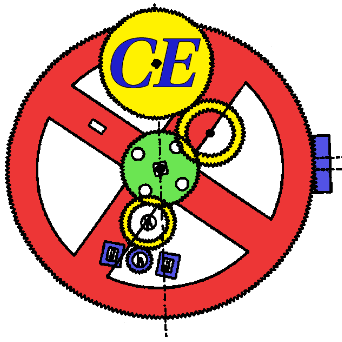


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2006

MSc THESIS

Reverse engineering of Java Card applets using power analysis

Dennis Vermoen

Abstract

Power analysis of smart cards is commonly used to obtain information about implemented cryptographic algorithms. We propose a similar methodology for reverse engineering of Java Card applets. In order to acquire power traces, we present a new microcontroller based smart card reader with an accurate adjustable trigger function. Because power analysis only does not provide enough information, we refine our methodology by involving additional information sources. Issues like distinguishing between instructions performing similar tasks and reverse engineering of conditional branches and nested loops are also addressed. The proposed methodology is applied to a commercially available Java Card smart card and the results are reported. We conclude that our augmented power analysis can be successfully used to acquire information about the instructions executed on a Java Card smart card.

CE-MS-2006-05

Faculty of Electrical Engineering, Mathematics and Computer Science

Reverse engineering of Java Card applets using power analysis

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Dennis Vermoen
born in Rotterdam, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Reverse engineering of Java Card applets using power analysis

by Dennis Vermoen

Abstract

Power analysis of smart cards is commonly used to obtain information about implemented cryptographic algorithms. We propose a similar methodology for reverse engineering of Java Card applets. In order to acquire power traces, we present a new microcontroller based smart card reader with an accurate adjustable trigger function. Because power analysis only does not provide enough information, we refine our methodology by involving additional information sources. Issues like distinguishing between instructions performing similar tasks and reverse engineering of conditional branches and nested loops are also addressed. The proposed methodology is applied to a commercially available Java Card smart card and the results are reported. We conclude that our augmented power analysis can be successfully used to acquire information about the instructions executed on a Java Card smart card.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2006-05

Committee Members :

Advisor:	G.N. Gaydadjiev, CE, TU Delft
Advisor:	M.F. Witteman, Riscure BV
Chairperson:	S. Vassiliadis, CE, TU Delft
Member:	J.C.A van der Lubbe, ICT, TU Delft

Contents

List of Figures vi

List of Tables vii

Acknowledgements ix

1 Introduction 1
- 1.1 Research questions 2
- 1.2 Conventions 2
- 1.3 Organisation of this thesis 3

2 Background information 5
- 2.1 Smart cards 5
  - 2.1.1 Typical smart card architecture 5
  - 2.1.2 Communication 6
- 2.2 Java Card technology 7
  - 2.2.1 Java Card applet example 7
  - 2.2.2 Java Card applet security 9
- 2.3 Power analysis 10
  - 2.3.1 Simple power analysis 11
  - 2.3.2 Differential power analysis 11
  - 2.3.3 Tool for power analysis 13

3 Improved acquisition system 15
- 3.1 Smart card reader 16
  - 3.1.1 Hardware 17
  - 3.1.2 Firmware 18
  - 3.1.3 Communication protocol 19
- 3.2 Oscilloscope 21
- 3.3 Acquisition software 21

4 Power analysis methodology & results 23
- 4.1 Trace set preprocessing 23
  - 4.1.1 Trace Resampling 23
  - 4.1.2 Trace Alignment 24
  - 4.1.3 Trace Shifting 26
  - 4.1.4 Trace Stretching 26
- 4.2 Template determination 28
  - 4.2.1 Reference applet 28
  - 4.2.2 Executing the reference applet 28

4.2.3 Obtaining power traces . . . 30
4.3 Template recognition . . . 32
4.3.1 Executing the (un)known applet . . . 33
4.3.2 Obtaining and recognising power traces . . . 33
4.4 Power analysis results . . . 34
4.4.1 Similar instructions . . . 35
4.4.2 Instruction duration . . . 35
4.4.3 Cryptographic operations . . . 38
4.4.4 Other Java Card smart cards . . . 39

5 Reverse engineering of Java Card applets **43**
5.1 Additional information sources . . . 44
5.1.1 Input data . . . 44
5.1.2 Impossible instruction sequences . . . 44
5.1.3 Unlikely instruction sequences . . . 46
5.1.4 Instruction statistics . . . 47
5.1.5 Instruction duration . . . 47
5.2 Execution trace processing . . . 48
5.2.1 Loop rerolling . . . 49
5.2.2 Conditional branches . . . 51
5.3 Decompilation . . . 51
5.4 Prototype reverse engineering program . . . 52

6 Conclusions **55**
6.1 Main contributions . . . 56
6.2 Future work . . . 57

Bibliography **60**

A JCVM instructions **61**

B Flowchart smart card reader **69**

List of Figures

1.1 Organisation of this thesis 3

2.1 Block diagram of a typical smart card 5
2.2 Contacts on a smart card 6
2.3 Structure of a command APDU 6
2.4 Structure of a response APDU 6
2.5 Example applet 8
2.6 Applet that contains several bugs 9
2.7 Power analysis on a smart card 10
2.8 Simple power analysis 11
2.9 Differential power analysis on simulated DES power traces 12
2.10 Inspector 13

3.1 Differences between the initial and the revised acquisition system 15
3.2 Difference between software trigger (a) and hardware trigger (b) 16
3.3 The new smart card reader 16
3.4 Smart card reader design 17
3.5 Delaying the acquisition 19
3.6 Smart card reader command structure 19

4.1 Frequency spectrum of the original power trace 24
4.2 End of a power trace affected by RPIs 27
4.3 Example reference applet 29
4.4 Bytecode of the `process` method 29
4.5 Single power trace while executing the reference applet 30
4.6 Average of 8000 power traces during the execution of the reference applet 31
4.7 JCVM visible in the power trace 31
4.8 Templates of the `baload`, `sstore` and `sload` instructions 32
4.9 Java source code, bytecode and the execution trace of `pow(3,2)` 33
4.10 Result of the template matching process 34
4.11 Correlation between the power trace and the type of `sload` operation performed 35
4.12 Difference between `sload_2` and `sload_3` 36
4.13 Average power traces of two `sdiv` instructions 37
4.14 Fragment of the applet to determine the `ifle` template 37
4.15 Duration of the `ifle` instruction 38
4.16 Power trace of a DES operation called from Java 38
4.17 Differences in power consumption 39
4.18 Fragment of the applet used for the quality comparison 40
4.19 Average power trace of *smart card 4* 41

5.1 Correlation between input data and the power trace 45
5.2 Probable instructions that follow the `sload_1` and `sload_2` instructions . 48

5.3 Probable instructions following the aload_1+sload_2 sequence 48
5.4 A for loop as generated by the Java compiler 49
5.5 Execution trace of the program depicted in Figure 5.4 49
5.6 Loop rerolling 50
5.7 Check impossible instruction sequences 52
5.8 Loop rerolling 53

B.1 Flowchart smart card reader firmware 69

List of Tables

3.1 I/O port functions 18

4.1 Correlation quality 40

5.1 Example execution trace obtained from the power analysis. 43
5.2 Examples of unlikely instruction sequences. 46
5.3 Examples of generic unlikely instruction sequences. 46
5.4 Java Card bytecode statistics 47
5.5 Decompiling Java Card bytecode 52

Acknowledgements

First of all, I would like to thank Marc Witteman for providing the opportunity to perform this interesting thesis project at Riscure BV and for his help during my thesis project. I would like to thank my advisor Georgi Gaydadjiev for reviewing my thesis and providing me with useful remarks and suggestions. I also want to thank my colleagues at Riscure BV, especially Yee Wei Law for reviewing my paper and Jasper van Woudenberg for reviewing my thesis. Finally, I would also like to thank my family for their support during my study.

Dennis Vermoen `<dennis@vermoen.com>`
Delft, The Netherlands
May 29, 2006

1 Introduction

Today, smart cards are being used in a growing number of different applications. For example, all Dutch passports issued after August 2006 contain a smart card that contains personal details and biometric information. In 2007, the Dutch *OV-chipkaart* will be used for public transport in The Netherlands.

At this moment, Java Card is the most commonly used operating system for smart cards. According to Sun Microsystems, Java Card technology grew from 750 million deployments in November 2004 to over 1.25 billion deployments in November 2005 [20, 21]. Because smart cards are typically used in applications that require a high degree of security, it is needless to say that security of Java Card applications is very important.

In this document, I describe my thesis project carried out at Riscure BV in Delft. Riscure is a security lab founded in 2001. It evaluates smart cards, terminals, smart phones and PDAs for banks, credit card companies, GSM operators, smart card manufacturers, organisations deploying digital IDs and companies in the pay television industry.

The main goal of this project is to research the possibilities of using power analysis to reverse engineer the source code of Java Card applets. Power analysis is a side channel analysis technique to acquire information about running processes on a device (such as a smart card) by monitoring the runtime current usage. Power analysis on smart cards is commonly used to obtain information about running cryptographic algorithms like DES or RSA [8, 9, 10, 22].

There are several reasons for performing this research project:

- Smart cards used to be programmed by experienced and specialised developers. With the introduction of Java Card in 1997, smart card technology became available to a wider developer community. This is however also a security risk, as developing an application for a smart card differs from developing an application for a desktop computer. When a Java Card applet is reverse engineered, possible security bugs in the applet could be revealed. A good example is presented in Section 2.2.2;

- Java Card developers sometimes implement proprietary cryptographic algorithms in their Java Card applets. During a security evaluation, reverse engineering these algorithms would possibly reveal vulnerabilities.

The experiments in this project are performed using several recent and commercially available Java Card smart cards. For the sake of convenience, we focus on one specific smart card in this document. This smart card is referred to as *Smart card 1*. Nevertheless, most of the proposed techniques can be applied in the general case.

Due to the sensitive nature of this research project, some of the results remain property of Riscure BV and are not disclosed in this document.

1.1 Research questions

The main research question in this project is: ”Is it possible to reverse engineer Java Card applets using power analysis?”. This question can be divided into several sub-questions:

- Is it possible to obtain a power profile for a specific instruction?
- Can specific instructions be recognised in a power trace?
- Can two instructions that perform similar tasks be distinguished from each other?
- Is it possible to generate Java Card source code from its execution trace?

Besides these main questions there are some miscellaneous research questions:

- What equipment is needed to reverse engineer Java Card applets?
- What information sources can be used in addition to power analysis?
- How effective are current countermeasures against power analysis?
- Are newer smart cards more secure than older smart cards, with respect to our study?

1.2 Conventions

The following conventions are used in this document:

- All Java source code, Java bytecode and Java types are written in `constant width`;
- All references to applets should be read as Java Card applets, unless otherwise specified. Likewise, all references to smart cards should be read as Java Card smart cards;
- The power traces depicted in this thesis do not depict values on their vertical axis, as the value itself is not used in the reverse engineering process. Moreover, the experiments during this project are performed using a smart card reader with a variable offset and gain which can be adjusted continuously, in order to use the oscilloscope resolution optimally. Therefore the absolute values may be different each time.

1.3 Organisation of this thesis

This thesis is organised as depicted in Figure 1.1. Chapter 2 covers the necessary background information about smart cards, Java Card technology and power analysis. Chapter 3 describes the development of the system used to perform our power measurements. Chapter 4 covers power analysis of Java Card applets. Transforming the results of the power analysis to the original Java source code is described in Chapter 5. Finally, the conclusions are drawn in Chapter 6.

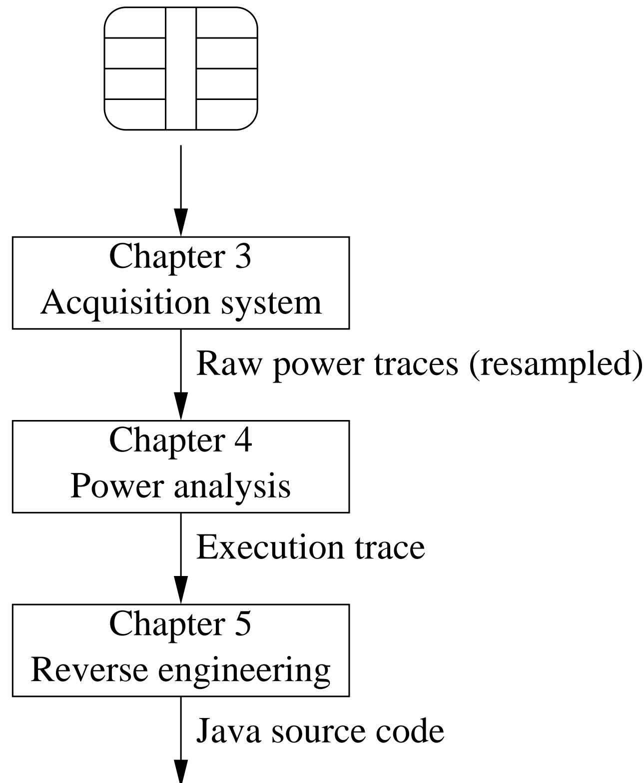


Figure 1.1: Organisation of this thesis

2 Background information

This chapter contains the necessary background information needed to follow the discussion in the rest of this document. In Section 2.1, smart cards are covered briefly. Then, in Section 2.2 Java Card technology is discussed. Finally, we discuss power analysis in Section 2.3.

2.1 Smart cards

This section briefly covers the architecture of smart cards. Furthermore, it explains the communication protocol that is used to communicate with smart cards.

2.1.1 Typical smart card architecture

Today, most smart cards are based on a small 8, 16 or 32-bit microprocessor. A block diagram of a typical smart card is depicted in Figure 2.1. A smart card usually contains three types of memory. First of all, ROM is used to store the operating system. Second, EEPROM is used to store non-volatile data like PINs and secret/private keys. Last, RAM is used as working memory. Smart cards are often used to perform encryption or decryption of data. Therefore, most smart cards are equipped with a secure cryptographic engine and random number generator. A *Universal Asynchronous Receiver Transmitter* (UART) is used to communicate with smart card readers. The communication protocol is specified in [5]. The test logic is used during the development of the smart card. It can be used to read specific addresses of the memory for debugging purposes. After the development, this logic is disabled. Security logic consists of sensors that check the input voltage and the clock signal, in order to protect the smart card against different attacks.

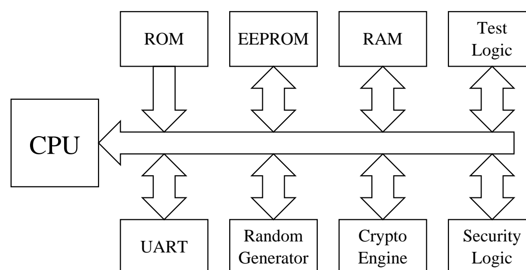


Figure 2.1: Block diagram of a typical smart card

A smart card is connected to a smart card reader using eight contacts. The assignment of the contacts is specified in [4]. Normally, only five of these contacts are used.

These are depicted in Figure 2.2. The VCC and GND contacts are used to supply power to the smart card. Depending on the type of smart card, VCC must be 5 V (class A) or 3 V (class B) [5]. Smart cards that can operate with both supply voltages also exist (i.e. class AB smart cards). Via the CLK contact, an external clock signal is supplied. The RST contact is used to reset (i.e. a *warm reset*) the smart card. The bidirectional I/O contact is used for serial communication with the smart card.

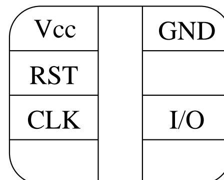


Figure 2.2: Contacts on a smart card

2.1.2 Communication

Smart cards communicate with smart card readers through *Application Protocol Data Units* (APDUs). There are two types of APDUs:

- Command APDUs (sent from the reader to the smart card);
- Response APDUs (sent from the smart card to the reader).

A command APDU consists of a class byte (**CLA**), an instruction byte (**INS**) and two parameter bytes (**P1** and **P2**). Optionally 1 to 255 extra data bytes can be sent using the **Data** field. The length of this field is specified in the **Lc** field. The **Le** field specifies the number of bytes expected in the response APDU **Data** field. The structure of a command APDU is depicted in Figure 2.3. Note that optional fields are enclosed by square brackets.

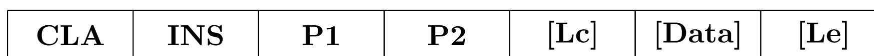


Figure 2.3: Structure of a command APDU

A response APDU consists of an optional **Data** field followed by a mandatory status word (**SW1** and **SW2**). The length of the **Data** field is specified in the **Le** of the preceding command APDU. The list of pre-specified status words is defined in [6]. The most common status word is `0x9000` which means "Normal processing, no further qualification". The structure of a response APDU is depicted in Figure 2.4.

Figure 2.4: Structure of a response APDU

2.2 Java Card technology

Java Card technology allows easy smart card application development using the Java programming language. It is compatible with the ISO 7816 standard for smart cards [4, 5, 6]. Therefore the communication protocol described in Section 2.1.2 can also be used to communicate with Java Card applets.

Smart cards have limited hardware resources. For example, most modern smart cards do not have more than 64 kilobytes of ROM and 4 kilobytes of RAM. Therefore, the *Java Card Virtual Machine* (JCVM) has the following limitations when compared to a standard *Java Virtual Machine* (JVM):

- The `char`, `double`, `float` and `long` data types are not available. The `int` data type is optional, but at this moment not available on most commercially available smart cards;
- Only one-dimensional arrays can be used;
- Dynamic class loading, garbage collection and threads are not supported.

The interested reader can find a more comprehensive list of differences between the JCVM and the JVM in [19]. Readers interested in reading more general information about Java Card technology can refer to [1].

The rest of this section covers two Java Card applet examples. The first example is used to explain how an applet is loaded on a smart card. The second example shows an insecure applet.

2.2.1 Java Card applet example

In order to understand how a Java Card applet is loaded on a smart card, we consider an example Java Card applet. The `SumApplet` class, as depicted in Figure 2.5, is a relatively simple applet that adds two values supplied through the command APDU. The sum of these numbers is returned in the response APDU.

In order to load, install and execute a Java Card applet on a smart card, the applet must extend the abstract class `Applet`. Each subclass must implement the `process` method. The `process` method handles all incoming command APDUs that are intended for this applet. The actual command APDU is supplied to the applet via the `apdu` parameter. Therefore the applet could also perform specific functions based on the instruction field of the command APDU. The constructor creates an instance of the `SumApplet` class. Furthermore, it calls the `register` method. This method registers the applet with the *Java Card Runtime Environment* (JCRE) and is implemented by the `Applet` class. The `install` method is called by the JCRE and creates an instance of the `SumApplet` class by calling its constructor. It is defined as `static`. This allows the `install` method to be called from the JCRE without an existing instance of the applet. The constructor and the `install` method are not enforced by the compiler (i.e. they are not defined as `abstract`). However, the default implementation throws a runtime exception if they are not implemented. Therefore they should be implemented, in order to develop a working Java Card applet.

```
1  package com.riscure.sum;
2
3  import javacard.framework.*;
4
5  public class SumApplet extends Applet
6  {
7     public SumApplet(byte[] bArray, short bOffset, byte bLength)
8     {
9        register();
10    }
11
12    public static void install(byte[] bArray, short bOffset, byte bLength)
13    {
14       new SumApplet(bArray, bOffset, bLength);
15    }
16
17    public void process(APDU apdu)
18    {
19       byte a, b, c;
20
21       if (selectingApplet())
22          return;
23
24       byte buffer[] = apdu.getBuffer();
25       short len = apdu.setIncomingAndReceive();
26
27       a = buffer[(short)(ISO7816.OFFSET_CDATA)];
28       b = buffer[(short)(ISO7816.OFFSET_CDATA + 1)];
29
30       c = (byte) (a + b);
31
32       buffer[0] = (byte) c;
33       apdu.setOutgoingAndSend((short) 0, (short) 1);
34    }
35 }
```

Figure 2.5: Example applet

2.2.1.1 Compilation

In order to load this applet on a Java Card smart card, first of all the applet is compiled with a regular Java compiler (i.e. `javac`) which translates the source code into bytecode. This results in a standard Java class file. The only difference between compiling for the Java Card platform and for the standard Java platform is the used API. When compiling for the Java Card platform, the classpath must be set to the Java Card API. This causes the compiler to use the `java.lang` package of the Java Card platform.

2.2.1.2 Conversion

As discussed earlier, the JCVM does not support some of the standard Java features. Therefore, standard Java class files must be converted using the Java Card `converter` tool, in order to verify and process the class files. This tool is also used to assign an *Application Identifier* (AID) to each converted applet. An AID can range from 5 to 16 bytes and uniquely identifies the applet. The `converter` tool produces a *converted applet* (CAP) file. In addition to the CAP file, it can also produce a *Java Card Assembly* (JCA) file, which is a textual representation of the CAP file. The JCA file can be used to directly edit the bytecode of the applet. The modified JCA file can be converted back to a CAP file using the `capgen` tool.

2.2.1.3 Applet installation

Loading an applet on a Java Card is performed using the *installer applet*. This applet is available by default on all Java Card smart cards. The `apdutool` tool generates command APDUs for the installer applet. First, it loads the contents of a CAP file on the smart card. Then, an instance of the applet is created. In Java Card technology, Java objects are stored into the EEPROM. Therefore, they remain on the smart card even if it is removed from the smart card reader.

2.2.2 Java Card applet security

The source code of Java Card applets is generally not accessible. When a Java Card applet could be reverse engineered using power analysis or other techniques, possible security bugs in the applet can be revealed. Suppose that a Java Card applet uses the implementation shown in Figure 2.6 to check if the user entered a valid PIN.

```
1  public boolean check(byte[] pin)
2  {
3     if (tryCounter > 0)
4     {
5        if (Util.arrayCompare(pin, (short)0, cardPin, (short)0, pin.length) == (byte)0)
6        {
7           tryCounter = tryLimit;
8           validatedPin = true;
9           return true;
10       }
11    }
12    validatedPin = false;
13    tryCounter--;
14    return false;
15 }
```

Figure 2.6: Applet that contains several bugs

`Util.arrayCompare` compares two arrays, beginning at the specified position from left to right. It returns the ternary result of the comparison: less than (-1), equal (0) or greater than (1).

This relatively simple Java method contains four errors on different levels.

- When a `pin` array of length 1 is used, only the first digit of the PIN is checked. This increases the chance of guessing the PIN by a factor 1000;
- The `tryCounter` variable is decremented when an invalid PIN is entered. When the `tryCounter` equals 0, a random PIN can be entered 129 times. Then `tryCounter` will then get the value 127 (provided that `tryCounter` is of type `byte`).

These two bugs can easily be identified when a Java Card applet is reverse engineered. The method also contains other (less obvious) problems.

- When observing the power consumption, one can easily determine the first incorrect digit, assumed that `Util.arrayCompare` returns right after an incorrect digit is found. Section 2.3.1 describes this in more detail. An attacker could even perform such *timing attack* without an oscilloscope. This can be done by measuring the time until the response APDU is received;
- In this example, `tryCounter` can easily be reset to 127. There is however another way to prevent `tryCounter` from being decremented. An attacker can switch off the power supply right before line 13 is executed.

Note that exploiting the two above bugs requires a custom smart card reader. Although the above example may seem a bit far-fetched, errors like this occur in real Java Card applets. When applets like this one could be reverse engineered, the above errors can be located and possibly exploited.

2.3 Power analysis

Power analysis is a side channel analysis technique to acquire information about running processes on a device (such as a smart card) by monitoring its runtime current usage. Figure 2.7 depicts the common way of power analysis on a smart card. The power consumption is observed by measuring the voltage drop over a low resistance resistor (e.g. 50Ω). The power consumption during the execution of a smart card process is refered to as a *power trace*. Multiple power traces can be stored in a *trace set*.

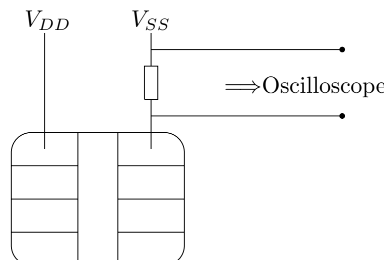


Figure 2.7: Power analysis on a smart card

2.3.1 Simple power analysis

A straightforward way to perform power analysis is *Simple Power Analysis* (SPA). SPA is the visual analysis of a few power traces. When SPA is applied on the applet shown in Figure 2.6, we obtain the power traces depicted in Figure 2.8. This figure shows single power traces while checking the PINs `0000`, `4000`, `4500` and `4560` respectively. Suppose the correct PIN is `4567`. As depicted in Figure 2.8, the time needed to complete the applet is variable. When examining the `check` method, the only statement that is likely to vary in time is the `Util.arrayCompare` method. This method returns after the first unsuccessful comparison. Therefore, checking the PIN `4000` takes longer than checking PIN `0000`. To obtain the correct second digit, an attacker only needs to try the ten possibilities for this digit. When the execution time of the applet increases, the digit is probably correct.

A brute force algorithm would only need to check 40 different PINs in the worst case situation. Although the smart card probably locks the PIN when it is entered incorrectly three times in a row, this attack increases the chance of guessing the PIN significantly.

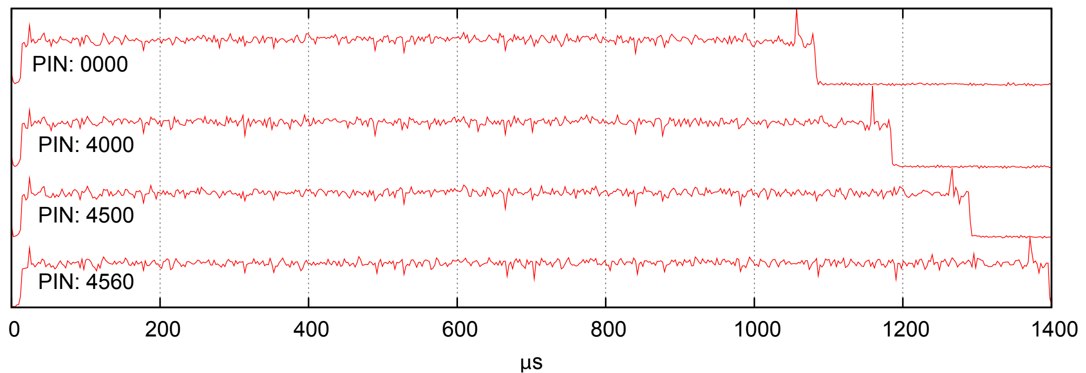


Figure 2.8: Simple power analysis

2.3.2 Differential power analysis

A more advanced technique is *Differential Power Analysis* (DPA) [9]. First, the algorithm is executed N times with alternating or random input values $D_i[j]$, where i denotes the i-th iteration, $0 \leq i < N$, and j denotes the j-th bit of the input data. N should be a significant large value (e.g. $N > 10,000$), depending on the amount of noise in the power consumption. During each execution of the algorithm, a power trace $T_i[t]$ is stored, where i denotes the i-th iteration and t denotes the time.

Using correlation, a measure of association between different variables can be obtained [13]. One can use correlation to find a measure of association between the samples of a power trace (i.e. T) and the augmented data (i.e. D). The correlation function results in a value between -1 and 1, where 1 means "identical in shape", 0 means that the variables are uncorrelated and -1 means "inverted in shape". In this document, a correlation of 1 is represented as 100%. Likewise a correlation of 0 is represented as 0%.

In most our study, a negative correlation is not relevant, as we only try to find patterns that are identical in shape.

Figure 2.9 depicts an example of differential power analysis on DES. The upper trace is one of the N input power traces, which are obtained using a simulation program developed by Riscure BV. The other three traces give the correlation between the input traces and a specific input bit. Reverse engineering of the DES algorithm is outside the scope of this thesis, but using these correlation traces, the secret DES key may be obtained.

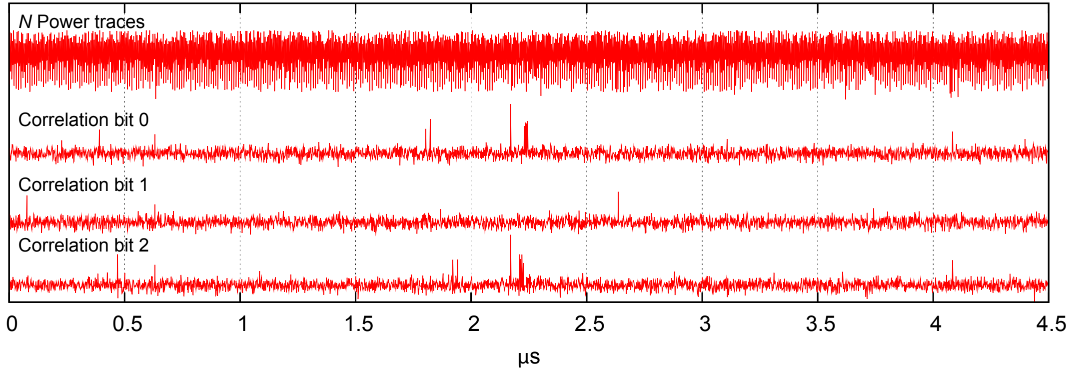


Figure 2.9: Differential power analysis on simulated DES power traces

In order to understand how the correlation between two variables can be computed, some other functions must be defined first. The variance of x is defined as:

$$var(x) = \frac{(\sum x_i - \overline{x})^2}{n-1} \tag{2.1}$$

where x_i represents the i-th element of x, $\overline{x}$ is the algebraic mean of x, and n is the size of x. The covariance of x and y provides a measure of how much x and y are related and is defined as:

$$cov(x, y) = \frac{\sum (x_i - \overline{x})(y_i - \overline{y})}{n-1} \tag{2.2}$$

The covariance is difficult to interpret though, because it depends on the scale of the input values. A better measure, independent on the absolute values of the input is given by the correlation function which is defined as:

$$corr(x, y) = \frac{cov(x, y)}{\sqrt{var(x) \cdot var(y)}} \tag{2.3}$$

2.3.3 Tool for power analysis

Inspector is a software tool to perform side-channel analysis, developed by Riscure BV. It provides a framework to analyse, edit and filter data obtained using side-channel analysis. Inspector can be extended using modules implemented in Java. These modules have access to an input trace set and can generate an output trace set. By default, Inspector has modules that can be used for editing, statistical analysis, filtering and the analysis of cryptographic operations. Figure 2.10 depicts an example screenshot of Inspector. As the screenshot shows, multiple traces in a trace set can be displayed simultaneously.

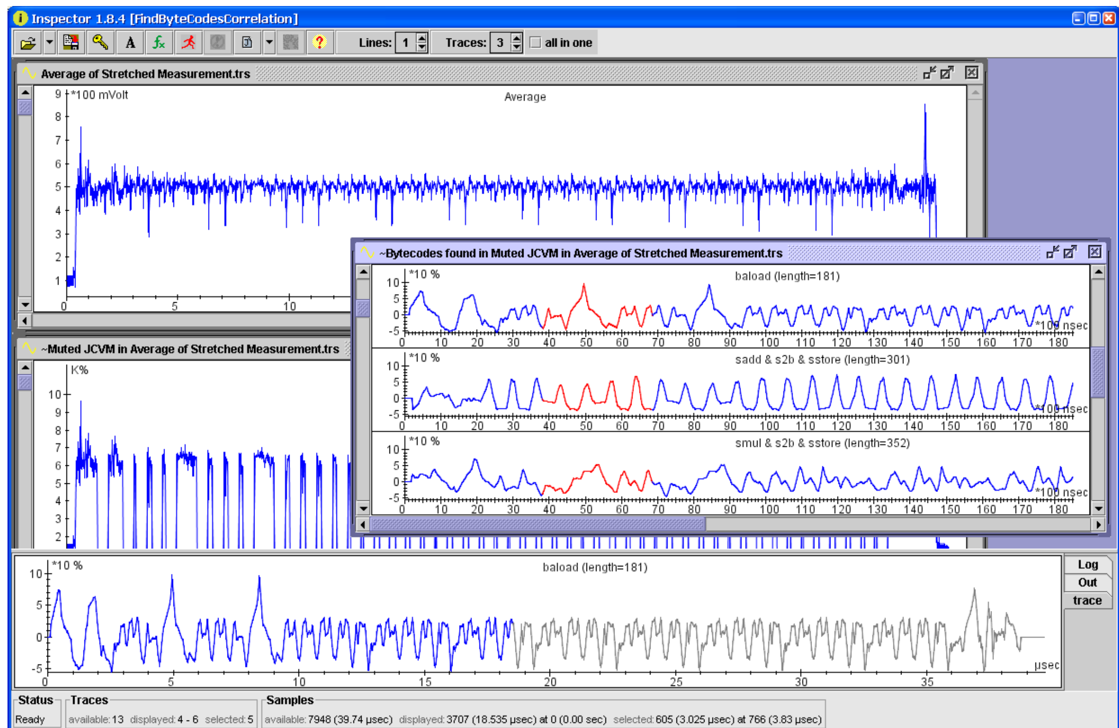


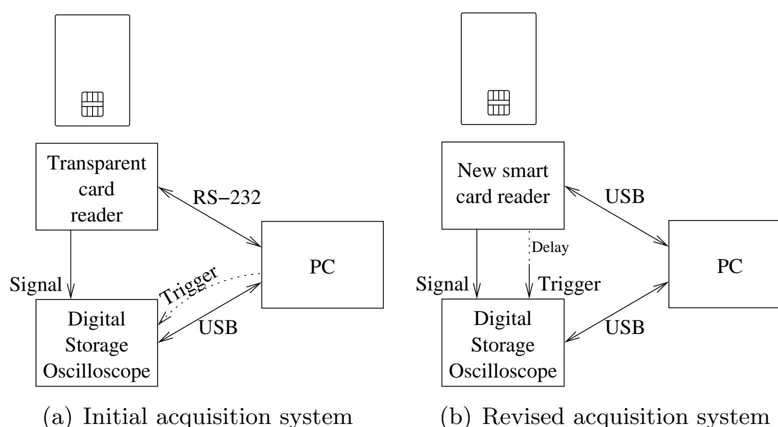
Figure 2.10: Inspector

3 Improved acquisition system

In order to collect power traces while executing a Java Card applet, an acquisition system is required. The system should perform the following two functions:

- Communication with the smart card (i.e. sending and receiving APDUs);
- Acquire power traces at a specified moment.

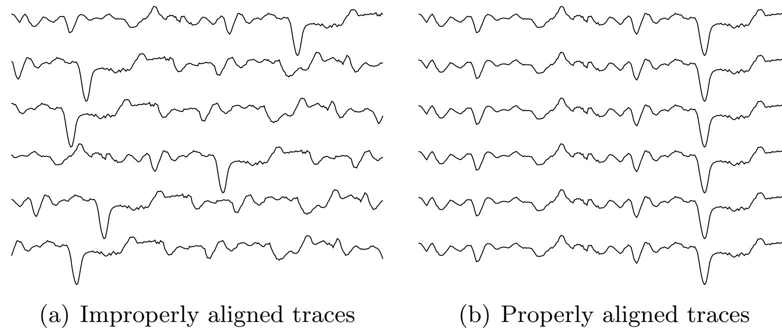
In early experiments, we used an existing acquisition system depicted in Figure 3.1a. In this measurement system, the PC sends a command APDU over a serial RS-232 connection to a *transparent card reader*. As the name implies, this reader sends and receives bytes transparently. In contrast to commercially available smart card readers, it does not implement any communication protocols. Right after sending the last byte of the command APDU, the oscilloscope is triggered using a special command sent through the USB interface. Unfortunately it turned out that triggering the oscilloscope using software, especially when sampling at high sample rates, did not result in properly aligned power traces, as depicted in Figure 3.2a.



(a) Initial acquisition system (b) Revised acquisition system

Figure 3.1: Differences between the initial and the revised acquisition system

To solve this problem, we developed an improved smart card reader. The design of this system is depicted schematically in Figure 3.1b. The advantage of our smart card reader is that it can automatically trigger the oscilloscope after sending the last byte of the command APDU. Furthermore, the trigger signal can also be delayed with µs precision, in order to analyse different parts of the APDU processing (i.e. the `process` method). The traces produced using our smart card reader are properly aligned as depicted in Figure 3.2b.



(a) Improperly aligned traces (b) Properly aligned traces

Figure 3.2: Difference between software trigger (a) and hardware trigger (b)

This chapter describes the system that measures power consumption. Section 3.1 describes the design of the smart card reader. Next, the different oscilloscopes used during this project are described in Section 3.2. Finally, Section 3.3 covers the acquisition software developed as part of this thesis project.

3.1 Smart card reader

This section describes the design of the smart card reader. Section 3.1.1 covers the design of the smart card reader hardware. Section 3.1.2 describes the smart card reader firmware. The communication protocol used by the smart card reader is explained in Section 3.1.3. A picture of the final version of the smart card reader is shown in Figure 3.3.

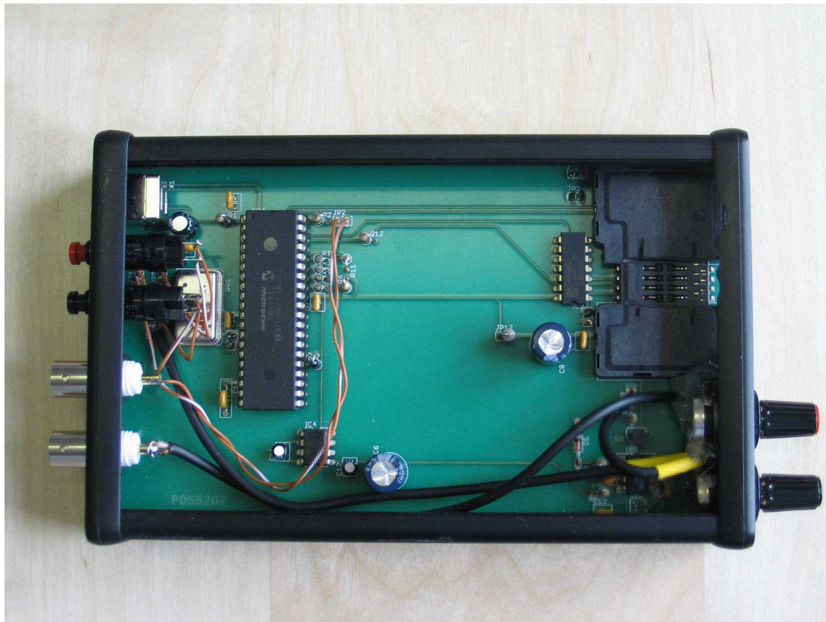


Figure 3.3: The new smart card reader

3.1.1 Hardware

The smart card reader is built upon a Microchip PIC18F4550 microcontroller. This microcontroller runs at 48 MHz and incorporates both a high-speed USB interface for host communication and a UART which is used to communicate with the smart card. The smart card reader is USB-powered. The voltage obtained from the USB interface (i.e. 5 V) is also provided to the smart card. Therefore, this smart card reader supports only class A (5 V) and class AB (3 and 5 V) smart cards. Class B (3 V) smart cards are not supported by the current hardware version. Figure 3.4 depicts the design of the smart card reader.

Both the TX and RX pins of the microcontroller are connected to the smart card's I/O channel, as a smart card contains a single bidirectional I/O port. A resistor is placed between TX and the smart card to prevent a possible short circuit.

Table 3.1 describes the function of the microcontroller I/O pins. The first column contains the pin numbers. The second column indicates if the pin is used as input or output. The third column describes the pin function.

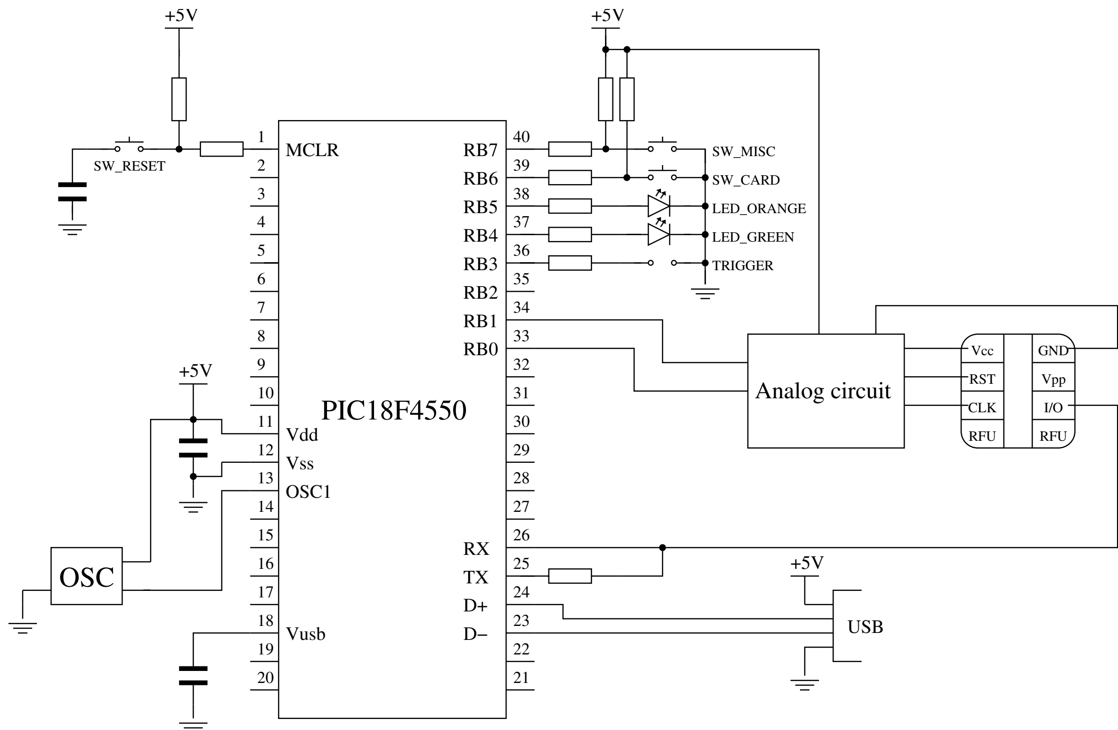


Figure 3.4: Smart card reader design

I designed the digital circuit, based on a schematic proposed in [12]. I also developed the software for the microcontroller, while colleagues from Riscure BV developed the analog circuit to measure the power consumption. The analog circuit is not depicted, but it is similar to the circuit depicted in Figure 2.7. Instead of a single resistor, this circuit allows the gain and the offset of the power signal to be adjusted. This is advantageous because in some situations the oscilloscope resolution cannot be used efficiently.

Port	In/Out	Function
RB0	Out	Smart Card power. Used to turn the power to the smart card on or off (1=powered on, 0=powered off).
RB1	Out	Smart Card reset. Used to control the smart card reset signal (1=normal operation, 0=reset).
RB2	-	Reserved for future use
RB3	Out	Trigger. Used to trigger an oscilloscope.
RB4	Out	Green LED. Used to indicate that the smart card reader is powered on. In addition, it indicates that it is triggering the oscilloscope when it is blinking.
RB5	Out	Orange LED. Used to monitor the smart card power status. When the LED is off, no card is inserted. A blinking yellow LED indicates that there is a card inserted, but it isn't powered on. A steady yellow LED indicates that a smart card is inserted and powered on.
RB6	In	Card detection switch. Using this switch the card reader knows whether there is a card present or not (1=no card inserted, 0=card inserted).
RB7	In	Push button that can be used for various user specific functions.

Table 3.1: I/O port functions

3.1.2 Firmware

The microcontroller is equipped with a bootloader. Using this bootloader, new versions of the firmware can be programmed in the microcontroller through the USB interface. The microcontroller is also equipped with the Microchip USB framework which emulates an RS-232 serial interface over USB [11]. This has two advantages:

- As most recent operating systems contain standard drivers for this class of devices, writing a device specific driver is not required;
- The impact on software previously developed by Riscure BV is minimal, as this software used the RS-232 serial interface to communicate with the old transparent smart card reader.

The microcontroller firmware consists of an infinite loop, in which the received commands are processed. The flowchart of the software containing the main loop is depicted in Figure B.1. After executing certain commands, the microcontroller can trigger the oscilloscope by generating a short pulse on one of its digital outputs (i.e. RB3). If required, the trigger signal can also be delayed with µs precision. This allows us to control the oscilloscope window position very precisely, as depicted in Figure 3.5. This is important, as most oscilloscopes can only store a limited amount of data when running at high sample rates.

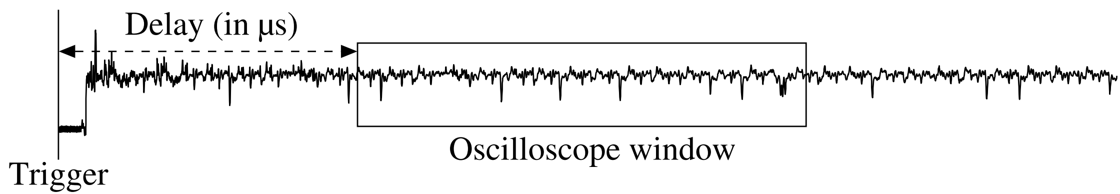


Figure 3.5: Delaying the acquisition

3.1.3 Communication protocol

The smart card reader is controlled over the USB port using a simple protocol. The *Command* field specifies the command that has to be executed. The length of the data is specified in the *Length* field and the data itself is specified in the *Data* field. The structure of a command to the smart card reader is shown in Figure 3.6. The specific commands supported by the smart card reader are covered in the following sections. The structure of the response (if any) from the reader depends on the executed command.

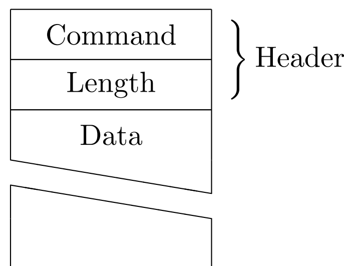


Figure 3.6: Smart card reader command structure

3.1.3.1 COMMAND_ARM

`COMMAND_ARM` arms the smart card reader. When the reader is armed, it will generate a trigger pulse after completion of the next command. This makes it possible to trigger after the last byte of a command APDU is sent or after the smart card is reset. The *Length* field of this command is always 4. The *Data* field contains 4 bytes representing the trigger delay time in µs. Therefore, the trigger can be delayed for almost $2^{32} \times 1\mu s \approx 72$ minutes. This command does not send back a response.

3.1.3.2 COMMAND_POWER_UP, COMMAND_POWER_DOWN & COMMAND_RESET_SMART_CARD

`COMMAND_POWER_UP` and `COMMAND_POWER_DOWN` are used to power up and power down the smart card respectively. When a card is inserted, the power to the smart card is always disabled. The `COMMAND_RESET_SMART_CARD` is used to reset the smart card. It is a combination of the `COMMAND_POWER_DOWN` and `COMMAND_POWER_UP` commands. The *Length* field of this command is always 0. Therefore, the *Data* field must be empty. It does not directly return a response.

3.1.3.3 COMMAND_CARD_DATA

`COMMAND_CARD_DATA` is used to send command APDUs to the smart card. All data that is received from the smart card is sent to the PC whenever it is received. The *Length* field of this command depends on the length of the command APDU. The *Data* field contains the command APDU. The response contains the response APDU.

3.1.3.4 COMMAND_ENABLE_WDT & COMMAND_DISABLE_WDT

When the acquisition system is used to acquire a significant number of traces (e.g. when it is running for a few days), the system may crash. Especially in our early experiments, this happened frequently. Therefore, the smart card reader is equipped with a watchdog timer that resets the microcontroller when no trigger event occurred after approximately eight seconds. It is recommended to enable the watchdog timer, using the `COMMAND_ENABLE_WDT` command, before starting long acquisitions. When an acquisition is finished, the `COMMAND_DISABLE_WDT` command can be used to disable the watchdog timer. The *Length* field of this command is always 0. Therefore, the *Data* field must be empty. This command does not send a response.

3.1.3.5 COMMAND_CARD_INSERTED

The smart card connector that is used in the smart card reader contains a card detection switch which is connected to one of the microcontroller inputs. Using the `COMMAND_CARD_INSERTED` command, the PC can check if a smart card is inserted in the smart card reader. The *Length* field of this command is always 0. Therefore, the *Data* field must be empty. This command always sends back a one byte response that indicates whether a smart card is inserted in the reader (1) or not (0).

3.2 Oscilloscope

During this thesis project, two different oscilloscopes were used. Therefore, an `Oscilloscope` interface was created. Each oscilloscope must implement all methods defined in the `Oscilloscope` interface. This has two advantages. First, it allows changing the oscilloscope with minimal impact on the acquisition software. Second, new oscilloscopes can be used, provided that a *Java Native Interface* (JNI) driver is developed.

Most experiments were performed using a PicoScope 3206 USB oscilloscope. This oscilloscope has a sample rate of 200 MS/s and can store approximately 1 million samples. Therefore, at the maximum sample rate, this oscilloscope can acquire power traces with a length of 5 ms. The oscilloscope can be controlled using an SDK, which is provided as a C library. Because Inspector and its modules are implemented in Java, we developed a JNI driver which provides a bridge between a Java class and the C library.

Some of our experiments were performed using a CompuScope 8500 PCI digitiser from Gage Applied Technologies, Inc. This oscilloscope has a sample rate of 500 MS/s and is capable of storing approximately 8 million samples. It can therefore acquire 16 ms power traces at the maximum sample rate. For this oscilloscope, a JNI driver has been already developed by Riscure BV. Therefore, this driver could be used without many modifications.

3.3 Acquisition software

The acquisition software is used to control the smart card reader and store the data obtained from the oscilloscope. The software is implemented as an Inspector module that generates a trace set.

Algorithm 1 Pseudocode Acquisition module

```
1: procedure AcquisitionModule(numberOfTraces)
2:     initialise oscilloscope 200 MS/s, 1,000,000 samples, 1V range   ▷ Initialise oscilloscope
3:     power on smart card                                          ▷ Power on smart card
4:     send apdu 00 A4 04 00 08 A0 00 00 00 FF 00 00 01 00            ▷ Select test applet AID
5:     for i = 0 to numberOfTraces do
6:         command ← randomise A0 00 00 00 02 00 00 00, 6, 2          ▷ Randomise two data bytes
7:         trace.data ← command[6..7]                                ▷ Store data in trace
8:         arm 900 μs delay                                          ▷ Arm smart card reader
9:         send apdu command                                         ▷ Send command APDU
10:        trace.samples ← get oscilloscope samples                  ▷ Get oscilloscope samples
11:        resample trace.samples at 4 MHz                           ▷ Resample trace
12:        traceset[i] ← trace                                       ▷ Store oscilloscope samples
13:    end for
14:    power off smart card                                          ▷ Power off smart card
15: end procedure
```

Algorithm 1 shows the pseudocode for the `Acquisition` module. This module performs the following tasks:

- The oscilloscope is initialised. In this case the sample rate is set to 200 MS/s. The range is set to 1 volt. The number of samples is set to 1,000,000;
- The smart card is powered up;
- The command APDU that selects the Java Card applet is sent;
- The actual power traces are acquired in a loop. In this loop, the following operations are performed:
  - A command APDU is constructed. If necessary, the data bytes of this command APDU can be randomised. The **randomise** command overwrites the specified bytes with random data. In this case two bytes at index six and seven are assigned random values;
  - The smart card reader is armed and the trigger delay is specified. The **arm** command causes the oscilloscope to wait for an external trigger. In addition, the smart card reader will send the delayed trigger pulse after the next received command. In this example, the trigger pulse is generated 900 µs after the command APDU is sent to the smart card using the **send apdu** command on line 9;
  - If necessary, the traces can be resampled. The **resample** command resamples a power trace. Resampling is a technique to reduce the file size. It is explained in Section 4.1.1. Of course resampling is not always necessary.
- The smart card is powered down.

Note that this is just a simple example. The above algorithm can be modified depending on the specific experiment that is performed. For example, the delay can be modified or resampling may be disabled.

Power analysis methodology & results 4

The previous chapter covered the acquisition system that is used to obtain power traces from a smart card. This chapter describes how this acquisition system is used to perform power analysis on known and unknown Java Card applets. In order to recognise the various instructions supported by the JCVM in a power trace, each of these instructions must be represented by a unique pattern (hereinafter referred to as *template*). In order to determine these templates, a reference smart card is required. This smart card should be freely programmable and identical to the smart card that contains the unknown applet. By loading known applets on the reference smart card, we can determine the templates and their corresponding instructions. Note that these templates can only be used to reverse engineer one particular type of smart card. However, the described procedure is applicable in the general case, assuming a programmable smart card is available.

In Section 4.1, we describe the Inspector modules that we developed to preprocess the obtained power traces. We present techniques to reduce the amount of data and align the traces, in order to create an average trace. Section 4.2 shows how to determine which part of a power trace represents a specific instruction. We will show an example Java Card applet. Then, we will execute the applet and determine the templates for the executed instructions. In Section 4.3, the template recognition process is described. We will try to recognise the stored templates in an unknown power trace. Section 4.4 contains the results of several interesting experiments.

4.1 Trace set preprocessing

Before the obtained power traces can be analysed, some preprocessing has to be performed. The following sections cover the Inspector modules that we developed or improved during this project.

4.1.1 Trace Resampling

Performing power analysis often requires a significant number of traces to be obtained. When capturing 10,000 traces at 200 MHz containing 1,000,000 8-bit samples each, the total file size of such trace set is $10000 \times 1000000 \approx 9.5$ GB. Resampling is a straightforward technique to reduce the file size, at cost of losing some of the information. When the trace set is resampled at 4 MHz (i.e. the external frequency of the smart card), the number of samples of the above example is reduced by a factor of 50. The resampled traces contain 20,000 samples, each containing the average of 50 samples from the original trace. The resampled trace set would require only 760 MB. Therefore it is advantageous to resample the traces before storing them. In addition, smaller trace sets speed up sequential filter and analysis processes.

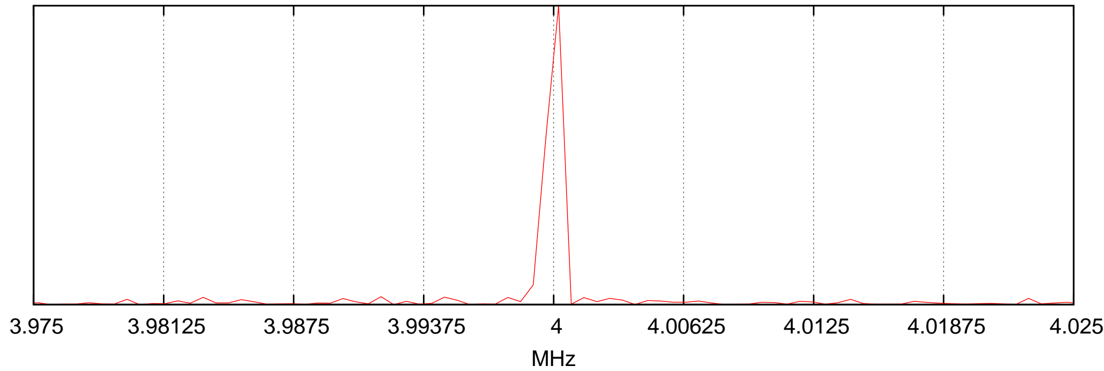


Figure 4.1: Frequency spectrum of the original power trace

The resample frequency of 4 MHz was chosen because it is equal to the external frequency of the smart card. However, the frequency spectrum of a power trace, as depicted in Figure 4.1 shows that the oscillator frequency is not exactly 4 MHz. It is therefore advantageous to resample at the frequency corresponding to the peak in the frequency spectrum of the power signal. Algorithm 2 automatically searches for this frequency within a certain margin of the specified resample frequency.

First of all, some variables are initialised. The frequency spectrum of the input trace is stored in *spectrum*. The strongest frequency in *spectrum* is stored in *resampleFreq*. The variable *factor* contains the ratio between the resample frequency and the original sample frequency. The number of samples added to the current index in the resampled trace is stored in *count*. The variable *lastIndex* stores the trace index of the previous resampled in the loop.

Next, the following operations are performed for each sample in the input trace. The current index of the resampled array (*resampleIndex*) is calculated. When the current index of the resampled array increased since the previous loop iteration, the previous index is divided by the number of samples added (i.e. *count*), in order to obtain an average. At the end of each loop iteration, *lastIndex* is assigned the old value of *resampleIndex*. Furthermore, *count* is incremented by one.

It should be noted that all time critical measurements that require high precision must of course not be resampled.

4.1.2 Trace Alignment

The measurements performed with the proposed smart card reader are properly aligned at the beginning of the trace, as depicted in Figure 3.2b. In some situations it is advantageous to align traces at the end of the execution. This can be done using the `Align` module. In order to align traces, a small pattern that is present in all traces, must be selected in one of the traces. The `Align` module aligns the traces, such that the selected pattern is aligned in all of the traces. This module stores shift values in a file. These values indicate the number of samples that each trace must be shifted, such that the traces are properly aligned. The actual shifting is performed by the `Shift` module, which

Algorithm 2 Pseudocode Resample module

```
 1: function RESAMPLE(trace, sampleFreq, resampleFreq, margin)
 2:     spectrum ← FREQUENCY_SPECTRUM(trace)
 3:     resampleFreq ← MAX(spectrum[resampleFreq − margin..resampleFreq + margin])
 4:     factor ← resampleFreq / sampleFreq
 5:     count ← 0
 6:     lastIndex ← 0
 7:     resampleLength ← ⌈factor × trace.length⌉
 8:     for i = 0 to trace.length do
 9:         resampleIndex ← ⌊factor × i⌋
10:         if resampleIndex > lastIndex and count > 0 then
11:             resampledTrace[lastIndex] ← resampledTrace[lastIndex] / count
12:             count ← 0
13:         end if
14:         if resampleIndex < resampleLength then
15:             resampledTrace[resampleIndex] ← resampledTrace[resampleIndex] + trace[i]
16:         end if
17:         lastIndex ← resampleIndex
18:         count ← count + 1
19:     end for
20:     if lastIndex <= resampleLength and count > 0 then
21:         resampledTrace[lastIndex] ← resampledTrace[lastIndex] / count
22:     end if
23:     return resampledTrace
24: end function
```

is described in the next section. The pseudocode of the alignment module is depicted in Algorithm 3. This module requires 5 parameters. The *traceSet* parameter contains the trace set that has to be aligned. The *index* parameter specifies the index of the trace that is used as the reference trace (i.e. the trace from which a fragment is selected). The parameters *start* and *end* indicate the start and the end of this fragment in the number of samples. The $maxShift$ parameter specifies the maximal allowed shift value.

Algorithm 3 Pseudocode Alignment module

```
 1: function ALIGN(traceSet, index, start, end, maxShift)
 2:     pattern[ ] ← traceSet[index].samples[start..end]
 3:     patternLength ← end − start
 4:     traceLength ← traceSet[index].samples.length
 5:     for i = 0 to traceSet.length do
 6:         maxCorrelation ← 0
 7:         for shift = −maxShift to maxShift do
 8:             if start + shift >= 0 and start + shift + patternLength < traceLength then
 9:                 currentPattern ← traceSet[index].samples[start + shift..start + shift + patternLength]
10:                 correlation ← CORRELATION(pattern, currentPattern)
11:                 if correlation > maxCorrelation then
12:                     maxCorrelation ← correlation
13:                     shiftValues[i] ← shift
14:                 end if
15:             end if
16:         end for
17:     end for
18:     return shiftValues
19: end function
```

4.1.3 Trace Shifting

As explained in the previous section, the `Shift` module performs the actual shifting of the traces, using the shift values stored in a file by the `Align` module. The pseudocode of the `Shift` module is depicted in Algorithm 4.

In some situations it is difficult to align a trace set because it contains too much noise. The `Shift` module can solve this problem. Suppose trace A, containing noise, needs to be aligned. This noise could be reduced by filtering out specific frequencies. The result is stored in trace set B. We then apply the `Align` module on trace set B and store the shift values as described earlier. Using the `Shift` module, we can apply the shift values obtained from B on A. This way, a noisy trace set can still be aligned.

Algorithm 4 Pseudocode Shift module

```
1: function SHIFT(traceSet, shiftValues)
2:     traceLength ← traceSet[0].samples.length
3:     for i = 0 to traceSet.length do
4:         for j = 0 to traceLength do
5:             index ← (traceLength + j + shiftValues[i]) modulo traceLength
6:             trace.samples[j] ← traceSet[i].samples[index]
7:         end for
8:         resultTraceSet[i] ← trace
9:     end for
10:    return resultTraceSet
11: end function
```

4.1.4 Trace Stretching

Some smart cards try to prevent power analysis by using *Random Process Interrupts* (RPIs). These interrupts stop the execution of the applet for a short time. Therefore, the execution time of an applet varies and its power traces are not properly aligned. Computing an average of a trace set that is not aligned correctly will add noise to the average trace [2]. Aligning at the end of a trace does not solve the problem, as this causes the beginning of the trace to shift too. For this situation, the `Stretch` module was developed. Figure 4.2 depicts the end of the power traces of a simple applet. This module corrects these shifted traces by stretching the entire trace over a constant length (e.g. the average execution time). The pseudocode of the `Stretch` module is shown in Algorithm 5. Of course this technique does not remove the RPI effects from the power traces, but it does reduce noise when an average of the trace set is computed.

The probability that an RPI is inserted can be calculated. Suppose that an RPI is inserted with a constant probability p. The average length of the trace set is defined as:

$$\overline{n} = n + pn = n(1 + p) \tag{4.1}$$

where n is the length of the trace without RPIs. The standard deviation σ is defined as:

$$\sigma = \sqrt{p(1-p)n} \tag{4.2}$$

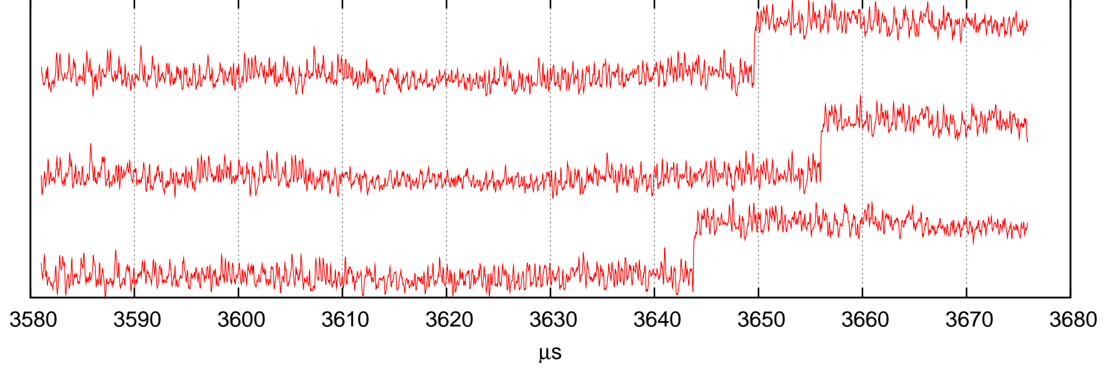


Figure 4.2: End of a power trace affected by RPIs

Algorithm 5 Pseudocode Stretch module

```
1: function STRETCH(traceSet, shiftValues, position, newLength)
2:     traceLength ← traceSet[0].samples.length
3:     for i = 0 to traceSet.length do
4:         stretch = newLength − traceLength − shiftValues[i]
5:         stretchFactor = 1 + stretch/position
6:         for j = 0 to newLength do
7:             index ← ROUND(j/stretchFactor)
8:             trace.samples[j] ← traceSet[i].samples[index]
9:         end for
10:        resultTraceSet[i] ← trace
11:    end for
12:    return resultTraceSet
13: end function
```

By substitution, we obtain:

$$\sigma = \sqrt{p(1-p)\frac{\overline{n}}{1+p}} = \sqrt{p\frac{1-p}{1+p}\overline{n}} \tag{4.3}$$

The actual value of σ can also be computed based on the trace set. If σ is known, p can also be calculated by rewriting Equation 4.3.

$$p = \frac{\overline{n} - \sigma^2 - \sqrt{\overline{n}^2 - 6\overline{n}\sigma^2 + \sigma^4}}{2\overline{n}} \vee p = \frac{\overline{n} - \sigma^2 + \sqrt{\overline{n}^2 - 6\overline{n}\sigma^2 + \sigma^4}}{2\overline{n}} \tag{4.4}$$

Finally, the actual length of the trace n without RPIs can be calculated as follows.

$$n = \frac{\overline{n}}{1+p} \tag{4.5}$$

4.2 Template determination

In this section, we show the various steps to determine templates for a specific smart card. First of all, a reference applet that contains all instructions for which templates should be determined, is required. An example of such applet is given in Section 4.2.1. Second, the reference applet has to be executed, as explained in Section 4.2.2. Third, power traces of the applet execution have to be obtained, as described in Section 4.2.3.

4.2.1 Reference applet

Figure 4.3 depicts an example of a reference applet that we used in this study. The applet is similar to the example applet in Figure 2.5. This applet assigns the first data byte of the command APDU to variable `b`. Then some addition and multiplication operations are performed. An addition statement is compiled to Java Card bytecode as depicted in Figure 4.4 (lines 32 to 36). A multiplication statement is compiled in a similar way (lines 27 to 31). Although, in contrast to the applet in Figure 2.5, this applet does not return any output data, it just performs a sequence of similar operations. Therefore, the power trace is expected to show a repeated pattern. A repeated pattern is advantageous when identifying the individual instructions. Figure 4.4 depicts the bytecode of the `process` method of this reference applet, generated by the compiler (i.e. the standard Java compiler and the Java Card converter).

4.2.2 Executing the reference applet

Because we developed the reference applet ourselves, the AID is known. In this case, we assigned the AID: `A0 00 00 00 FF 00 00 01` to the applet. The test applet can be selected by sending the following command APDU to the smart card:

```
> 0x00 0xA4 0x04 0x00 0x08 0xA0 0x00 0x00 0x00 0xFF 0x00 0x00 0x01 0x00
```

Instruction `0xA4` is the SELECT FILE command as implemented by the JCRE. This command selects the applet with AID `0xA0 0x00 0x00 0x00 0xFF 0x00 0x00 0x01`. The first parameter (i.e. `0x04`) indicates that an AID has to be selected. The second parameter (i.e. `0x00`) indicates that the first or only occurrence has to be selected. The length field is set to eight, as the length of the following AID consists of 8 bytes. If the applet exists on the card, the following response APDU will be received:

```
< 0x90 0x00
```

After the applet is selected, the JCRE dispatches all incoming command APDUs to the applet via the `process` method. However, some instructions like SELECT FILE are always processed by the JCRE. An extensive explanation of this mechanism is given in [18]. The `process` method is executed using the following command APDU.

```
> 0x00 0x00 0x00 0x00 0x01 0xAB 0x00
```

```java
package com.riscure.test;

import javacard.framework.*;

public class TestApplet extends Applet
{
   public TestApplet(byte[] bArray, short bOffset, byte bLength)
   {
      register();
   }

   public static void install(byte[] bArray, short bOffset, byte bLength)
   {
      new TestApplet(bArray, bOffset, bLength);
   }

   public void process(APDU apdu)
   {
      byte a = (byte) 0x03, b, c;

      if (selectingApplet())
         return;

      byte buffer[] = apdu.getBuffer();
      short len = apdu.setIncomingAndReceive();

      b = buffer[(short)(ISO7816.OFFSET_CDATA)];
      c = (byte)(a + b);
      c = (byte)(a * b);
      c = (byte)(a * b);
      c = (byte)(a + b);
      c = (byte)(a * b);
      c = (byte)(a + b);
      c = (byte)(a + b);
      c = (byte)(a * b);
   }
}
```

Figure 4.3: Example reference applet

```
L0: sconst_3;
    sstore_2;
    aload_0;
    invokevirtual 2;
    ifeq L2;
L1: return;
L2: aload_1;
    invokevirtual 3;
    astore 5;
    aload_1;
    invokevirtual 4;
    sstore 6;
    aload 5;
    sconst_5;
    baload;
    sstore_3;
    sload_2;
    sload_3;
    sadd;
    s2b;
    sstore 4;
    sload_2;
    sload_3;
    smul;
    s2b;
    sstore 4;
    sload_2;
    sload_3;
    smul;
    s2b;
    sstore 4;
    sload_2;
    sload_3;
    sadd;
    s2b;
    sstore 4;
    sload_2;
    sload_3;
    smul;
    s2b;
    sstore 4;
    sload_2;
    sload_3;
    sadd;
    s2b;
    sstore 4;
    sload_2;
    sload_3;
    sadd;
    s2b;
    sstore 4;
    sload_2;
    sload_3;
    smul;
    s2b;
    sstore 4;
    return;
```

Figure 4.4: Bytecode of the `process` method

This applet does not check the **CLA** and **INS** fields contained in the command APDU. Therefore, all command APDUs will execute the `process` method. Variable `b` is assigned the value `0xAB`. After processing the addition and multiplication statements, the following response APDU will be received:

```
< 0x90 0x00
```

This response indicates that the applet is executed successfully.

4.2.3 Obtaining power traces

The acquisition of the power trace depicted in Figure 4.5 is started immediately after sending the last byte of a command APDU. The last part of the power trace represents the response APDU sent serially by the smart card (LSB first, one start bit, even parity and one stop bit). In this case the response APDU was `0x9000`, because the Java Card applet executed successfully. The `0x9000` response code is returned after approximately 3 ms. Therefore, the execution of the actual Java Card applet must be the first part of the power trace (i.e. from 1ms to 3ms).

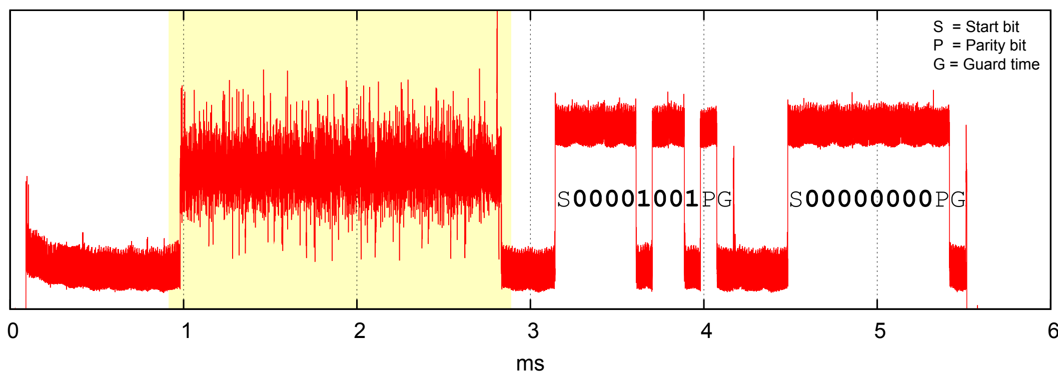


Figure 4.5: Single power trace while executing the reference applet

Now that the approximate start time and duration of the Java Card applet are determined, a larger number of power traces can be obtained. From these power traces, an average power trace can be constructed, as depicted in Figure 4.6. Note that a repeated pattern is clearly visible in the power trace. Moreover, the `smul` instructions can easily be distinguished from the `sadd` instructions.

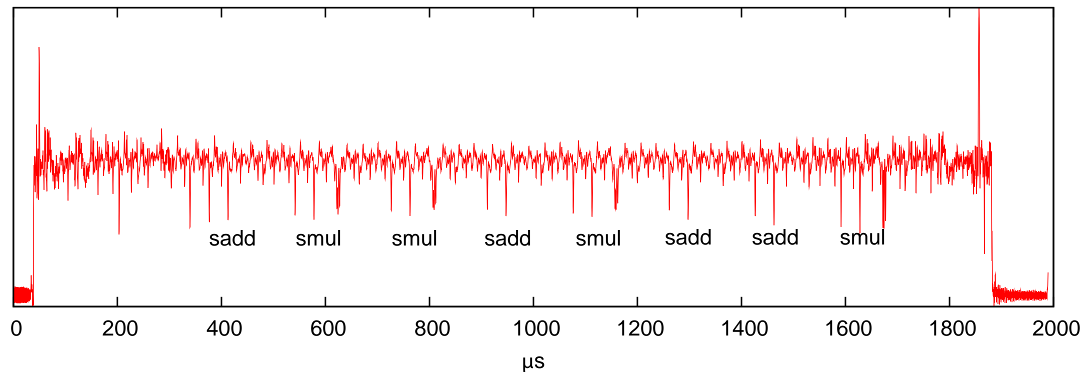


Figure 4.6: Average of 8000 power traces during the execution of the reference applet

4.2.3.1 JCVM template

There exist smart cards with a hardware implementations of the JCVM. For example, STMicroelectronics sells smart card cores that allow direct execution of the majority of the Java Card byte codes [17]. However, at this moment, most Java Card smart cards contain a software interpreter to execute bytecode. Therefore, it is likely that the JCVM fetch, decode and execute stages can be recognised in a power trace. As the fetch and decode stages of the JCVM always perform the same task, the power trace is not expected to differ significantly. In contrast, the actual execution of the instruction is expected to be different. Figure 4.7 shows a repeated pattern that represents the fetch and decode stages of the JCVM. This pattern is referred to as *JCVM template*. The existence of this template is advantageous, because this template can be used to split the power trace into separate parts representing the individual instructions.

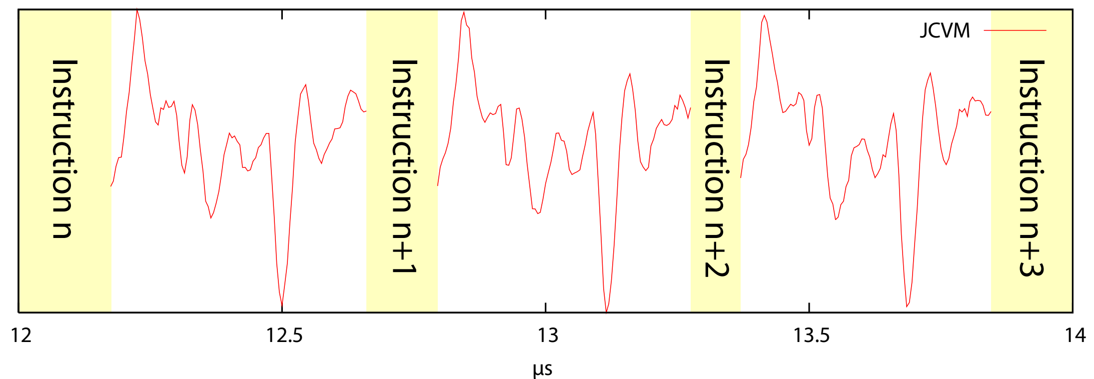


Figure 4.7: JCVM visible in the power trace

4.2.3.2 Instruction templates

Recognising instructions in a power trace requires each instruction to be represented by a unique template. In order to determine the template of a specific instruction, an applet that contains this instruction must be developed. For example, the source code fragment depicted in Figure 4.3 can be used to determine templates of 10 different instructions (i.e. `aload`, `baload`, `return`, `s2b`, `sadd`, `sconst`, `sload`, `sload`, `smul` and `sstore`).

By comparing the separated parts of the power trace with the bytecode of the known Java Card applet, it is possible to link these parts to a instruction and store them as a template for that specific instruction. Figure 4.8 depicts the templates obtained for the `baload`, `sstore` and `sload` instructions respectively. We will see later that some instruction sequences occur more frequently. Therefore, frequently occurring sequences can be stored in a single template. These templates are referred to as *combined templates*. For example, `smul+sstore` is a combined template.

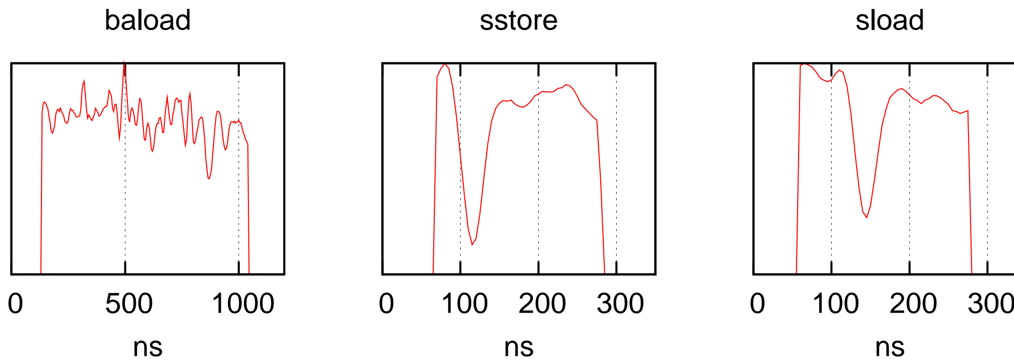


Figure 4.8: Templates of the `baload`, `sstore` and `sload` instructions

It is important to realise that the executed instructions must be matched against the execution trace of the applet, as the order of the executed instructions may differ from the structured bytecode due to loops and conditional statements. Figure 4.9 depicts an example. The `pow` method contains a `for` loop to compute a^n. The figure depicts the Java source code, the bytecode generated by the compiler and the execution trace of `pow(3,2)`. This execution trace is created manually. For more complex algorithms, the execution trace can be obtained using the *Java Card C Reference Implementation* (CREF). This tool simulates a Java Card smart card on a PC.

4.3 Template recognition

In this section, we present the template recognition process. This process is quite similar to the template determination process described in the previous section. First, the applet must be executed, as explained in Section 4.3.1. Then, power traces of the applet execution have to be obtained, as described in Section 4.3.2.

```
1  // Java source code
2  public short pow(short a, short n)
3  {
4     short ret = 1;
5
6     for (short d = 0; d < n; d++)
7        ret *= a;
8
9     return ret;
10 }
```

```
1  // Bytecode
2  L0: sconst_1;
3      sstore_3;
4      sconst_0;
5      sstore 4;
6      goto L2;
7  L1: sload_3;
8      sload_1;
9      smul;
10     sstore_3;
11     sinc 4 1;
12 L2: sload 4;
13     sload_2;
14     if_scmplt L1;
15 L3: sload_3;
16     sreturn;
```

```
1  // Execution trace of pow(3,2)
2  L0: sconst_1;
3      sstore_3;
4      sconst_0;
5      sstore 4;
6      goto L2;
7  L2: sload 4;
8      sload_2;
9      if_scmplt L1; // branch!
10 L1: sload_3;
11     sload_1;
12     smul;
13     sstore_3;
14     sinc 4 1;
15 L2: sload 4;
16     sload_2;
17     if_scmplt L1; // branch!
18 L1: sload_3;
19     sload_1;
20     smul;
21     sstore_3;
22     sinc 4 1;
23 L2: sload 4;
24     sload_2;
25     if_scmplt L1; // no branch!
26 L3: sload_3;
27     sreturn;
```

Figure 4.9: Java source code, bytecode and the execution trace of `pow(3,2)`

4.3.1 Executing the (un)known applet

As explained earlier, an applet has to be selected using its AID, before it can be executed. On some smart cards it is possible to obtain a list of available AIDs that would allow execution of unknown applets. Besides the AID, the supported instructions and their respective classes have to be obtained. Although these can be found using a simple brute force technique, another way is to eavesdrop on the I/O channel of the smart card while it is used in a real application [16]. An advantage of this method is that also P1, P2 and the length of the data field of the command APDUs will be revealed. This technique can of course also be used to determine the AID.

4.3.2 Obtaining and recognising power traces

When the AID and the APDUs are known, the applet can be executed while measuring the smart card's power consumption. The templates determined earlier can then be used to process the unknown power trace. We developed an Inspector module that matches n templates against an average power trace automatically by using the correlation technique described in Section 2.3.2. Algorithm 6 shows the pseudocode of the Inspector module `RecogniseBytecodes`. The parameter *trace* contains an average power trace of the applet execution.

The result of this program is a set of n traces containing the correlation of the power trace with each template. The result of the template matching process is depicted in Figure 4.10. The average power trace (shown in red on the first row) and its correlation

Algorithm 6 Pseudocode RecogniseInstructions module

```
1: function RecogniseInstructions(trace)
2:     templates[ ] ← LoadTemplates()
3:     for i = 0 to templates.length do
4:         template ← templates[i]
5:         correlationTrace[i].name ← template.name
6:         for j = 0 to trace.length − template.length do
7:             correlationTrace[i].samples[j] ← Correlation(template, trace[j..j + template.length])
8:         end for
9:     end for
10:    return correlationTrace
11: end function
```

with the JCVM template and the templates for `sload`, `baload`, `sadd+s2b+sstore` and `smul+s2b+sstore` respectively are shown. From Figure 4.10 can be concluded that the instruction sequence `smul+s2b+sstore` is probably executed three times during the applet execution.

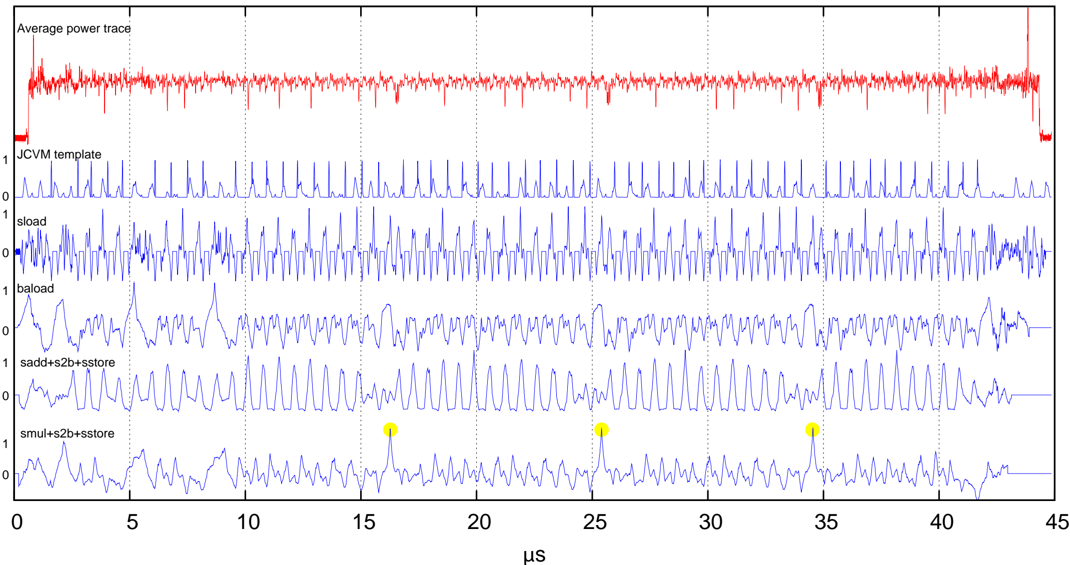


Figure 4.10: Result of the template matching process

4.4 Power analysis results

Although the techniques described in this chapter perform well for most instructions, there exist some special cases. First of all, Section 4.4.1 describes that instructions which perform similar tasks, show very similar power profiles. Second, Section 4.4.2 shows that the duration of some instructions is not fixed, which makes it more difficult to create a template for these instructions. However, we will see that the duration provides useful information. Next, we will discuss cryptographic operations in Section 4.4.3. Finally, in Section 4.4.4, results with some other smart cards are described.

4.4.1 Similar instructions

Some instructions that are available in the JCVM are used to optimise common operations. For example, loading a `short` value from local variable 2 or 3 can be performed using `sload_2` or `sload_3` respectively. We acquired the power traces at 200 MHz, as distinguishing between similar instructions, such as `sload_2` and `sload_3`, is difficult based on resampled measurements. We performed 12500 measurements of the power consumption during the execution of an `sload_2` instruction and another 12500 measurements during the execution of an `sload_3` instruction. We augmented each trace with the opcode of the executed instruction. Figure 4.11 depicts the average power consumption of all measurements. It also depicts the correlation between the measurements and the augmented opcode.

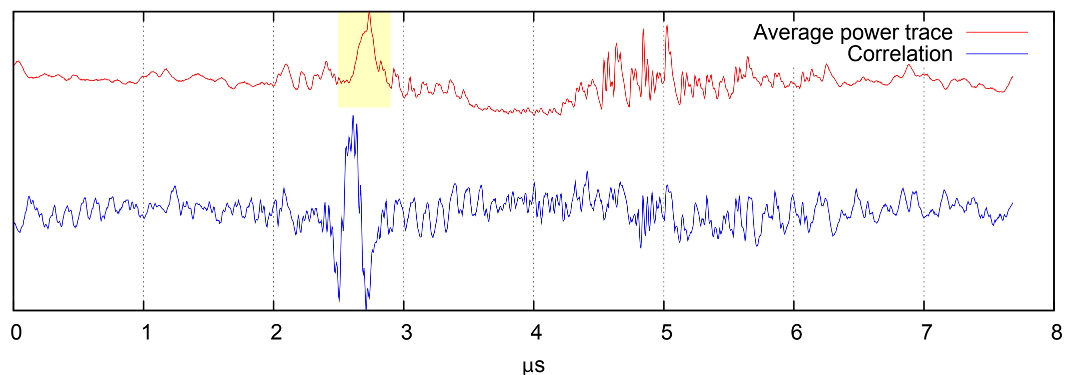


Figure 4.11: Correlation between the power trace and the type of `sload` operation performed

Figure 4.11 shows that there is a minimal difference in the power consumption between `sload_2` and `sload_3`. This difference is indicated by the peak in the correlation function (from 2.5 µs to 2.7 µs). Figure 4.12 depicts the difference between `sload_2` and `sload_3` in this region. Note that the difference is minimal, as there is only a significant difference during a small period of time (i.e. approximately 400ns). Although it is possible to determine the type of `sload` operation, a significant number of traces must be collected and therefore it is very time consuming.

4.4.2 Instruction duration

In this section, we show that the duration of some instructions is not fixed. First of all, Section 4.4.2.1 shows that the duration of the `sdiv` instruction is dependent on the quotient. Therefore it is difficult to recognise this instruction using a fixed-size template. Next, the duration of the conditional branch instruction (like `ifle`) is also dependent on input data, as described in Section 4.4.2.2.

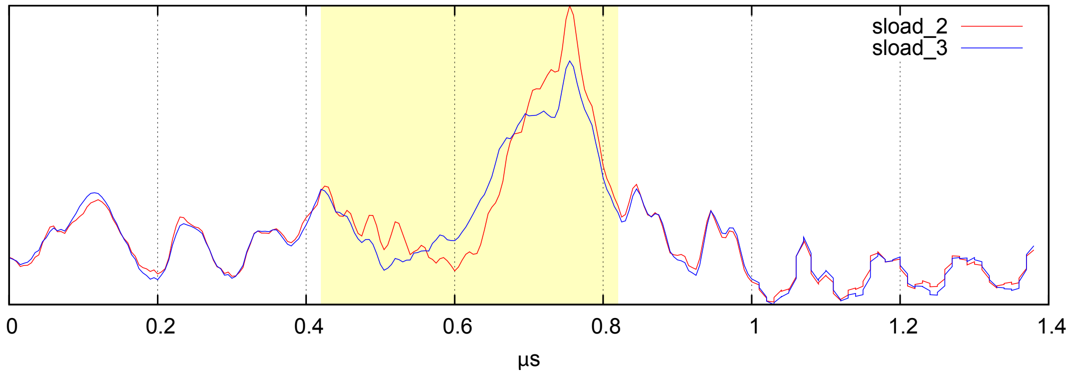


Figure 4.12: Difference between `sload_2` and `sload_3`

4.4.2.1 `sdiv` duration

At first, a template for the `sdiv` instruction was difficult to determine, because the time needed for the division operation is variable. The duration of the `sdiv` instruction appeared to be dependent on the quotient. We investigated this dependence by collecting 2000 traces of each of the following divisions.

- 256 / 11 = 23 (0b0000000000010111)
- 1239 / 3 = 413 (0b0000000110011101)

The average power traces of these `sdiv` operations are depicted in Figure 4.13. Note that the individual bits of the quotient can be easily read from this power trace. Probably a simple algorithm using left shifts and subtractions is used on the smart card we investigated to implement the division. A 1-bit in the quotient requires more steps in the algorithm (i.e. subtracting the divisor from the partial remainder and incrementing the quotient).

It should be noted that this is a serious vulnerability. Besides the fact that the quotient can be read from the power trace, it is also possible to determine the input values if one of them is known. If one of the input values of the division instruction (i.e. dividend or divisor) can be changed, it is even possible to determine the other value statistically by just analysing the time required for the operation.

A more secure algorithm for division on secure devices like smart cards, is proposed in [7]. Unfortunately, we were not able to perform power analysis on this algorithm, as we do not have a smart card that implements it.

4.4.2.2 Conditional branch instructions

Figure 4.14 shows a fragment of the reference applet that we used to determine the template for the `ifle` instruction. Variable `a` is supplied to the applet via the command APDU. The statements performed in the `if` part and `else` part are intentionally kept identical.

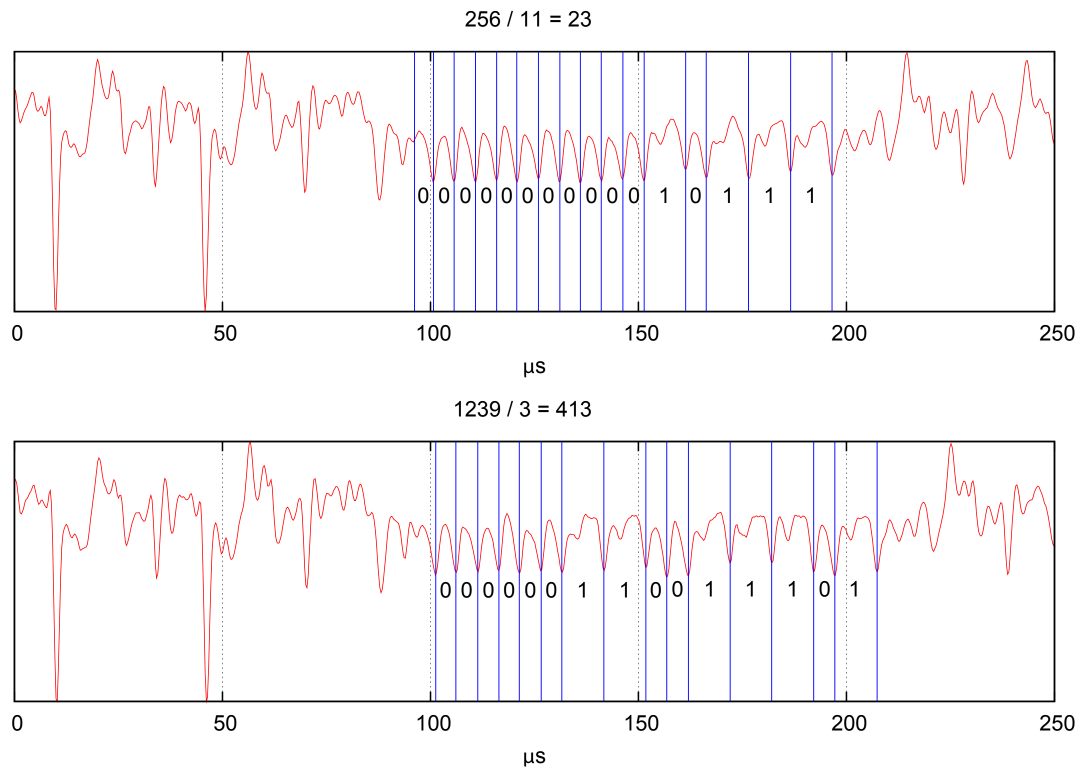


Figure 4.13: Average power traces of two `sdiv` instructions

```
1       a = buffer[(short)(ISO7816.OFFSET_CDATA)];
2
3       if (a > 0)
4         a = 1;
5       else
6         a = 1;
```

Figure 4.14: Fragment of the applet to determine the `ifle` template

By alternating variable `a` between 0 and 1, we tried to determine the template for the `ifle` instruction, we discovered that the duration of this instruction depends on whether the branch is taken or not. Figure 4.15 depicts this difference. Note that the rest of the trace is identical, but shifted by 4 µs.

Such behaviour can be explained by the fact that the JCVM has to compute the address at which the execution should be proceeded. Figure 4.15 depicts the difference between a branch which is not taken and a taken branch respectively. While decompiling, it may be helpful to know whether a branch is taken or not. For example, one can draw conclusions about operands (i.e. variable `a` in this case). Therefore, we stored two templates for this instruction. Both are augmented with a flag that indicates whether the branch was taken or not.

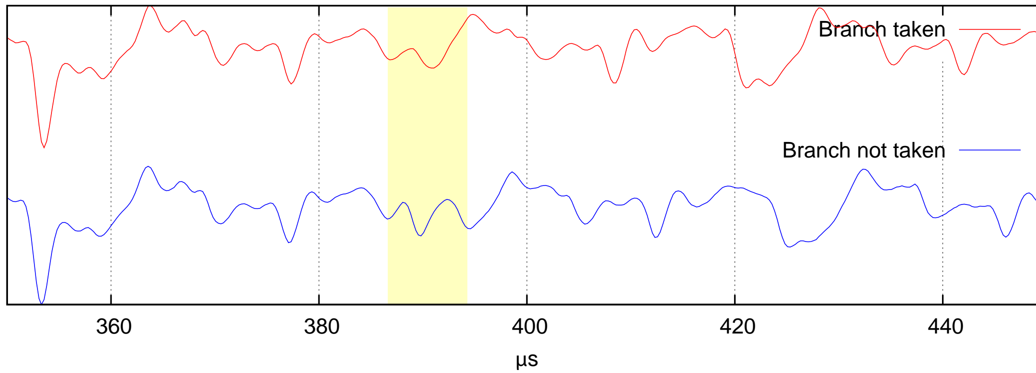


Figure 4.15: Duration of the `ifle` instruction

4.4.3 Cryptographic operations

Most Java Card applets invoke cryptographic algorithms like DES or RSA. These algorithms are commonly implemented as native software functions or in hardware and can be invoked using the Java Card cryptography API. Although reverse engineering of native functions and functions implemented in hardware are outside the scope of this project, it is interesting to see if these operations can be identified in a power trace. Figure 4.16 depicts a power trace of an applet that executes a single DES operation using the Java Card cryptography API. From this power trace, we can create a template. This allows us to recognise the execution of DES operations in a Java Card applet.

Although recognising a DES operation in a power trace is not difficult, there is one practical problem. As described in Section 3.2, the PicoScope oscilloscope can measure 5 ms at the maximum sample rate. Because the actual DES operation alone, already takes approximately 10 ms, we are forced to decrease the sample rate and therefore reduce the precision of our measurements. Acquiring the power trace in several iterations would be another possible solution to this problem.

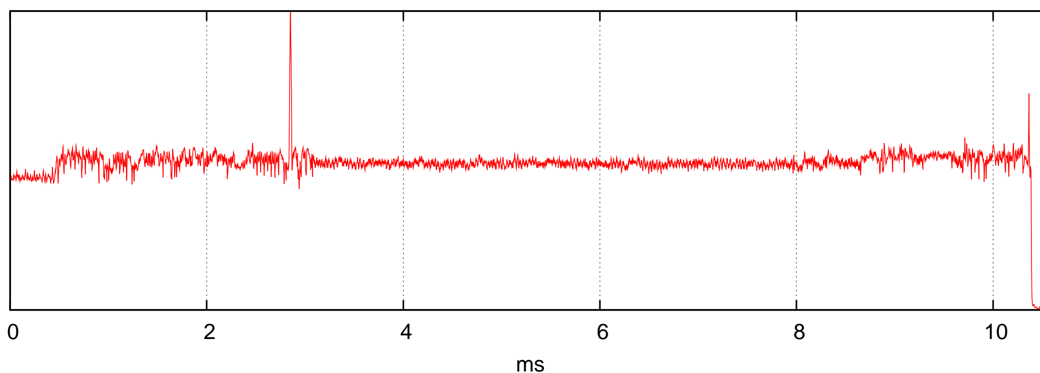


Figure 4.16: Power trace of a DES operation called from Java

4.4.4 Other Java Card smart cards

In this section, we will see the power traces of a few other smart cards. All of these smart cards are loaded with the same example reference applet (i.e. the applet depicted in Figure 4.3). First of all, we show the average power traces of three revisions of a smart card (same brand and type). These three revisions are referred to as *Smart card 1*, *Smart card 2* and *Smart card 3*, purchased in 2003, 2005 and 2006 respectively. Then, in Section 4.4.4.2 we will discuss *Smart card 4*. The brand of this smart card differs from the other three smart cards.

4.4.4.1 Smart card 1, 2 and 3

The differences between the three smart cards of the same brand and type, while executing the same reference applet, are depicted in Figure 4.17. Besides the difference in power consumption and speed, *Smart card 1* is similar to *Smart card 2*. *Smart card 3* however, shows a different power profile. The power consumption is higher and the execution is slower. When inspecting the power trace of *Smart card 3*, one can see ripples that represent the execution of instructions.

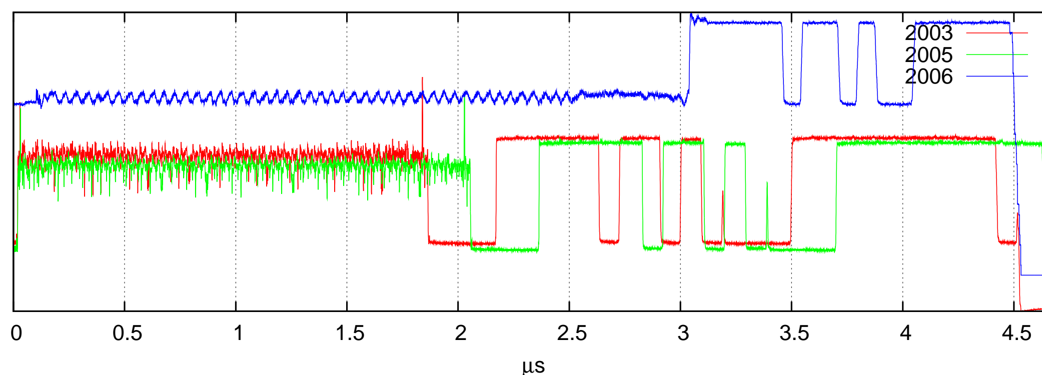


Figure 4.17: Differences in power consumption

An interesting question is whether the newer revisions of the smart cards are more secure than the older smart cards. We investigated this using the following experiment. The `baload` instruction has a power profile that can be easily recognised on all three smart cards. Therefore, we forced the execution of this instruction during each addition or multiplication statement, as shown in Figure 4.18.

We executed the applet 5,000 times on all smart cards. Then, we resampled the traces at 4 MHz, aligned them and computed an average power trace.

Next, we manually selected the first multiplication statement (line 11) in the power trace and we correlated this selected part against the power trace, as if it was a template. We found that the selected multiplication statement has a correlation with the other multiplication statement, as well as with the addition statement. Table 4.1 shows the results of this experiment. The first column indicates the smart card on which the experiment is performed. The second column shows the correlation of the selected mul-

```
1  public void process(APDU apdu)
2  {
3     byte a = (byte) 0x03, b, c;
4
5     if (selectingApplet())
6        return;
7
8     byte buffer[] = apdu.getBuffer();
9     short len = apdu.setIncomingAndReceive();
10
11    c = (byte)(a * buffer[(short)(ISO7816.OFFSET_CDATA)]);
12    c = (byte)(a * buffer[(short)(ISO7816.OFFSET_CDATA)]);
13    c = (byte)(a + buffer[(short)(ISO7816.OFFSET_CDATA)]);
14 }
```

Figure 4.18: Fragment of the applet used for the quality comparison

tiplication statement with the other one. The third column shows the correlation of the selected multiplication statement with the addition statement.

Smart card	**Correlation with** multiplication	**Correlation with** addition
1 (2003)	97.78%	64.59%
2 (2005)	99.34%	67.81%
3 (2006)	95.55%	90.16%

Table 4.1: Correlation quality

As shown in Table 4.1, the first two smart cards do not differ significantly. It is likely that the higher correlation of *Smart card 2* is caused by the slower applet execution on that card. The correlation of 95.55% for *Smart card 3* indicates that the multiplication can be recognised. However, the correlation with the addition statement is also significant. Therefore, with respect to distinguishability of the multiplication and addition statements, *Smart card 3* is more secure than *Smart card 1* and *Smart card 2*.

4.4.4.2 Smart card 4

Smart card 4 was the first Java Card smart card that we tried to reverse engineer. Unfortunately this smart card contains a countermeasure that adds noise by shifting. Therefore, the average power trace becomes a flat line, as depicted in Figure 4.19.

A more accurate inspection of the power traces revealed that the power trace is similar each 88th iteration. We did not further investigate this smart card. However, we do expect that a repeated pattern (i.e. the execution of instructions) will become visible when an average of each 88th power trace is made. Unfortunately this means that we should execute the applet 88 times to obtain one power trace.

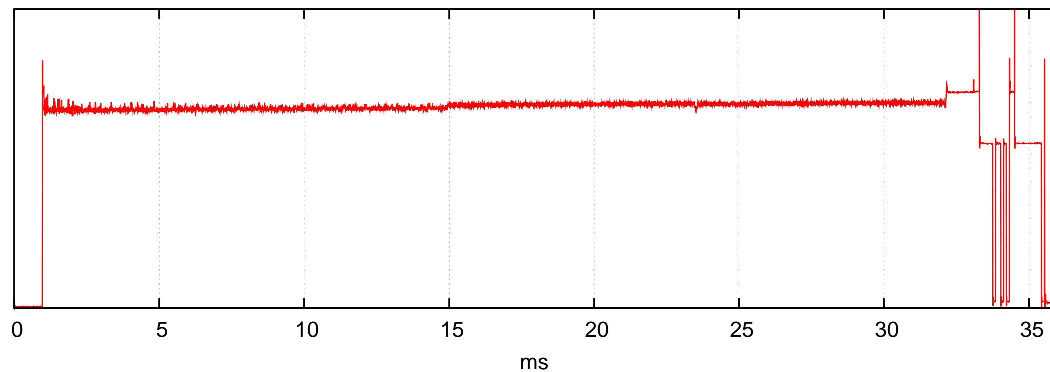


Figure 4.19: Average power trace of *smart card 4*

5 Reverse engineering of Java Card applets

The previous chapter described how to obtain and process power traces. In this section we will discuss the reverse engineering process, based on the results of the power analysis.

Table 5.1 shows the results of the power analysis of an applet that performs two addition statements. The table is based on the information obtained using the template recognition process, as described in Section 4.3. The first column shows the expected execution trace, which is of course not available when an unknown applet is reverse engineered. The second column shows the execution trace obtained from the power analysis. This column contains the instructions with the highest correlation. The third column shows instructions with a lower correlation. As shown, the execution trace obtained from the power analysis contains uncertainties and even one error. On line number 6 the `aload` instruction has a better correlation than the `sload` instruction. Errors like this one can be detected and recovered using techniques described in this chapter.

Table 5.1: Example execution trace obtained from the power analysis.

Expected	**Recognised**	**Alternatives**
`sload`	`sload` (93%)	`aload` (89%)
	JCVM	
`sload`	`sload` (92%)	`aload` (91%), `sconst` & `sstore` (57%)
	JCVM	
`sadd`	`sadd` (91%)	`sload` (55%), `aload` (51%)
	JCVM	
`s2b & sstore`	`s2b` & `sstore` (91%)	`sload` (51%)
	JCVM	
`sload`	`sload` (92%)	`aload` (78%), `sconst` & `sstore` (54%)
	JCVM	
`sload`	~~`aload` (92%)~~	`sload` (91%)
	JCVM	
`sadd`	`sadd` (90%)	`sload` (54%), `aload` (53%)
	JCVM	
`s2b & sstore`	`s2b` & `sstore` (90%)	`sload` (53%)

Reverse engineering using power analysis differs from reverse engineering of an executable on a PC. Although reverse engineering of an executable on a PC is certainly not trivial, it is easier than reverse engineering using power analysis, because of the following reasons:

- On a PC, the structured program is available. Power analysis results in an execution trace. In case of a conditional branch, only one execution path is visible;
- In contrast to an executable program on a PC, the execution trace obtained using power analysis may contain errors or uncertainties due to a noisy signal.

In order to improve the quality of our reverse engineering process, additional information sources are desirable. These information sources are covered in Section 5.1. By using the techniques described in this section, an improved execution trace is obtained. The processing of this execution trace is described in Section 5.2. Finally, Section 5.3 describes a technique to decompile the structured bytecode to the original Java source code.

5.1 Additional information sources

In this section, the additional information sources are described. In Section 5.1.1, correlation with input data is covered. In Section 5.1.2, we will see that not all instruction sequences are possible. In addition, some instruction sequences are unlikely to occur, as explained in Section 5.1.3. In Section 5.1.4, statistics of Java Card bytecode are shown. Section 5.1.5 shows that the duration of an instruction can also be used to obtain information.

5.1.1 Input data

Besides correlating a power trace with templates, as described in the previous chapter, correlation with input data contained in the command APDU can also be used to determine which instruction uses the input data. The example in Figure 5.1 depicts the average power trace of a single `smul` instruction. In addition, it also depicts the correlation with one of the operands of the `smul` instruction and the correlation with a random byte, which is not used by the `smul` instruction. From Figure 5.1 can be concluded that it is possible to determine if a specific input value is used by an instruction.

5.1.2 Impossible instruction sequences

Not all instructions can follow each other in a valid execution trace of a Java Card applet. During the reverse engineering process, it is advantageous to keep an operand type stack. Although storing the operands themselves is difficult, storing their types is much easier. Based on the elements on top of the operand type stack, some instructions can be excluded from the set of possible following instructions. Note that this technique will reduce the search space and hence speed up the overall process.

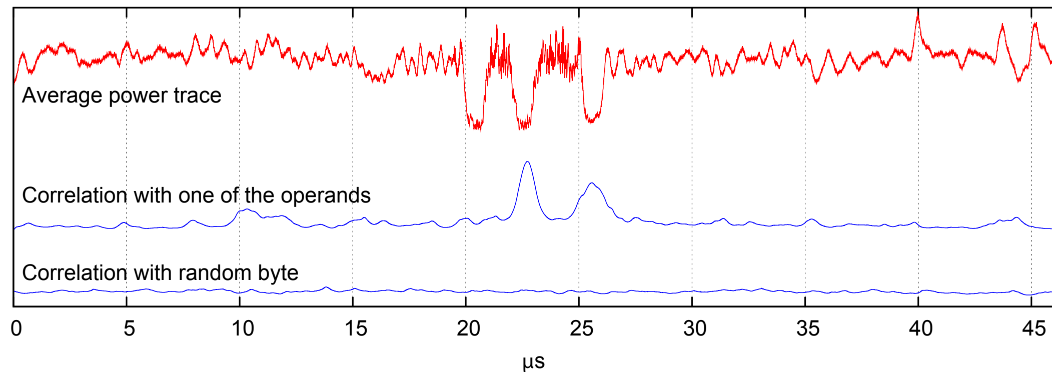


Figure 5.1: Correlation between input data and the power trace

For example, the `aload_<index>` instruction would push an object reference onto the operand stack, as described in [19]:

> ... The *objectref* in the local variable at *index* is pushed onto the operand stack. ...

The `sadd` instruction is described in the JCVM specification as follows:

> ... Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The `short` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack. ...

This technique can be applied to the example of Table 5.1. The impossible instruction sequence in this example is:

`sload`, `aload`, `sadd`

The `sadd` instruction expects two `short` values on top of the operand stack, while the `aload` instruction pushes an *objectref*. Therefore, the `aload` candidate must be replaced by an alternative instruction (i.e. the `sload` instruction that matches for 91%). This results in:

`sload`, `sload` (first alternative), `sadd`

In this case, it is assumed that the `sload` instruction on line 5 and the `sadd` instruction on line 7 are correctly recognised.

The pseudocode of the program that checks impossible instruction sequences, is shown in Algorithm 7. The GetPoppedTypes and GetPushedTypes functions return an array of the popped and pushed types respectively.

Algorithm 7 Pseudocode CheckSequence program

```
 1: procedure CheckSequence(sequence[ ])
 2:     stack = [ ]
 3:     for i = 0 to sequence.length do
 4:         pop[ ] ← GetPoppedTypes(sequence[i])
 5:         push[ ] ← GetPushedTypes(sequence[i])
 6:         for j = 0 to pop.length do
 7:             popFromStack ← stack.pop()
 8:             if popFromStack ≠ pop[j] then
 9:                 Print("Line number: " + i)
10:                 Print("Instruction: " + sequence[i])
11:                 Print("Expected type: " + pop[j])
12:                 Print("Found type: " + popFromStack)
13:             end if
14:         end for
15:         for j = 0 to pop.length do
16:             stack.push(push[j])
17:         end for
18:     end for
19: end procedure
```

5.1.3 Unlikely instruction sequences

Besides impossible instruction sequences, as described in the previous section, there are instruction sequences that are unlikely to occur even they are allowed by the JCVM. Table 5.2 shows some examples of unlikely instruction sequences.

Table 5.2: Examples of unlikely instruction sequences.

Sequence	**Description**
`sconst_0`, `sdiv`	Divide by constant 0
`sneg`, `sneg`	Negate two times
`sadd`, `pop`	Add two `short` values and pop the result
`sload_2`, `sstore_2`	Assign variable to itself (e.g. `a = a;`)

Automatically determining all of these sequences is difficult and manually determining the sequences is a lot of work. Therefore, it is advantageous to define the unlikely instruction sequences in a more generic manner, as shown in Table 5.3. The first generic sequence covers 48 instruction sequences, as the *ALU operation* group contains 24 instructions and the *Pop operation* group contains 2 instructions.

Table 5.3: Examples of generic unlikely instruction sequences.

Sequence	**Description**
ALU operation, *Pop operation*	Perform an ALU operation and pop the result
sload_x, *sstore_x*	Assign local variable x to itself

5.1.4 Instruction statistics

The previous section described unlikely instruction sequences. These sequences can not be determined easily. Another way of determining unlikely instruction sequences is by using statistical information based on a large execution trace. As obtaining a large execution trace from a smart card is difficult, we used the CREF to generate such execution trace. At the time the execution trace was made, the CREF was installing some Java Card applets. Although the implementation is not known, this will probably execute some complex methods. The execution trace contains a sequence of nearly 100,000 instructions.

Bytecode of Java Card applets on a smart card is usually generated by the Java compiler. Therefore, certain instructions will occur more often than others. As described in [15], 90% of an execution trace contains less than 45 out of the 250 standard Java instructions. Furthermore, approximately 30-40% of the execution trace consists of load operations (e.g. `sload_1`). We performed a similar experiment for Java Card bytecode. The results of this experiment are shown in Table 5.4.

Table 5.4: Java Card bytecode statistics

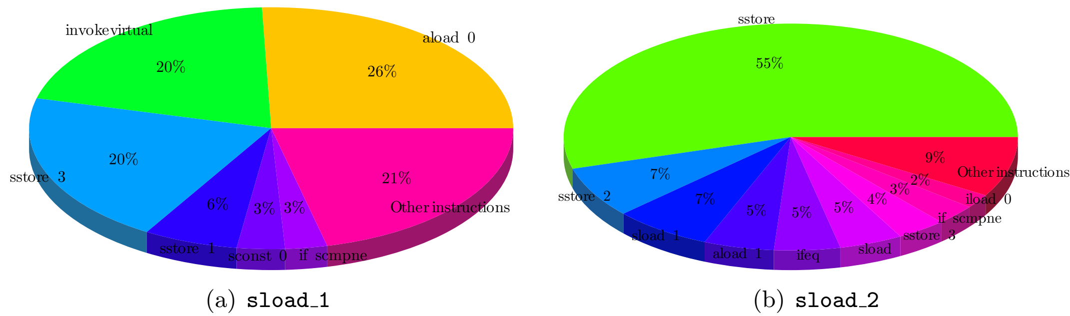
Instructions group	**Percentage**
Load	31.9%
Stack	19.4%
Method call	15.5%
Store	9.3%
ALU	9.1%
Branches	6.4%
Constant pool	5.1%
Jump	1.5%
Other instructions	1.8%

Besides statistics about individual instructions, it is also possible to obtain statistics about instruction sequences. Suppose that instruction I is executed. Based on the CREF execution trace, we can find the possible instructions that follow I and the chance that each of them follows I. Figure 5.2 depicts the possible instructions that follow the `sload_1` and `sload_2` instructions. We can see in the figure, that if `sload_1` is executed, there is a probability of 26% that an `aload_0` is executed next.

The same technique can also be used to obtain the possible instructions that follow I. In addition, we can find the instructions that precede or follow an instruction sequence. For example, Figure 5.3 depicts the possible instructions that follow the instruction sequence `aload_1+sload_2`.

5.1.5 Instruction duration

In some situations the duration of an instruction provides useful information. As already explained in Section 4.4.2.2, we found that the duration of a conditional branch instruction indicates if a branch is taken or not. For example, the duration of the `if_scmplt`



(a) `sload_1` (b) `sload_2`

Figure 5.2: Probable instructions that follow the `sload_1` and `sload_2` instructions

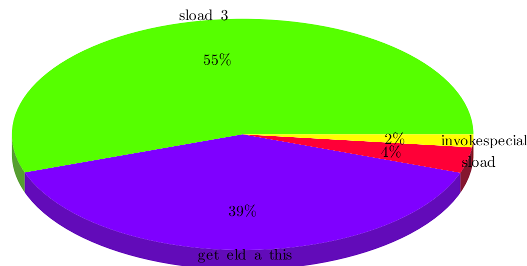


Figure 5.3: Probable instructions following the `aload_1+sload_2` sequence

instruction is approximately 5.75µs. In case of a taken branch the duration increases by 4.5µs to 10.25µs.

5.2 Execution trace processing

Using the techniques described in the previous sections, it is possible to obtain the most likely execution trace. In order to reverse engineer a Java Card applet completely, the execution trace should be transformed to structured bytecode. This step is certainly not trivial, because an execution trace may contain (nested) loops. The structured bytecode of a `for` loop will always be generated as depicted in Figure 5.4. Basically, it contains a `goto` instruction, 2 labels and an `if_scmplt` instruction.

In the ideal case, the reverse engineering process would generate the execution trace depicted in Figure 5.5.

The rest of this section covers techniques to recover the original structured source code as good as possible. Section 5.2.1 shows how an unrolled `for` loop can be rerolled. In Section 5.2.2, we explain why conditional branch instructions make reverse engineering process more difficult.

```
1       sconst_0       // initialisation
2       sstore_2
3       goto L2
4   L1: sload_3        // loop body
5       sload_2
6       sadd
7       sstore_3
8       sload_2        // increment loop variable
9       sconst_1
10      sadd
11      s2b
12      sstore_2
13  L2: sload_2        // condition
14      bspush    3
15      if_scmplt L1
```

Figure 5.4: A `for` loop as generated by the Java compiler

```
1  sconst_0         15 sstore_2         29 bspush    3
2  sstore_2         16 sload_2          30 if_scmplt
3  goto             17 bspush    3      31 sload_3
4  sload_2          18 if_scmplt        32 sload_2
5  bspush    3      19 sload_3          33 sadd
6  if_scmplt        20 sload_2          34 sstore_3
7  sload_3          21 sadd             35 sload_2
8  sload_2          22 sstore_3         36 sconst_1
9  sadd             23 sload_2          37 sadd
10 sstore_3         24 sconst_1         38 s2b
11 sload_2          25 sadd             39 sstore_2
12 sconst_1         26 s2b              40 sload_2
13 sadd             27 sstore_2         41 bspush    3
14 s2b              28 sload_2          42 if_scmplt
```

Figure 5.5: Execution trace of the program depicted in Figure 5.4

5.2.1 Loop rerolling

Consider the execution trace depicted in Figure 5.5. It shows the execution of an unrolled `for` loop which is iterated 3 times. As shown in Figure 5.6 the execution trace can be divided into several parts. First of all, lines 3-6 indicate the presence of a loop. The `goto` statement is used to branch to the conditional part of the loop which loads a `short` value (`sload`), pushes a constant (`bspush`) and branches if the `short` comparison succeeds (`if_scmplt`). Second, the lines following the `goto` statement (i.e. lines 4-6) can be used to split the execution trace of the loop into parts. This process is depicted in Figure 5.6. The end of the loop is reached when the conditional branch instruction is not followed by the loop body. In addition, the duration of the conditional branch instruction may also indicate the end of the loop, as explained earlier.

Besides reconstructing the loop, this technique has other advantages. First of all it is possible to derive the labels originally used on lines 3, 6, 15, 24 and 33. Second, it is very common that the same loop variable is used in the initialisation, condition and increment part of the loop. Therefore it is likely that the instructions on lines 2, 4, 8, 12, 13, 17, 21, 22, 26, 30 and 31 share the same local variable index. Third, based on the number of loop iterations, the operand of the `bspush` instruction can be reconstructed.

Although this technique works for this relatively simple example, it is difficult to completely automate this process. Especially when nested loops are involved or when a loop contains conditional statements. Moreover the execution traces may contain errors. Detecting a nested loop as such is less difficult, because the nested loop will cause an additional `goto` statement.

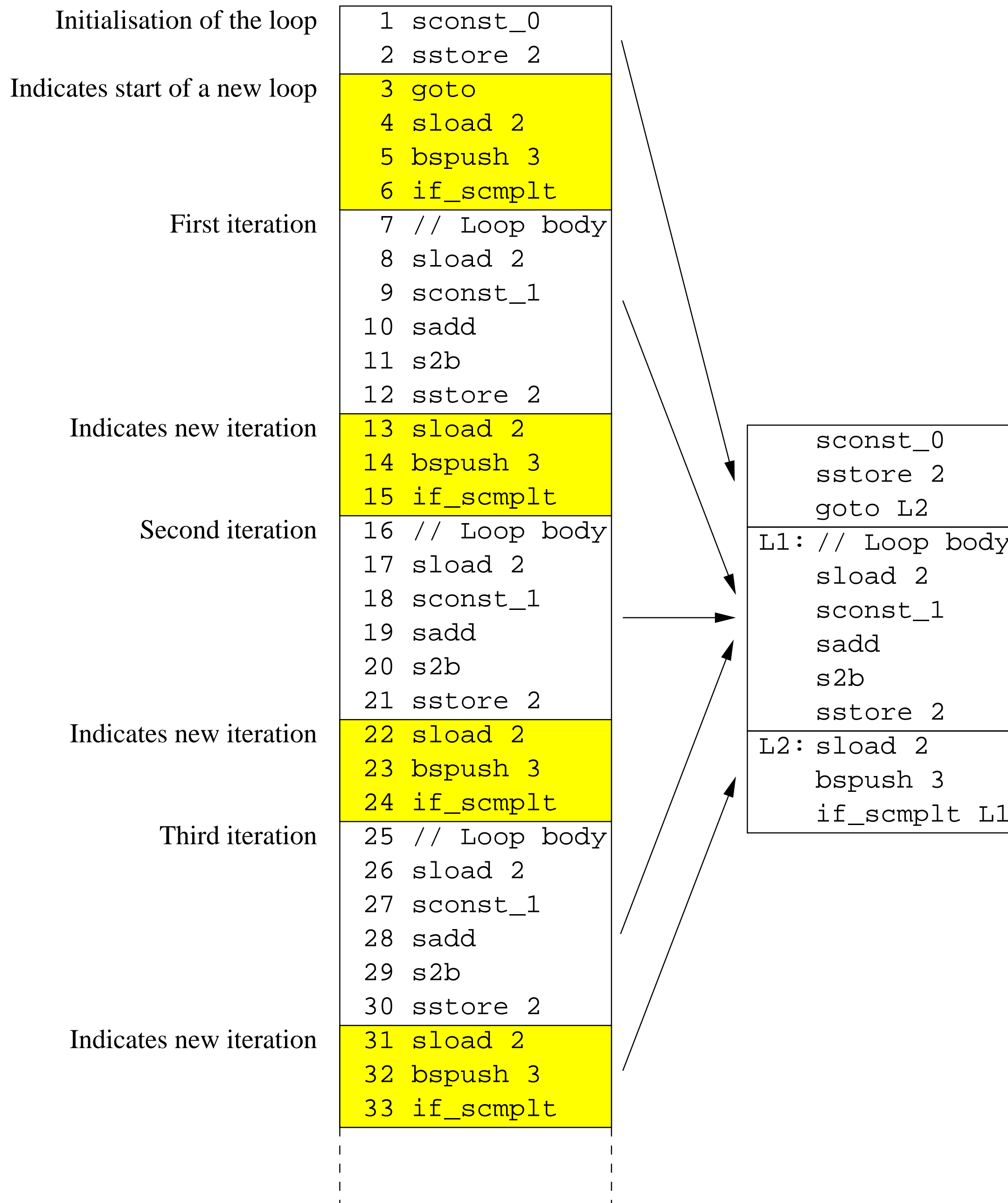


Figure 5.6: Loop rerolling

5.2.2 Conditional branches

Conditional branch instructions, such as `if_scmplt`, make the reverse engineering process more difficult. By varying the input data, it is possible to enforce another part of the source code to be executed. Without knowledge of the source code it can be difficult to determine on what input data a conditional branch instruction is dependent. There are two ways to determine such dependency:

- Use correlation between random input data and the power profile of the conditional branch instruction. When a conditional branch depends on input data, it is likely that a correlation is found. Then, altering this data will possibly cause the branch to execute another part of the program;
- Inspect the (partially) reverse engineered applet first and try to derive what input data is used in the condition.

It is however possible that a varying input data does not affect the conditional branch, for example when it is based on an internal state or data from a random generator. In this case the condition has to be determined from the partially reverse engineered source code.

5.3 Decompilation

When the structured bytecode is available, it is relatively easy to reconstruct source-level expressions. In [14], a technique to automatically decompile Java bytecode into Java source code is presented. Although the referred paper focuses on decompiling standard Java bytecode, we implemented a Java Card version. Suppose that reverse engineering of a Java Card applet results in the structured Java Card bytecode shown in the first column of Table 5.5.

Although `goto` statements are not allowed in Java, the reverse engineered source code, as shown in the third column, is much more readable than the instructions in the first column. Note that the local variables are prefixed with the first character of their type (e.g. s for short), because the original names are not available. As described in [14], there exist techniques to eliminate the `goto` statements.

Table 5.5: Decompiling Java Card bytecode

Instruction	Stack	Source
sconst_0	"0"	
sstore_2		s2 = 0;
goto L2		goto L2;
L1: sload_3	"s3"	L1:
sload_2	"s3","s2"	
sadd	"s3 + s2"	
sstore_3		s3 = s3 + s2;
sload_2	"s2"	
sconst_1	"s2","1"	
sadd	"s2 + 1"	
s2b	"(byte) (s2 + 1)"	
sstore_2		s2 = (byte) (s2 + 1);
L2: sload_2	"s2"	L2:
bspush 3	"s2","3"	
if_scmplt L1		if (s2 < 3) goto L1;

5.4 Prototype reverse engineering program

The techniques described earlier are implemented in a prototype program. This program has several functions.

First of all it can find impossible instruction sequences, as described in Section 5.1.2. In addition, unlikely instruction sequences could be checked, as described in Section 5.1.3

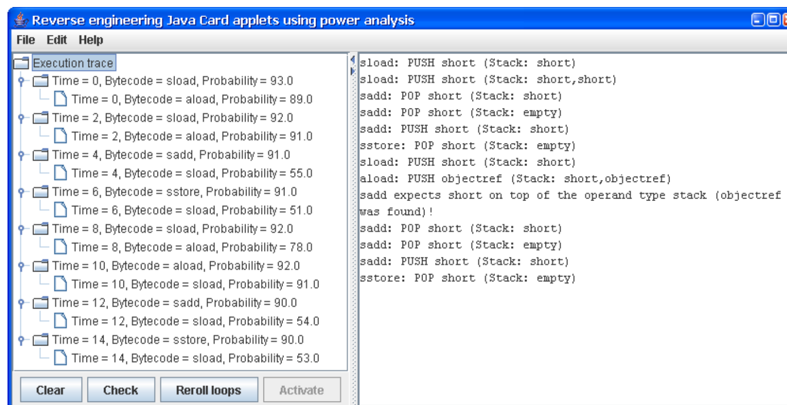


Figure 5.7: Check impossible instruction sequences

Second, the program depicts errors and allows the user to replace the incorrect instruction by one of the alternatives. This is done by selecting the correct instruction and clicking the **Activate** button.

Finally, the program can reroll loops, using the technique described in Section 5.2.1. Note that besides rerolling the loop, the program also detects the number of iterations. Normally, it is very difficult to obtain an operand. Using this technique, the operand of the `bspush` instruction is obtained. In addition, the original labels are reconstructed.

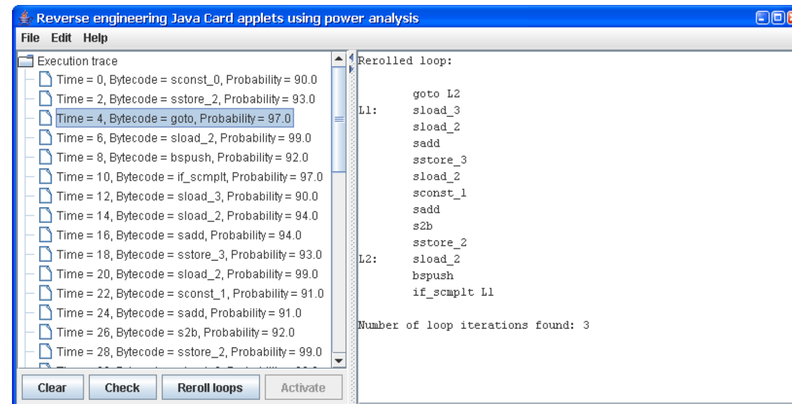


Figure 5.8: Loop rerolling

6 Conclusions

In this report, we showed that power analysis can be used to acquire information about executed instructions on a Java Card smart card. In order to acquire power traces, we built a custom smart card reader that can precisely trigger an oscilloscope. Using acquisition software, this smart card reader is successfully used to obtain power traces of the execution of Java Card applets.

In order to obtain an average power trace, we presented several preprocessing techniques that are applied to the obtained trace set. First of all, we presented a resampling technique to reduce the memory needed to store the power trace, at cost of losing some information. Second, we showed several techniques to align the power traces. Finally, an average of the trace set could be calculated. Based on this average trace and the known execution trace of the applet, we were able to determine the templates for the executed instructions.

These templates could be recognised in a power trace of an unknown Java Card applet using correlation techniques. Experiments showed that different instructions can be identified relatively easy. In contrast, instructions that perform similar tasks cannot be distinguished easily. It is therefore possible that the execution trace obtained using power analysis contains errors or uncertainties. We proposed the following additional information sources that could be used to reduce the number of errors and uncertainties in the execution trace:

- Each instruction has a set of types that is popped from the operand stack and a set of types that is pushed on the operand stack. A sequence of two instructions i_1 and i_2 is valid if and only if the set of types pushed by i_1 matches the set of types popped by i_2. Therefore, some bytecode sequences cannot occur in a valid Java Card applet;

- Some instruction sequences perform useless tasks and are therefore not likely to appear in Java Card applets. It is difficult to determine these sequences automatically. Therefore it is advantageous to specify these sequences as groups, as described in Section 5.1.3;

- Statistical results of other Java Card applets can identify frequently occurring instruction sequences or sequences that are not likely to occur;

- Correlation with input data can be used to determine which instructions depend on input data. This can be advantageous in several situations. For example, when it is known that (a part of) the input data is used by a conditional branch instruction, the branch direction can possibly be affected by modifying the input data;

- The instruction execution duration may provide additional information. For example, we found that from the `sdiv` instruction on *Smart card 1*, the hamming weight of the quotient can be derived from the length of the `sdiv` instruction. Moreover, the actual value of the quotient can be read from the power trace.

The information sources described above, in addition to the results of the template recognition process, result in an execution trace. We presented techniques to generate structured bytecode from this execution trace using loop rerolling. Most of the time however, this step will be difficult, as the execution trace may also contain nested loops and other conditional statements.

Finally, we showed that structured bytecode, once it is available, can be decompiled to Java source code using algorithms which are similar to algorithms that are used to decompile regular Java applications.

The experiments performed during this project were mainly focused on one particular smart card, purchased in 2003. We investigated some other cards as well (i.e. two cards from the same vendor and one from another vendor). A newer revision of the same smart card, purchased in 2005, showed similar power traces. Reverse engineering applets running on the latest revision of this smart card, purchased in 2006, appeared to be more difficult. However, the individual instructions are still visible. Therefore, we expect that reverse engineering applets on the newest revision of this smart card is still possible to some extent, provided that enough power traces are acquired.

During the project, we have identified a couple of countermeasures. These countermeasures make the reverse engineering process more difficult, but we showed that it is still possible to get information about executed instructions.

6.1 Main contributions

The main contributions of this thesis are:

- Improved acquisition system that consists of:
  - A new microcontroller based smart card reader with accurate hardware trigger;
  - JNI driver for the PicoScope 3206 USB oscilloscope;
  - Acquisition software that controls the smart card reader and the oscilloscope.
- Improved techniques to align power traces;
- Technique to determine templates for the various Java Card instructions in a power trace;
- Technique to recognise templates in a power trace;
- Additional information sources that can be used during the reverse engineering process;

- Techniques to generate structured bytecode from an execution trace (e.g. loop rerolling);
- A prototype program that implements various techniques described in this report.

6.2 Future work

During this project, we thought of the following research topics that can be investigated in the future.

- It would be interesting to see if the results in this document can be improved when an oscilloscope with a higher sample rate (i.e. more than 500 MS/s) and larger memory is used;
- This project focused on reverse engineering Java Card applets using power analysis. Besides power analysis, other side channel analysis techniques such as *Electro-Magnetic Analysis* (EMA) exist. EMA has successfully been applied to widely used cryptographic algorithms (e.g. DES and RSA), as described in [3]. It is possible that smart cards with countermeasures against power analysis are vulnerable to these different side channel analysis techniques;
- We discussed countermeasures that we encountered while performing power analysis. It would be interesting to investigate countermeasures that prevent Java Card applets from being reverse engineered using power analysis;
- This project focused on contact smart cards. It would be interesting to see if the techniques described in this document can also be applied to contactless smart cards;
- The different steps of the reverse engineering process, described in this document, require manual actions, especially during the template determination. Developing a program that performs the template determination for all instructions automatically would be interesting.

Bibliography

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

[2] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous, *Differential Power Analysis in the Presence of Hardware Countermeasures*, CHES 2000: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems (London, UK), Springer-Verlag, 2000, pp. 252–263.

[3] Karine Gandolfi, Christophe Mourtel, and Francis Olivier, *Electromagnetic analysis: Concrete results*, CHES 2001: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (London, UK), Springer-Verlag, 2001, pp. 251–261.

[4] International Organization for Standardization, *ISO/IEC 7816-2 – Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 2: Dimensions and location of the contacts*, 1999.

[5] ———, *ISO/IEC 7816-3 – Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 3: Electronic signals and transmission protocols*, 1999.

[6] ———, *ISO/IEC 7816-4 – Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange*, 2005.

[7] Marc Joye and Karine Villegas, *A protected division algorithm.*, CARDIS, USENIX, 2002.

[8] Paul C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems.*, CRYPTO (Neal Koblitz, ed.), Lecture Notes in Computer Science, vol. 1109, Springer, 1996, pp. 104–113.

[9] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun, *Differential power analysis.*, CRYPTO (Michael J. Wiener, ed.), Lecture Notes in Computer Science, vol. 1666, Springer, 1999, pp. 388–397.

[10] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan, *Examining smart-card security under the threat of power analysis attacks.*, IEEE Trans. Computers **51** (2002), no. 5, 541–552.

[11] Microchip Technology Inc., *Application Note – Migrating Applications to USB from RS-232 UART with Minimal Impact on PC Software*, 2004, Also available at `http://ww1.microchip.com/downloads/en/AppNotes/00956b.pdf`.

[12] ———, *PICDEM FS USB Demonstration Board User's Guide*, 2004, Also available at `http://ww1.microchip.com/downloads/en/DeviceDoc/51526a.pdf`.

[13] W.H. Press, S.A. Teukolsky, W.T. Vettering, and B.P. Flannery, *Numerical Recipes in C++ – Second Edition*, Cambridge University Press, Cambridge, UK, 2002.

[14] Todd A. Proebsting and Scott A. Watterson, *Krakatoa: Decompilation in java (does bytecode reveal source?)*, COOTS, 1997, pp. 185–198.

[15] R. Radhakrishnan, J. Rubio, and L. John, *Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels*, 1999.

[16] Wolfgang Rankl and Wolfgang Effing, *Smart Card Handbook, 3rd Edition*, John Wiley & Sons, Ltd., New York, NY, USA, 2003.

[17] STMicroelectronics, `http://www.st.com/stonline/products/families/smartcard/sc_sol_secure_ics_st22.htm`, 2005.

[18] Sun Microsystems, Inc., *Java Card 2.1.1 Runtime Environment Specification*, 2000.

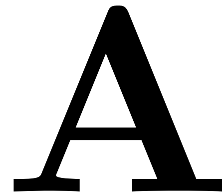
[19] ———, *Java Card 2.1.1 Virtual Machine Specification*, 2000.

[20] ———, `http://www.sun.com/smi/Press/sunflash/2004-11/sunflash.20041102.1.xml`, 2004.

[21] ———, `http://www.sun.com/smi/Press/sunflash/2005-11/sunflash.20051115.2.xml`, 2005.

[22] Marc Witteman, *Advances in smartcard security*, Information Security Bulletin **7** (2002), 11–22, Also available at `http://www.riscure.com/articles/ISB0707MW.pdf`.

JCVM instructions



Dec	Hex	Mnemonic	Description
0	0x00	nop	Do nothing
1	0x01	aconst_null	Push null
2	0x02	sconst_m1	Push short constant -1
3	0x03	sconst_0	Push short constant 0
4	0x04	sconst_1	Push short constant 1
5	0x05	sconst_2	Push short constant 2
6	0x06	sconst_3	Push short constant 3
7	0x07	sconst_4	Push short constant 4
8	0x08	sconst_5	Push short constant 5
9	0x09	iconst_m1	Push int constant -1
10	0x0A	iconst_0	Push int constant 0
11	0x0B	iconst_1	Push int constant 1
12	0x0C	iconst_2	Push int constant 2
13	0x0D	iconst_3	Push int constant 3
14	0x0E	iconst_4	Push int constant 4
15	0x0F	iconst_5	Push int constant 5
16	0x10	bspush	Push byte
17	0x11	sspush	Push short
18	0x12	bipush	Push byte
19	0x13	sipush	Push short
20	0x14	iipush	Push int
21	0x15	aload	Load reference from local variable
22	0x16	sload	Load short from local variable
23	0x17	iload	Load int from local variable
24	0x18	aload_0	Load reference from local variable 0
25	0x19	aload_1	Load reference from local variable 1
26	0x1A	aload_2	Load reference from local variable 2
27	0x1B	aload_3	Load reference from local variable 3

Dec	Hex	Mnemonic	Description
28	0x1C	sload_0	Load short from local variable 0
29	0x1D	sload_1	Load short from local variable 1
30	0x1E	sload_2	Load short from local variable 2
31	0x1F	sload_3	Load short from local variable 3
32	0x20	iload_0	Load int from local variable 0
33	0x21	iload_1	Load int from local variable 1
34	0x22	iload_2	Load int from local variable 2
35	0x23	iload_3	Load int from local variable 3
36	0x24	aaload	Load reference from array
37	0x25	baload	Load byte or boolean from array
38	0x26	saload	Load short from array
39	0x27	iaload	Load int from array
40	0x28	astore	Store reference into local variable
41	0x29	sstore	Store short into local variable
42	0x2A	istore	Store int into local variable
43	0x2B	astore_0	Store reference into local variable 0
44	0x2C	astore_1	Store reference into local variable 1
45	0x2D	astore_2	Store reference into local variable 2
46	0x2E	astore_3	Store reference into local variable 3
47	0x2F	sstore_0	Store short into local variable 0
48	0x30	sstore_1	Store short into local variable 1
49	0x31	sstore_2	Store short into local variable 2
50	0x32	sstore_3	Store short into local variable 3
51	0x33	istore_0	Store int into local variable 0
52	0x34	istore_1	Store int into local variable 1
53	0x35	istore_2	Store int into local variable 2
54	0x36	istore_3	Store int into local variable 3
55	0x37	aastore	Store into reference array
56	0x38	bastore	Store into byte or boolean array
57	0x39	sastore	Store into short boolean array
58	0x3A	iastore	Store into int boolean array
59	0x3B	pop	Pop top operand stack word
60	0x3C	pop2	Pop top two operand stack words
61	0x3D	dup	Duplicate top operand stack word
62	0x3E	dup2	Duplicate top two operand stack words
63	0x3F	dup_x	Duplicate top operand stack words and insert below

Dec	Hex	Mnemonic	Description
64	0x40	swap_x	Swap top two operand stack words
65	0x41	sadd	Add short
66	0x42	iadd	Add int
67	0x43	ssub	Subtract short
68	0x44	isub	Subtract int
69	0x45	smul	Multiply short
70	0x46	imul	Multiply int
71	0x47	sdiv	Divide short
72	0x48	idiv	Divide int
73	0x49	srem	Remainder short
74	0x4A	irem	Remainder int
75	0x4B	sneg	Negate short
76	0x4C	ineg	Negate int
77	0x4D	sshl	Shift left short
78	0x4E	ishl	Shift left int
79	0x4F	sshr	Arithmetic shift right short
80	0x50	ishr	Arithmetic shift right int
81	0x51	sushr	Logical shift right short
82	0x52	iushr	Logical shift right int
83	0x53	sand	Boolean AND short
84	0x54	iand	Boolean AND int
85	0x55	sor	Boolean OR short
86	0x56	ior	Boolean OR short
87	0x57	sxor	Boolean XOR short
88	0x58	ixor	Boolean XOR int
89	0x59	sinc	Increment local short variable by constant
90	0x5A	iinc	Increment local int variable by constant
91	0x5B	s2b	Convert short to byte
92	0x5C	s2i	Convert short to int
93	0x5D	i2b	Convert int to byte
94	0x5E	i2s	Convert int to short
95	0x5F	icmp	Compare int
96	0x60	ifeq	Branch if short comparison with zero succeeds (equal)
97	0x61	ifne	Branch if short comparison with zero succeeds (not equal)

Dec	Hex	Mnemonic	Description
98	0x62	`iflt`	Branch if `short` comparison with zero succeeds (less than)
99	0x63	`ifge`	Branch if `short` comparison with zero succeeds (greater than or equal)
100	0x64	`ifgt`	Branch if `short` comparison with zero succeeds (greater than)
101	0x65	`ifle`	Branch if `short` comparison with zero succeeds (less than or equal)
102	0x66	`ifnull`	Branch if reference is null
103	0x67	`ifnonnull`	Branch if reference is not null
104	0x68	`if_acmpeq`	Branch if reference comparison succeeds (equal)
105	0x69	`if_acmpne`	Branch if reference comparison succeeds (not equal)
106	0x6A	`if_scmpeq`	Branch if `short` comparison succeeds (equal)
107	0x6B	`if_scmpne`	Branch if `short` comparison succeeds (not equal)
108	0x6C	`if_scmplt`	Branch if `short` comparison succeeds (less than)
109	0x6D	`if_scmpge`	Branch if `short` comparison succeeds (greater than or equal)
110	0x6E	`if_scmpgt`	Branch if `short` comparison succeeds (greater than)
111	0x6F	`if_scmple`	Branch if `short` comparison succeeds (less than or equal)
112	0x70	`goto`	Branch always
113	0x71	`jsr`	Jump subroutine
114	0x72	`ret`	Return from subroutine
115	0x73	`stableswitch`	Access jump table by `short` index and jump
116	0x74	`itableswitch`	Access jump table by `int` index and jump
117	0x75	`slookupswitch`	Access jump table by key match and jump
118	0x76	`ilookupswitch`	Access jump table by key match and jump
119	0x77	`areturn`	Return reference from method
120	0x78	`sreturn`	Return `short` from method
121	0x79	`ireturn`	Return `int` from method
122	0x7A	`return`	Return `void` from method
123	0x7B	`getstatic_a`	Get static reference field from class

Dec	Hex	Mnemonic	Description
124	0x7C	getstatic_b	Get static byte or boolean field from class
125	0x7D	getstatic_s	Get static short field from class
126	0x7E	getstatic_i	Get static int field from class
127	0x7F	putstatic_a	Set static reference field in class
128	0x80	putstatic_b	Set static byte or boolean field in class
129	0x81	putstatic_s	Set static short field in class
130	0x82	putstatic_i	Set static int field in class
131	0x83	getfield_a	Fetch reference field from object
132	0x84	getfield_b	Fetch byte or boolean field from object
133	0x85	getfield_s	Fetch short field from object
134	0x86	getfield_i	Fetch int field from object
135	0x87	putfield_a	Set reference field in object
136	0x88	putfield_b	Set byte or boolean field in object
137	0x89	putfield_s	Set short field in object
138	0x8A	putfield_i	Set int field in object
139	0x8B	invokevirtual	Invoke instance method; dispatch based on class
140	0x8C	invokespecial	Invoke instance method; special handling for superclass, private, and instance initialization method invocations
141	0x8D	invokestatic	Invoke a static class method
142	0x8E	invokeinterface	Invoke interface method
143	0x8F	new	Create new object
144	0x90	newarray	Create new array
145	0x91	anewarray	Create new array of reference
146	0x92	arraylength	Get length of array
147	0x93	athrow	Throw exception or error
148	0x94	checkcast	Check whether object is of given type
149	0x95	instanceof	Determine if object is of given type
150	0x96	sinc_w	Increment local short variable by constant
151	0x97	iinc_w	Increment local int variable by constant
152	0x98	ifeq_w	Branch if short comparison with zero succeeds (wide index, equal)
153	0x99	ifne_w	Branch if short comparison with zero succeeds (wide index, not equal)
154	0x9A	iflt_w	Branch if short comparison with zero succeeds (wide index, less than)

Dec	Hex	Mnemonic	Description
155	0x9B	ifge_w	Branch if short comparison with zero succeeds (wide index, greater than or equal)
156	0x9C	ifgt_w	Branch if short comparison with zero succeeds (wide index, greater than)
157	0x9D	ifle_w	Branch if short comparison with zero succeeds (wide index, less than or equal)
158	0x9E	ifnull_w	Branch if reference is null (wide index)
159	0x9F	ifnonnull_w	Branch if reference is not null (wide index)
160	0xA0	if_acmpeq_w	Branch if reference comparison succeeds (wide index, equal)
161	0xA1	if_acmpne_w	Branch if reference comparison succeeds (wide index, not equal)
162	0xA2	if_scmpeq_w	Branch if short comparison succeeds (wide index, equal)
163	0xA3	if_scmpne_w	Branch if short comparison succeeds (wide index, not equal)
164	0xA4	if_scmplt_w	Branch if short comparison succeeds (wide index, less than)
165	0xA5	if_scmpge_w	Branch if short comparison succeeds (wide index, greater than or equal)
166	0xA6	if_scmpgt_w	Branch if short comparison succeeds (wide index, greater than)
167	0xA7	if_scmple_w	Branch if short comparison succeeds (wide index, less than or equal)
168	0xA8	goto_w	Branch always (wide index)
169	0xA9	getfield_a_w	Fetch reference field from object (wide index)
170	0xAA	getfield_b_w	Fetch byte or boolean from object (wide index)
171	0xAB	getfield_s_w	Fetch short field from object (wide index)
172	0xAC	getfield_i_w	Fetch int field from object (wide index)
173	0xAD	getfield_a_this	Fetch reference field from current object
174	0xAE	getfield_b_this	Fetch byte or boolean field from current object
175	0xAF	getfield_s_this	Fetch short field from current object
176	0xB0	getfield_i_this	Fetch int field from current object
177	0xB1	putfield_a_w	Set reference field in object (wide index)
178	0xB2	putfield_b_w	Set byte or boolean field in object (wide index)

Dec	Hex	Mnemonic	Description
179	0xB3	putfield_s_w	Set short field in object (wide index)
180	0xB4	putfield_i_w	Set int field in object (wide index)
181	0xB5	putfield_a_this	Set reference field in current object
182	0xB6	putfield_b_this	Set byte or boolean field in current object
183	0xB7	putfield_s_this	Set short field in current object
184	0xB8	putfield_i_this	Set int field in current object
254	0xFE	impdep1	Reserved opcode (Cannot appear in valid CAP file)
255	0xFF	impdep2	Reserved opcode (Cannot appear in valid CAP file)

Flowchart smart card reader B

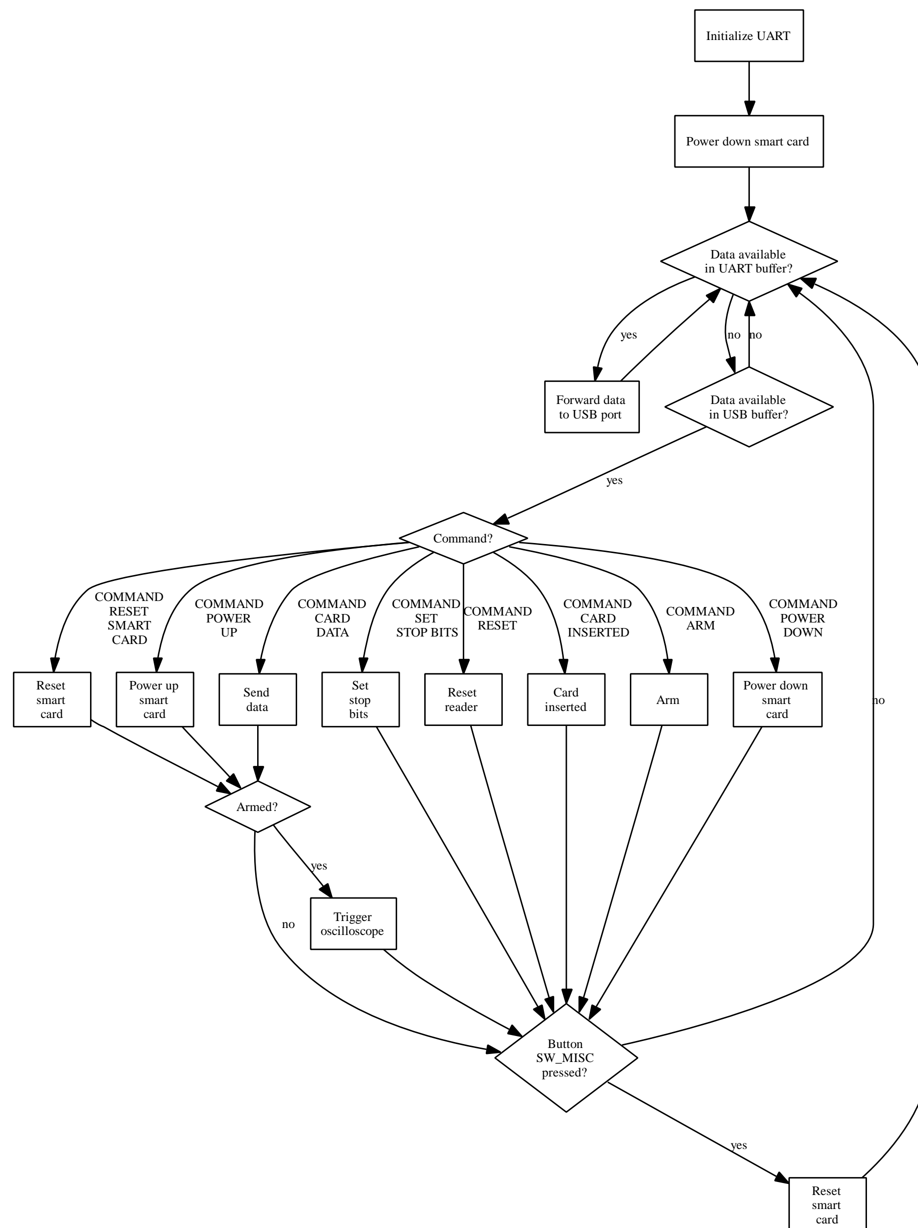


Figure B.1: Flowchart smart card reader firmware