_b  183  B7     putfield_s_this
136  137  89     putfield_s  184  B8     putfield_i_this
137  138  8A     putfield_i
138  139  8B     invokevirtual  254  FE     impdep1
139  140  8C     invokespecial  255  FF     impdep2   ";
140
141
142
143  $vals = explode(" ", $toparse);
144  $i=0;
145  $temp = "";
146  $trans = array();
147  foreach($vals as $var)
148  {
149    if($var != "")
150    {
151  //    echo $var."\t ".$i."<br>";
152      if($i==0)
153        $temp = $var;
154      if($i == 2)
155      {
156        $mnem = trim($var);
157        $trans[chr($temp)] = ("<pre>".($temp<128?$temp:$temp-255).
158                              "\t".dechex($temp)."\t".$mnem."</pre>") ;
159      }
160      $i++;
161      $i %= 3;
162    }
163  }
164
165  //now fill all the nonexistant values with 'correct' ones so it's
166  // entirely 255 values are filled...
167
168  for($i=0; $i<255; $i++)
169    if( $trans[chr($i)] == "" )
170    {
171      $trans[chr($i)] = ("<pre>".$i."\t".dechex($i).
172                         "\tNONEXISTANT:".($i - 256)."</pre>");
173    }
174
175  if(($data = file_get_contents("temp.cap")))
176  {
177    $outp = strtr($data, $trans);
178    echo $outp;
179  }
180  else
181    echo "File get contents FAILED!<br>\n";
182
183  ?>
184  </body>
185  </HTML>
```

E.2 Sample

In this section is shown a small program, and the bytecode it gives after conversion.

E.2.1 Actual program

```
/*
 * Created on May 19, 2005
 * Created by Koos Gadellaa
 * Copyright 2005
 *
 */
package miniapp;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;

/**
 * @author Koos Gadellaa
 *
 * Copyright 2005
 */
public class MiniApplet extends Applet {

    public static void install(byte[] bArray, short bOffset, byte bLength) {
        // OP-compliant JavaCard applet registration
        new MiniApplet().register(
            bArray,
            (short) (bOffset + 1),
            bArray[bOffset]);
    }
    public void process(APDU apdu) {
        // Good practice: Return 9000 on SELECT
        if (selectingApplet()) {
            return;
        }

        byte[] buf = apdu.getBuffer();
        switch (buf[ISO7816.OFFSET_INS]) {
            case (byte) 0xb1 :

                short q = apdu.setIncomingAndReceive();
                short exp_len = apdu.setOutgoing();
                apdu.setOutgoingLength((short) 2);
                if (exp_len != 2)
                    ISOException.throwIt((short) 8);

                buf[0] = 0x4f; //O
```

```
                buf[1] = 0x4b; //K
                apdu.sendBytes((short) 0, (short) 2);
                return;
            default :
                ISOException.throwIt((short) 7);
                // good practice: If you don't know the INStruction, say so:
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }
}
```

E.2.2 output of method.cap

Below is the output of the program. As it is quite large already for even a small program. It reads top to bottom, left to right.

7	7	sconst_4
0	0	nop
109	6d	if_scmpge
0	0	nop
1	1	aconst_null
16	10	bspush
24	18	aload_0
-115	8c	invokespecial
0	0	nop
2	2	sconst_m1
122	7a	return
5	5	sconst_2
48	30	sstore_1
-112	8f	new
0	0	nop
3	3	sconst_0
61	3d	dup
-115	8c	invokespecial
0	0	nop
9	9	iconst_m1
24	18	aload_0
29	1d	sload_1
4	4	sconst_1
65	41	sadd
24	18	aload_0
29	1d	sload_1
37	25	baload
-116	8b	invokevirtual
0	0	nop
0	0	nop
122	7a	return
3	3	sconst_0
35	23	iload_3
24	18	aload_0
-116	8b	invokevirtual
0	0	nop
4	4	sconst_1
96	60	ifeq
3	3	sconst_0
122	7a	return
25	19	aload_1
-116	8b	invokevirtual
0	0	nop
6	6	sconst_3
45	2d	astore_2
26	1a	aload_2
4	4	sconst_1
37	25	baload
115	73	stableswitch
0	0	nop
52	34	istore_1
0	ff	impdep2
-78	b1	putfield_a_w
0	ff	impdep2
-78	b1	putfield_a_w
0	0	nop
9	9	iconst_m1
25	19	aload_1
-116	8b	invokevirtual
0	0	nop
8	8	sconst_5
50	32	sstore_3
25	19	aload_1
-116	8b	invokevirtual
0	0	nop
10	a	iconst_0
41	29	sstore
4	4	sconst_1
25	19	aload_1
5	5	sconst_2
-116	8b	invokevirtual
0	0	nop
7	7	sconst_4
22	16	sload
4	4	sconst_1
5	5	sconst_2
106	6a	if_scmpeq
7	7	sconst_4
16	10	bspush
8	8	sconst_5
-114	8d	invokestatic
0	0	nop
1	1	aconst_null
26	1a	aload_2
3	3	sconst_0
16	10	bspush
79	4f	sshr
56	38	bastore
26	1a	aload_2
4	4	sconst_1
16	10	bspush
75	4b	sneg
56	38	bastore
25	19	aload_1
3	3	sconst_0
5	5	sconst_2
-116	8b	invokevirtual
0	0	nop
5	5	sconst_2
122	7a	return
16	10	bspush
7	7	sconst_4
-114	8d	invokestatic
0	0	nop
1	1	aconst_null
17	11	sspush
109	6d	if_scmpge
0	0	nop
-114	8d	invokestatic
0	0	nop
1	1	aconst_null
122	7a	return

Bibliography

[1] Uwe Hansmann, Martin S. Nickous, Thomas Schäck, Achim Schneider, Frank Seliger. *Smart Card Application Development Using Java.* Springer-Verlag.

[2] Zhiqun Chen. *Java Card Technology for Smart Cards.* Addison-Wesley

[3] *Eclipse Basic Tutorial.* http://www.eclipse.org/documentation/pdf/org.eclipse.platform.doc.user_2.1.3.pdf

[4] *Java Development User Guide.* http://www.eclipse.org/documentation/pdf/org.eclipse.jdt.doc.user_2.1.3.pdf

[5] Microsoft. *Smart Card Base Components* http://www.microsoft.com/downloads/release.asp?releaseid=36755

[6] Uwe Hansmann, Martin S. Nickous, Thomas Schäck, Achim Schneider, Frank Seliger. *Readfile Sample.* http://www.opencard.org/SCJavaBook/SCJavaBook.zip ReadFile.Java

[7] *Java Forum - Java Card* http://forum.Java.sun.com/forum.jspa?forumID=23

[8] *PC/SC Workgroup* http://www.pcscworkgroup.com/

[9] Dan Boneh, Richard A. DeMillo, Richard J. Lipton. *On the Importance of Eliminating Errors in Cryptographic Computations.* Journal of Cryptology 2001, Volume 14, pages 37-51. Springer-Verlag.

[10] Ronald Rivest, Adi Shamir, Leonard Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Communications of the ACM, 21:120–126, February 1978, Page 223

[11] A.K. Lenstra. *Memo on RSA signature generation in the presence of faults* manuscript, Sept. 28, 1996. Available from the author, arjen.lenstra@citicorp.com

[12] Johannes Blömer, Jean-Pierre Seifert. *Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)* Financial Cryptography 2003, LNCS 2742, Springer-Verlag.

[13] Eli Biham, Adi Shamir. *Differential Fault Analysis of Secret Key Cryptosystems*
Advances in Cryptlogy - CRYPTO '97, LNCS 1294, Springer-Verlag.

[14] Oliver Kömmerling, Markus G. Kuhn. *Design Principles for Tamper-Resistant Hardware*
USENIX workshop on Smartcard Technology 1999

[15] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert *Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures*
Proceedings of CHES '02, LNCS 2523, Springer-Verlag

[16] Ross Anderson, Markus Kuhn *Low Cost Attacks on Tamper Resistant Hardware*
Security Protocols, 5th international Workshop 1997, LNCS 1361, Springer-Verlag

[17] Sergi P. Skorobogogatov, Ross J.Anderson *Optical Fault Induction Attacks*
Proceedings of CHES '02, LNCS 2523, Springer-Verlag

[18] J.-J. Quisquater, D. Samyde *Eddy Current for Magnetic Analysis with Active Sensor*
E-Smart 2002, NOVAMEDIA

[19] Virtual Machine Specification, Java Card Platform, Version 2.2.1 *Sun Microsystems*

[20] Pierre Bieber, Jacques Cazin, Abdellah Al-Marouani, Pierre Girard, Jean-Louis Lanet, Virginie Wels, Guy Zanon *The PACAP Prototype: a Tool for Detecting Java Card Illegal Flow*
Programming and Security, LNCS 2041, Springer-Verlag.

[21] Xavier Leroy *On-Card Bytecode Verification for Java Card*
E-Smart 2001, LNCS 2140, Springer-Verlag

[22] Xavier Leroy *Bytecode verification on Java Smart Cards*
Software — Practice & Experience volume 32

[23] R.M. Cohen *Defensive Java Virtual Machine version 0.5 alpha*
1997, http://www.computationallogic.com/software/djvm/

[24] Eva Rose *Lightweight bytecode verification*
Kluwer Academic Papers, 2003

[25] Serge Chaumete, Damien Sauveron *New security problems raised by open multiapplication smart cards*
2004

[26] John McCormac, Knut Vikor, Martyn Williams, Rene Vreeman, Linus Sugoy, Brian McIlwrath *Hacking Pay TV*
http://www.iol.ie/ kooltek/faq.html, chapter 2

[27] Michael Montgomery, Ksheerabdhi Krishna *Secure Object Sharing in Java Card*
USENIX Workshop on Smartcard Technology 1999

[28] Sudhakar Govindavajhala, Andrew W. Appel *Using Memory Errors to Attack a Virtual Machine*
2003 IEEE Symposium on Security and Privacy, May 11–14, 2003

[29] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, Claire Whelan *The Sorcerer's Apprentice Guide to Fault Attacks*
Workshop on Fault Detection and Tolerance in Cryptography, June 2004.

[30] F.bao, R.H.Deng, Y. Han, A.Jeng, A.D.Narasimhalu, T.Ngair *Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults*
Proceedings of the 5th Workshop on Secure Protocols, LNCS 1361, Springer-Verlag.

[31] Alexander May, *Cryptanalysis of Unbalanced RSA with Small CRT-Exponent*,
CRYPTO 2002, LNCS 2442, Springer-Verlag.

[32] Sung-Ming Yen, Seungjoo Kim, Seongam Lim, Sang-Jae Moon, *RSA Speedup with Chinese Remainder Theorem Immune against Hardware Fault Cryptanalysis*,
ICICS 2001, LNCS 2288, Springer-Verlag

[33] Johannes Blömer, Martin Otto, Jean-Pierre Seifert, *A New CRT-RSA Algorithm Secure Against Bellcore Attacks*,
ACM CCS 2003, ACM Press

[34] Sung-Ming Yen, Marc Joye, *Checking before output may not be enough against fault-based cryptanalysis*,
IEEE Transactions on Computers, September 2000.

[35] David Wagner, *Cryptanalysis of a Provably Secure CRT-RSA Algorithm*,
ACM conference on Computer and communications security 2004

[36] *MultOS website*,
http://www.multos.com/

[37] DG Abraham, GM Donal, GP Double, JV Stevens, *Transaction Security System*,
IBM Systems Journal volume 30 no. 2 (1991)

[38] Ross Anderson, *Security Engineering*
Wiley Publishing, 2001

[39] *Underwriters Laboratories, Inc*
http://www.ul.com

[40] Arjen K. Lenstra, Eric R. Verheul, *Selecting Cryptographic Key Sizes*,
Journal of Cryptology 14, Springer-Verlag, 2001